VULNERABLE BY DESIGN: MITIGATING DESIGN FLAWS IN HARDWARE AND SOFTWARE

PH.D. THESIS

RADHESH KRISHNAN KONOTH

VRIJE UNIVERSITEIT AMSTERDAM, 2020

Faculty of Science

The research reported in this dissertation was conducted at the Faculty of Science — at the Department of Computer Science — of the Vrije Universiteit Amsterdam

This work was supported by the *MALPAY* consortium, consisting of the Dutch national police, ING, ABN AMRO, Rabobank, Fox-IT, and TNO. This thesis represents the position of the author and not that of the aforementioned consortium partners

VRIJE UNIVERSITEIT

VULNERABLE BY DESIGN: MITIGATING DESIGN FLAWS IN HARDWARE AND SOFTWARE

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor of Philosophy aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op maandag 7 december 2020 om 13.45 uur
in de online bijeenkomst van de universiteit,
De Boelelaan 1105

door

Radhesh Krishnan Konoth

geboren te Koorkenchery, India

promotor: prof.dr.ir. H. J. Bos

copromotor: dr. K. Razavi

ഞാൻ ഈ പുസ്തകം എന്റെ അച്ഛനും അമ്മയ്ക്കും സമർപ്പിക്കുന്നു

“Those that have an attitude of service towards others
are the beauty of society."

- Amma

Acknowledgements

Let me begin by acknowledging the people who supported me during this wonderful journey.

First and foremost, I am greatly indebted to my supervisor Herbert Bos for providing me the opportunity to do a Ph.D. Thank you Herbert for taking that risk and for believing in me. This journey would not have been possible without your support and guidance. Your dedication, leadership, and the ability to sell any idea are such an inspiration. I truly couldn't have wished for a better supervisor. You are the best!

I am deeply grateful to my co-promoter Kaveh Razavi. Only amazing things happened from the moment you joined the team. You gave me the idea and opportunity to work on a crazy Rowhammer defense project and to do an internship at UCSB. Both ideas ended up as two amazing chapters of this Ph.D. thesis. Every conversation with you was inspiring and you always pushed me to create the best research. I am grateful for all those pushes, guidance, and support. You made this journey a lot more exciting for me. I am indebted.

Next, I would like to thank members of my Ph.D. thesis committee: Clémentine Maurice, Christian Rossow, Erik van der Kouwe, Michel van Eeten, Marten van Dijk and Wan Fokkink. Thank you for your allotting time and energy to review my thesis.

During my Ph.D. I was fortunate to work with a lot of smart researchers. My sincere thanks to Victor, Andrei, Marco, Cristiano, Dennis, Elias, Enes, Wan, Martina, Veelasha, Emanuele, Björn, Christopher, Giovanni. This thesis wouldn't have existed without your support.

VUSec is a group of amazing researchers. Everyone there made this journey very special to me. Thank you Sanjay, Manolis, Taddeüs, Koen, Ben, Erik van der Kouwe, Erik Bosman, Chen, Istvan, Koustubha, Lucian, Natalie, Pietro, Sebastiaan, Stephan, Elia, Brian, Hany, Emanuele, Alyssa, Marius, and Manuel.

I am grateful to all the teachers of my life, friends, and family. Thank you

Renuka, Vipin, Zubin, Praveen, Sreekumar, Roshan, Vinod, Candice, Elisabeth, Amrutha, Sreejith, and Akshara (my only niece, yet).

I would like to thank my dad and my mom for all the motivation, support, and love. This Ph.D. thesis is a result of your efforts, sacrifices, and prayers. Thank you from the bottom of my heart.

At the heart of it all, let me express my humble gratitude towards Amma (Mata Amritanandamayi Devi) for her inspiration, guidance, and grace. Without her support, neither would I have started this journey nor would I have concluded this journey. Thank you, Amma.

Radhesh
Amsterdam, December 2020

Contents

Publications

Parts of this dissertation have been published earlier. The text in this thesis differs from the published versions in minor editorial changes that were made to improve readability. The following publications form the core of this thesis.

R. K. Konoth, V. van der Veen, and H. Bos. **How anywhere computing just killed your phone-based two-factor authentication**. In *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016. [Appears in Chapter 2.]

R. K. Konoth, B. Fischer, W. Fokkink, E. Athanasopoulos, K. Razavi, and H. Bos. **SecurePay: Strengthening two-factor authentication for arbitrary transactions**. In *Proceedings of the* 5^{th} *IEEE European Symposium on Security and Privacy (EuroS&P)*. Sep. 2020. [Appears in Chapter 3.]

R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi. **ZebRAM: Comprehensive and compatible software protection against rowhammer attacks**. In *Proceedings of the* 13^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2018. [Appears in Chapter 4.]

R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna. **Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense**. In *Proceedings of the* 25^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2018. [Appears in Chapter 5.]

The following publications are not included in this dissertation.

A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi. **Throwhammer: Rowhammer attacks over the network and defenses**. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*. Jul. 2018.

B. Kollenda, E. Göktaş, T. Blazytko, P. Koppe, R. Gawlik, R. K. Konoth, C. Giuffrida, H. Bos, and T. Holz. **Towards automated discovery of crash-resistant primitives in binary executables**. In *Proceedings of the* 47^{th} *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Jun. 2017.

1 Introduction

Design flaws and implementation bugs are two different types of security defects. Studies estimate that design flaws, errors that occur at the design phase of the product development lifecycle, constitute 50% of the vulnerabilities in computer systems [105]. Often they are the results of features introduced by vendors and product developers for improving the usability or performance of computer systems. On the other hand, implementation bugs are the errors introduced at the implementation phase of the product development lifecycle. Design flaws and implementation bugs can occur in both software and hardware components of computer systems.

Today, a lot of research and tools exist to identify and mitigate implementation bugs. However, these tools cannot mitigate the security impacts of design flaws because, unlike exploiting an error in the implementation like a memory corruption bug, they are more about bad actors taking advantage of an unintended consequence of a feature. Thus, it is typically difficult to detect, and complex to patch a design flaw when compared to an implementation bug — often requiring solutions unique to each attack. Clearly, more research is desirable for identifying and mitigating cyber threats stemmed from design flaws.

Motivation and Problem Statement

As the vendors and product developers are constantly introducing new features for enhancing the usability and performance of the computer systems, increasing the level of complexity, size and attack surface, the secure software development process is becoming more and more challenging. Requirement gathering, design, implementation, and testing are the four common stages of the software development lifecycle. Many organizations consider security only at later stages of this software development lifecycle — especially at the implementation and testing phase, where they can utilize static and dynamic analysis tools. However, such tools can only find security defects (bugs) introduced in the implementation

Table 1.1. Design principles by Saltzer and Schroeder

Design Principle	Description
Economy of mechanism	A simpler design is easier to test and therefore the design should be as simple and small as possible.
Fail-safe defaults	Access should be denied by default and only be permitted when explicit permission exists.
Complete mediation	Every access to every object must be checked for authority.
Open design	The security of a mechanism should not depend on the secrecy of its design or implementation.
Separation of privilege	Access should be granted based on more than one piece of information. Protection mechanism that requires two keys are more robust and flexible.
Least privilege	Every process and user should be given the least set of privileges that it needs in order to complete its task.
Least common mechanism	The protection mechanism should be shared as little as possible among users.
Psychological acceptability	The protection mechanism should be easy to use.

phase, not the security defects introduced in the earlier stages of the product development lifecycle (such as design flaws). Since it is more expensive and difficult to fix such security defects at a later stage, it would be a better approach to layer security throughout the product development lifecycle and natively build into it from the beginning.

We can classify any security defect into either an *implementation bug* or a *design flaw*, and such defects can occur at both hardware and software. However, it is more expensive and sometimes impractical to fix a security defect that occurred at the hardware-level. Hence, the hardware components should be even more carefully designed and implemented than the software components.

An implementation error at the software level is typically called a *software bug*. A *software bug* is a coding error that causes the program to behave in unintended ways. Most software contains such bugs — yet they are fixable. Nowadays, exploiting a software bug such as a memory-corruption bug to compromise computers and gain access to organizations is all too common. A lot of research has been already done on identifying and mitigating such threats. Similarly, an implementation bug at the hardware level is known as a *hardware bug*. Typically, they are much harder to fix. Sometimes the hardware is supported by embedded firmware (for instance, the microcode in CPUs). In that case, the hardware bug may be mitigated by a firmware update.

In contrast, an error that occurrs at the design phase of the software (or hardware) development lifecycle is called a *design flaw* or *logic flaw*. Sometimes it consists of an otherwise legitimate function or feature with unintended conse-

Table 1.2. Design principles by Viega and McGraw

Design Principle	Description
Secure the weakest link	The level of security is only as strong as the weakest link in the system.
Practice defense in depth	Use multiple complementary security mechanisms, so that a failure in one does not mean total insecurity.
Fail securely	Design the system so it fails in a secure manner.
Follow the principle of least privilege	Only the minimum access necessary to perform an operation should be granted, and that access should be granted only for the minimum amount of time.
Compartmentalize	Segment a system into multiple components that are protected independently to reduce the damage of an attack.
Keep it simple	Avoid unnecessary complexity by keeping the system as simple as possible.
Promote privacy	Promote privacy for the users and the system.
Remember that hiding secrets is hard	This principle assumes that even the most secure systems are amenable to inside attacks.
Be reluctant to trust	Instead of making assumptions that need to hold true, you should be reluctant to extend trust.
Use your community resources	Use well-known community resources that have been widely scrutinized and used.

quences that attackers seek out to exploit. For instance, attackers can exploit them to gain root access to highly secure systems, leading to data theft, disruption of critical infrastructure, and other serious consequences. Unlike bugs, a design flaw is typically difficult to patch; moreover, automated tools cannot always find such security defects easily.

Design principles are specific guidelines that can be followed during the design phase of the product development lifecycle to avoid such flaws. These principles demonstrate what to consider to enhance security as well as to aid in the development of a new computer system. In 1975, Saltzer and Schroeder [103] presented a series of design principles for secure systems, which apply especially to protection mechanisms (see Table 1.1). Building on that later in 2001, Viega and McGraw [106] also introduced a set of design principles to improve the secure software development process (see Table 1.2).

Unfortunately, many developers either consider security only in the implementation phase or fail to adhere to these design principles thereby introducing design flaws [122]. The *Rowhammer bug* is an example of a hardware level design flaw. DRAM vendors are making DRAM chips denser to increase DRAM capacity and the lower energy consumption. Unfortunately, this design choice to optimize the cost-per-bit by cramming bits very close together increases the possibility of

memory errors in the DRAM chip owing to the smaller difference in charge between a "0" bit and a "1" bit. Previous research showed that it is possible to force memory errors in DDR3 memory by activating a row many times in quick succession, causing capacitors in neighboring *victim* rows to leak their charge before the memory controller has a chance to refresh them. This rapid activation of memory rows to flip bits in neighboring rows is known as the *Rowhammer attack*. Subsequent research has shown that the bit flips induced by Rowhammer are highly reproducible and can be exploited in a multitude of ways. These bit flips/memory errors happen in hardware without any indication, hence the system fails to detect such memory errors. This design of memory chips violates the design principle called *Fail securely* (see Table 1.2). The basic idea behind *Fail securely* is that when a system fails, it should do so securely; the confidentiality and integrity of a system should remain intact even though availability may have been lost. The attackers must not be permitted to gain access rights to privileged objects during a failure that are normally inaccessible. Clearly, the vendors did not think of the security implications of such memory errors in the design phase.

This leads us to ask the following questions: (i) Do similar exploitable design flaws also exist in modern software systems that deal with highly sensitive operations such as financial transactions? (ii) Can we still mitigate the cyber threats stemming from these design flaws under the assumption that the flaws themselves cannot be fixed (for practical reasons)? (iii) Is the current set of design principles comprehensive enough to prevent today's cyber threats? These are the overarching questions that we try to answer in this thesis. Of course, we cannot solve all design flaws in hardware and software in a generic manner. We will focus on two specific issues in this thesis, with very specific solutions. However, they may serve as examples that even though the security problem is deep in the design of the a system we may still be able to fix them later.

Research Questions

The goal of this work is to study and build computer defenses that primarily focus on preventing exploitation based on design flaws. First, we focus on new cyber threats that emerged from such design flaws at both the software and hardware level. Then, we study whether the current set of design principles is comprehensive enough to prevent today's cyber threats. For the purpose of this study the following research questions have been defined:

Question (1): *Given that exploitable design flaws exist in hardware, can we also*

discover them in software that deals with highly sensitive operations such as financial transactions?

Question (2): *Given that it is harder to fix such a design/logical flaw when compared to patching a typical software bug, can we still mitigate the cyber threat stemming from the design flaw that we identified as part of our first research question under the assumption that the flaw itself cannot be fixed for practical reasons? What will be the cost of such a solution?*

Question (3): *Given that it is often more complex to patch a design/logical flaw in a hardware component, can we still mitigate the cyber threat originated from the Rowhammer bug using a software-based solution? What will be the cost of such a solution?*

Question (4): *Given that cyber attacks have evolved over time to become more stealthy and complex, is the current set of design principles comprehensive enough to prevent today's cyber threats?*

Organization

This dissertation makes several contributions, with results published in refereed conferences and workshops (Page xv). The remainder is organized as follows:

- **Chapter 2** presents BANDROID, an attack that exploits a design or logical flaw at the software level to break mobile-based two-factor authentication provided by a wide range of (financial) web services. In broad strokes, the scenario is as follows. If attackers have control over the browser on the PC of a user using Google services (like Gmail, Google+, etc.), they can push any app with any permission on any of the user's Android devices using this *remote install* feature, and activate it — allowing one to bypass two-factor authentication via the phone. Thus, we show that it is practical to design a cyber-attack that exploits a design/logical flaw (which in this case is the *remote install* feature). We conclude the chapter discussing the design principle this usability feature violates.

  I share first authorship on BANDROID with Victor van der Veen. In particular, I found the vulnerability in Google Play and built a proof of concept. During the first years of my PhD, I strengthened the exploit for Android by adding a *name masquerading* technique and developed an exploit for the

iOS platform. Victor focused on generalizing my findings allowing us to present a coherent story.

Chapter 2 appeared in the *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC 2016)* [44].

- **Chapter 3** presents SecurePay, a software-based defense to mitigate the Bandroid attack which is the result of a design flaw (as well as other attacks on 2FA). In the previous chapter, we saw that the separation between the factors has weakened by the *remote install* feature and one compromised device may be enough to break current mobile-based 2FA mechanisms. In this chapter, we identify the basic principles for securing any transaction using mobile-based 2FA and build a defense against the Bandroid cyber attack. We argue that the computing system should not only provide isolation between the two factors, but also the integrity of the transaction while involving the user in confirming the authenticity of the transaction. We show for the first time how these properties can be provided on commodity mobile phones, securing 2FA-protected transactions even when the operating system on the phone is fully compromised by a cyber attack like the Bandroid attack. The design of SecurePay is deliberately minimalistic to adhere to Saltzer and Schroeder's design principles of Economy of Mechanism, Least Common Mechanism, Least Authority, and Privilege Separation [103].

  As the first author of the paper, my focus was mainly on designing, implementing and evaluating SecurePay.

  Chapter 3 appeared in the *Proceedings of the* 5^{th} *IEEE European Symposium on Security and Privacy (EuroS&P 2020)* [42]. SecurePay won the *Best Paper Award* at EuroS&P 2020.

- **Chapter 4** presents ZebRAM, a novel and comprehensive software-level protection against the Rowhammer attack which exploits a design flaw in the memory hardware (DRAM). The vendor's design choice to optimize the cost-per-bit by cramming bits so close together increased the possibility of memory errors. Kim et al. [39] show that intentionally activating a row many times in a short duration (i.e., Rowhammering) can cause the charge in the capacitors to leak in close-by rows. This rapid activation of memory rows to flip bits in neighboring rows is known as the Rowhammer attack. As the bit flips induced by Rowhammer happen without any indication, the

system fails to detect it. Subsequent research has shown that the bit flips induced by Rowhammer are highly reproducible and can be exploited in a multitude of ways and even occur in the newer DDR4 memory chips [28].

ZebRAM isolates every DRAM row that contains data with guard rows that absorb any Rowhammer induced bit flips. This is the only way to protect against all forms of Rowhammer. Rather than leaving guard rows unused, ZebRAM improves performance by using the guard rows as efficient, integrity-checked and optionally compressed swap space. We conclude the chapter by discussing the design principle this memory chip design violates and the cost of the software-based solution.

As the first author of the paper, my focus was mainly on designing, implementing and evaluating security provided by ZebRAM. Marco Oliverio, the second author of the paper, evaluated the performance overhead of the ZebRAM.

Chapter 4 appeared in the *Proceedings of the* 13^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)* [43].

- **Chapter 5** presents MineSweeper, an in-depth study of a new class of cyber threat called *cryptojacking*. This new class of cyber threats neither violates any of the current design principles nor exploits an implementation bug. It does not break today's widely-accepted security model known as the CIA triad (standing for Confidentiality, Integrity, and Availability); still it is a very practical and stealthy cyber attack that monetizes of a victim's computational resources. In this chapter, we first perform a comprehensive analysis on Alexa's Top 1 Million websites to shed light on the prevalence and profitability of *cryptojacking*. Then, we present possible countermeasures against this cyber threat. Finally, we conclude the chapter by proving that we need to update the design principles to address this class of new cyber threat.

  As the first author of the paper, my focus was mainly on designing and implementing the large-scale analysis framework and designing the countermeasures.

  Chapter 5 appeared in the *Proceedings of the* 25^{th} *ACM Conference on Computer and Communications Security (CCS 2018)* [45].

- **Chapter 6** concludes this thesis, recapitulating our main results and discussing perspectives for future work.

2 Killing Phone-Based Two-Factor Authentication Exploiting a Design Flaw

Exponential growth in smartphone usage combined with recent advances in mobile technology is causing a shift in (mobile) app behavior: application vendors no longer restrict their apps to a single platform, but rather add synchronization options that allow users to conveniently switch from mobile to PC or vice versa in order to access their services. This process of integrating apps among multiple platforms essentially removes the gap between them. Current, state of the art, mobile phone-based two-factor authentication (2FA) mechanisms, however, heavily rely on the existence of such separation. They are used in a variety of segments (such as consumer online banking services) to protect against malware. For example, with 2FA in place, attackers should no longer be able to use their PC-based malware to instantiate fraudulent banking transactions. Clearly, the synchronization options are violation of the design principle called *compartmentalize* (see Table 1.2). In this chapter, we analyze the security implications of diminishing gaps between platforms and show that the ongoing integration and desire for increased usability results in violation of key principles for mobile phone 2FA. As a result, we identify a new class of vulnerabilities dubbed *2FA synchronization vulnerabilities*. To support our findings, we present practical attacks against Android and iOS that illustrate how a Man-in-the-Browser attack can be elevated to intercept One-Time Passwords sent to the mobile phone and thus bypass the chain of 2FA mechanisms as used by many financial services.

2.1 Introduction

Approaching an impressive 1.25 billion sales in 2014 with an expected audience of over 1.75 billion, smartphones have become an important factor in many people's day-to-day life [144, 206]. Daily activities performed on these mobile devices include those that can be done on PC as well: accessing e-mail, searching the web, social networking, or listening to music [148]. To enhance usability, both application developers and platform vendors are making an effort to blur boundaries between the two platforms. This is reflected in synchronization features like Firefox Sync and Samsung SideSync or sophisticated market places like Google Play and Microsoft's Windows Store that allow users to manage their mobile phone remotely.

A second important trend in web computing is the increasing number of applications that provide the possibility to harden user accounts by enabling *2 Factor Authentication* (2FA) for them. 2FA is a form of multi-factor authentication and provides unambiguous identification of users by means of the combination of two different components, i.e., something the user *knows* (PIN code, password) and something the user *possesses* (bank card, USB stick token). With 2FA enabled, if attackers steal a user's password, they still require access to the second component before they can impersonate the victim.

Not surprisingly, software vendors often embody the second component of 2FA in the form of a mobile phone. To authenticate, the web application sends a one-time-valid, dynamic passcode to the user's mobile phone (for instance via SMS, e-mail, or a dedicated application), which must then be entered along with the user's credentials in order to complete the authentication. Since users usually carry their phone all the time, *Mobile Phone 2FA* does not introduce additional costs and can be implemented relatively easy. Examples of well-known companies that provide mobile phone 2FA include Amazon, Apple, Dropbox, Google, Microsoft, Twitter, Yahoo, and many more, including a large number of financial institutions[1]. The latter is represented by many of the biggest financial organizations in the world such as Bank of America, Wells Fargo, JP Morgan Chan, ICBC in China, and ING in The Netherlands.

In this chapter, we analyze the security implications of *Anywhere Computing* and show that seamless platform integration comes at the cost of weakening the (commonly perceived) strong mobile phone 2FA mechanism. We define a new class of vulnerabilities dubbed *2FA synchronization vulnerabilities* and show how these can be exploited by an attacker. In particular, we present reliable attacks

[1] http://twofactorauth.org

against both Android and iOS, two platforms that represent a combined market share of over 90% [188]. Our threat model is the same as that of 2FA: we assume that a victim's PC has been compromised, allowing an attacker to perform Man-in-the-Browser (MitB) attacks. In this scenario, mobile phone 2FA should guarantee that the attacker cannot perform authorized operations without having also access to the user's phone. By exploiting certain 2FA synchronization vulnerabilities, however, we show that mobile phone 2FA as used by many online services for secure authentication, including financial institutions, can be easily bypassed.

In more detail, our first attack utilizes Google Play's remote app installation feature to install a specifically crafted *vulnerable* app onto registered Android devices of the victim which is then silently activated and used to hijack One-Time Passwords (OTPs). Our iOS attack, on the other hand, exploits a new OS X feature that enables the synchronization of SMS messages between iPhone and Mac.

Although the security of 2FA has been subject of prior work [20], we believe that our work is the first to address weaknesses relating to ongoing synchronization and usability enhancement efforts.

Contributions In summary, our contributions are the following:

1. We identify a new class of vulnerabilities, *2FA synchronization vulnerabilities*, that weaken the security guarantees of mobile phone 2FA.

2. We present practical attacks against Android and iOS that exploit multiple 2FA synchronization vulnerabilities and show how these can be used to successfully bypass mobile phone 2FA.

3. We discuss the security implications of our findings and provide recommendations for various stakeholders. Based on our findings, we conclude that SMS-based 2FA should be considered unsafe.

The remainder of this chapter is organized as follows. In Section 2.2, we outline current efforts deployed by vendors that ease platform integration and provide a definition of *2FA synchronization vulnerabilities*. Section 2.3 details our attacks against Android and iOS which can be used to bypass mobile phone 2FA. We discuss security implications and recommendations in Section 2.4, followed by a related work study on the evolution of Man-in-the-Browser attacks and 2FA in Section 2.5. We conclude in Section 2.6.

2.2 Synchronization

To maximize connectivity and to ensure that users never miss another status update, vendors continuously come up with ways to close the gap between PC and mobile devices. In this section, we separate these integration techniques into two categories: (1) remote services as provided by mobile operating system vendors and (2) integration of applications across the different platforms using synchronization features. Finally, we define *2FA synchronization vulnerabilities* in detail and show example vulnerabilities that we later use to break mobile phone 2FA.

2.2.1 Remote Services

Mobile operating system market leader Google provides a *remote install* service in its Play Store that allows users to install Android applications on any of their phones or tablets, from a desktop computer. The process is painless and straightforward: a user (1) logs into the Google Play store, (2) picks an app of his interest, (3) hits the *install* button, (4) accepts the app's permissions, (5) chooses the device on which this app should be installed, and (6) confirms installation. The app is now automatically pushed and installed onto the selected phone—as soon as it has connectivity. Since all the app's permissions are requested and confirmed in the browser already, the only trace left on the phone is a *<app name> successfully installed* notification message. Similar features have been deployed in app stores of both Microsoft (Windows Phone) and Apple (iOS).

Naturally, platform vendors have adopted security policies to prevent exploitation of this feature. Focusing on Android, for example, Google, deployed two: (1) silent remote install only works for apps on Google Play, which is actively monitored for malware by Google Bouncer; and (2) newly installed apps default to a *deactivated* state which means that even if the app defines specific event receivers (e.g., on `BOOT_COMPLETED` to start a service at boot-time, or `SMS_RECEIVED` to listen for incoming SMS text messages), it cannot use these until the app is explicitly activated by the user. Activation is triggered by starting the app for a first time, either by selecting it from the launcher or by sending it an intent from another app (e.g., by opening a link from the mobile browser) [157].

In addition to remote install, platform vendors also provide features that help users in locating or wiping a lost device [128, 158, 180].

2.2.2 App Synchronization

Besides remote *services*, developers try to increase usability even further by incorporating cross-platform synchronization features in their *applications*. This

is best illustrated by looking at recent changes in browsers. Browsers once were self-contained software pieces that ran on a single device. Popular browsers like Google Chrome or Mozilla Firefox, however, nowadays offer integrated synchronization services. By using these features, users no longer have to configure browsers individually, but can automatically synchronize all their saved passwords, bookmarks, open tabs, browser history and settings across multiple devices [160, 184]. It is expected that Microsoft's Edge introduces similar functionality soon [197].

Another example of application synchronization is Apple's Continuity which features, among others, synchronization of SMS text messages between iOS (8.1 and up) and Mac OS X (10.10 Yosemite and later): "with Continuity, all the SMS and MMS text messages you send and receive on your iPhone also appear on your Mac, iPad, and iPod touch" [129].

2.2.3 2FA Synchronization Vulnerabilities

Given the ongoing efforts by both platform vendors and application developers to bridge the gap between the end-user's desktop and his or her mobile devices, we identify a new class of vulnerabilities that, while increasing usability, jeopardize 2FA security guarantees.

Definition *A 2FA synchronization vulnerability is a usability feature that deliberately blurs the boundaries between devices, but, potentially combined with other vulnerabilities, inadvertently weakens the security guarantees of 2FA.*

As an example, consider the previously discussed remote app installation feature: a clear product of a design decision aiming to enhance usability. Although such option successfully improves usability indeed—users can conveniently manage their mobile device from their browser—it comes with an obvious security risk: if attackers manage to get control over a user's browser, they can extend control to the user's mobile devices as well by pushing arbitrary apps to them. We thus identify the remote install feature as a 2FA synchronization vulnerability.

Focussing again on Android, Google's deployed security measures make that without additional vulnerabilities, attackers cannot abuse this synchronization vulnerability alone to bypass mobile phone 2FA. Finding such vulnerabilities is easy though. First, fundamental weaknesses in Google Bouncer expose multiple ways to bypass malware detection, giving attackers a sufficient time window to push malicious apps to Google Play and thus to mobile devices. Second, we identify numerous ways to activate apps after installation, either by exploiting

end-users' curiosity (*hey, what is this app?*) or by relying on additional synchronization vulnerabilities, for example in browser apps: previously discussed features can be used by an attacker to synchronize malicious bookmarks or browser tabs that, when opened on the mobile device, can activate deactive apps.

A second attack exploits the clear 2FA synchronization vulnerability introduced in recent Mac OS X releases. If Continuity is enabled, there is no need for attackers to control a victim's phone: they can read SMS messages from an infected Mac directly.

It is important to realize that 2FA synchronization vulnerabilities are not necessarily caused by bad developer habits or configuration mistakes. More often, they will be the result of a design decision-making process. This means that it is much harder to convince vendors of their mistakes: a 2FA synchronization vulnerability does not leak data or enable code execution, but must be considered within the mobile phone 2FA threat model before it becomes a threat.

2.3 Exploiting 2FA Synchronization Vulnerabilities

By exploiting the synchronization vulnerabilities discussed in Section 2.2, we can construct attacks that break mobile phone 2FA. In this section, we present practical implementations of such attacks against the two major mobile operating systems: Google Android and Apple iOS. Additionally, we show that synchronization vulnerabilities also imperil mobile phone 2FA implementations that use a dedicated app to transfer the OTP.

Our attacks operate on the basic threat model of 2FA: we assume that the attacker already has control over the victim's PC, possibly including a MitB, and is specifically interested in bypassing mobile phone 2FA.

2.3.1 Android

The intention of our Android attack is to exploit the remote install feature of Google Play to push a malicious app onto the user's mobile device. This app can then intercept and forward OTPs sent as SMS messages to a server that is controlled by the attacker. Given that the attackers have control over the user credentials (stolen by the MitB), this gives them sufficient means to bypass 2FA.

Google's deployed mitigation techniques slightly complicate our scenario. In order to successfully break 2FA, we need to address two defenses: (1) we need to bypass Google Bouncer before we can publish our SMS stealing app in Google Play, and (2) we need the user to activate the app before it can intercept and forward SMS messages.

Bypassing Google Bouncer Since Google's remote install feature only allows app installation from trusted sources, attackers first need to get an SMS stealing app published in Google Play. For this, they need to bypass Bouncer, Google's automated malware analysis tool that uses both static and dynamic analysis to identify malicious behavior [0]. Once an application is uploaded to Google Play, Bouncer starts analyzing it for known malware, spyware and trojans.

Although the inner workings of Bouncer are kept confidential, prior work has shown that it is easily circumvented [223, 67]. This is confirmed by a recent case study where Avast identified a number of popular Play Store apps that had over a million downloads to be in fact malware [136].

Orthogonal to recent work, our approach to trick Bouncer into accepting rogue apps is publishing a *vulnerable* application [82]. By pushing a poorly coded `WebView` application, for example, attackers no longer have to hide malicious code from Bouncer, but can simply move it to a web server that will be contacted by the app to display regular data [60]. An alternative, even harder to detect scheme, involves exposing a backdoor in native code via a memory corruption vulnerability [9].

To show the practicality of our attack, we successfully published an SMS 'backup' app in Google Play. Upon SMS reception, our app first writes the message content to a file, followed by loading a remote webpage inside a hidden webview component. The prepared webview component, however, is made vulnerable by exposing a `ProcessBuilder` class via the `addJavascriptInterface` API. This allows the remote webpage to execute arbitrary commands within the app's context using JavaScript.

Removing malicious code from the app makes it undetectable for Google Bouncer's static analysis. To also hide from dynamic analysis, we construct the remote webpage in such a way that it does not serve malicious commands when the incoming connection is made from a Google machine. In practice, to avoid accidental misuse, we instructed the webpage to only serve malicious code if accessed from an IP address that is under our control.

App Activation Once installed, Android puts new apps in a *deactivated* state. While deactivated, an app will not run for any reason, except after (1) a manual launch of its main activity via the launcher, or (2) an explicit intent from another app (e.g., a clicked link from the mobile browser) [166]. Attackers must thus somehow steer their victim into starting the app manually. We identify two reliable approaches to achieve this.

1. The most naive method is to hide the malicious activity inside an attrac-

tive container. By using a challenging or even provocative app name or icon, a user may be tempted into opening the app manually, simply out of curiosity.

2. Armed with both synchronization vulnerabilities and the victim's Google credentials obtained by the MitB, an attacker can manipulate saved bookmarks, recent tabs, or URLs used in e-mail, cloud documents, social media, etcetera, in such a way that, when clicked, they redirect to a malicious webpage. This page, controlled by the attacker, can then send the aforementioned intent to activate the malicious app.

To prevent a user from detecting the rogue app after it has been activated, we complement it with stealth features. Strictly abiding to the Android developers guidelines, we constructed our app in such a way that, once activated, it removes it's main icon from the launcher. Additionally, we use a name masquerading technique to maximize discretion: (1) the app name shown in the notification bar is different from (2) the name of the app as found in the launcher, which in its turn differs from (3) the official app name as shown in the *app overview* (accessible from the settings view). This works because (1) during app submission, the Google Developers Console does not check whether the provided app name matches the official app name as found in the uploaded `.apk`, and (2) the `<activity-alias>` tag inside the app's manifest allows us to declare additional activity names.

The process of installing a vulnerable app and activating it is shown in Figure 2.1. The stealthy installation via bookmarks (or recent tabs or some other object of synchronization) combined with name obfuscation makes it hard to tell that an app is malicious, even for experienced users.

Breaking 2FA With the malicious/vulnerable app and activation methods in place, attackers can start their attack from the hijacked browser by requesting remote installation for the rogue app. We implemented a MitB trojan for the Google Chrome browser that can do this. Once installed, our extension can use Google session cookies to start remote app installation and prepare app activation. The plugin basically consists of three phases:

1. **Hijack a Google session**. Our plugin waits for a Google authentication cookie to become available. This happens when the user logs into a Google component (e.g., Gmail, YouTube, Drive, etcetera). Optionally, it forwards the typed credentials or cookies over the network to the attacker.

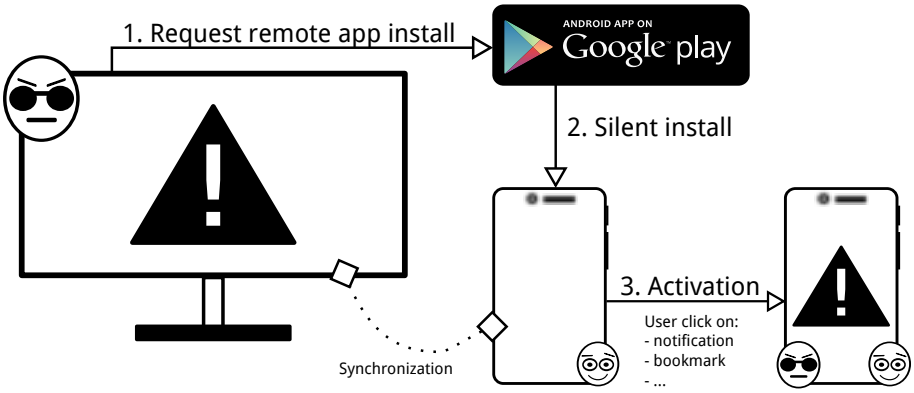

Figure 2.1. Malicious app installation process. Attackers (1) use their deployed MitB to request the installation of a vulnerable app from Google Play, and replace all the browser's bookmarks with malicious variants. Google then (2) pushes the app onto the mobile phone of the victim. Finally (3) the user is steered into activating the app. Activation is achieved by exploiting browser features to synchronize the malicious bookmarks to the phone, or by exploiting the user's curiosity.

2. **Remote install**. Using the hijacked Google session, the trojan sends a request to Google Play to retrieve a list of *Device ID*s of all Android devices linked to this particular Google account. Next, for each device, the plugin requests remote installation of the vulnerable app. Since app permissions are approved from within the PC-based browser only, the app will be silently installed, leaving only a *<app name> successfully installed* installation notification on the device.

3. **Activation**. To allow app activation, our extension rewrites all stored bookmarks and recent tabs so that they point to an attacker-controlled page while the original URL is provided as parameter: http://mal.icio.us/proxy.php?url=<original_url>. When opened using the mobile Chrome browser, this page performs a redirect to rogueapp://<original_url> which triggers activation of the rogue app. The app then immediately fires another intent that redirects the mobile browser to <orignal_url>, leaving practically no footprint.

Once activated, the malicious app can be used in conjunction with the PC-based trojan to successfully bypass mobile phone 2FA. Fraudulent financial transactions, for example, can be initiated by attackers once their PC-based trojan has captured banking credentials of their victims. To confirm such transaction, the mobile component intercepts the OTP sent via SMS, and forwards it to the attacker. This attack scenario is depicted in Figure 2.2.

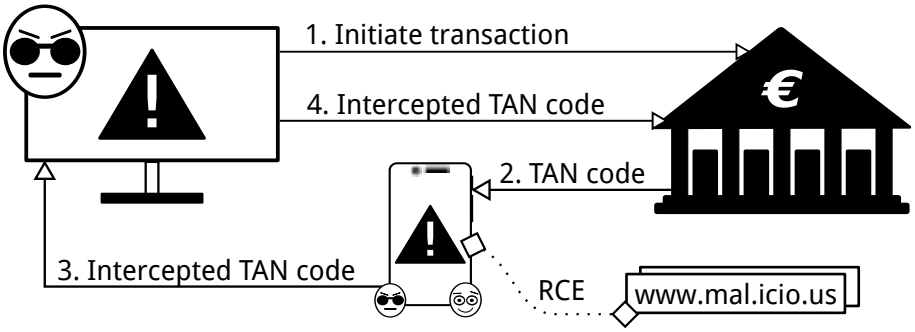

Figure 2.2. Completing fraudulent transactions while bypassing 2FA. After our app processes the TAN code, it loads a remote webpage into a `WebView` component that allows the attacker to perform Remote Code Execution (RCE). This way, attackers can hide their malicious activity from Google Play.

2.3.2 iOS

Similar to our Android attack, mobile phone 2FA on the iOS platform can be bypassed by publishing a rogue app to Apple's App Store and installing it from an infected PC via the iTunes remote-install feature. Wang et. al., already demonstrated how a vulnerable app could slip through Apple's strict review process and how such app can be used to access private APIs reserved for system apps to read SMS messages [204, 82]. Additionally, Bosman and Bos showed how a vulnerable app and *sigreturn oriented programming* allow to execute any set of system calls needed to pull of any attack [9].

As of iOS 8.3, released in April 2015, however, it is no longer possible to receive a so-called `kCTMessageReceivedNotification` to let an app act on incoming text messages without using a specific entitlement (similar to the Android `RECEIVE_SMS` permission). Since this functionality stems from a so-called private API, requesting such permission violates the App Store Review Guidelines and will result in an app rejection, effectively breaking this type of attack. The recent release of Mac Os X 10.10 Yosemite, however, opens up a new attack scenario.

As outlined in Section 2.2, Mac OS X Continuity features options to synchronize SMS and MMS text messages between multiple Apple devices. When enabled, SMS messages that are received on a linked iPhone, are forwarded and stored in plain-text in the `~/Library/Messages/chat.db` file on the Mac.

Breaking 2FA With Continuity enabled, attackers can break 2FA by instructing their MitB to monitor the `chat.db` database for changes and forward new

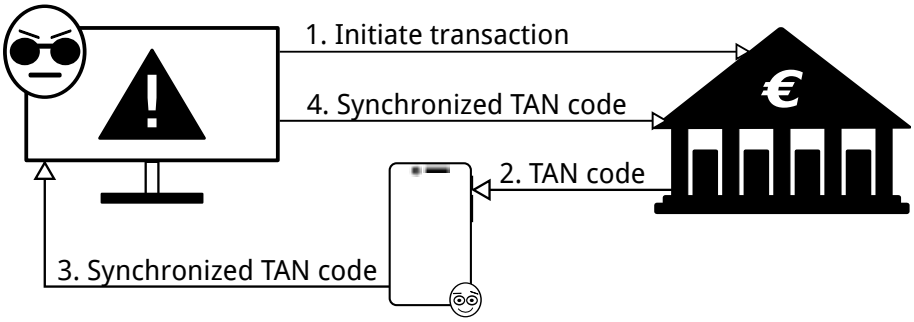

Figure 2.3. Breaking 2FA on Apple Continuity. If enabled, Mac OS X 10.10 automatically synchronizes SMS messages between different Apple devices, breaking the second factor.

messages to a remote server immediately after receipt. To show the practicality of this attack, we implemented a Firefox extension that uses the `FileUtils.jsm` API to read contents of synchronized SMS messages as soon as they are delivered to the iPhone. The Continuity attack is illustrated in Figure 2.3.

2.3.3 Dedicated 2FA Apps

Many online and offline applications are in the process of complementing their authentication mechanism with an optional 2FA step, often dubbed Two-Step Verification (2SV). Open source implementations are provided by Google (Google Authenticator) and Microsoft (Azure Authenticator) and can already be enabled for dozens of popular services, including Google, Microsoft Online, Amazon Web Services, Dropbox, Facebook, WordPress, Joomla, and KeePass.

Due to sandboxing techniques, our previously described attacks cannot access OTPs that are generated by 2SV authenticator apps. During the process of setting up an authenticator app, however, users are advised to provide the underlying system a backup phone number. The rationale behind this is that if, for some reason, users fail to access the authenticator app, they can fallback to requesting an OTP sent over SMS.

Assuming that many users provide a backup phone number that is used by the same smartphone that runs the authenticator app, an attacker can easily bypass these dedicated 2FA apps: (1) having access to stolen credentials harvested by the MitB, an attacker initiates the login procedure; (2) for logins via the Google Authenticator, for example, when prompted to enter a verification code, the attacker instructs the login page to *try another way to sign in*, followed by selecting the *Send a text message to your phone* option. From here, our previously described at-

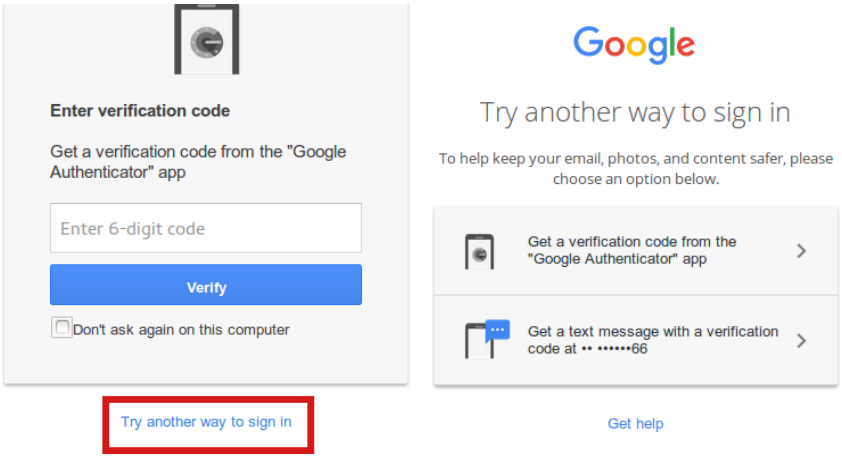

Figure 2.4. Bypassing dedicated 2FA apps. The screenshot on the left shows Google 2SV requesting a verification code from the Google Authenticator. Note the *Try another way to sign in* option near the bottom of the window. When clicked, the right-hand figure shows the fallback option to get a text message with an OTP sent over SMS. An attacker in control of the PC-browser is therefore able to dictate what 2FA technique is used.

tacks can be used to completely bypass the 2FA mechanism. Figure 2.4 illustrates how an attacker can fallback to SMS based OTPs when using Google Authenticator.

2.4 Discussion

In the previous sections, we showed how an attacker can bypass a variety of mobile phone 2FA mechanisms by exploiting synchronization vulnerabilities. We now study feasibility and practicalities of our attacks in more detail. Additionally, we discuss our efforts regarding responsible disclosure, as well as recommendations for involved parties.

2.4.1 Feasibility

Reviewing our Android attack described in Section 2.3.1, we conclude that exploiting synchronization vulnerabilities to bypass 2FA can be done in a reliable and stealthy way on Google's mobile operating system. Attackers can reduce their footprint to a bare minimum by breaking the attack down in different steps: (1) a **preparation phase** wherein attackers acquire access to infected PCs, possibly via a *Malware as a Service*-provider [13]; (2) an **app-installation phase**

wherein attackers push a vulnerable app to Google Play and instruct their victims to remotely install it. Depending on the target audience of the attacker, this can be done within a time window of only a couple of hours, after which the rogue app can again be removed from Google's servers; (3) an **app-activation phase** wherein attackers gracefully wait until victims activate the malicious app. Our app-hiding tricks make that attackers can safely wait days so that a large group of victims get to activate the rogue app; and (4) an **attack phase** wherein attackers perform an automated attack that requires access to OTPs sent over SMS. One typical example of such attack is transferring funds from saving accounts to an account that is controlled by the attackers.

Although more prerequisites must be met for our iOS attacks described in Section 2.3.2, they complement each other nicely: the *vulnerable app* approach does not work on iPhones running the latest iOS version, while our Continuity attack requires that victims *do* use more up to date versions of iOS and Mac OS X. The latter, however, also requires that (1) victims have enabled message synchronization (which setup process requires interaction with both Mac and iPhone), and (2) both devices are connected to the same wireless network. Although this does not necessarily make the attack less feasible, it may slightly reduce its scalability given that synchronization is off by default and increase the detection rate by attentive users (the content of received SMS messages will pop up on both devices).

Finally, although the remote-install 2FA synchronization vulnerability is also prevalent on the Windows Phone (WP) platform, Microsoft does not (yet) provide an API for reading received SMS messages programmatically. Additionally, to the best of our knowledge, WP does not provide SMS synchronization features like Apple's Continuity. It is because of this that we were unable to break mobile phone 2FA on WP.

2.4.2 Recommendations and Future Work

An important step towards preventing the presented sophisticated MitB-based attacks against mobile phone 2FA, is to raise awareness among the various stakeholders. *Mobile platform vendors* should be aware that the release of new synchronization features may introduce security risks for their end-users. As such, vendors should be extremely careful when enabling new features by default instead of making them optional. It is their obligation to inform *end-users* that enabling or using certain synchronization features might jeopardize security guarantees of mobile phone 2FA. Only then can the user make a considered decision to give up security in favor of usability.

Reviewing our proposed attacks, this means that Apple, for example, should warn users about potential security risks when they set up Continuity. Moreover, if the user decides to enable this feature, synchronizing only messages sent by *trusted* phone numbers — those that are found in the user's contact list — would eliminate our attack scenario, assuming that TAN codes are sent by an unknown sender or SMS gateway. Additionally, we recognize a major task for platform vendors to safeguard their remote-install features. In our view, users should always be forced to explicitly approve new app installations on their mobile device. This way, attackers can no longer silently push apps, but always require manual user-interaction. Ignorant users may still be phished into approving unknown install requests, of course, but such change would eradicate our completely automated attack scenario. We believe that the current app-activation security policy alone as deployed by vendors is too weak, given that additional synchronization vulnerabilities can be used to achieve activation.

Startled users who do not want to wait for a fix from their vendor, can protect themselves from exploitation by using a separate account for each device. This way, remote-install features have zero knowledge about which devices an app can be pushed to. Naturally, the downside of such approach is losing the ability to use synchronization features at all. Authenticator users, in addition, should update their settings so that their backup is a phone number that is attached to a dumb phone. These phones are remarkably harder to get infected.

Besides raising user-awareness, future work should focus on the detection of SMS stealing apps at runtime, given that existing mobile Anti-Virus apps are useless to this respect—they are confined to their own filesystem sandbox and thus cannot access directories of other apps, monitor the phone's file system, or analyze dynamic behavior of installed applications [25]. Instead, system modifications that can monitor the global smartphone state are required. To this, the redesigned permission model of Android Marshmallow in which apps are no longer automatically granted all of their specified permissions at install time, but rather prompt users to grant individual permissions at runtime, is promising. Unfortunately, this model will only be used by applications that are specifically compiled for Marshmallow and can thus still be bypassed.

As an ultimate resort, we recommend that financial institutions consider the removal of mobile 2FA from their business processes and switch to token based 2FA instead—such token must of course be able to show transaction details, so that Man-in-the-Middle attacks can be detected by the user during transaction processing. Naturally, such switch will cause large expenses; each institution will have to consider whether moving away from mobile 2FA is feasible by comparing

costs, gained security, and risk analysis results. Even so, given the attack scenarios we conclude that 2FA on smartphones is currently entirely compromised and no safer than single factor authentication.

2.4.3 Responsible Disclosure

To show the practicality of bypassing Google Bouncer, we uploaded a first version of our SMS stealing app to Google Play on July 8, 2015, where it has been publicly available for over two months. The app got removed on September 10, 2015, only a few hours *after* we had shared its name and a video demonstration of our attack with the head of Android Platform Security, while we already reported our attack scenario and recommendations to the Android security team months before the initial publication. Responses so far, unfortunately, indicate that Google believes that our proposed attack is not feasible in practice, despite all evidence to the contrary (including actual demos[2]).

We notified Apple about our findings on November 30, 2015, but we did not receive a technical response.

2.5 Background and Related Work

In this section, we provide a brief historical overview and related work discussion of the two fundamental components covered in this chapter: Man-in-the-Browser attacks and Two-Factor Authentication. Additionally, we discuss state-of-the-art attacks against mobile-phone 2FA which rely on cross-platform infection. We focus on online banking schemes in particular, as this always was, and still is, one of the services subject to a vast amount of criminal activity.

2.5.1 Man-in-the-Browser

At first, online financial services depended completely on single-factor authentication (e.g., by using a secret key). For attackers, keyloggers were enough to steal credentials of associated users. However, they also generated vast amount of useless data, forcing the attacker to parse a huge amount of log output in order to retrieve meaningful credentials. Parsing keylog data was considered a challenging and time consuming task for an attacker, as it is hard to automate. As an alternative, cyber criminals deployed phishing campaigns, followed quickly by form grabbing attacks. The latter proved to be an effective and robust mechanism to steal useful information.

[2] https://youtu.be/k1v_rQgS0d8

Well known banking trojans like Zeus and SpyEye were the first to implement form grabbing by hooking web browser APIs [170, 216]. The fundamental idea behind form grabbing is to intercept all form information before it is sent to the network via HTTP requests. Form grabbing can be implemented in different ways: (1) *sniffing* all outgoing requests using a PCAP-based library—something that has the disadvantage of only working for unencrypted data [205]; (2) *API hooking* the browser's dynamic library to steal all the requests and responses made by the user before they get encrypted [205]; and (3) using a *malicious plugin* to easily register callbacks within the browser for events like *page load* or *file download* in order to intercept any request or response.

Malicious plugins and API hooking techniques can be used to do more than just form grabbing. Using a plugin, an attacker can modify HTTP responses received by the browser or covertly perform illegitimate operations on behalf of the user. This is commonly known as a Man-in-the-Browser (MitB) attack [164].

Guhring has identified various ways of which a trojan can perform a MitB attack and discusses pros and cons of various countermeasures that could be taken [164]. Boutin studies how webinjects are used by a trojan in the browser and discusses the underground economy behind selling webinjects [134]. Buescher et al., analyzed different types of hooking methods used by financial trojans [12]. They propose an approach for detecting and classifying trojans by looking at the manipulations they perform on a browser. However, their approach is mainly based on detecting API hooks. As a consequence, MitB attacks that are implemented using plugins cannot be detected using this technique.

2.5.2 Two-Factor Authentication

Most account fraud and identity theft relate to accounts that use only single-factor authentication [149]. To defend against MitB attacks, financial services started using different types of multi-factor authentication mechanisms. The most elementary mechanism is that of a list of Transaction Authorization Numbers (TAN codes) as provided by the online service, from which the user can choose one to perform a secure transaction. A more convenient method that has been adopted by a majority of financial services is generating a new TAN code for each transaction and sending this via an out-of-band channel to the user. Naturally, SMS is a cheap and efficient candidate channel: almost everybody owns a mobile phone.

To defend against MitB attacks that hijack an ongoing transaction by modifying its details (receiver's bank account number or the amount of money transferred), financial services are starting to include transaction details along with

the TAN code in the out-of-band SMS message. Users can then verify the transaction by inspecting these details in the SMS and only confirm if these match their expectation.

On August 8, 2001, the Federal Financial Institutions Examination Council agencies (FFIEC) issued guidance entitled *Authentication in an Electronic Banking Environment* [149]. FFIEC encourages financial institutions to use mobile phone-based 2FA as described above to secure their user's transactions.

Aloul et al., show how an app on a trusted mobile device can be used for generating one-time passwords, or how a mobile device itself can be used as a medium for out-of-band communication to financial services [2]. This is what most current deployed 2FA implementations use today. Mulliner analyzes attacks that target SMS interception in general and shows how a smartphone trojan can steal OTPs received via SMS. He proposes to use a dedicated channel which cannot be controlled by normal applications for receiving the OTP [58]. This is based on the assumption that mobile trojans do not have root privileges. Schartner et al., describe an attack against SMS based OTPs in the scenario where a transaction is made from the mobile device itself [124]. Since the transaction involves a single device (smartphone), a malware in the device can sniff both credentials and OTPs received via SMS.

Konoth et al., describe how Google's 2FA implementation can be bypassed using a MitB attack on an untrusted device [46]. Dmitrienko et al., analyzed 2FA implementations of major online service providers such as Google, Twitter, Dropbox and Facebook [20]. Their work identifies various weaknesses in existing implementations that allow an attacker to bypass 2FA and also illustrates a general attack against 2FA. However, unlike ours, their attack relies on complex cross-platform infection.

2.5.3 Cross-platform infection

Cardtrap.A is the first discovered malware that features a cross-platform infection implementation. The trojan first infects a symbian smartphone. When the user inserts the memory card of the mobile phone into a Windows PC, it attempts to infect the PC [168]. In 2006, researchers found that it is possible for PC malware to infect a smartphone by exploiting Microsoft's ActiveSync synchronization software [147]. Furthermore, Wang et al., explain how a sophisticated adversary can spread malware to another device through a USB connection [84]. Finally, Dmitrienko et al., demonstrated via prototypes the feasibility of both PC-to-mobile and mobile-to-PC cross platform attacks [20].

2.6 Conclusion

With the ongoing integration of platforms—the result of a strong desire for enhanced usability—keeping our web accounts safe has become increasingly challenging. The synchronization features and cross-platform services that are integrating different platforms are clearly a violation of the design principle called *compartmentalize* (see Table 1.2). The basic idea behind the *compartmentalization* is to segment a system into multiple compartments or units that are protected independently so that a vulnerability in one unit will not jeopardize the security of the other units.

In this chapter, we showed how synchronization features and cross-platform services can be used to elevate a regular PC-based Man-in-the-Browser to an accompanying Man-in-the-Mobile threat which can be used to successfully bypass mobile phone 2FA. The root cause is that imprudent synchronization functionality has obliterated the security boundaries on which 2FA solutions depend.

Due to the large number of financial institutions that rely on mobile phone 2FA for secure transaction processing, we expect that cyber criminals extend their activities by implementing attacks similar to ours, putting those institutions and their customers at risk.

3 Resurrecting Phone-Based Two-Factor Authentication Using a Software-Based Solution

Secure transactions on the Internet often rely on two-factor authentication (2FA) using mobile phones. In most existing schemes, the separation between the factors is weak and a compromised phone may be enough to break 2FA. In this chapter, we identify the basic principles for securing any transaction using mobile-based 2FA. In particular, we argue that the *computing system* should not only provide *isolation* between the two factors, but also the *integrity* of the transaction, while involving the user in confirming the *authenticity* of the transaction. We show for the first time how these properties can be provided on commodity mobile phones, securing 2FA-protected transactions even when the operating system on the phone is fully compromised. We explore the challenges in the design and implementation of SecurePay, and evaluate the first formally-verified solution that utilizes the ARM TrustZone technology to provide the necessary integrity and authenticity guarantees for mobile-based 2FA. For our evaluation, we integrated SecurePay in ten existing apps, all of which required minimal changes and less than 30 minutes of work. Moreover, if code modifications are not an option, SecurePay can still be used as a secure drop-in replacement for existing (insecure) SMS-based 2FA solutions.

3.1 Introduction

Today's Two-Factor Authentication (2FA) schemes for secure online banking and payment services often use smartphones for the second factor during initial authentication or subsequent transaction verification. As a result, all current solutions are vulnerable to sophisticated attacks and offer only weak security guarantees (see also Table 3.3). Specifically, attackers may compromise the phone (including the kernel [156, 163, 217]) and break the second factor. This is true for mobile-only banking services, but also for solutions that use a separate device (typically, a PC) to initiate a transaction.

Starting with the former, users increasingly rely exclusively on mobile applications for using their bank services, purchasing products, or booking trips [133, 207]. Using web-based payment services through a smartphone brings convenience, as users can now access them anytime and anywhere—even when access to a personal computer is not possible. However, such convenience comes at a cost to security guarantees offered by 2FA. Specifically, 2FA works if and only if the two factors remain independent and isolated, because it requires a compromise of *both* factors to initiate fraudulent transactions—a difficult task when devices are decoupled. This is not the case, however, when a *single* device, such as a smartphone, serves both factors, since the attacker needs to compromise only the potentially vulnerable smartphone for breaking 2FA.

Worse, even PC-initiated transactions are not safe if the attacker obtains root privileges on the mobile device. In that case, attackers can replace or tamper with the mobile apps, intercept messages and display misleading information. Unfortunately, compromising smartphones is a realistic threat especially given a compromised PC, because even though the phone and PC are physically separate, the devices are often not independent [44].

Surprisingly, despite ample proof that today's phone-based 2FA is weak in practice [20, 162, 44] and despite a range of proposed solutions [126, 182, 76, 85, 219, 93], this is not at all a solved problem. In practice, the issues may be as basic as a lack of separation between the two factors. However, even if there is a strong separation, there are other, equally fundamental issues. For instance, even research solutions tend to focus on a limited threat model that excludes fully compromised phones where an attacker obtains root access, infects the kernel and/or the corresponding app. Given that 2FA is often used in high-value interactions (e.g., banking) and full system compromises are a common occurrence [99, 192, 80, 81], such a limited threat model is wholly insufficient.

Getting what appears to be a simple issue right is remarkably hard and we

will show that even the most state-of-the-art solutions are lacking. When studied closely, 2FA exhibits many subtle issues in both the bootstrap phase (generating and registering keys) and operational phase (performing transaction). Given the many failed attempts at secure solutions, verifying the correctness of a solution in all possible corner cases is difficult for a human analyst. Instead, we propose the use of automated proofs to *guarantee* the security properties of our solution.

In terms of basic principles, we say that a transaction's *authenticity* is ensured if we prevent an attacker from initiating a transaction on behalf of the user (or server) without being noticed. Meanwhile, a transaction's *integrity* is preserved if an attacker is not able to modify the content of the messages exchanged or displayed. Given these basic principles, we argue that even if existing solutions separate the two factors in 2FA, they tend to focus on authentication and ignore integrity, even though secure transaction requires both authenticity and integrity.

In this chapter, we present SecurePay, a novel, principled design to regain the strength of 2FA even in the presence of fully compromised devices, while retaining the convenience of mobile devices by securing a minimal amount of core functionality for handling the second factor in the secure world provided by the Trusted Execution Environment (TEE). All other code runs in the normal world. SecurePay applies both to mobile-only transactions *and* to transactions initiated on a personal computer with the mobile device serving as the second factor only. Unlike previous work [126, 145, 161, 182, 76, 85, 219], SecurePay builds on the solid foundation of a minimalistic protocol for guaranteeing authenticity *and* integrity for any transaction-based system, for which we additionally provide a formal security proof.

The protocol is deliberately minimalistic to adhere to Saltzer and Schroeder's principles of Economy of Mechanism, Least Common Mechanism, Least Authority, and Privilege Separation [103]. It is the first solution to include just the minimum of generic 2FA functionality in the TEE to cater to *all* banking, payment, and similar services—allowing each to secure their transactions with verifiable OTPs (one-time passcodes) in a 2FA solution.

In particular, SecurePay's secure world provides three essential functions. First, it is responsible for generating SecurePay public-private key pairs. The private key *never* leaves the secure world, while banking and similar services will use the public key to encrypt the verification OTPs for transactions. Second, it will decrypt the verification OTPs that it receives from banking and similar services (via the mobile app in the normal world as an intermediary). Since the SecurePay private key never leaves the TEE, it is the only entity capable of de-

crypting these messages. Finally, it is capable of displaying the encrypted messages to the user in a secure and unforgeable manner. In other words, when users see messages displayed by SecurePay's trusted code, they can be certain that it was generated by the trusted components and that it was not tampered with. SecurePay ensures this through a software-only solution based on a secret that is shared between the user and the TEE.

These three functions allow any payment or similar service to implement secure transactions. To back this up, we incorporated SecurePay in ten different apps with minimal effort. Moreover, on the client-side, SecurePay may also serve as a drop-in replacement for existing (unsafe) solutions such as SMS—without any code changes whatsoever. To the best of our knowledge, SecurePay is the first generic system that supports arbitrary transaction services without requiring additional hardware or significant changes on the client side, while ensuring the authenticity and integrity of the transactions initiated from a PC or smartphone, even in the case of fully compromised devices. We designed SecurePay as an effective and practical solution, utilizing TEE features available in all modern devices.

Contributions In summary, our contributions are the following:

1. We analyze current 2FA-based techniques for protecting Internet banking, show why they are weak, and identify the key functionality that we need to isolate.

2. We present SecurePay, a generic 2FA design capable of guaranteeing the integrity and authenticity of any transaction (financial or otherwise) initiated from a mobile app or PC even in the face of a complete system compromise—where current solutions target *only* authenticity for more limited threat models.

3. We implemented and evaluated SecurePay on an actual smartphone[1].

4. We provide a formal proof of SecurePay's security guarantees.

[1]While this sounds simple enough, TEEs in phones are normally not accessible to researchers. It was only possible thanks to a vendor's support.

3.2 Background

In this section, we discuss the necessary background information for both our threat model and the SecurePay design.

3.2.1 Mobile transactions and 2FA

Two-factor authentication (2FA) is a well-established mechanism for hardening authentication schemes. Typical designs for web-based transactions require users to demonstrate not only their knowledge of some secret credentials (such as a password), but also their possession of some artifact (such as mobile phone). Since mobile phones are easily the most popular choice for a large class of 2FA implementations, we limit ourselves to phones only. In virtually all such schemes, after a user submits the credentials to a server, the server sends an OTP to the user's phone, often in the form of a short sequence of digits. The assumption is that the ability to also submit the OTP proves possession of that phone. In principle, 2FA secures clients against the misuse of passwords that may have leaked [35, 98]. An adversary who steals a user's password and/or compromises the user's personal computer is still unable to impersonate the victim, as long as the service can transmit the second factor safely to the smartphone, that is assumed to be non-compromised.

Besides initial authentication, 2FA also commonly protects sensitive transactions, such as bank transfers. In particular, e-banking and e-commerce services enforce a 2FA procedure whenever the *already authenticated* user performs a particular action, such as transferring money from one bank account to another, or verifying an on-line purchase. For instance, the Bank of America's SafePass offers security based on 2FA. Without it, customers can transfer only small amounts of money, while for higher-value transfers, the bank sends a 6-digit OTP as an SMS text message to the user's phone. In this context, 2FA no longer functions as an enhanced user authentication mechanism, but rather as additional procedure to verify the *authenticity* of the transaction. There is a subtle but important difference between these two uses of 2FA. Specifically, for on-line transactions just verifying the *authenticity* of the transaction is not enough, and *integrity* must be preserved also, for instance to prevent strong and stealthy attackers from modifying the user-issued transaction without the user's knowledge.

Unfortunately, in the absence of strong guarantees of separation between the factors, and authenticity and integrity of the transaction, an attacker who compromises the mobile can also intercept and/or tamper with the OTP. Worse, all 2FA solutions for mobile devices today are vulnerable to strong attackers capable

of compromising the system completely by obtaining root access, infecting the kernel, or replacing the banking app with a malicious repackaged version. Such strong attackers may be able to initiate a transaction without the user's knowledge, breaking the *authenticity* of the transaction, or hijack the user's transaction by displaying misleading information on the display, breaking the *integrity* of the transaction.

3.2.2 Separating the factors

Fortunately, modern devices offer strong isolation primitives in the form of a TEE, such as ARM's TrustZone. As we shall see, if carefully used, they can restore the isolation between the factors in a 2FA scheme. A TEE offers a hardware-supported secure environment that protects both code and data from all other code—with respect to authenticity and integrity. Even the kernel of the operating system in the normal world is unable to view or tamper with anything in the secure world.

However, even with a TEE, facilitating a secure transaction using 2FA in the face of a strong attacker is deceptively difficult, and all existing solutions show weaknesses in important cases. For instance, in Section 3.6.6 we will see that solutions such as the TrustPay design are vulnerable to *man-in-the-mobile* attacks even though it uses a TEE, because isolation is only the first step; integrity and authenticity of the transaction are equally important. Similarly, we will also discuss the weaknesses of VButton [50] design that result from the lack of a Trusted UI indicator.

Moreover, the naive solution of simply allowing each and every bank, payment service, or transaction application to run custom code in the TEE is undesirable, as doing so violates the Principle of Least Authority as well as that of Privilege Separation [103], and increases both the number of bugs and the attack surface in the TEE (jeopardizing the security of the entire system). Instead, we should keep the amount of code running in the TEE to a minimum and offer the least amount of functionality possible to cater to all possible 2FA applications (adhering to the Principle of Least Common Mechanism).

Besides TEEs, it is also possible to separate the factors by an additional (external) hardware device. For instance, the only solutions with a similar threat model to ours, such as ZTIC [85], require such additional hardware in the form of a USB stick. These devices need constant updates to support new service providers. Moreover, ZTIC protects only PC-initiated transactions and previous work has shown that extending it to mobile-only scenarios is difficult [101].

3.2.3 Trusted Execution Environment (TEE)

A TEE is a secure execution environment that runs in parallel with the operating system of a smartphone or similar device. Several hardware vendors have introduced different hardware-assisted TEEs: Intel Identity Protection Technology, Intel Software Guard eXtension, ARM TrustZone etc. [72, 91]; however, in this chapter we focus on TrustZone, as mobile devices tend to use ARM processors. Most such devices support a TrustZone-based TEE [222], which they implement using a set of proprietary methods. TEEs manage *trusted* applications which provide security services to *untrusted* applications running on the commodity operating system of the mobile device. For this purpose, the GlobalPlatform (GP) consortium is developing a set of standards for TEEs [120, 22, 111]. It includes APIs for the creation of trusted applications [109], as well as for interacting with other trusted applications securely [110]. Each trusted application should be able to run independently and should be prevented from accessing additional resources of other (trusted) applications. Nowadays, TEE developers implement TEEs in compliance with the GP API specifications.

ARM TrustZone [118, 119, 86] provides a TEE by enabling the system to run in two execution domains in parallel: the *normal* and the *secure* world (Figure 3.6). The current state of the system is determined by the value of the *Non Secure (NS)* bit of the secure configuration register. The secure domain is privileged to access all hardware resources like CPU registers, memory and peripherals, while the normal world is constrained. There is an additional CPU mode called *monitor*, which serves as a gatekeeper between the two domains. This monitor mode is also privileged, regardless of the status of the NS bit. Likewise, the memory address spaces is divided into regions, which are marked as secure or non-secure using the TrustZone Address Space Controller. Finally, peripherals can be marked as accessible to the secure world by programming the TrustZone Protection controller.

To switch from the normal to the secure world, the system must initially switch to monitor mode by executing a *Secure Monitor Call (SMC)*. Essentially, this allows the monitor mode to verify whether switching from one world to the other should be permitted. If the request is determined valid, it modifies the NS bit accordingly and completes the world switch.

3.3 Threat model and assumptions

We assume a strong and stealthy attacker, who has obtained the user's credentials (e.g., via a password leak, or a compromised device), and fully compromised

the user's smartphone. In other words, all the code in the normal world, including the operating system kernel, should be considered malicious. The attacker seeks to perform malicious transactions on behalf of the victim. For example, the attacker may replace the banking app in the user's device with a malicious one. While strong, this is a realistic threat model on today's smartphones and probably *should* be the threat model for highly sensitive applications such as payment systems. Many exploits exist that allow attackers to run malicious apps on a user's phone in a stealthy manner [143, 44, 214] and escalate the privilege to root [192, 80, 81]. As we discuss in Section 3.8, previous work fails to protect against such strong attacker models.

We also assume that users have secured their accounts with 2FA, so that the attacker must bypass 2FA restrictions (stealthily) for every malicious transaction. As the attackers have compromised the mobile device, they have access not just to the user's credentials, but also to the second factor OTP which we assume the payment service to deliver *only* to the compromised device. As attackers may have compromised the phone entirely, denial-of-service attacks are not relevant. Hardware-level attacks such as bus snooping [108, 96] and cold boot [34, 57] are out of scope as they require physical access to the phone.

Bootstrap We assume that TEE provides a secure boot process to ensure the integrity of the executables running in the secure world. In fact all modern devices achieve the secure-boot by implementing a chain of trust [167, 115]. On a device-reset event, the boot code from ROM verifies and loads the secure bootloader. The secure bootloader initializes TEE and loads the non-secure bootloader – after verifying its integrity. Finally, the non-secure bootloader verifies and loads the normal world OS.

3.4 Design

In this section, we first describe the requirements for a secure and compatible mobile-based 2FA before we explain SecurePay's design.

3.4.1 Requirements for a secure and compatible design

Transaction-based systems use traditional phone-based 2FA solutions to guarantee the authenticity of a transaction and the 2FA works only as long as one of the factors is not completely compromised. The main weakness of such 2FA-based solutions under our threat model is that the two factors are not sufficiently isolated and as a result, these solutions cannot guarantee both the authenticity and the integrity of the transaction.

As an example, consider Alice, who uses e-banking (using either her mobile phone or PC) to transfer some money to Bob. When Alice is about to make the money transfer, in the ideal case her bank sends an additional short code (generally referred to as a One-Time Password or OTP) with the transaction summary to Alice's phone to verify the requested transaction. However, if the phone is compromised, the attacker can (i) silently initiate a transaction, read the OTP and send it to the bank to confirm a fraudulent transaction breaking the *authenticity* of the transaction, or (ii) display a falsified transaction summary to the user (that matches the user's expectation) and trick her to confirm a different, fraudulent transaction—breaking the *integrity* of the transaction.

Thus, we identify the following key requirements that must be satisfied for any design for secure 2FA:

1. Isolation: we must ensure the separation of the domains manipulating the two factors in 2FA.
2. Integrity: attackers should not be able to tamper with (or read and display in modified form) a transaction's OTP messages as sent by the payment service.
3. Authenticity: users must be looped in to enforce the authenticity of the transaction.
4. Secure bootstrapping: users must be able to securely register the device to the service they wish to engage in 2FA authentication.

Besides these strict requirements, we increase both security and usefulness of our solution by three additional constraints:

5. Least common mechanism: the TEE should support the minimum functionality needed to support most applications and no more [103].
6. Provable security: given the pivotal role of 2FA for many highly sensitive services, we demand a formal proof of our design's security guarantees.
7. Compatibility: to facilitate adoption, we demand that it should work with existing services.

3.4.2 SecurePay

We propose SecurePay, our design of a secure and compatible 2FA solution that satisfies all of the aforementioned requirements. To satisfy (1), SecurePay uses

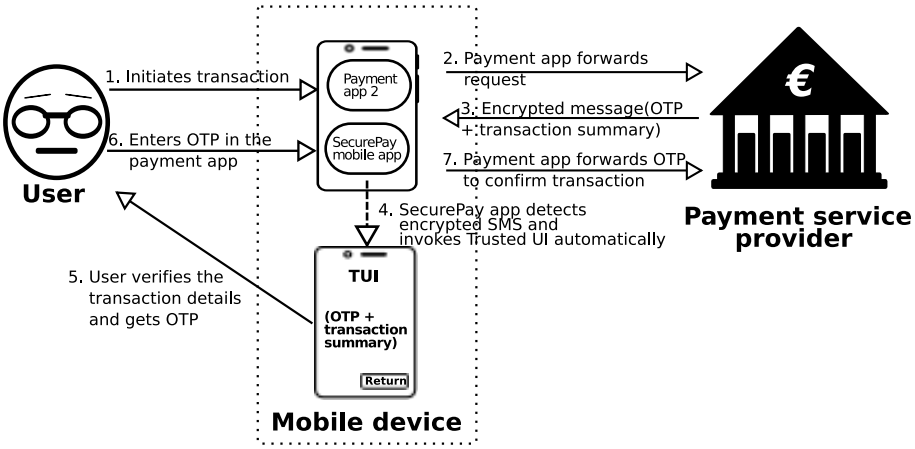

Figure 3.1. The work-flow of SecurePay-based transactions in transparent (drop-in replacement) mode.

TEE, such as ARM's TrustZone for creating a hardware-enforced isolated environment. To satisfy (2), SecurePay uses off-the-shelf public-key cryptography and the TEE for protecting the integrity of the transaction. To satisfy (3), SecurePay relies on a software-based secure display implementation, the output of which can only be produced by legitimate code which is recognizable as such by the user. To satisfy (4), SecurePay provides a tamper-resistant mechanism enforced by the TEE that allows users to securely register with a service provider that allows authentication through 2FA.

Furthermore, to satisfy our softer requirements, SecurePay provides a minimal TCB that runs in the TEE (trusted app) and we have formally verified that its protocol provides *authenticity* and *integrity* for transactions. To provide compatibility, SecurePay is capable of utilizing SMS as the communication channel between the user and service provider, and provides a normal-world component, the SecurePay app, to communicate the received encrypted SMS to SecurePay's trusted app (TA) in the TEE. Thus the service providers do not have to modify their mobile app to utilize SecurePay. Upon receiving the encrypted OTP and transaction summary, the user can invoke the SecurePay app to display the decrypted message on the secure screen (fully controlled by the TEE).

To further explore the compatibility aspects, SecurePay provides two modes of operation, one that is a fully transparent drop-in replacement for existing SMS-based schemes, and another which requires a small modification to the service providers' apps but offers full integration to simplify the user interaction.

For the drop-in replacement mode, Figure 3.1 shows the workflow as a se-

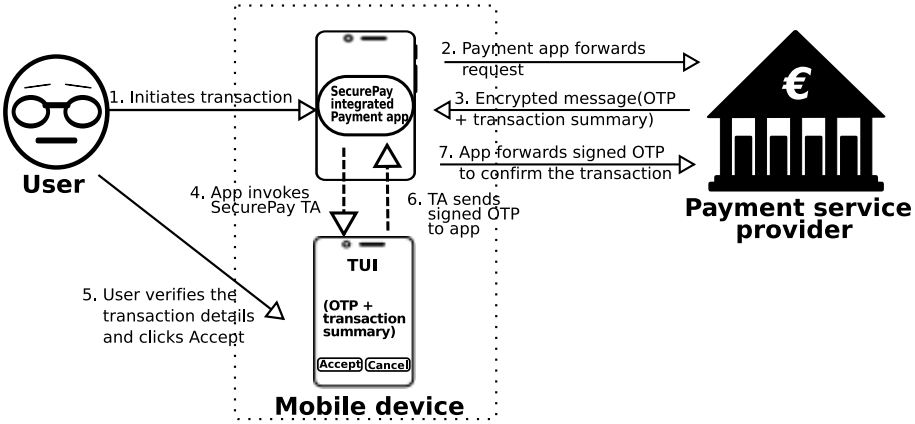

Figure 3.2. The work-flow for full integration of SecurePay (SecurePay integrated mode).

Table 3.1. Type of OTP used by different services

Service Provider	Type of OTP	Length of OTP
Google	digits	6
ING Bank	digits	6
Bitfinex	digits	5
Bank of America	digits	6
Citibank	digits	6
Deutsche Bank	digits	6
Dhanlaxmi Bank	digits	6
Axis Bank	digits	6
Alpha Bank	digits	6
TransferWise	digits	6

quence of steps. First, the user initiates a transaction from an app on the phone (or in the browser running on her PC as shown in Figure 3.4). The service provider receives this request and responds with an SMS containing an encrypted message (using the public key of the user's SecurePay) that includes the transaction summary and an OTP. The SecurePay's secure app receives this SMS and launches the trusted UI to display the transaction summary and the OTP to the user. Once the user verifies the authenticity of the transaction, she can switch to the service provider's app and enter the OTP, exactly like she would do with existing SMS-based OTPs. This OTP is then forwarded to the service provider and if it matches the one sent by the provider earlier, the transaction completes successfully. This version of SecurePay does not require any modification on the service provider's app on the phone, but it does require the user to memo-

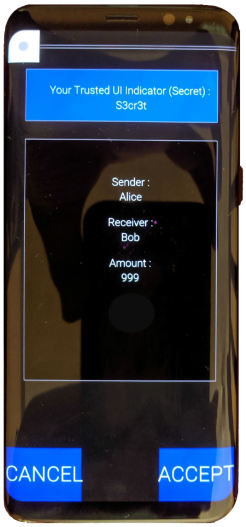

Figure 3.3. Full integration of SecurePay: the user simply presses accept or cancel on the trusted screen (SecurePay integrated mode).

rize (or note down) the OTP and enter it in the service provider's mobile app (or web interface) to confirm the transaction. Fortunately, studies in psychology and HCI [100] have shown that humans can remember without difficulty on average 7 items in their short memory. Table 3.1 shows that most services are using fewer digits (5 to 6) as OTP.

SecurePay can lift this requirement and hide the OTP entirely with a small modification of the service provider's app to fully integrate SecurePay. Figure 3.2 shows the necessary steps for confirming a transaction in this version of SecurePay. The main difference is that the service provider's app directly communicates with the SecurePay trusted app using the SecurePay library (discussed in Section 3.5). Similar to the previous version, the user initiates a transaction using the service provider's app. The service provider then sends an SMS with an encrypted summary and an OTP to the phone. Given that we do not want to increase SecurePay's code base, we let the provider's app (instead of SecurePay) receive this information (via SMS or Internet) and forward it to the SecurePay trusted app, which decrypts the message and shows the user the transaction summary (but no OTP). The only thing the user needs to do is accept or reject this transaction after looking at the summary (see Figure 3.3). The SecurePay trusted app then transparently signs the OTP and sends it to the service provider's app, which in turn forwards the signed OTP to the service provider. Upon receiving

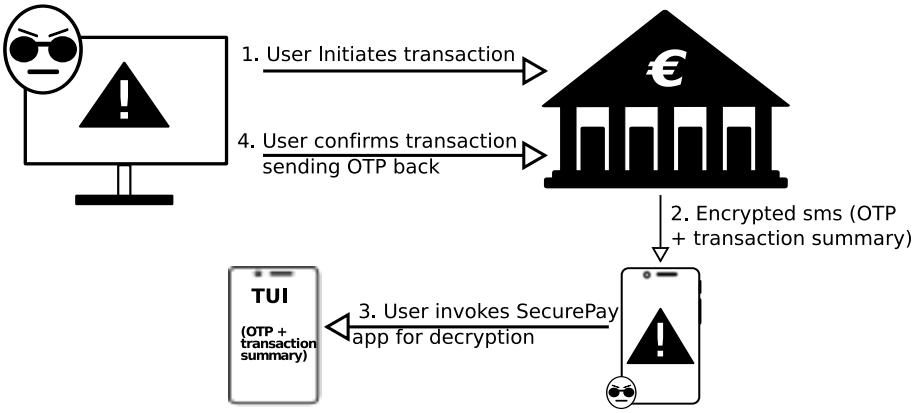

Figure 3.4. Shows how a SecurePay-enabled user issues transactions securely through a PC even if both PC and associated mobile device are infected by malicious code.

the signed OTP, the provider completes the transaction if the OTP matches the one sent earlier. This version of SecurePay provides more convenience for the user, but requires a small modification of the service provider's app (around 20 lines of code on average and little effort, as we will show in our evaluation).

Next, we discuss implementation details of SecurePay before analyzing its security guarantees and evaluating its performance.

3.5 Implementation

The architecture of SecurePay is depicted in Figure 3.6. In our prototype on Android 8, the mobile operating system runs in the normal world and manages all mobile apps, while Kinibi, Trustonic's TEE, runs in the secure world and manages all trusted apps. We tested a full implementation of SecurePay on a Samsung Galaxy S8 mobile device. In this section, we first discuss the implementation of SecurePay's components, then introduce its secure bootstrapping process, and finally, explain in detail how a user can initiate and complete a transaction securely even if all (normal-world) software on the user's devices (both PC and mobile) is fully compromised.

3.5.1 SecurePay components

SecurePay contains two main components: the SecurePay trusted app (TA) and the SecurePay Android library that enables any mobile app to communicate to the SecurePay TA. The SecurePay TA runs inside the secure world, beyond the reach of normal apps running in the normal world. The mobile app, running

in the normal world, can access the TA only through the APIs implemented by SecurePay Android library.

The SecurePay Android library is an Android Archive (AAR) file which can be linked to any Android app. It implements the API that allows apps in the normal world to access the functionalities provided by the SecurePay TA and comprises 1,546 LoC. Internally, the library uses the Global Platform (GP) TEE client API to implement these APIs.

For the drop-in replacement mode, we built the SecurePay mobile app, the normal world component of the SecurePay (Figure 3.1), using the SecurePay Android library. The end user can use the SecurePay mobile app to (i) generate the key pair, (ii) retrieve the public key, and (iii) decrypt and display the SMS on the secure screen. Once the secure screen is visible, the user is sure that the SecurePay TA is in control and the content of the screen can be trusted. How SecurePay implements its secure screen will be explained later in the section.

In SecurePay integrated mode, the SecurePay TA transparently sends the decrypted OTP back to the payment app (Figures 3.2 and 3.3), once the user verifies and accepts the transaction. This can be implemented using the SecurePay Android library, but in this case, the service provider has to modify its app to receive the encrypted transaction summary and OTP from the server, and to invoke the SecurePay TA to decrypt and display it on the secure screen. In practice, doing so took less than 30 minutes in all ten apps we tried (Section 5.6). Then, the user can verify the transaction details and press *accept* or *cancel* on the secure screen (Figures 3.3 and 3.2). If the user accepts the transaction, the SecurePay TA signs the OTP with the private key and sends it back to the service provider. If not, the SecurePay TA terminates the session. Note that the signing of the OTP by the SecurePay TA serves to prevent the attacker from trying to guess or bruteforcing the OTP reply to the server.

Note that in both modes, the OTP only leaves the secure world if the user *accepts* the transaction by either clicking on the button on the secure screen or entering the OTP in the normal world app. This is how SecurePay ensures the authenticity of the transaction, while the integrity of the transaction is ensured using public-private cryptography, a secure screen and secure bootstrapping (which we discuss in the next subsection).

The SecurePay trusted app The trusted core of SecurePay comprises 4,565 LoC running in the *Kinibi* secure world—a GP-compliant TEE which implements secure storage APIs and many common cryptographic APIs. As a consequence, SecurePay should work out of the box with any GP-compliant TEE.

Specifically, the TEE Internal API defined by the GlobalPlatform Association

and implemented by Kinibi supports most common cryptographic functions such as message digests, symmetric ciphers, message authentication codes (MAC), authenticated encryption, asymmetric operations (encryption/decryption or signing/verifying), key derivation, and random data generation. On top of these primitives, Kinibi implements a powerful *secure storage* layer which guarantees the confidentiality and integrity of sensitive general-purpose data, such as key material, as well as the atomicity of all operations on secure storage.

Using these APIs, the SecurePay TA supports three minimal functions. First, it can generate an asymmetric key pair of which the private key never leaves the TEE. Second, it can display the QR code of the public key on the secure screen (Figure 3.5). Third, it can decrypt messages encrypted with the public key when requested to do so from an (unprivileged) user app in the normal world and securely display it to the user on a secure display—with guaranteed authenticity and integrity, even if the attacker has administrator access to the phone. We now explain these functions in more detail.

`Generate_keys()`: The mobile OS automatically invokes this function at first boot (or full device reset). When normal-world code invokes the function, the SecurePay TA first checks whether a key pair already exists in its Kinibi-enforced GP-compliant secure storage and only if the pair does not exist, will it generate a new RSA key pair (using Kinibi's cryptographic API). Of this keypair, it returns only the public key to the normal world.

`Display_public_key()`: When invoked, the SecurePay TA checks whether a key pair already exists in its secure storage. If it exists, it extracts the public key component and displays its QR code on the secure screen (Figure 3.5).

`Display_summary()`: When invoked, the SecurePay TA decrypts a message using the private key stored in secure storage and displays it on the secure screen. It is used to handle the OTP from the transaction service (such as a bank). Recall that after the user initiated a transaction and the mobile app has sent the transaction details to, say, her bank, the banking service encrypts the transaction details and a freshly generated OTP using the user's public key and sends it back to the mobile device. Now, the SecurePay mobile app invokes `Display_summary()` to display the transaction summary and OTP on the secure screen. The API takes a boolean input parameter which decides whether the trusted app should return the *signed* OTP to the normal world if the user clicks on the *accept* button (Figure 3.3). To minimize confusion for the user, if the parameter is set to false, the SecurePay TA only displays a *return* button instead of *accept* and *cancel*.

Secure screen The main challenge in realizing a trusted user interface (TUI/secure screen/trusted screen) is ensuring that users can tell if they are actually

dealing with a trusted application, and not with a user interface injected and controlled by a malicious app [72]. Since a switch from normal-world to secure world code is done via a GP TEE client API which internally calls the `SMC` instruction, an attacker who has full control over the victim's device can easily bypass the switch and project an attacker-controlled user interface instead—tricking the user into believing that the active interface is now the trusted one.

Existing approaches are not suitable for SecurePay and typically require additional hardware. For instance, to realize a TUI, TrustOTP (TOTP) [76] shares a single screen between the normal and the secure world, but with two different frame buffers, of which one is accessible only from the secure world. Moreover, to make sure that the attacker cannot bypass world switching, TrustOTP uses a separate non-maskable interrupt, triggered by a special button on the phone for passing control to the secure world. Unfortunately, such solutions do not work for SecurePay. Since the GP compliant TEE is expected to run multiple trusted apps in the secure world, a single interrupt is insufficient, while adding separate hardware interrupts for each of them is impractical. In addition and equally important in practice, current COTS phones lack such special-purpose buttons to begin with.

As an alternative, one could also use a single piece of additional hardware, such as a specialized LED, as an indicator of whether the display is controlled by the normal or the secure world [89]. By configuring a GPIO port to be only accessible from the secure world and connecting to a special LED on the phone, the user knows that if the LED is on, the secure world is in control of the display. Again, as smartphones today do not have such a LED-based indicator, this is not a practical solution for our purposes either.

Hence, SecurePay implements a software-only solution whereby the trusted code authenticates its output to the display by means of an easily recognizable shared secret similar to previous work [53, 54]. The example secret is an image or secret text that is known only to the user and the SecurePay TA. Examples are shown in Figures 3.5 and 3.3, where the simple logo in the top left corner serves as a simple example of a secret image and "*S3cr3t*" as an example of the secret text (the "Trusted UI Indicator") in the figure. The secrets are explicitly loaded into the TEE at first boot (or full system reset), when the device was assumed to be in a pristine state and they are stored in the Kinibi's *secure storage* layer which guarantees both confidentiality and integrity of the data. Since only the trusted code knows the secrets, the user knows that if the device displays the secret image and/or the text on screen, the trusted code must be in control of the screen and the frame buffer, and no other code can access it. Even the Android

Figure 3.5. Registering with the bank by showing the QR code of public key on the trusted screen

OS literally does not have any access to the hardware or the frame buffer during the period that the secure screen (TUI) is active — meaning that malware cannot capture the data displayed on the screen or simulate touches, even if the phone is rooted.

3.5.2 SecurePay registration and bootstrap

Enabling SecurePay with an actual service involves communicating the public key to the service. In case the user owns several devices, all devices must register with the SecurePay-protected service. Registration takes place when the user installs the client part (i.e., the mobile application) on the device. For successful registration, the user must communicate the public key securely to the service—in terms of integrity, not necessarily confidentiality.

Since we assume that the mobile device may be already compromised at registration time, we must prevent attackers from registering their own public keys with a user's account, either by initiating a binding request themselves, or by replacing the public key with their own when the user's binding request is in transit. Like all secure transaction systems, SecurePay requires a secure bootstrap procedure to handle this. Various solutions are possible, ranging from custom hardware extensions to in-person registration at a physical office (whereby

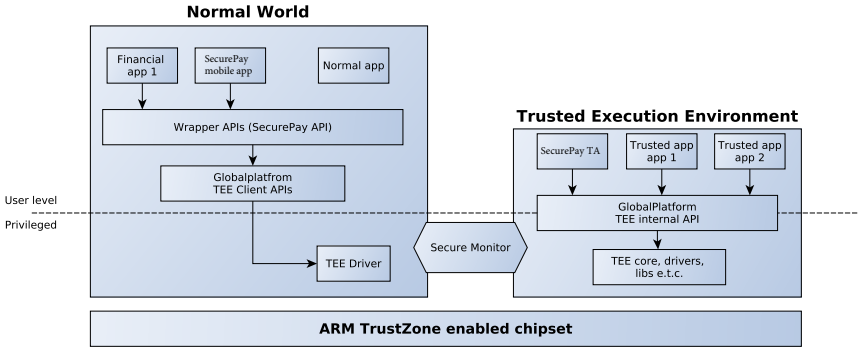

Figure 3.6. Multiple apps can use the same SecurePay TA.

a public key displayed on the phones secure display is manually bound to the account number).

Initial registration in our design simply assumes the presence of a secure terminal—for instance, at an ATM machine or a physical branch office. After installing the SecurePay mobile app, the user can invoke the SecurePay TA to display the QR code of the public key on the trusted screen as shown in the figure 3.5. Note that the user has to make sure the display is currently controlled by the trusted app by verifying the personalized image or secret text before sharing it to the bank. Finally, the bank simply scans the QR code to retrieve the public key safely from the user's device.

Note that the registration for SecurePay is comparable to or simpler than that of many other payment services. For example, to enable e-banking, many banks require physical presence at a branch office and/or hardware tokens. More importantly, the threat model for SecurePay is considerably stronger than that of existing systems—protecting the user against attacks launched from a fully compromised device, where the attacker controls even the device's operating system kernel.

3.6 Evaluation

3.6.1 Security of mobile transactions

An attacker can get privileged access on a victim's device in two ways: exploit a software/hardware vulnerability or trick the user to install a repackaged version of a mobile app. Many reports [130, 135] show that cyber criminals are often successful in tricking the users into installing a repackaged version of fi-

nancial apps using social engineering techniques. For the purposes of this chapter, we exploited a hardware vulnerability of the Nexus 5 phone [80] to get root access and replace the official financial app with a repackaged version. The latter hijacks transactions and sends money to attacker-controlled accounts. Once a transaction is completed, the malicious app displays a fake transaction summary to the user on the infected device instead of the details of what actually happened. Countering such attacks, that can take place when a bank transaction is carried out solely by using a compromised device, is extremely hard.

SecurePay can help the user (victim) stop any hijacked transaction from even happening in the first place. With SecurePay enabled, once the e-banking service receives a user's transaction, it encrypts the transaction details and a freshly generated OTP with the user's public key stored at the bank's server and sends back the result to the financial app. The repackaged version of the app receives the encrypted message, but, unless the private key has leaked from the trusted storage, decrypting it is not possible. The only way to decrypt the message is to relay it to SecurePay, which, being in the secure world, controls the private key. However, once SecurePay decrypts the message, it forwards the plain text to the secure display. The user is able to inspect the modified transaction and signal an abort message to the e-banking service by entering an invalid OTP. Note that the bank needs to generate a new OTP for every transaction request it receives in order to prevent the attacker from reusing an old OTP.

3.6.2 Security of non-mobile transactions

Many well-known banking trojan horses like Zeus [216], Dyre [125], and Dridex [121] use malicious plugins or API hooking techniques to modify the HTTP responses received by a browser or to silently perform illegal operations on behalf of the user [170]. This is commonly known as a Man-in-the-Browser (MitB) attack [164].

Let us assume that the user is making a financial transaction from an infected (non-mobile) host, such as a PC, but SecurePay powers the user's mobile device and financial application. For example, Alice initiates a transaction to transfer $100 to Bob using her browser running on her PC. An attacker, through a MitB attack, modifies the transaction to $1,000 to be transferred to an attacker-controlled account. Once the e-banking service receives the transaction, it encrypts the transaction summary and a freshly computed OTP, and sends the encrypted message back to Alice's smartphone, using a push notification or SMS. Since Alice runs the SecurePay mobile app, the message is handled by the SecurePay TA and the transaction summary is displayed, along with the OTP, on the trusted dis-

play. Alice reviews the transaction, and since it has been modified, aborts the transaction.

We stress that in this scenario, the attacker may well have full control over the user's mobile device and even her PC and web account credentials. However, in spite of this, thanks to SecurePay, it is still not possible for any hijacked transaction to actually take place.

3.6.3 Verification using TAMARIN

We formally verified SecurePay's *authenticity* and *integrity* security properties using TAMARIN v1.4.1 (see Appendix A). The TAMARIN [56] prover supports the automated, unbounded, symbolic analysis of security protocols. It features expressive languages for specifying protocols, adversary models, and properties, and efficient support for deduction and equational reasoning. A security protocol is specified through multi-set rewrite rules and facts. A rewrite rule takes a number of facts and rewrites them to other facts. Initially, the state contains no facts and only rewrite rules that do not require input facts can be applied. An exception is the generation of a fresh nonce, which is always possible.

With regards to SecurePay we have two such initiator rules. The first rule models the initiation of a binding request for a new device. In this case, the fresh nonce is the private key of the device to be added. Since this rule can always be applied, the proof is performed for infinitely many devices. The second rule is the initiation of a new transaction, which can also be performed infinitely many times. The nonce is the transaction data, which is the initial input from the user to perform a transaction. This also means that the models hold for infinitely many transactions.

There are two flavors of facts. Persistent facts remain part of the state after they are consumed by a rewrite rule, hence they can never be removed. Linear facts are removed from the state when they are consumed by a rewrite rule. The latter may be used to model multiple steps of a role. Each rewrite step produces a linear fact that is consumed by the successor step. We give an example:

```
rule step 1 :
  [ Fr ( ~nonce ) ]
 ==>
  [ Step1Completed ( ) , Out ( ~nonce ) ]

rule step 2 :
  [ Step1Completed ( ) , In ( response ) ]
 ==>
  [ Step2Completed ( ) ]
```

The first rewrite rule generates a fresh nonce and sends it into the network using the fact `Out (~nonce)`. The second step waits for a response from the network using the fact `In (response)`. The second rewrite rule is only ready to be performed after the first rewrite rule has been performed (modelled using the fact `Step1Completed`). Furthermore, persistent facts can be used to model the completion of a SecurePay binding request. Upon completion of the binding request the bank will create the fact `!PublicKeyForAccount (account, publicKey)`. This fact can be consumed (many times) to encrypt messages sent from the bank to the device holder. The usage of persistent facts in the model allows that the complete SecurePay protocol (binding requests and transactions) can be verified in a single model.

The adversary model employed by Tamarin is the well-known Dolev-Yao model [21]. The intruder learns every message which is sent over the network, can change its content and may generate new messages from the knowledge obtained so far. In terms of Tamarin this means that an intruder observes all `In`-facts and can produce an arbitrary number of `Out`-facts. All other facts are not observable by the intruder. Perfect encryption is assumed, meaning that the intruder does not learn anything from an encrypted message for which she does not own the key. Since in SecurePay the normal world is compromised, all messages to or from the normal world are compromised. We model this fact by exposing all messages that are traversing through the normal world to the network, i.e. the intruder.

We identify the following entities, which are involved in the protocol: i) the human performs binding requests and transactions, ii) the trusted app generates key pairs and displays messages on the secure screen, and iii) the bank processes binding requests and verifies transactions.

Our Tamarin specification of SecurePay separates the multi-set rewrite rules of the trusted zone, the bank and the human entity by prefixing all rule names. An overview of all rewrite rules and their intended relations are depicted in Figure 3.7 (some facts are omitted for readability). Rewrite rules that do not consume any fact can be executed arbitrarily many times, hence we consider infinitely many devices and accounts. Similarly, because the fact `HumanInitiatesTransaction` only consumes persistent facts, it can be executed arbitrarily many times if we witness a single `HumanOpensAccount`. It follows that we also consider arbitrarily many transactions. Note that the rewrite rules may not appear in the same order in every trace. Because the intruder can create an arbitrary number of messages from previously obtained knowledge, we consider a multitude of rule interleavings. We verified the security properties for all possible cases.

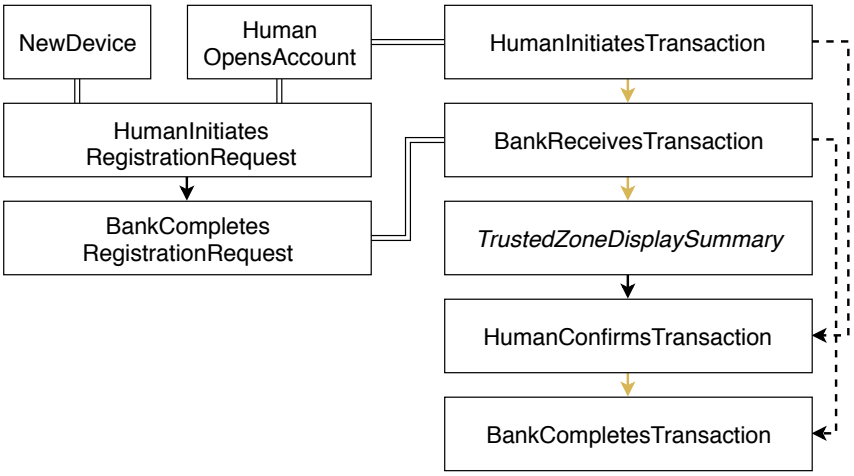

Figure 3.7. A visualization of the Tamarin specification of SecurePay. Boxes denote rewrite rules. Arrows between boxes denote facts, pointing from producer to consumer. Double lines denote persistent facts. Yellow arrows depict messages sent through the network or the normal world, i.e. interceptable by the intruder. Black lines are other linear facts, not observable by the intruder. Dashed arrows depict facts that denote a successor step within a role.

Unlike many security protocols, SecurePay has a control flow. Namely, for each transaction, the user decides to input the correct or a wrong OTP and the bank decides to accept or reject a transaction based on the received OTP. We model this behavior by restricting the application of rewrite rules using equality reasoning. An example for the above is `BankCompletesTransaction`. This rule can only be executed if the OTP contained in the network message matches the one that is generated by the fact `BankReceivesTransaction`.

Tamarin imposes the security properties on a global view by inspecting the trace. Additional to input and output facts, rewrite rules can specify action facts. Whenever a rule is applied, the action facts are appended to the initially empty trace. The trace is used to verify security properties, which are called lemmas in Tamarin. Lemmas are expressed using first-order logic formulas which must hold in all traces that are reachable from the initial configuration. To prove authentication and integrity, we specify lemma 3.1.

$$\begin{aligned} &\forall t. TransactionCompleted(t) \rightarrow \\ &(\exists . HonestTransactionInitiated(t)) \end{aligned} \tag{3.1}$$

Both `HonestTransactionInitiated` and `TransactionCompleted` are facts

and contain the transaction details including the account owner. Intuitively, the lemma means that a transaction must only complete if it was initiated by the account owner.

Tamarin automatically verifies security properties, expressed in first-order logic, for all possible execution traces, in a backward fashion. Hereby it employs constraint solving. It is either concluded that a given property holds for all execution traces that are possible from the initial protocol configuration, or a counter-example is produced. Since this verification problem is undecidable, inevitably such a backward run may not terminate. To achieve termination, the user may formulate and, with the help of Tamarin, prove so-called source lemmas that construct the possible sources for a fact. Source lemmas are solved using induction on the trace length.

To achieve termination of SecurePay's verification, one source lemma was needed. The lemma shows all possible sources for the encrypted message that is decrypted by the trusted zone. For all messages concerning a transaction, either the message comes from the bank or the intruder must know the OTP (that is contained in it). Using induction, Tamarin can prove this lemma automatically.

To prove that replay attacks are not possible, we specify lemma 3.2.

$$\begin{aligned} &\forall t, t', i, j. TransactionCompleted(t)@i \wedge \\ &\quad TransactionCompleted(t')@j \wedge i \neq j \\ &\qquad \rightarrow \\ &(\exists k, l. HonestTransactionInitiated(t)@k \wedge \\ &HonestTransactionInitiated(t')@l \wedge k \neq l) \end{aligned} \tag{3.2}$$

The variables i, j, k, l are time variables of the logical clock that is part of Tamarin. They uniquely identify the respective facts, that are proceeding the @ symbol. By definition, two facts cannot occur at the same time. Using inequality, we assume two arbitrary but distinct transactions and verify that the account owner(s) must have initiated two distinct transactions. We make no assumptions about t and t', therefore, we also verify replay attacks accross different accounts.

Using Tamarin, we verified *authentication* for SecurePay: every transaction must be initiated by the human that owns the account. Furthermore, we verified *integrity*: a transaction can only complete if both the human and the bank agree on the transaction details. The proof holds for an unlimited number of devices and transactions. Additionally, we verified that a replay attack is not possible.

Table 3.2. Open-source apps modified to utilize SecurePay

Android apps	LoC added	Time taken
Wordpress login	20	$<$ 30 minutes
InboxPager login	20	$<$ 30 minutes
Openshop.io login	20	$<$ 30 minutes
OpenRedmine login	20	$<$ 30 minutes
Quill login	20	$<$ 30 minutes
Yaaic login	20	$<$ 30 minutes
Seadriod login	20	$<$ 30 minutes
Slide login	20	$<$ 30 minutes
Kandriod login	20	$<$ 30 minutes
Photobook login	20	$<$ 30 minutes

3.6.4 Performance evaluation

We evaluate the performance overhead of our system by integrating the SecurePay normal-world library into a native library and then bundling it with a custom Android banking app. Since transactions are relatively infrequent events, throughput is not of paramount importance. Instead, we demand that each operation involved in registration and transaction verification takes a "reasonable" time—no longer than one or two seconds, say. For this, we measure the time taken by the core operations of SecurePay using the `System.nanoTime()` function, available in the Java library. We invoke each core operation 1,000 times and report the average value in seconds. We conduct this experiment on a Samsung Galaxy S8.

SecurePay takes 1.34 seconds to generate a 2,048 bit RSA key pair and to retrieve the public key from the secure world, and 1.91 seconds to generate an RSA key pair and the display QR code of the public key on a trusted screen. Note that key generation happens only once. Finally, it takes 1.29 seconds to decrypt and display a transaction summary of 100 bytes on the trusted screen, including SMS retrieval from the inbox, a world switch, decrypting and displaying the message. The performance of SecurePay is directly proportional to the performance of each component in the TEE (such as the cryptographic services, secure-storage services, trusted UI, etc.). We expect that for all practical applications, SecurePay can be enabled on commodity smartphones with little additional overhead.

3.6.5 Integration effort

Any mobile vendor implementing a TEE according to the GP specification can use the SecurePay TA [153]. Financial app developers can easily integrate SecurePay into their apps by linking the provided user-space library.

The apps that are already using SMS-based mobile 2FA do not require any change in their mobile app. However, the developers need to add 45 lines of code on the server side to encrypt the transaction summary and/or OTP using the user's public key. For the developers who want to use the second (fully integrated) model, we measured the effort that it requires to integrate SecurePay in a mobile application. For this purpose, we picked 10 open-source Android apps that have a *login activity* from Github [152, 228] and recorded the time required for a full integration of SecurePay. To ensure the diversity we picked the apps randomly from the following categories: shopping, business, social network, productivity, etc.

Table 3.2 shows that the app developer can link the SecurePay Android library and fully integrate SecurePay using only 20 LoC. 16 of these LoC are to configure the app to use the SecurePay TA, while 4 are to invoke the relevant SecurePay API (mentioned in the implementation section 3.5.1). The table also shows that it took one researcher (unfamiliar with the target app) less than 30 minutes to add SecurePay support to any app.

3.6.6 Comparison with similar efforts

TrustPAY [93] proposes a design to ensure security and to protect the privacy of a mobile payment (m-payment); however, under the threat model considered in the current chapter, it fails to protect both. In this part, we explain how TrustPAY works, possible attacks against it and how SecurePay successfully deals with such problems. We depict the underlying protocol of TrustPay in Figure 3.8 (taken directly and unmodified from TrustPAY) for protecting an m-payment transaction. In short, TrustPAY works as follows. Any normal world (NW) app can use TrustPAY to make a secure m-payment following a series of steps:

1. The TrustPAY component of the NW app requests a new/existing RSA key pair to the TrustPAY trusted app (TA), which runs inside the secure world (TA checks for an existing key pair; in case this is not found, it generates a new RSA key pair).

2. TA saves the newly generated private key (T_RSA_{PRI}) inside the secure world and shares the public key (T_RSA_{PUB}) at the NW app.

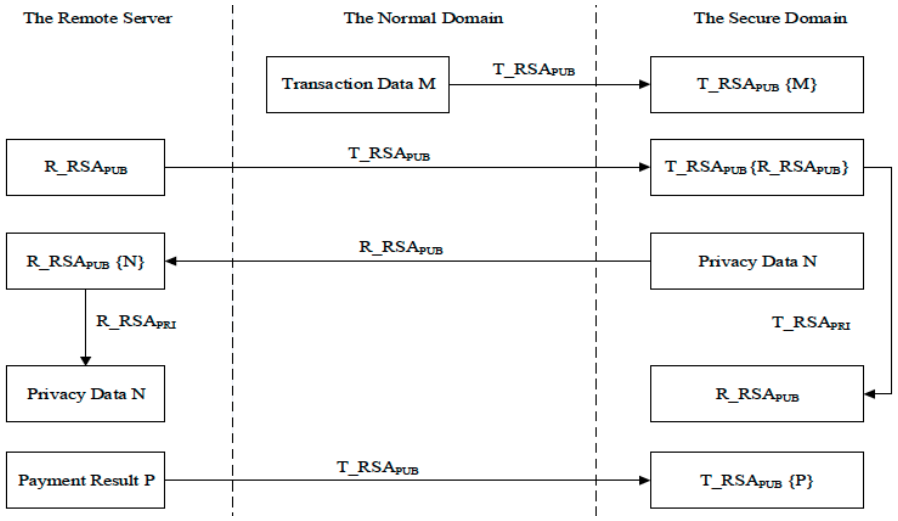

Figure 3.8. TrustPAY for m-payments.

3. Once the user places an order, the financial app in NW encrypts the order information using T_RSA_{PUB} key and sends the encrypted order information to the TA.

4. The TA decrypts the order information using its T_RSA_{PRI} key and displays it on the Trusted UI.

5. TrustPAY requests the bank for its public key by sharing T_RSA_{PUB}.

6. The bank encrypts its public key (R_RSA_{PUB}) with T_RSA_{PUB} and sends it to the TA.

7. The user can now verify the order details displayed on the Trusted UI and if she wants to pay the order, the user needs to enter the account number, password and verification code on the same Trusted UI.

8. Finally, the TA encrypts the user's private data with the public key of the bank, and sends the information back to the bank.

Let us analyze how an attacker can leak the user's private data (for example the account number or password) and hijack an in principle TrustPAY-protected transaction. We consider TrustPAY and the threat model assumed in this chapter, i.e., the attacker already has root access on the device. In this case, when TrustPAY requests the bank's public key, the attacker can send an attacker-controlled RSA public key to the TA (instead of the bank's public key). This means that,

Table 3.3. Comparison: SecurePay ensures the integrity and authenticity of a transaction, protects from *man-in-the-mobile* (MitM) and MitB attacks, supports both PC and mobile platforms, requires no change of the client for supporting a new service provider, and does not require additional hardware.

2FA Soln.	Authen-ticity	Integrity	MitM	MitB	Mobile	PC	Generic client	No hard-ware cost
TOTP [76]	✓	✗	✗	✗	✓	✓	✗	✓
TrustPAY [93]	✓	✗	✗	✗	✓	✗	✗	✓
VButton	✓	✓	✗	✗	✓	✗	✗	✓
ZTIC [85]	✓	✓	✗	✓	✗	✓	✗	✗
RSA SecurID [145]	✓	✗	✗	✗	✓	✓	✗	✗
Yubikey [219]	✓	✗	✗	✗	✓	✓	✗	✗
E.dentifier2 [126]	✓	✗	✗	✗	✗	✓	✗	✗
Authenticator [161, 182]	✓	✗	✗	✗	✓	✓	✗	✓
SecurePay	✓	✓	✓	✓	✓	✓	✓	✓

when the user confirms the payment, TA encrypts the user's private data and confirmation status of the order using the attacker-controlled public key. Now the attacker can decrypt the user's private data, modify the transaction/order confirmation status, and encrypt it with the bank's public key, before the normal app relays it to the remote server. The remote server then decrypts the request with the bank's private key and confirms the fraudulent m-payment. Moreover, it should be noted that the attacker can also display the user-expected transaction/order details on the Trusted UI by just encrypting it with TA's public key (T_RSA_{PUB}), which is accessible to NW. As we have already discussed above, SecurePay can protect the user from such *man-in-the-mobile* attacks. Moreover, SecurePay can also be used for PC-initiated financial transactions, which are not supported by TrustPAY.

In concurrent work, VButton [50] provides a system for enabling a mobile service provider to verify the authenticity of a user-driven operation originated from an untrusted client device. VButton requires integration at both client- and server-side to validate each user-driven operation. Moreover, it neither provides a Trusted UI indicator nor a secure way to register the public key to the service provider/attestation server. Without a Trusted UI indicator and a secure boot-

strap protocol, VButton is susceptible to timing-based and MitB attacks. In the timing-based attack, an attacker can show a different value to the user in the untrusted UI and switch to the Trusted UI with the correct value right before the user confirms the transaction. This can be mitigated with a check by the developer to ensure the user has had enough time in the Trusted UI, but this is not explored in the paper. In the MitB attack, the attacker could register their own public keys with a user's account, either by initiating a binding request themselves or by replacing the public key with their own when the user's binding request is in transit. Furthermore, the lack of a secure registration process also makes VButton vulnerable to relay attack as mentioned in the paper [50]. Compared to VButton, SecurePay's design is simple, complete, practical and formally proven to be secure. SecurePay can even be used as a secure drop-in replacement for existing (insecure) SMS-based 2FA without requiring any code change at the client-side.

Most of the hardware-based solutions [6, 145, 219] that are available in the market also fail to protect users from the aforementioned attacks, because these solutions can only be used to ensure the authenticity of the action and ignore its integrity properties. Moreover, these solutions have a variety of drawbacks. First, most of them are only for PCs. Second, hardware solutions are cumbersome in firmware upgrades. Third, they typically cost tens of dollars per token [146] and, fourth, they are inconvenient to carry around.

ZTIC [85] proposes a hardware-based solution for defending against *man-in-the-middle*. Compared to SecurePay, ZTIC ensures the integrity of PC-initiated banking transactions only. Furthermore, ZTIC requires a predetermined list of banks and additional modifications to the client, such as installing HTTP parsing profiles, user credentials, and X.509 certificates for supporting each new service. SecurePay does not require changes of the client code, requires no extra hardware, and protects both PC-initiated and mobile-initiated transactions.

We depict how SecurePay compares to related solutions using a series of key properties in Table 3.3. To the best of our knowledge, SecurePay is the only system that (1) requires no change at the client side to support a new financial service, (2) ensures the integrity *and* authenticity of transactions even for fully compromised clients, (3) does not require any additional hardware (beyond the TEE already present in almost all smartphones today), and finally, (4) protects from strong attack vectors such as MitB and MitM.

3.7 Discussion

We discuss other security aspects of SecurePay.

Availability: SecurePay does not guarantee availability at all. We assume that the mobile app runs on a compromised device. For instance, attackers can simply turn off the code that implements the SecurePay API. In that case, any forthcoming transactions will fail. Even so, no malicious transactions are possible.

Replay attack: We assume that (i) the remote service generates a unique OTP for each transaction request it receives, (ii) the remote service accepts the OTP only once, and (iii) the OTP expires after a short period to protect from replay attack. In Section 3.6.3 we formally verified that SecurePay is not vulnerable to replay attack.

SIM-jacking: SIM-jacking is an attack where the attacker convinces a victim's carrier to switch victim's phone number over to a SIM card that the attacker owns to bypass the current phone-based 2FA. SIM-jacking attacks have been widely used to hack into social media accounts, steal cryptocurrencies, and break into bank accounts [132, 150, 171]. SecurePay is not vulnerable to such attacks because SecurePay is not dependent on the SIM card.

Insecure TEEs: SecurePay assumes a secure implementation of TEE. If the TEE implementation has bugs, the attackers can exploit them to steal the private key from the TEE. Orthogonal to this work, formal verification can be used to ensure TEE is free of software bugs [40].

Microarchitectural attacks on TEEs: Defending against microarchitectural attacks on TEE is orthogonal to this research. Currently, SecurePay assumes a secure implementation of TEE. For instance, precautions such as constant-time software and microarchitectural resource flushing are known techniques again cache and speculation attacks. The attacks based on power/voltage glitching can be mitigated by following the standard practice of disallowing access to power/-voltage regulators from the normal world. In the case of Rowhammer [27, 28, 80, 81], to the best of our knowledge, there is no real-world attack that can compromise TEE. The only known attack triggers uncontrolled flips in TEE only when normal world's memory is allocated next to TEE. This is almost never the case in real devices (including our test phone). Nevertheless, even this weak attack can be mitigated by adding guard rows [11, 43].

3.8 Related work

2FA has been used to authenticate and protect financial transactions for many years. Multiple different ways to implement 2FA have been used: SMS-based, software-based, and hardware-based.

The most widely adopted approach nowadays is SMS-based Mobile 2FA[2], probably because it has practical advantage over some other methods in that it requires no additional hardware to store and handle the secondary authentication token. Services using SMS-based 2FA send an OTP and transaction summary in the form of an SMS to the user's mobile device, so that the user can verify the transaction details and confirm the transaction by entering the received OTP.

Recently, the National Institute of Standards and Technology at the US Department of Commerce stated that since SMS messages can be intercepted and redirected, implementers should consider an alternative authentication mechanism [162].

Besides SMS, over the last few years software-based 2FA implementations for authentication and transaction verification have become very popular. Software-based OTPs are usually generated by means of a form of software application. This could be an app running on the smartphone that generates OTPs from the *seed record* along with the device clock and an OTP generating algorithm. Google's *Authenticator* [161] and Microsoft's *Azure Authenticator* [182] are examples of such solutions and can be enabled for dozens of web services like Google, Microsoft Online, WordPress, Joomla, Amazon Web Services, Facebook and Dropbox. However, as we discussed earlier, software-based 2FA solutions cannot protect the user in the scenario where her mobile device is compromised.

As an alternative to such systems, hardware-based 2FA solutions rely on a separate piece of hardware, equipped with a small screen that is capable of generating OTP and displaying it. Today, several hardware-based solutions are available in the market, such as Yubikey [219], RSA SecurID [145] and E.dentifier2 [126]. Hardware-based 2FA is considered to be better than SMS-based and software-based solutions. However, it comes with an additional cost and causes inconvenience. Moreover, these solutions are used for authentication purposes – not for ensuring the integrity of a transaction.

Alexandra et al. [20] analyzed potential attacks against mobile 2FA and provided possible solutions against those attacks. Research [44] has shown how synchronization features and cross-platform services can be used to elevate a regular PC-based Man-in-the-Browser to an accompanying Man-in-the-Mobile

[2] https://twofactorauth.org/

threat and bypass SMS-based 2FA. Our work provides protection from *all* these attack vectors.

Marforio et. al. [53, 54] addressed the problem of how to securely set up a personalized security indicator in mobile banking to protect from phishing attacks. SecurePay uses a similar approach to implement secret-based trusted-UI. However, SecurePay assumes that the underlying mobile OS is fully compromised at the time of the transaction. Lenin et al. [75] propose a design for secure e-commerce transactions, but fail to protect from MitM attacks. Note that since the underlying Nizza architecture provides a software-level secure execution domain, one could port SecurePay to it to support any type of electronic commerce application (in addition to AppCore's cart-based ones). Norman et al. [26] demonstrate how to implement a secure graphical user interface to provide isolation between clients to prevent spying on each other, but does not protect users from MitB/MitM. The problem for SecurePay is different, since here we assume the entire (normal world) system may be compromised, including even the operating system kernel, and the objective is to guarantee that the user can distinguish outputs from the trusted app from those of regular programs.

A series of academic efforts involve the development of trusted applications for security solutions. Azab et al. [4] propose a system that provides real-time protection of the OS kernel using TrustZone (TZ). Santos et al. [74] use TrustZone to build a trusted-language runtime to protect the confidentiality and integrity of .NET mobile applications running in the normal world. Li et al. [49] propose a verifiable mobile advertisement framework to detect and prevent advertisement frauds using TrustZone. Marforio et al. [52] propose a location-based second-factor authentication mechanisms for payment at point-of-sale. Pirker at al. [66] propose a framework to protect the privacy of the user when a payment is made by mobile apps. In contrast, our work ensures authenticity and integrity of a transaction–rather than privacy. Truz-Droid [89] proposes a design to integrate the TEE with the mobile operating system to allow any app to leverage the TEE and builds a prototype on a Hikey board. Unlike SecurePay, Truz-Droid requires modification of the operating system, additional hardware support (a LED controlled by the TEE), and still, neither provides an easily adoptable drop-in solution for the banking apps nor supports PC-initiated transactions.

TrustOTP [76] has shown how to convert smartphones into secure OTP tokens. TrustOTP can be used to protect authenticity of any transaction but unlike our work, it does not also protect the integrity of a financial transaction. Moreover, TrustOTP has to be updated with the OTP generating algorithm used by each service provider, while SecurePay is decoupled from the OTP generat-

ing algorithm used by the service provider. Meanwhile, TrustPAY [93] proposes a payment system to ensure security and to protect privacy of mobile transactions. However, as we discussed in section 5.6 in more detail, certain flaws in their design allow the OS running in the normal world to leak the user's private information and to modify transaction details. Similarly, as discussed in the section 5.6, VButton proposes a system to enable a mobile service provider to verify the authenticity of user-driven operation; however, it lacks the Trusted UI indicator and the secure public-key registration process which are required to protect from timing-based, MitB and relay attacks.

Kellner et al. [38] claim that there is tremendous popularity among regular users for customizing their devices through jailbreaks. Jailbreaks remove vital security mechanisms, which are necessary to ensure a trusted environment that allows to protect sensitive data, such as login credentials and transaction numbers (TANs). The study shows that all but one banking app, available in the App Store, can be fully compromised by trivial means without reverse-engineering, manipulating the app, or other sophisticated attacks. Hence, the study pleads for more advanced defensive measures for protecting user data. The formally verified SecurePay design is a practical solution for this problem.

Finally, regarding the TEE, various vendors offer their own TEEs: OP-TEE [177], Trustonic [212], QSEE [195], SierraTEE [201], T6 [211], and MobiCore [151], and each of these TEEs comes with an SDK which helps developers to build trusted apps for the secure world.

3.9 Conclusions

In this chapter, we explored the risks associated with using a single mobile device for payments, even when enhanced authentication, such as 2FA is in place. We stressed that strong attackers can compromise the potentially vulnerable device and render 2FA completely useless. In parallel, we argued that sensitive applications, such as payment systems, gain limited security with using 2FA for several actions—not just for signing in, but also for issuing sensitive transactions. Following up, we defined a strong threat model, where stealthy attackers compromise smartphones for hijacking user-initiated payments that are otherwise protected with 2FA. We therefore identified the necessary requirements for facilitating a system, that leverages 2FA for securing the user's actions, even when compromised. The key property of our analysis is that 2FA should not be considered for protecting authenticity only, but also for the integrity of individual actions (i.e., the contents of a financial transaction). We, finally, presented SecurePay, a fully

working prototype, based on commodity technologies such as ARM's TrustZone, for realizing smartphones that allow users to perform Internet banking (and similar transaction activities) securely, even when their device is compromised.

4 Software Protection for a Hardware-level Design Flaw (Rowhammer Bug)

Vendors are packing an ever-increasing number of transistors in memory chips in response to consumer demands. Unfortunately, this design choice increases the possibility of memory errors in DRAM chips owing to the smaller difference in charge between a "0" bit and a "1" bit. As a result, it is possible to force memory errors in DDR3 memory by activating a row many times in quick succession, causing capacitors in neighboring victim rows to leak their charge before the memory controller has a chance to refresh them. This rapid activation of memory rows to flip bits in neighboring rows is known as the Rowhammer attack. As these bits flips/memory errors happen without any indication, the system fails to detect such memory errors. As a result, attackers are able to corrupt sensitive data structures (such as page tables, cryptographic keys, object pointers, or even instructions in a program), and circumvent all existing defenses.

This chapter introduces ZebRAM, a novel and comprehensive software-level protection against Rowhammer. ZebRAM isolates every DRAM row that contains data with guard rows that absorb any Rowhammer-induced bit flips; the only known method to protect against all forms of Rowhammer. Rather than leaving guard rows unused, ZebRAM improves performance by using the guard rows as efficient, integrity-checked and optionally compressed swap space. ZebRAM requires no hardware modifications and builds on virtualization extensions in commodity processors to transparently control data placement in DRAM. Our evaluation shows that ZebRAM provides strong security guarantees while utilizing all available memory.

4.1 Introduction

The Rowhammer vulnerability, a defect in DRAM chips that allows attackers to flip bits in memory at locations to which they should not have access, has evolved from a mere curiosity to a serious and very practical attack vector for compromising PCs [10], VMs in clouds [71, 87], and mobile devices [27, 80]. Rowhammer allows attackers to flip bits in DRAM rows simply by repeatedly reading neighboring rows in rapid succession. Existing software-based defenses have proven ineffective against advanced Rowhammer attacks [3, 11], while hardware defenses are impractical to deploy in the billions of devices already in operation [123]. This chaper introduces ZebRAM, a comprehensive software-based defense preventing all Rowhammer attacks by isolating every data row in memory with guard rows that absorb any bit flips that may occur.

Practical Rowhammer attacks Rowhammer attacks can target a variety of data structures, from page table entries [225, 80, 81, 87] to cryptographic keys [71], and from object pointers [10, 27, 79] to opcodes [31]. These target data structures may reside in the kernel [225, 80], other virtual machines [71], the same process address space [10, 27], and even on remote systems [79]. The attacks may originate in native code [225], JavaScript [10, 32], or from co-processors such as GPUs [27] and even DMA devices [79]. The objective of the attacker may be to escalate privileges [10, 80], weaken cryptographic keys [71], compromise remote systems [79], or simply lock down the processor in a denial-of-service attack [37].

Today's defenses are ineffective Existing hardware-based Rowhammer defenses fall into three categories: refresh rate boosting, target row refresh, and error correcting codes. Increasing the refresh rate of DRAM [39] makes it harder for attackers to leak sufficient charge from a row before the refresh occurs, but cannot prevent Rowhammer completely without unacceptable performance loss and power consumption increase. The target row refresh (TRR) defense, proposed in the LPDDR4 standard, uses hardware counters to monitor DRAM row accesses and refreshes specific DRAM rows suspected to be Rowhammer victims. However, TRR is not widely deployed; it is optional even in DDR4 [113]. Moreover, researchers still regularly observe bit flips in memory that is equipped with TRR [224]. As for error correcting codes (ECC), the first Rowhammer publication already argued that even ECC-protected DRAM is susceptible to Rowhammer attacks that flip multiple bits per memory word [39]. While this is complicating

attacks, they do not stop fully stop them as shown by the recent ECCploit attack [17]. Furthermore, ECC memory is unavailable on most consumer devices.

Software defenses do not suffer from the same deployment issues as hardware defenses. These solutions can be categorized into primitive weakening, detection, and isolation.

Primitive weakening makes some of the steps in Rowhammer attacks more difficult, for instance by making it harder to obtain physically contiguous uncached memory [225], or to create the cache eviction sets required to access DRAM in case the memory *is* cached. Research has already shown that these solutions do not fundamentally prevent Rowhammer [27].

Rowhammer detection uses heuristics to detect suspected attacks and refresh victim rows before they succumb to bit flips. For instance, ANVIL uses hardware performance counters to identify likely Rowhammer attacks [3]. Unfortunately, hardware performance counters are not available on all CPUs, and some Rowhammer attacks may not trigger unusual cache behavior or may originate from unmonitored devices [27].

A final, and potentially very powerful defense against Rowhammer is to *isolate* the memory of different security domains in memory with unused *guard rows* that absorb bit flips. For instance, CATT places a guard row between kernel and user memory to prevent Rowhammer attacks against the kernel from user space [11]. Unfortunately, CATT does not prevent Rowhammer attacks between user processes, let alone attacks *within* a process that aim to subvert cryptographic keys [71]. Moreover, the lines between security domains are often blurry, even in seemingly clear-cut cases such as the kernel and user-space, where the shared page cache provides ample opportunity to flip bits in sensitive memory areas and launch devastating attacks [31].

ZebRAM: isolate everything from everything Given the difficulty of correctly delineating security domains, the only *guaranteed* approach to prevent all forms of Rowhammer is to isolate *all* data rows with guard rows that absorb bit flips, rendering them harmless. The guard rows, however, break compatibility: buddy allocation schemes (and certain devices) require physically-contiguous memory regions. Furthermore, the drawback of this approach is obvious—sacrificing 50% of memory to guard rows is extremely costly. This chapter introduces ZebRAM, a novel, comprehensive and compatible software protection against Rowhammer attacks that isolates everything from everything else *without* sacrificing memory consumed by guard rows. To preserve compatibility, ZebRAM remaps physical memory using existing CPU virtualization extensions. To utilize guard rows,

ZebRAM implements an efficient, integrity-checked and optionally compressed swap space in memory.

As we show in Section 4.7, ZebRAM incurs an overhead of 5% on the SPEC CPU 2006 benchmarks. While ZebRAM remains expensive in the memory-intensive `redis` instance, our evaluation shows that ZebRAM's in-memory swap space significantly improves performance compared to our basic solution that leaves the guard rows unused, in some cases eliminating over half of the observed performance degradation. In practice, the recent Meltdown/Spectre vulnerabilities show that for a sufficiently serious threat, even expensive fixes are accepted [189]. First and foremost, however, this work investigates an extreme point in the design space of Rowhammer defenses: the first complete protection against all forms of Rowhammer, without sacrificing memory, at a cost that is a function of the workload.

Contributions In summary, our contributions are the following:

1. We describe ZebRAM, the first comprehensive software protection against all forms of Rowhammer.
2. We introduce a novel technique to utilize guard rows as fast, memory-based swap space, significantly improving performance compared to solutions that leave guard rows unused.
3. We implement ZebRAM and show that it achieves both practical performance and effective security in a variety of benchmark suites and workloads.
4. ZebRAM is open source to support future work.

4.2 Background

This section discusses background on DRAM organization, the Rowhammer bug, and existing defenses.

4.2.1 DRAM Organization

We now discuss how DRAM chips are organized internally, which is important knowledge for launching an effective Rowhammer attack. Figure 4.1 illustrates the DRAM organization.

The most basic unit of DRAM storage is a *cell* that can hold a single bit of information. Each DRAM cell consists of two components: a capacitor and a

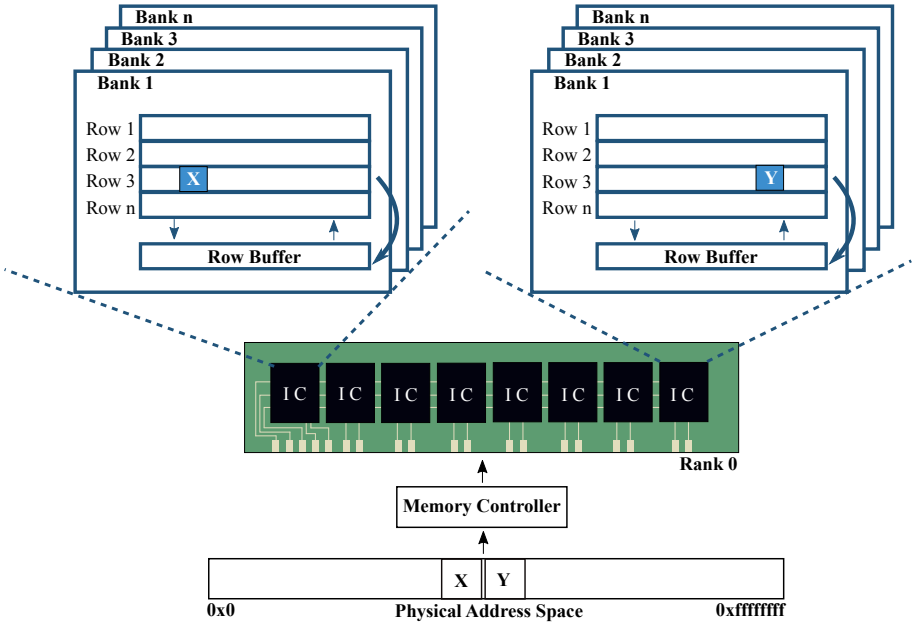

Figure 4.1. DRAM organization and example mapping of two consecutive addresses.

transistor. The capacitor stores a bit by retaining electrical charge. Because this charge leaks away over time, the memory controller periodically (typically every 64 ms) reads each cell and rewrites it, restoring the charge on the capacitor. This process is known as *refreshing*.

DRAM cells are grouped into *rows* that are typically 1024 cells (or *columns*) wide. Memory accesses happen at row granularity. When a row is accessed, the contents of that row are put in a special buffer, called the *row buffer*, and the row is said to be *activated*. After the access, the activated row is written back (i.e., recharged) with the contents of the row buffer.

Multiple rows are stacked together to form *banks*, with multiple banks on a DRAM *integrated circuit (IC)* and a separate row buffer per bank. In turn, DRAM ICs are grouped into *ranks*. DRAM ICs are accessed in parallel; for example, in a DIMM that has eight ICs of 8 bits wide each, all eight ICs are accessed in parallel to form a 64 bit *memory word*.

To address a memory word within a DRAM rank, the system memory controller uses three addresses for the bank, row and column, respectively. Note that the mapping between a physical memory address and the corresponding rank-index, bank-index and row-index on the hardware module is nonlinear. Consequently, two consecutive physical memory addresses can be mapped to memory

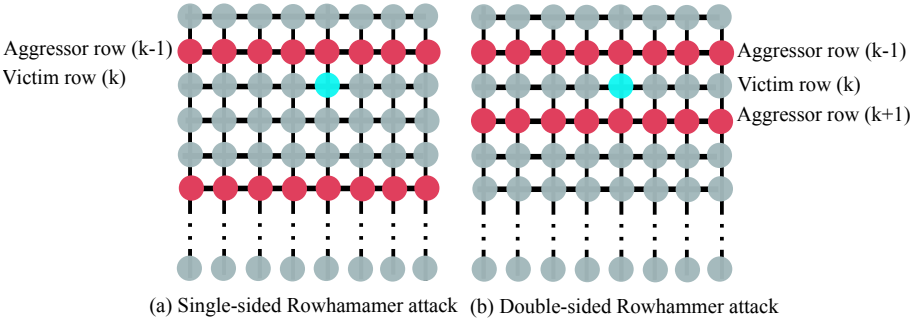

Figure 4.2. Flipping a bit in a neighboring DRAM row through single-sided (a) and double-sided (b) Rowhammer attacks.

cells that are located on different ranks, banks, or rows (see Figure 4.1). As explained next, knowledge of the address mapping is vital to effective Rowhammer.

4.2.2 The Rowhammer Bug

As DRAM chips become denser, the capacitor charge reduces, allowing for increased DRAM capacity and lower energy consumption. Unfortunately, this increases the possibility of memory errors owing to the smaller difference in charge between a "0" bit and a "1" bit.

Research shows that it is possible to force memory errors in DDR3 memory by activating a row many times in quick succession, causing capacitors in neighboring *victim* rows to leak their charge before the memory controller has a chance to refresh them [39]. This rapid activation of memory rows to flip bits in neighboring rows is known as the *Rowhammer attack*. Subsequent research has shown that bit flips induced by Rowhammer are highly reproducible and can be exploited in a multitude of ways, including privilege escalation attacks and attacks against co-hosted VMs in cloud environments [10, 32, 70, 71, 225, 80, 87].

The original Rowhammer attack [225] is now known as *single-sided* Rowhammer. As Figure 4.2 shows, it uses many rapid-fire memory accesses in one *aggressor* row $k-1$ to induce bit flips in a neighboring victim row k. A newer variant called *double-sided* Rowhammer hammers rows $k-1$ and $k+1$ on both sides of the victim row k, increasing the likelihood of a bit flip (see Figure 4.2). Recent research shows that bit flips can also be induced by hammering only one memory address [31] (*one-location* hammering). Regardless of the type of hammering, Rowhammer can only induce bit flips on directly neighboring DRAM rows.

In contrast to single-sided Rowhammer, the double-sided variant requires knowledge of the mapping of virtual and physical addresses to memory rows.

Since DRAM manufacturers do not publish this information, this necessitates reverse engineering the DRAM organization.

4.2.3 Rowhammer Defenses

Research has produced both hardware- and software-based Rowhammer defenses.

The original Rowhammer work [39, 59] explores multiple hardware-based defenses, some of which already deployed: the hardware manufacturer, Lenovo [174], deployed firmware updates that double refresh rates as a Rowhammer defense. Unfortunately, this has been proven insufficient to defend against Rowhammer [3]. Other suggested defense is error-correcting DRAM chips (ECC memory), which can detect and correct a 1-bit error per ECC word (64-bit data). Unfortunately, ECC memory cannot correct multi-bit errors [1, 17, 123] and is not readily available in consumer hardware. Probabilistic Adjacent Row Refresh (PARA) [39, 59] refreshes adjacent rows of an activated row with a small probability to mitigate Rowhammer. PARA requires hardware modification, complicating its deployment at scale. Similar to PARA, the new LPDDR4 standard [112] specifies two features which together defend against Rowhammer: *Target Row Refresh (TRR)* enables the memory controller to refresh rows adjacent to a certain row, and *Maximum Activation Count (MAC)* specifies a maximum row activation count before adjacent rows are refreshed. Despite these defenses, Gruss et al. [224] still report bit flips in TRR memory.

ANVIL [3], a software defense, uses Intel's performance monitoring unit (PMU) to detect physical addresses that cause many cache misses indicative of Rowhammer.[1] It then recharges suspected victim rows by accessing them. Unfortunately, the PMU does not accurately capture memory accesses through DMA, and not all CPUs feature PMUs. Moreover, the current implementation of ANVIL does not accurately take into account DRAM address mapping and has been reported to be ineffective because of it [78].

Another software-based defense, B-CATT [94], implements a bootloader extension to blacklist all the locations vulnerable to Rowhammer, thus wasting the memory. However, Gruss et al. [31] show that this approach is not practical as it may blacklist over 95% of memory locations; similar results were reported by Tatar et al. [78] showing DIMMs with 99+% vulnerable memory locations. In addition, in our experiments, we have observed different bit flip patterns over time for the same module, making B-CATT incomplete.

Yet another software-based defense called CATT [11] proposes an alterna-

[1] Rowhammer attacks repeatedly clear hammered rows from the CPU cache to ensure that they hammer DRAM memory, not the cache.

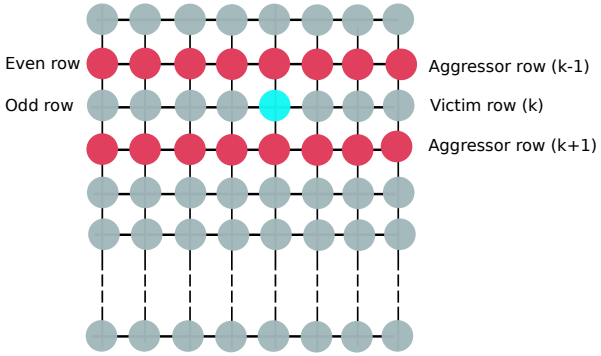

Figure 4.3. Hammering even-numbered rows can only induce bit flips in odd-numbered rows and vice versa.

tive memory allocator for the Linux kernel that isolates user and kernel space in physical memory, thus ensuring that user-space attackers cannot flip bits in kernel memory. However, CATT does not defend against attacks between user-space processes, and previous work [31] shows that CATT can be bypassed by flipping bits in the code of the sudo program.

4.3 Threat Model

The Rowhammer attacks found in prior research aim for privilege escalation [10, 32, 70, 71, 225, 80, 87], compromising co-hosted virtual machines [71, 87] or even attacks over the network [79]. Our approach, ZebRAM, addresses all these attacks through its principle of isolating memory rows from each other. Our prototype implementation of ZebRAM focuses only on virtual machines, stopping all of the aforementioned attacks launched from or at a victim virtual machine, assuming the hypervisor is trusted. We discuss possible alternative implementations (e.g., native) in Section 4.9.2.

4.4 Design

To build a comprehensive solution against Rowhammer attacks, we should consider Rowhammer's fault model: bit flips only happen in adjacent rows when a target row is hammered as shown in Figure 4.3. Given that any row can potentially be hammered by an attacker, all rows in the system can be abused. To protect against Rowhammer in software, we can follow two approaches: we either need to protect the entire memory against Rowhammer or we need to limit the rows that the attacker can access. Protecting the entire memory is not secure

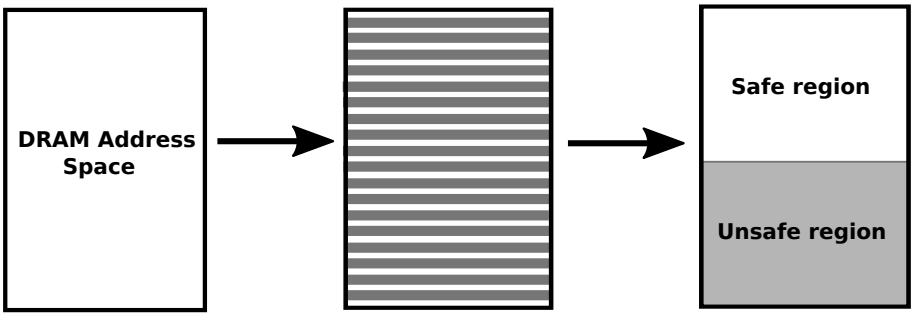

Figure 4.4. Splitting the memory into safe and unsafe regions using even and odd rows in a zebra pattern.

even in hardware [123, 80] and software attempts have so far been shown to be insecure [31]. Instead, we aim to design a system where an attacker can only hammer a subset of rows directly.

Basic ZebRAM In order to make sure that Rowhammer bit flips cannot target any data, we should enforce the invariant that all *adjacent rows are unused.* This can be done by making sure that either all odd or all even rows are unused by the system. Assuming odd rows are unused, all even rows will create a *safe region* in memory; it is not possible for an attacker to flip bits in this safe regions simply because all the odd rows are inaccessible to the attacker. The attacker can, however, flip bits in the odd rows by hammering the even rows in the safe region. Hence, we call the odd rows the *unsafe region* in memory. Given that the unsafe region is unused, the attacker cannot flip bits in the data used by the system. This simple design with its zebra pattern shown in Figure 4.4 already stops all Rowhammer attacks. It however has an obvious downside: it wastes half of the memory that makes up the unsafe region. We address this problem later when we explain our complete ZebRAM design.

A more subtle downside in this design is incompatibility with the Buddy page allocation scheme used in commodity operating systems such as Linux. Buddy allocation requires contiguous regions of physical memory in order to operate efficiently and forcing the system not to use odd rows does not satisfy this requirement. Ideally, our design should utilize the unsafe region while providing (the illusion of) a contiguous physical address space for efficient buddy allocation as shown on the right side of Figure 4.4. To address this downside, our design should provide a translation mechanism that creates a linear physical address space out of the safe region.

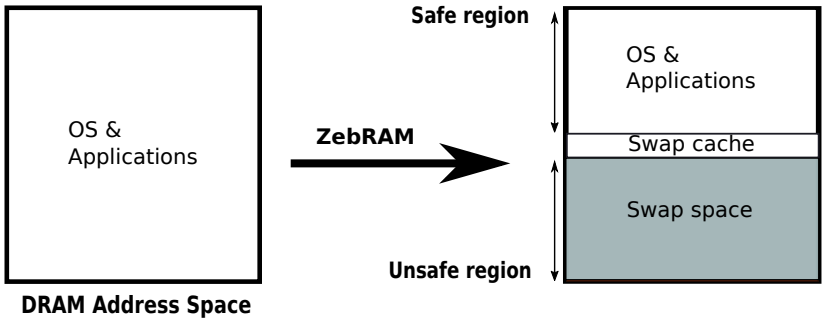

Figure 4.5. ZebRAM logically divides system memory into a safe region for normal use, a swap space made from the unsafe region, and a swap cache to protect the safe region from accesses made to the unsafe region.

ZebRAM If we can find a way to securely use the unsafe region, then we can gain back the memory wasted in the basic ZebRAM design. We need to enforce two invariants if we want to make use of the unsafe region for storing data. First, we need to make sure that we properly handle potential bit flips in the unsafe region. Second, we need to ensure that accessing the unsafe region does not trigger bit flips in the safe region. Our proposed design, ZebRAM, shown in Figure 4.5 satisfies all these requirements. To handle bit flips in the unsafe region, ZebRAM performs software integrity checks and error correction whenever data in the unsafe region is accessed. To protect the safe region from accesses to the unsafe region, ZebRAM uses a cache in front of the unsafe region. This cache is allocated from the safe region and ZebRAM is free to choose its size and replacement policy in a way that protects the safe region. Finally, to provide backward-compatibility with memory management in commodity systems, ZebRAM can employ translation mechanisms provided by hardware (e.g., virtualization extensions in commodity processors) to translate even rows into a contiguous physical address space for the guest.

To maintain good performance, ZebRAM ensures that accesses to the safe region proceed without interposition. As mentioned earlier, this can potentially cause bit flips in the unsafe region. Hence, all accesses to the unsafe region should be interposed for bit flip detection and correction. To this end, ZebRAM exposes the unsafe region as a swap device to the protected operating system. With this design, ZebRAM reuses existing page replacement policies of the operating system to decide which memory pages should be evicted to the swap (i.e., unsafe region). Given that most operating systems use some form of Least Recently Used (LRU), the working set of the system remains in the safe region, preserving performance. Once the system needs to access a page from the unsafe region,

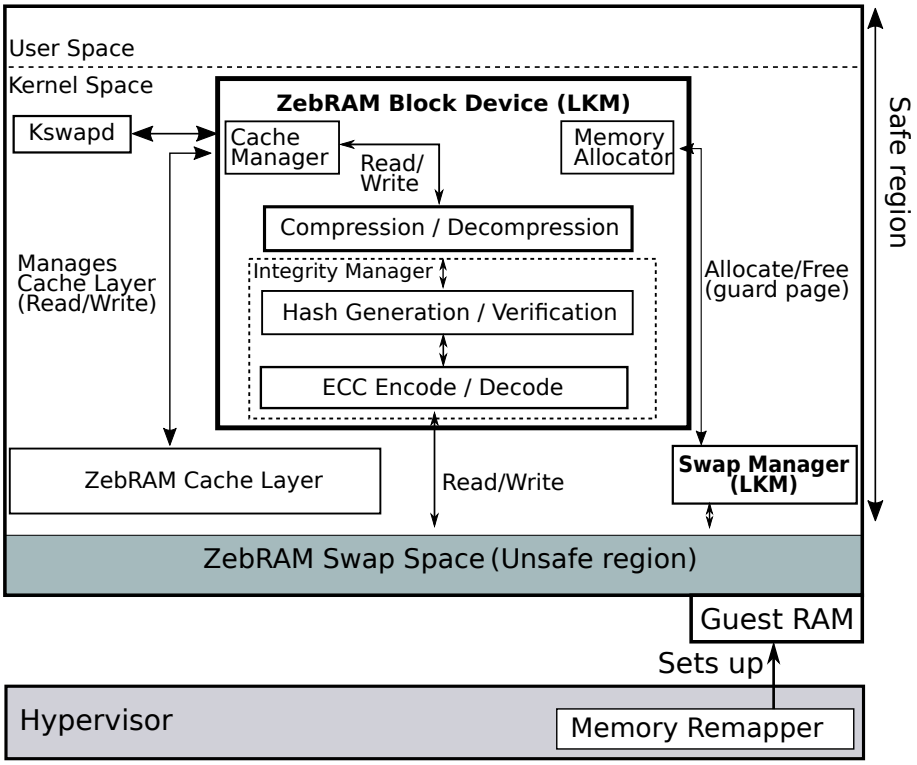

Figure 4.6. ZebRAM Components.

the operating system selects a page from the safe region (e.g., based on LRU) and creates necessary meta data for bit flip detection (and/or correction) using the contents of the page and writes it to the unsafe region. At this point, the system can bring the page to the safe region from the unsafe region. But before that, it uses the previously calculated meta data to perform bit flip detection and correction. Note that the swap cache (for protecting the safe region) is essentially part of the safe region and is treated as such by ZebRAM.

Next, we discuss our implementation of ZebRAM's design before analyzing its security guarantees and evaluating its performance.

4.5 Implementation

In this section, we describe a prototype implementation of ZebRAM on top of the Linux kernel. Our prototype protects virtual machines against Rowhammer attacks and consists of the following four components: the *Memory Remapper*, the *Integrity Manager*, the *Swap Manager*, and the *Cache Manager*, as shown in

Figure 4.6. Our prototype implements Memory Remapper in the hypervisor and the other three components in the guest OS. It is possible to implement all the components in the host to make ZebRAM guest-transparent. We discuss alternative implementations and their associated trade-offs in Section 4.9.2. We now discuss these components as implemented in our prototype.

4.5.1 ZebRAM Prototype Components

Memory Remapper implements the split of physical memory into a safe and unsafe region. One region contains all the even-numbered rows, while the other contains all the odd-numbered rows. Note that because hardware vendors do not publish the mapping of physical addresses to DRAM addresses, we need to reverse engineer this mapping following the methodology established in prior work [65, 78, 87].

Because Rowhammer attacks only affect directly neighboring rows, a Rowhammer attack in one region can only incur bit flips in the other region, as shown in Figure 4.3. In addition, ZebRAM supports the conservative option of increasing the number of guard rows to defend against Rowhammer attacks that target a victim row not directly adjacent to the aggressor row. However, our experience with a large number of vulnerable DRAM modules shows that with the correct translation of memory pages to DRAM locations, bit flips trigger exclusively in rows adjacent to a row that is hammered.

Integrity Manager protects the integrity of the unsafe region. Our software design allows for a flexible choice for error detection and correction. For error correction, we use a commonly-used Single-Error Correction and Double-Error Detection (SECDED) code. As shown in recent work [17], SECDED and other similar BCH codes can still be exploited on DIMMs with large number of bit flips. Our database of Rowhammer bit flips from 14 vulnerable DIMMs [78] shows that only 0.00015% of all memory words with bit flips can bypass our SECDED code (found in 2 of the 14 vulnerable DIMMs) and 0.13% of them can cause a detectable corruption (found in 7 of the 14 vulnerable DIMMs). To provide strong detection guarantees, while providing correction possibilities, ZebRAM provides the possibility to mix SECDED with collision resistant hash functions such as SHA-256 at the cost of extra performance overhead.

Swap Manager uses the unsafe region to implement an efficient swap disk in memory, protected by the Integrity Manager and accessible only by the OS. Using the unsafe region as a swap space has the advantage that the OS will only

access the slow, integrity-checked unsafe region when it runs out of fast safe memory. As with any swap disk, the OS uses efficient page replacement techniques to minimize access to it. To maximize utilization of the available memory, the Swap Manager also implements a compression engine that optionally compresses pages stored in the swap space.

Note that ZebRAM also supports configurations with a dedicated swap disk (such as a hard disk or SSD) in addition to the memory-based swap space. In this case, ZebRAM swap is prioritized above any other swap disks to maximize efficiency.

Cache Manager implements a fully associative cache that speeds up access to the swap space while simultaneously preventing Rowhammer attacks against safe rows by reducing the access frequency on memory rows in the unsafe region. The swap cache is faster than the swap disk because it is located in the safe region and does not require integrity checks or compression. Because attackers must clear the swap cache to be able to directly access rows in the unsafe region, the cache prevents attackers from efficiently hammering guard rows to induce bit flips in safe rows.

Because the cache layer sits in front of the swap space, pages swapped out by the OS are first stored in the cache, in uncompressed format. Only if the cache is full does the Cache Manager flush the *least-recently-added (LRA)* entry to the swap disk. The LRA strategy is important, because it ensures that attackers must clear the *entire* cache after every row access in the unsafe region.

4.5.2 Implementation Details

We implemented ZebRAM in C on an Intel Haswell machine running Ubuntu 16.04 with kernel v4.4 on top a Qemu-KVM v2.11 hypervisor. Next we provide further details on the implementation various components in the ZebRAM prototype.

Memory Remapper To efficiently partition memory into guard rows and safe rows, we use *Second Level Address Translation (SLAT)*, a hardware virtualization extension commonly available in commodity processors. To implement the Memory Remapper component, we patched Qemu-KVM's `mmap` function to expose the unsafe memory rows to the guest machine as a contiguous memory block starting at physical address `0x3ffe0000`. We use a translation library similar to that of Throwhammer [79] for assigning memory pages to odd and even rows in the Memory Remapper component.

Integrity Manager The Integrity Manager and Cache Manager are implemented as part of the ZebRAM block device, and comprise 369 and 192 LoC, respectively. The Integrity Manager uses `SHA-256` algorithm for error detection, implemented in mainline Linux, to hash swap pages, and keeps the hashes in a linear array stored in safe memory. Additionally, the Integrity Manager by default uses an ECC derived from the extended Hamming(63,57) code [97], expurgated to have a message size an integer multiple of bytes. The obtained ECC is a $[64, 56, 4]_2$ block code, providing single error correction and double error detection (SECDED) for each individual (64-bit) memory word—functionally on par with hardware SEC-DED implementations.

Swap Manager The Swap Manager is implemented as a Loadable Kernel Module (LKM) for the guest OS that maintains a stack containing the Page Frame Numbers (PFNs) of free pages in the swap space. It exposes the RAM-based swap disk as a readable and writable block device that we implemented by extending the `zram` compressed RAM block device commonly available in Linux distributions. We changed `zram`'s `zsmalloc` slab memory allocator to only use pages from the Swap Manager's stack of unsafe memory pages. To compress swap pages, we use the LZO algorithm also used by `zram` [230]. The Swap Manager LKM contains 456 LoC while our modifications to `zram` and `zsmalloc` comprise 437 LoC.

Cache Manager The Cache Manager implements the swap cache using a linear array to store cache entries and a radix tree that maps ZebRAM block device page indices to cache entries. By default, ZebRAM uses 2% of the safe region for the swap cache.

Guest Modifications The guest OS is unchanged except for a minor modification that uses Linux's boot memory allocator API (`alloc_bootmem_low_pages`) to reserve the unsafe memory block as swap space at boot time. Our changes to Qemu-KVM comprise 2.6K lines of code (LoC), while the changes to the guest OS comprise only 4 LoC. Furthermore, the Linux kernel may eagerly write dirty pages into the swap device based on its `swappiness` tunable. In ZebRAM, we use a swappiness of 10 instead of the default value of 60 to reduce the number of unnecessary writes to the unsafe region.

Table 4.1. ZebRAM's effectiveness defending against a ZebRAM-aware Rowhammer exploit.

Run no.	1 bit flip in 64 bits	2 bit flips in 64 bits	Total bit flips	ZebRAM detection performance Detected bit flips	Corrected bit flips
1	4,698	2	4,702	4,702	4,698
2	5,132	0	5,132	5,132	5,132
3	2,790	0	2,790	2,790	2,790
4	4,216	1	4,218	4,218	4,216
5	3,554	0	3,554	3,554	3,554

4.6 Security Evaluation

This section evaluates ZebRAM's effectiveness in defending against traditional Rowhammer exploits. Additionally, we show that ZebRAM successfully defends even against more advanced ZebRAM-aware Rowhammer exploits. We evaluated all attacks on a Haswell i7-4790 host machine with 16GB RAM running our ZebRAM-based Qemu-KVM hypervisor on Ubuntu 16.04 64-bit. The hypervisor runs a guest machine with 4GB RAM and Ubuntu 16.04 64-bit with kernel v4.4, containing all necessary ZebRAM patches and LKMs.

4.6.1 Traditional Rowhammer Exploits

Under ZebRAM's memory model, traditional Rowhammer exploits on system memory only hammer the safe region, and can therefore trigger bit flips only in the integrity-checked unsafe region by construction. We tested the most popular real-world Rowhammer exploit variants to confirm that ZebRAM correctly detects these integrity violations.

In particular, we ran the single-sided Rowhammer exploit published by Google's Project Zero,[2] as well as the one-location[3] and double-sided[4] exploits published by Gruss et al. on our testbed for a period of 24 hours. During this period the single-sided Rowhammer exploit induced two bit flips in the unsafe region, while the one-location and double-sided exploits failed to produce any bit flips. ZebRAM successfully detected and corrected all of the induced bit flips.

The double-sided Rowhammer exploit failed due to ZebRAM's changes in the DRAM geometry, alternating safe rows with unsafe rows. Conventional double-sided exploits attempt to exploit a victim row k by hammering the rows $k-1$ and $k+1$ below and above it, respectively. Under ZebRAM, this fails because the hammered rows are not really adjacent to the victim row, but remapped to

[2] https://github.com/google/rowhammer-test
[3] https://github.com/IAIK/flipfloyd
[4] https://github.com/IAIK/rowhammerjs/tree/master/native

be separated from it by unsafe rows. Unaware of ZebRAM, the exploit thinks otherwise based on the information gathered from the Linux' `pagemap`—due to the virtualization-based remapping layer—and essentially behaves like an unoptimized single-sided exploit. Fixing this requires a ZebRAM-aware exploit that hammers two consecutive rows in the safe region to induce a bit flip in the unsafe region. As described next, we developed such an exploit and tested ZebRAM's ability to thwart it.

4.6.2 ZebRAM-aware Exploits

To further demonstrate the effectiveness of ZebRAM, we developed a ZebRAM-aware double-sided Rowhammer exploit. This section explains how the exploit attempts to hammer both the safe and unsafe regions, showing that ZebRAM detects and corrects all the induced bit flips.

4.6.2.1 Attacking the Unsafe Region

To induce bit flips in the unsafe region (where the swap space is kept), we modified the double-sided Rowhammer exploit published by Gruss et al. [32] to hammer every pair of two consecutive DRAM rows in the safe region (assuming the attacker is armed with an ideal ZebRAM-aware memory layout oracle) and ran the exploit five times, each time for 6 hours. As Table 4.1 shows, the first exploit run induced a total of 4,702 bit flips in the swap space, with 4,698 occurrences of a single bit flip in a 64-bit data word and 2 occurrences of a double bit flip in a 64-bit word. ZebRAM successfully corrected all 4,698 single bit flips and detected the double bit flips. As shown in Table 4.1, the other exploit runs produced similar results, with no bit flips going undetected. Note that ZebRAM can also detect more than two errors per 64-bit word due to its combined use of ECC and hashing, although our experiments produced no such cases.

4.6.2.2 Attacking the Safe Region

In addition to hammering safe rows, attackers may also attempt to hammer unsafe rows to induce bit flips in the safe region. To achieve this, an attacker must trigger rapid writes or reads of pages in the swap space. We modified the double-sided Rowhammer exploit to attempt this by opening the swap space with the *open* system call with the *O_DIRECT* flag, followed by repeated *preadv* system calls to directly read from the ZebRAM swap disk (bypassing the Linux page cache).

Because the swap disk only supports page-granular reads, the exploit must

read an entire page on each access. Reading a ZebRAM swap page results in at least two memory copies; first to the kernel block I/O buffer, and next to user space. The exploit evicts the ZebRAM swap cache before each swap disk read to ensure that it accesses rows in the swap disk rather than in the cache (which is in the safe region). After each page read, we use a `clflush` instruction to evict the cacheline we use for hammering purposes. Note that this makes the exploit's job easier than it would be in a real attack scenario, where the exploit cannot use `clflush` because the attacker does not own the swap memory. A real attack would require walking an entire cache eviction buffer after each read from the swap disk.

We ran the exploit for 24 hours, during which time the exploit failed to trigger any bit flips. This demonstrates that the slow access frequency of the swap space—due to its page granularity, integrity checking, and the swap cache layer—successfully prevents Rowhammer attacks against the safe region.

To further verify the reliability of our approach, we re-tested our exploit with the swap disk's cache layer, compression engine, and integrity checking modules disabled, thus providing overly optimistic access speeds (and security guarantees) to the swap space for the Rowhammer exploit. Even in this scenario, the page-granular read enforcement of the swap device alone proved sufficient to prevent any bit flips. Our time measurements using `rdtsc` show that even in this optimistic scenario, memory dereferences in the swap space take 2,435 CPU cycles, as opposed to 200 CPU cycles in the safe region, removing any possibility of a successful Rowhammer attack against the safe region.

4.7 Performance Evaluation

This section measures ZebRAM's performance in different configurations compared to an unprotected system under varying workloads. We test several different kinds of applications, commonly considered for evaluation by existing systems security defenses. First, we test ZebRAM on the SPEC CPU2006 benchmark suite to measure its performance for CPU-intensive applications. We also benchmark ZebRAM the popular `nginx` and Apache web servers, as well as the `redis` in-memory key-value store. Additionally, we present microbenchmark results to better understand the contributing factors to ZebRAM's overhead.

Testbed Similar to our security evaluation, we conduct our performance evaluation on a Haswell i7-4790 machine with 16GB RAM running Ubuntu 16.04 64-bit with our modified Qemu-KVM hypervisor. We run the ZebRAM modules and

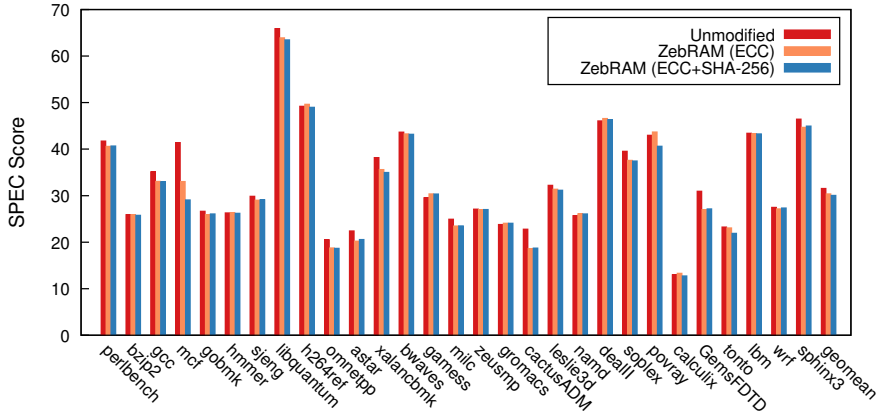

Figure 4.7. SPEC CPU 2006 performance results.

the benchmarked applications in an Ubuntu 16.04 guest VM with kernel v4.4 and 4GB of memory using a split of 2GB for the safe region and 2GB for the unsafe region. To establish a baseline, we use the same guest VM with an unmodified kernel and 4GB of memory. In the baseline measurements, the guest VM has direct access to all its memory, while in the ZebRAM performance measurements half of the memory is dedicated to the ZebRAM swap space. In all reported memory usage figures we include memory used by the Integrity Manager and Cache Manager components of ZebRAM. For our tests of server applications, we use a separate Skylake i7-6700K machine as the client. This machine has 16GB RAM and is linked to the ZebRAM machine via a 100Gbit/s link. We repeat all our experiments multiple times and observe marginal deviations across runs.

SPEC 2006 We compare performance scores of the SPEC 2006 benchmark suite in three different setups: (i) unmodified, (ii) ZebRAM configured to use only ECC, and (iii) ZebRAM configured to use ECC and SHA-256. The ZebRAM (ECC) and ZebRAM (ECC and SHA-256) show a performance overhead over the unmodified baseline of 4% and 5%, respectively (see Figure 4.7). The reason behind this performance overhead is that as the ZebRAM splits the memory in a zebra pattern, the OS can no longer benefit from huge pages. Also, note that certain benchmarks, such as `mcf`, exhibits more than 5% overhead because they use ZebRAM's swap memory as their working set do not fit in the safe region.

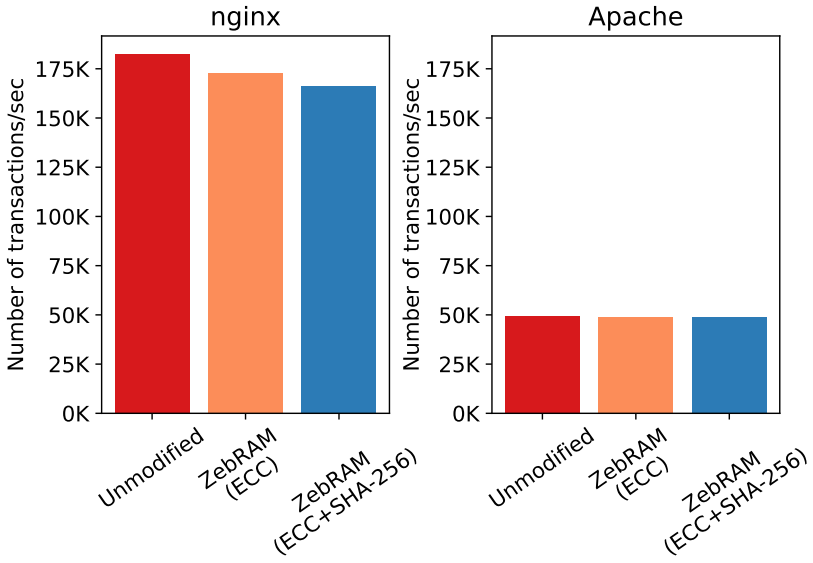

Figure 4.8. Nginx and Apache throughput at saturation.

Web servers We evaluate two popular web servers: nginx (1.10.3) and Apache (2.4.18). We configure the virtual machine to use 4 VCPUs. To generate load to the web servers we use the wrk2 [233] benchmarking tool, retrieving a default static HTML page of 240 characters. We set up nginx to use 4 workers, while we set up Apache with the prefork module, spawning a new worker process for every new connection. We also increase the maximum number of clients allowed by Apache from 150 to 500. We configured the wrk2 tool to use 32 parallel keep-alive connections across 8 threads. When measuring the throughput we check that CPU is saturated in the server VM. We discard the results of 3 warmup rounds, repeat a one-minute run 11 times, and report the median across runs. Figure 4.8 shows the throughput of ZebRAM under two different configurations: (i) ZebRAM configured to use only ECC, and (ii) ZebRAM configured to use ECC and SHA-256. Besides throughput, we want to measure ZebRAM's latency impact. We use wrk2 to throttle the load on the server (using the rate parameter) and report the 99th percentile latency as a function of the client request rate in Figure 4.9.

The baseline achieves 182k and 50k requests per second on Nginx and Apache respectively. The ZebRAM's first configuration (only ECC) reaches 172k and 49k while the second configuration reaches 166k and 49k.

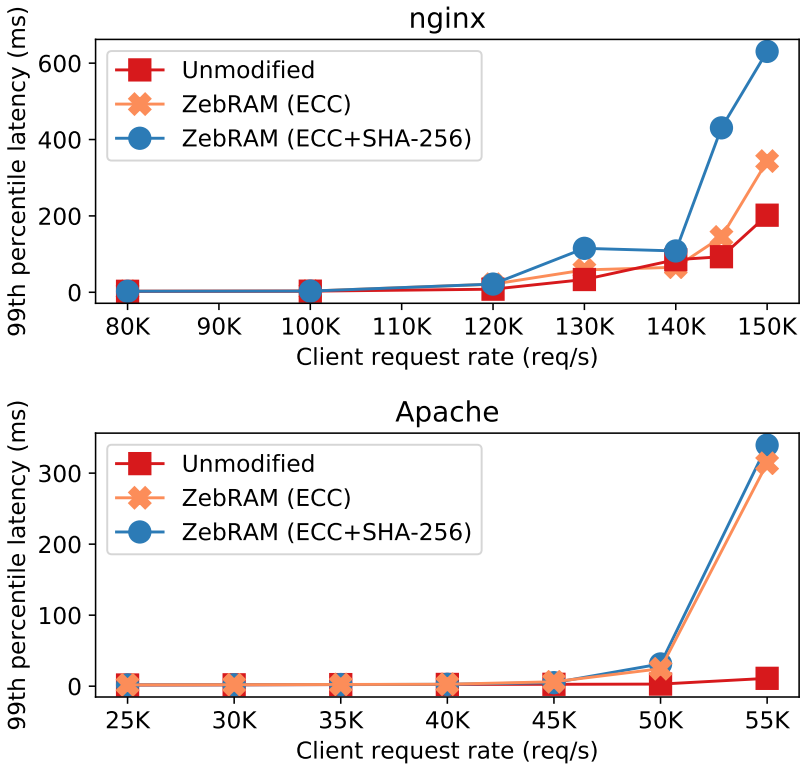

Figure 4.9. Nginx and Apache latency (99th percentile).

Before saturation, the results show that ZebRAM imposes no overhead on the 99th percentile latency. After then, both configurations of ZebRAM show a similar trend with linearly higher 99th percentile response time.

Overall, ZebRAM's performance impact on both web servers and SPEC benchmarks is low and mostly due to the inability to efficiently use Linux' THP support. This is expected, since as long as the working set can comfortably fit in the safe region (e.g., around 400MB for our web server experiments) the unsafe memory management overhead is completely masked. We isolate and study such overhead in more detail in the following.

Microbenchmarks To drill down the overhead of each single feature of ZebRAM, we measure the latency of swapping in a page from the ZebRAM device under different configurations. To measure the latency, we use a small binary that sequentially writes on every page of a large eviction buffer in a loop. This ensures

Table 4.2. Page swap-in latency from the ZebRAM device.

Configuration	median (ns)	90th (ns)	99th (ns)
copy	2,362.0	4,107.0	8,167.0
SHA-256	13,552.0	14,209.0	17,092.0
cache + comp + SHA-256	8,633.0	13,191.0	18,678.0
cache + comp + SHA-256 + ECC	9,862.0	15,118.0	20,794.0

that, between two accesses to the same page, we touch the entire buffer, evicting that page from memory. To be sure that Linux swaps in just one page for every access, we set the page-cluster configuration parameter to 0. In this experiment, two components interact with ZebRAM: our binary triggers swap-in events from the ZebRAM device while the `kswapd` kernel thread swaps pages to the ZebRAM device to free memory. The interaction between them is completely different if the binary uses exclusively loads to stress the memory. This is because the kernel would optimize out unnecessary flushes to swap and batch together TLB invalidations. Hence, we choose to focus on stores to study the performance in the worst-case scenario and because read-only workloads are less common than mixed workloads.

We reserve a core exclusively for the binary so that `kswapd` does not (directly) steal CPU cycles from it. We measure 1,000,000 accesses for each different configuration. Table 4.2 presents our results. We also run the binary in a loop and profile its execution with the `perf` Linux tool to measure the time spent in different functions. Due to function inlining, it is not always trivial to map a symbol to a particular feature. Nevertheless, `perf` can provide insights into the overhead at a fine granularity. In the first configuration, we disable the all features of ZebRAM and perform only memory copies into the ZebRAM device. As the copy operation is fast, the `perf` tool reports that just 4% percent of CPU cycles are spent copying. Interestingly, 47% of CPU cycles are spent serving Inter Process Interrupts from other cores. This is because, while we are swapping in, `kswapd` on another core is busy freeing memory. For this purpose, `kswapd` needs to unmap pages that are on their way to be swapped out from the process's page tables. This introduces TLB shootdowns (and IPIs) to invalidate other cores' TLB stale entries. It is important to notice that the faster we swap in pages, the faster `kswapd` needs to free memory. This unfortunately results in a negative feedback loop that represent one of the major sources of overhead when the large number of swap-in events continuously force `kswapd` to wake up.

Adding hashing (SHA-256) on top of the previous configuration shows an increase in latency, which is also reflected in the CPU cycles breakdown. The `perf`

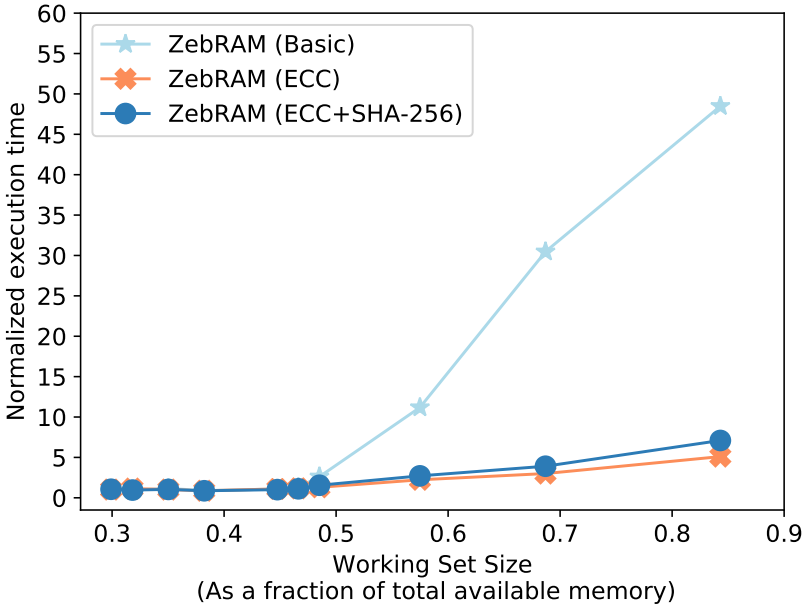

Figure 4.10. Redis throughput at saturation.

tool reports that 55% of CPU cycles are spent swapping in pages (copy + hashing), while serving IPIs accounts for 29%. Adding cache and compression on top of SHA-256 decreases the latency median and increases the 99th percentile. This is because, on a cache hit, the ZebRAM only needs to copy the page to userspace; however, on a cache miss, it has to verify the hash of the page and decompress the page too. The `perf` tool reports 42% of CPU cycles are spent in the decompression routine and 26% in serving IPI requests for other cores and less than 5% in hashing and copying. This confirms the presence of the swap-in/swap-out feedback loop under high memory pressure. Adding ECC marginally increases the latency, the `perf` tool reports similar CPU usage breakdown for the version without ECC.

Larger working sets As expected, ZebRAM's overheads are mostly associated to swap-in/swap-out operations, which are masked when the working set can fit in the safe region but naturally become more prominent as we grow the working set. In this section, we want to evaluate the impact of supporting increasingly larger working sets compared to a more traditional swap implementation. For this purpose, we evaluate the performance of a key-value store in four different

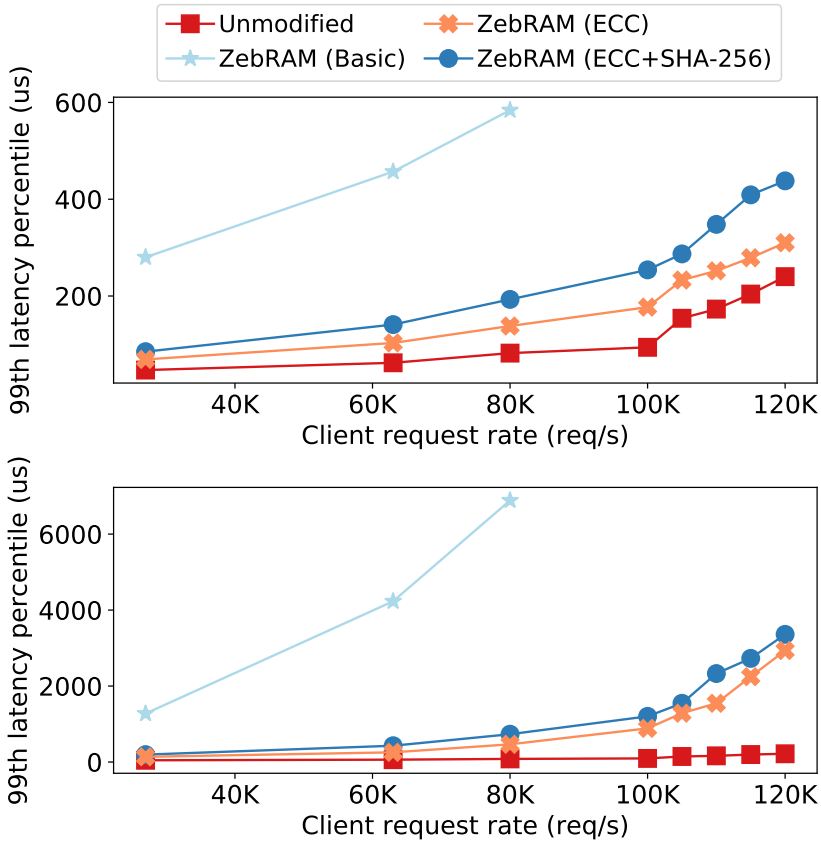

Figure 4.11. Redis latency (99th percentile). The working set size is 50% of RAM (top) and 70% of RAM (bottom).

setups: (i) unmodifed system, (ii) the basic version of ZebRAM (iii) ZebRAM configured with ECC, and (iv) ZebRAM configured with ECC and SHA-256. The basic version of ZebRAM uses just one of the two domains in which ZebRAM splits the RAM and swaps to a fast SSD disk when the memory used by the OS does not fit into it. We use YCSB[18] to generate load and induce a target working set size against a redis (4.0.8) key-value store. We setup YCSB to use 1KB objects and perform a 90/10 read/write operations ratio. Each test runs for 20 seconds and, for each configuration, we discard the results of 3 warmup rounds and report the median across 11 runs. We configure YCSB to access the dataset key space uniformly and we measure the throughput at saturation for different data set sizes.

Figure 4.10 depicts the reported normalized execution time as a function of the working set size (in percentage compared to the total RAM size). As shown in the figure, when the working set size is small enough (e.g., 44%) the OS hardly reclaims any memory, hence the unsafe region remains unutilized and the normalized execution time is only 1.08x for the basic version of ZebRAM while the normalized execution time is between 1.04x and 1.10x for all other configurations of ZebRAM. As we increase the working set size, the OS starts reclaiming pages and the normalized execution time increases accordingly. However, the increase is much more gentle for ZebRAM compared to the basic version of ZebRAM and the gap becomes more significant for larger working set sizes. For instance, for a fairly large working set size (e.g., 70%), ZebRAM (ECC) has 3.00x normalized execution time, and ZebRAM (ECC and SHA-256) has 3.90x, compared to the basic version of ZebRAM at 30.47x.

To study the impact of ZebRAM on latency, we fix the working set size to 50% and 70% of the total RAM and repeat the same experiment while varying the load on the server. Figure 4.11 presents our results for the 99th latency percentile. At 50%, results of (i) the ZebRAM configured with ECC, (ii) the ZebRAM configured with ECC and SHA-256, and (iii) baseline (unmodified) follow the same trend. The ZebRAM's first configuration (only ECC) reports a 99th latency percentile of 138us for client request rates below 80,000, compared to 584us for ZebRAM (basic). At 70%, the gap is again more prominent, with ZebRAM reporting a 99th latency percentile of 466us and ZebRAM (basic) reporting 6,887us.

Overall, ZebRAM can more gracefully reduce performance for larger working sets compared to a traditional (basic ZebRAM) swap implementation, thanks to its ability to use an in-memory cache and despite the integrity checks required to mitigate Rowhammer. As our experiments demonstrate, given a target performance budget, ZebRAM can support much larger working sets compared to the ZebRAM's basic implementation, while providing a strong defense against arbitrary Rowhammer attacks. This is unlike the basic ZebRAM implementation, which optimistically provides no protection against similar bit flip-based attacks. Unfortunately, such attacks, which have been long-known for DRAM [39], have recently started to target flash memory as well [14, 47].

4.8 Related work

This section summarizes related work on Rowhammer attacks and defenses.

Attacks In 2014, Kim et al. [39] were the first to show that it is possible to flip bits in DDR3 memory on x86 CPUs simply by accessing other parts of memory. Since then, many studies have demonstrated the effectiveness of Rowhammer as a real-world exploit in many systems.

The first practical Rowhammer-based privilege escalation attack, by Seaborn and Dullien [225], targeted the x86 architecture and DDR3 memory, hammering the memory rows by means of the native x86 `clflush` instruction that would flush the cache and allow high-frequency access to DRAM. By flipping bits in page table entries, the attack obtained access to privileged pages.

Not long after these earliest attacks, researchers greatly increased the threat of Rowhammer attacks by showing that is possible to launch them from JavaScript also, allowing attackers to gain arbitrary read/write access to the browser address space from a malicious web page [10, 32].

Moreover, newer attacks started flipping bits in memory areas other than page table entries, such as object pointers (to craft counterfeit objects [10]), op-codes [31], and even application-level sensitive data [71].

For instance, Flip Feng Shui demonstrated a new attack on VMs in cloud environments that flipped bits in RSA keys in victim VMs to make them easy to factorize, by massaging the physical memory of the co-located VMs to land the keys on a page that was hammerable by the attacker. Around the same time, other researchers independently also targeted RSA keys with Rowhammer but now for fault analysis [5]. Concurrently, also, Xiao et al. [87] presented another cross-VM attack that manipulates page table entries in Xen.

Where the attacks initially focused on PCs with DDR3 configurations, later research showed that ARM processors and DDR4 memory chips are also vulnerable [80]. While this opened the way for Rowhammer attacks on smartphones, the threat was narrower than on PCs, as the authors were not yet able to launch such attacks from JavaScript. This changed recently, when research described a new way to launch Rowhammer attacks from JavaScript on mobile phones and PC, by making use of the GPU. Hammering directly from the GPU by way of WebGL, the authors managed to compromise a modern smart phone browser in under two minutes. Moreover, this time the targeted data structures are doubles and pointers: by flipping a bit in the most significant bytes, the attack can turn pointers into doubles (making them readable) and doubles into pointers (yielding arbitrary read/write access).

All Rowhammer attacks until that point required local code execution. Recently, however, researchers demonstrated that even remote attacks on servers are possible [79], by sending network traffic over high-speed network to a victim

process, using RDMA NICs. As the server that is receiving the network packets is using DMA to write to its memory, the remote attacker is able to flip bits in the server. By carefully manipulating the data in a key-value store, they show that it is possible to completely compromise the server process.

It should be clear that Rowhammer exploits have spread from a narrow and arcane threat to target two of the most popular architectures, in all common computing environments, different types of memory (and arguably flash [14]), while covering most common threat models (local privilege escalation, hosted JavaScript, and even remote attacks). ZebRAM protects against all of the above attacks.

Defenses Kim et al. [39] propose multiple defenses against Rowhammer. These defenses have proven insufficient [3, 17] and infeasible to deploy on the required massive scale. The new LPDDR4 standard [112] specifies two features which together defend against Rowhammer: TRR and MAC. Despite these defenses, van der Veen et al. still report bit flips on a Google pixel phone with LPDDR4 memory [226] and Gruss et al. [224] report bit flips in TRR memory. While nobody has demonstrated Rowhammer attacks against ECC memory yet, the real problem with such hardware solutions is that most systems in use today do not have ECC, and replacing all DRAM in current devices is simply infeasible.

In order to protect from Rowhammer attacks, many vendors simply disabled features in their products to make life harder for attackers. For instance, Linux disabled unprivileged access to the *pagemap* [225], Microsoft disabled memory deduplication [181] to defend from the Dedup Est Machina attack [10], and Google disabled [209] the ION contiguous heap in response to the Drammer attack [80] on mobile ARM devices. Worryingly, not a single defence is currently deployed to protect from the recent GPU-based Rowhammer attack on mobile ARM devices (and PCs), even though it offers attackers a huge number of vulnerable devices.

Finally, researchers have proposed targeted software-based solutions against Rowhammer. ANVIL [3] relies on Intel's performance monitoring unit (PMU) to detect and refresh likely Rowhammer victim rows. An improved version of ANVIL requires specialized Intel PMUs with a fine-grained physical to DRAM address translation. Unfortunately, Intel's (and AMD's) PMUs do not capture precise address information when memory accesses bypass the CPU cache through DMA. Hence, this version of ANVIL is vulnerable to off-CPU Rowhammer attacks. Unlike ANVIL, ZebRAM is secure against off-CPU attacks, since device drivers transparently allocate memory from the safe region.

CATT [11] isolates (only) user and kernel space in physical memory so that

user-space attackers cannot trigger bit flips in kernel memory. However, research [31] shows CATT to be bypassable by flipping opcode bits in the sudo program code. Moreover, CATT does not defend against attacks that target co-hosted VMs at all [11]. In contrast, ZebRAM protects against co-hosted VM attacks, attacks against the kernel, attacks between (and even within) user-space processes and attacks from co-processors such as GPUs.

Other recent software-based solutions have targeted specific Rowhammer attack variants. GuardION isolates DMA buffers to protect mobile devices against DMA-based Rowhammer attacks [81]. ALIS isolates RDMA buffers to protect RDMA-enabled systems against Throwhammer [79]. Finally, VUSion randomizes page frame allocation to protect memory deduplication-enabled systems against Flip Feng Shui [62].

4.9 Discussion

This section discusses feature and performance tradeoffs between our ZebRAM prototype and alternative ZebRAM implementations.

4.9.1 Prototype

Because the ZebRAM prototype relies on the hypervisor to implement safe/unsafe memory separation, and on a cooperating guest kernel for swap management, both host and guest need modifications. In addition, the guest physical address space will map highly non-contiguously to the host address space, preventing the use of huge pages. The guest modifications, however, are small and self-contained, do not touch the core memory management implementation and are therefore highly compatible with mainline and third party LKMs.

4.9.2 Alternative Implementations

In addition to our implementation presented in Section 4.5, several alternative ZebRAM implementations are possible. Here, we compare our ZebRAM implementation to alternative hardware-based, OS-based, and guest-transparent virtualization-based implementations.

Hardware-based Implementing ZebRAM at the hardware level would require a physical-to-DRAM address mapping where sets of odd and even rows are mapped to convenient physical address ranges, for instance an even lower-half and an odd upper-half. This can be achieved with by a fully programmable memory controller, or implemented as a configurable feature in existing designs. With such a

mapping in place, the OS can trivially separate memory into safe and unsafe regions. In this model, the Swap Manager, Cache Manager and Integrity Manager are implemented as LKMs just as in the implementation from Section 4.5. In contrast to other implementations, a hardware implementation requires no hypervisor, allows the OS to make use of (transparent) huge pages and requires minimal modifications to the memory management subsystem. While a hardware-supported ZebRAM implementation has obvious performance benefits, it is currently infeasible to implement because memory controllers lack the required features.

OS-based Our current ZebRAM prototype implements the Memory Remapper as part of a hypervisor. Alternatively, the Memory Remapper can be implemented as part of the bootloader, using Linux' boot memory allocator to reserve the unsafe region for use as swap space. While this solution obviates the use of a hypervisor, it also results in a non-contiguous physical address space that precludes the use of huge pages and breaks DMA in older devices. In addition, it is likely that this approach requires invasive changes to the memory management subsystem due to the very fragmented physical address space.

Transparent Virtualization-based While our current ZebRAM implementation requires minor changes to the guest OS, it is also possible to implement a virtualization-based variant of ZebRAM that is completely transparent to the guest. This entails implementing the ZebRAM swap disk device in the host and then exposing the disk to the guest OS as a normal block device to which it can swap out. The drawback of this approach is that it degrades performance by having the hypervisor interposed between the guest OS and unsafe memory for each access to the swap device, a problem which does not occur in our current implementation. The clear advantage to this approach is that it is completely guest-agnostic: guest kernels other than Linux, including legacy and proprietary ones are equally well protected, enabling existing VM deployments to be near-seamlessly transitioned over to a Rowhammer-safe environment.

4.10 Conclusion

The root cause of the Rowhammer bug is the vendor's design choice to optimize the cost-per-bit by cramming bits so close together. As the bit flips/memory errors induced by Rowhammer happen without any indication, the system fails to detect it. Clearly, this design of memory chips violates the design principle

called *Fail securely* (see Table 1.2). The basic idea behind the *Fail securely* is that when a system fails, it should do so securely; the confidentiality and integrity of a system should remain even though availability has been lost. The attackers must not be permitted to gain access rights to privileged objects that are normally inaccessible during a failure.

In this chapter, we have introduced ZebRAM, the first comprehensive software defense against all forms of Rowhammer. ZebRAM uses guard rows to isolate all memory rows containing user or kernel data, protecting these from Rowhammer-induced bit flips. Moreover, ZebRAM implements an efficient integrity-checked memory-based swap disk to utilize the memory sacrificed to the guard rows. Since Rowhammer's root cause is vendors optimizing the cost-per-bit by cramming bits so close together that an access can flip bits in adjacent cells, no reliable solution to undo such effects will be free of cost. Nevertheless, our evaluation shows ZebRAM to be a strong defense able to use all available memory at a cost that is a function of the workload. To aid future work, we release ZebRAM as open source.

5 An In-depth Look into a Modern Cyber Threat (Cryptojacking)

A wave of alternative coins that can be effectively mined without specialized hardware, and a surge in cryptocurrencies' market value has led to the development of cryptocurrency mining (*cryptomining*) services, such as Coinhive, which can be easily integrated into websites to monetize the computational power of their visitors. While legitimate website operators are exploring these services as an alternative to advertisements, they have also drawn the attention of cybercriminals: *drive-by mining* (also known as *cryptojacking*) is a new web-based attack, in which an infected website secretly executes JavaScript code and/or a WebAssembly module in the user's browser to mine cryptocurrencies without her consent.

This new class of cyber threat neither violates any of the current design principles nor exploits an implementation bug. It does not break today's widely-accepted security model called CIA triad (standing for Confidentiality, Integrity, and Availability); but, still is a very practical and stealthy cyber attack that monetizes of victim's computational resources. In this chapter, we perform a comprehensive analysis on Alexa's Top 1 Million websites to shed light on the prevalence and profitability of this attack. We study the websites affected by drive-by mining to understand the techniques being used to evade detection, and the latest web technologies being exploited to efficiently mine cryptocurrency. As a result of our study, which covers 28 Coinhive-like services that are widely being used by drive-by mining websites, we identified 20 active cryptomining campaigns.

Motivated by our findings, we investigate possible countermeasures against this type of attack. We discuss how current blacklisting approaches and heuristics based on CPU usage are insufficient, and present MineSweeper, a novel

detection technique that is based on the intrinsic characteristics of cryptomining code, and, thus, is resilient to obfuscation. Our approach could be integrated into browsers to warn users about silent cryptomining when visiting websites that do not ask for their consent.

5.1 Introduction

Ever since its introduction in 2009, Bitcoin [187] has attracted the attention of cybercriminals due to the possibility to perform and receive anonymous payments. In addition, the financial reward for using computing power for mining has incentivized criminals to experiment with *silent* cryptocurrency miners (*cryptominers*), which gained popularity among malware authors who were, after all, already in the business of compromising PCs and herding large numbers of them in botnets. However, as Bitcoin mining became too difficult for regular machines, the profits of mining botnets dwindled, and Bitcoin-mining botnets declined: an analysis by McAfee in 2014 suggested that malicious miners are not profitable on PCs and certainly not on mobile devices [173].

Since then, a wave of alternative coins (altcoins) has been introduced: the market now counts over 1,500 cryptocurrencies, out of which more than 600 see an active trade. At the time of writing, they represent over 50% of the cryptocurrency market [95]. Unlike Bitcoin, many of them are still mineable without specialized hardware. Furthermore, miners can organize themselves into *mining pools*, which allow members to distribute mining tasks and share the rewards. These new currencies, and an overall surge in market value across cryptocurrencies at the end of 2017 [24], has renewed interest in cryptominers and led to the proliferation of cryptomining services, such as Coinhive [140], which can easily be integrated into a website to mine on its visitors' devices from within the browser.

For cybercriminals, these services provide a low-effort way to monetize websites as part of *drive-by mining* (or *cryptojacking*) attacks: they either compromise webservers (through exploits [131, 175, 191, 203, 208], or taking advantage of misconfigurations [190]) and install JavaScript-based miners, distribute their miners through advertisements (including Google's DoubleClick on YouTube [155] and the AOL advertising platform [178]), or compromise third-party libraries [215] included in numerous websites. Attackers also have come up with creative tactics to conceal their attack, for example by using "pop-under" windows [154] (to maximize the time a victim spends on the mining website), or by abusing Coinhive's URL shortening service [221]. Finally, rogue WiFi hotspots [137] and compromised routers [169] allow attackers to inject the mining payload on a large scale into *any* website that their users visit.

However, in-browser mining is not malicious per-se: charities, such as UNICEF [176], launched dedicated websites to mine for donations, and legitimate websites are exploring mining in an attempt to monetize their content in the presence

of ad blockers [196]. Whether users accept cryptocurrency miners as an alternative to invasive advertisements, which raise privacy concerns due to wide-spread targeting and tracking [16, 55, 63], remains to be seen. For them, in-browser mining degrades their system's performance and increases its power consumption [102]. Therefore, the key distinction between these use cases and drive-by mining attacks is user consent and whether a website discloses its mining activity or not. For example, as a way to enforce user consent for in-browser mining, Coinhive launched AuthedMine [107], which explicitly requires user input. However, a related study has found that this API has not yet found widespread adoption [199]. Related work also suggested the introduction of a "do not mine" HTTP header [23], which, however, websites do not necessarily need to honor.

To study the prevalence of drive-by mining attacks, i.e., in-browser mining without requiring any user interaction or consent, we performed a comprehensive analysis of Alexa's Top 1 Million websites [127]. As a result of our study, which covers 28 Coinhive-like services, we identified 20 active cryptomining campaigns. In contrast to a previous study, which found cryptomining on low-value targets, such as parked websites, and concluded that cryptomining was not very profitable [23], we find that cryptomining can indeed make economic sense for an attacker. We identified several video players used by popular video streaming websites that include cryptomining code and which maximize the time users spend on a website mining for the attacker—potentially earning more than US$ 30,000 a month. Furthermore, we found that instead of JavaScript-based attacks, drive-by mining now largely takes advantage of WebAssembly (Wasm) to efficiently mine cryptocurrencies and maximize profits.

As a countermeasure, browsers [138, 213, 218], dedicated browser extensions [229, 232], and ad blockers have started to use blacklists. However, maintaining a complete blacklist is not scalable, and it is prone to false negatives. These blacklists are often manually compiled and are easily defeated by URL randomization [198] and domain generation algorithms (DGAs), which are already actively being used in the wild [220]. Other detection attempts look for high CPU usage as an indicator that cryptocurrency mining is taking place. This not only causes false positives for other CPU-intensive use cases, but also causes false negatives, as cryptocurrency miners have started to throttle their CPU usage to evade detection [23].

In this work, we focus on Wasm-based mining, the most efficient and widespread technique for drive-by mining attacks. We propose MineSweeper, a drive-by mining defense that is based on identifying the intrinsic characteristics of the mining itself: the execution of its hashing function. Our first approach is to per-

form static analysis on the Wasm code and to identify the hashing code based on the cryptographic operations it performs. Currently, attackers avoid heavy obfuscation of the Wasm code as it comes with performance penalties, and hence decreases profits. To deal with future evasion techniques, we present a second, more obfuscation-resilient detection approach: by monitoring CPU cache events at run time we can identify cryptominers based on their memory access patterns.

As browsers are currently struggling to find a suitable alternative to blacklists [159], the techniques used by MineSweeper could be adopted as a defense mechanism against drive-by mining, for example by warning users and enforcing their consent before allowing mining scripts to execute or blocking mining scripts altogether.

Contributions In summary, our contributions are the following:

1. We perform the first in-depth assessment of drive-by mining.
2. We discuss why current defenses based on blacklisting and CPU usage are ineffective.
3. We propose MineSweeper, a novel detection approach based on the identification of cryptographic functions through static analysis and monitoring of cache events during run time.

In the spirit of open science, we make the collected datasets and the code we developed for this work publicly available at https://github.com/vusec/minesweeper.

5.2 Background

A cryptocurrency is a medium of exchange much like the Euro or the US Dollar, except that it uses cryptography and blockchain technology to control the creation of monetary units and to verify the transaction of a fund. *Bitcoin* [187] was the first such decentralized digital currency. A cryptocurrency user can transfer money to another user by forming a transaction record and committing it to a distributed write-only database called *blockchain*. The blockchain is maintained by a peer-to-peer network of *miners*. A miner collects transaction data from the network, validates it, and inserts it into the blockchain in the form of a block. When a miner successfully adds a valid block to the blockchain, the network compensates the miner with cryptocurrency (e.g., Bitcoins). In the case of Bitcoin, this process is called *Bitcoin mining*, and this is how new Bitcoins en-

ter circulation. Bitcoin transactions are protected with cryptographic techniques that ensure only the rightful owner of a Bitcoin wallet address can transfer funds from it.

To add a block (i.e., a collection of transaction data) to the blockchain, a miner has to solve a cryptographic puzzle based on the block. This mechanism prevents malicious nodes from trying to add bogus blocks to the blockchain and earn the reward illegitimately. A valid block in the blockchain contains a solution to a cryptographic puzzle that involves the hash of the previous block, the hash of the transactions in the current block, and a wallet address to credit with the reward.

5.2.1 Cryptocurrency Mining Pools

The cryptographic puzzle is designed such that the probability of finding a solution for a miner is proportional to the miner's computational power. Due to the nature of the mining process, the interval between mining events exhibits high variance from the point of view of a single miner. Consequently, miners typically organize themselves into *mining pools*. All members of a pool work together to mine each block, and share the reward when one of them successfully mines a block.

The protocol used by miners to reliably and efficiently fetch jobs from mining pool servers is known as *Stratum* [116]. It is a cleartext communication protocol built over TCP/IP, using a JSON-RPC format. Stratum prescribes that miners who want to join the mining pool first send a *subscription* message, describing the miner's capability in terms of computational resources. The pool server then responds with a *subscription response* message, and the miner sends an *authorization request* message with its username and password. After successful authorization, the pool sends a *difficulty notification* that is proportional to the capability of the miner—ensuring that low-end machines get easier jobs (i.e., puzzles) than high-end ones. Finally, the pool server assigns these jobs by means of *job notifications*. Once the miner finds a solution, it sends it to the pool server in the form of a *share*. The pool server rewards the miner in proportion to the number of valid shares it submitted and the difficulty of the jobs.

5.2.2 In-browser Cryptomining

The idea of cryptomining by simply loading a webpage using JavaScript in a browser exists since Bitcoin's early days. However, with the advent of GPU- and ASIC-based mining, browser-based Bitcoin mining, which is $1.5x$ slower than na-

tive CPU mining [23], became unprofitable. Recently, the cause for the decline of JavaScript-based cryptocurrency miners has subsided: due to new CPU-mineable altcoins and increasing cryptocurrency market value, it is now profitable to mine cryptocurrencies with regular CPUs again. In 2017, Coinhive was the first to revisit the idea of in-browser mining. They provide APIs to website developers for implementing in-browser mining on their websites and to use their visitors' CPU resources to mine the altcoin Monero. Monero employs the CryptoNight algorithm [200] as its cryptographic puzzle, which is optimized towards mining by regular CPUs and provides strong anonymity; hence, it is ideal for in-browser cryptomining.[1] Moreover, the development of new web technologies that have been happening in parallel allows for more efficient—and thus profitable—mining in the browser.

5.2.3 Web Technologies

Web developers continuously strive to deploy performance-critical parts of their application in the form of native code and run it inside the browser securely. As such, there are on-going research and development efforts to improve the performance of native code execution in the web browser [33, 104]. Naturally, the developers of JavaScript-based cryptominers started exploiting these advancements in web technologies to speed up drive-by mining, thus taking advantage of two web technologies: *asm.js* and *WebAssembly*.

In 2013, Mozilla introduced asm.js, which takes C/C++ code to generate a subset of JavaScript code with annotations that the JavaScript engine can later compile to native code. To improve the performance of native code in the browser even further, in 2017, the World Wide Web Consortium developed WebAssembly (Wasm). Any C/C++/Rust-based application can be easily converted to Wasm, a binary instruction format for a stack-based virtual machine, and executed in the browser at native speed by taking advantage of standard hardware capabilities available on a wide range of platforms. Today, all four major browsers (Firefox, Chrome, Safari, and Edge) support Wasm.

The main difference between *asm.js* and *Wasm* is in the way in which the code is optimized. In asm.js, the JavaScript Just-in-Time (JIT) compiler of the browser converts the JavaScript to an Abstract Syntax Tree (AST). Then, it compiles the AST to non-optimized native code. Later, at run time, the JavaScript JIT engine looks for slow code paths and tries to re-optimize this code at run time. The

[1] Note that Monero is not the only altcoin that uses the CryptoNight algorithm: most CPU-mineable coins that exist today, such as Bytecoin, Bitsum, Masari, Stellite, AEON, Graft, Haven Protocol, Intense Coin, Loki, Electroneum, BitTube, Dero, LeviarCoin, Sumokoin, Karbo, Dinastycoin, and TurtleCoin are based on CryptoNight.

detection and re-optimization of slow code paths consume a substantial amount of CPU cycles. In contrast, Wasm performs the optimization of the whole module only once, at compile time. As a result, the JIT engine does not need to parse and analyze the Wasm module to re-optimize it. Rather, it directly compiles the module to native code and starts executing it at native speed.

5.2.4 Existing Defenses against Drive-by Mining

Until now, there is no reliable mechanism to detect drive-by mining. The developers of CoinBlockerLists [227] maintain a blacklist of mining pools and proxy servers that they manually collect from reports on security blogs and Twitter. Dr. Mine [231] attempts to block drive-by mining by means of explicitly blacklisted URLs (based on for example CoinBlockerLists). In particular, it detects JavaScript code that tries to connect to blacklisted mining pools. MinerBlock [229] further combines blacklists with detecting potential mining code inside loaded JavaScript files. Both approaches suffer from high false negatives: as we show in our analysis, most of the drive-by mining websites are using obfuscated JavaScript and randomized URLs to evade the aforementioned detection techniques.

Google engineers from the Chromium project recently acknowledged that blacklisting does not work and that they are looking for alternatives [159]. Specifically, they considered adding an extra permission to the browser to throttle code that runs the CPU at high load for a certain amount of time. Related studies also found high CPU usage from the website as an indicator of drive-by mining [186]. At the same time, another recent study shows that many drive-by miners are throttling their CPU usage to around 25% [23] and simply considering the CPU usage alone as the indicator of drive-by mining suffers from high false negatives. Even without taking the CPU throttling to such extremes, drive-by miners can blend in with other browsing activity, potentially leading to false positives for other CPU-intensive use cases, such as games [198].

Making matters worse, in-browser mining service providers, such as Coinhive, have no incentives to disrupt drive-by mining attacks: Coinhive keeps 30% of the cryptocurrency that is mined with its code. In reaction to abuse complaints, they reportedly keep all of the profits of campaigns, whose members still keep mining cryptocurrency even after their site key (i.e., the campaign's account identifier with Coinhive) has been terminated [172].

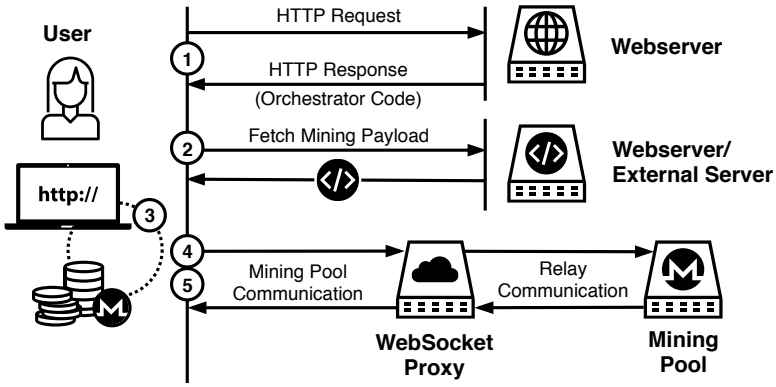

Figure 5.1. Overview of a typical drive-by mining attack.

5.3 Threat Model

We consider only drive-by mining rather than legitimate browser-based mining in our threat model, i.e., we measure only the prevalence of mining without users' consent. A website may host stealthy miners for many reasons. Some website owners knowingly include them on their sites without informing the users to monetize their sites on the sly. However, it is also possible that the owners are unaware that their site is stealing CPU cycles from their visitors. For instance, silent cryptocurrency miners may ship with advertisements or third-party services. In some cases, the attackers install the miners after they compromise a victim site. In this research, we measure, analyze, and detect all these cases of drive-by mining.

Figure 5.1 illustrates a typical drive-by mining attack. A cryptocurrency mining script contains two components: the *orchestrator* and the *mining payload*. When a user visits a drive-by mining website, the website (1) serves the orchestrator script, which checks the host environment to find out how many CPU cores are available, (2) downloads the highly-optimized cryptomining payload (as either Wasm or asm.js) from the website or an external server, (3) instantiates a number of web workers [117], i.e., spawns separate threads, with the mining payload, depending on how many CPU cores are available, (4) sets up the connection with the mining pool server through a WebSocket proxy server, and, (5) finally, fetches work from the mining pool and submits the hashes to the mining pool through the WebSocket proxy server. The protocol used for this communication with the mining pool is usually Stratum.

Table 5.1. Summary of our dataset and key findings.

Crawling period	March 12, 2018 – March 19, 2018
# of crawled websites	991,513
# of drive-by mining websites	1,735 (0.18%)
# of drive-by mining services	28
# of drive-by mining campaigns	20
# of websites in biggest campaign	139
Estimated overall profit	US$ 188,878.84
Most profitable/biggest campaign	US$ 31,060.80
Most profitable website	US$ 17,166.97

5.4 Drive-by Mining in the Wild

The goals of our large-scale analysis of active drive-by mining campaigns in the wild are two-fold: first, we investigate the prevalence and profitability of this threat to show that it makes economic sense for cybercriminals to invest in this type of attack—being a low effort heist with potentially high rewards. Second, we evaluate the effectiveness of current drive-by mining defenses, and show that they are insufficient against attackers who are already actively using obfuscation to evade detection. Based on our findings, we propose an obfuscation-resilient detection system for drive-by mining websites in Section 5.5.

As part of our analysis, we first crawl Alexa's Top 1 Million websites, log and analyze all code served by each website, monitor side effects caused by executing the code, and capture the network traffic between the visited website and any external server. Then, we proceed to detect cryptomining code in the logged data and the use of the Stratum protocol for communicating with mining pool servers in the network traffic of each website. Finally, we correlate the results from all websites to answer the following questions:

1. How prevalent is drive-by mining in the wild?
2. How many different drive-by mining services exist currently?
3. Which evasion tactics do drive-by mining services employ?
4. What is the modus operandi of different types of campaigns?
5. How much profit do these campaigns make?
6. Can we find common characteristics across different drive-by mining services that we can use for their detection?

Table 5.2. Types of mining services in our initial dataset and their keywords.

Mining Service	Keywords
Coinhive	new CoinHive\.Anonymous | coinhive.com/lib/coinhive.min.js | authedmine.com/lib/
CryptoNoter	minercry.pt/processor.js | \.User\(addr
NFWebMiner	new NFMiner | nfwebminer.com/lib/
JSECoin	load.jsecoin.com/load
Webmine	webmine.cz/miner
CryptoLoot	CRLT\.anonymous | webmine.pro/lib/crlt.js
CoinImp	www.coinimp.com/scripts | new CoinImp.Anonymous | new Client.Anonymous | freecontent.stream | freecontent.data | freecontent.date
DeepMiner	new deepMiner.Anonymous | deepMiner.js
Monerise	apin.monerise.com | monerise_builder
Coinhave	minescripts\.info'
Cpufun	snipli.com/[A-Za-z]+\" data-id='
Minr	abc\.pema\.cl | metrika\.ron\.si | cdn\.rove\.cl | host\.dns\.ga | static\.hk\.rs | hallaert\.online | st\.kjli\.fi | minr\.pw | cnt\.statistic\.date | cdn\.static-cnt\.bid | ad\.g-content\.bid | cdn\.jquery-uim\.download'
Mineralt	ecart\.html\?bdata= | /amo\.js\"> | mepirtedic\.com'

Table 5.1 summarizes our dataset and key findings. We start by discussing our data collection approach in Section 5.4.1, explain how we identify drive-by mining websites in Section 5.4.2, explore websites and campaigns in-depth, as well as estimate their profit in Section 5.4.3, and finally summarize characteristics that are common across the identified drive-by mining services in Section 5.4.4.

5.4.1 Data Collection

As the basis for our analysis, we built a web crawler for visiting Alexa's Top 1 Million websites and collecting data related to drive-by mining. During our preliminary analysis, we observed that many malicious websites serve a mining payload only when the user visits an internal webpage. Thus, in contrast to related studies [185, 102, 73] that based their analysis only on the websites' landing pages,[2] we configured the crawler to visit three random internal pages of each website. The crawler stayed for four seconds on each visited page. Moreover, we configured it to passively collect data from each visited website without simulating any user interactions. That is, the crawler did not give any consent for cryptomining.

[2]PublicWWW [194] only recently started indexing internal pages: https://twitter.com/bad_packets/status/1029553374897696768 (August 14, 2018)

Listing 5.1. Example usage of the Coinhive mining service.

```
<script src="https://coinhive.com/lib/coinhive.min.js">
</script>
<script>
    var miner = new CoinHive.Anonymous('CLIENT-ID',
                                       {throttle: 0.9});
    miner.start();
</script>
```

5.4.1.1 Cryptomining Code

To identify the cryptomining payloads that the drive-by mining website serves to client browsers, the web crawler saves the webpage, any embedded JavaScript, and all the requests originating from and responses served to the webpage. Then, our offline analyzer parses these logs to identify known drive-by mining services (such as Coinhive or Mineralt). As a first approximation, it does so using string matches, similar to existing defenses (see Section 5.2.4). However, this is only the first step in our analysis: as we show later, relying on pattern matching alone to detect drive-by mining easily leads to false negatives.

As explained in the previous section, the mining code consists of two components: the orchestrator and the optimized hash generation code (i.e., the mining payload), which we can both identify independently of each other.

Identification of the orchestrator Usually, websites embed the orchestrator script in the main webpage, which we can detect by looking for specific string patterns. For instance, Listing 5.1 shows a website using Coinhive's service for drive-by mining by including the orchestrator component (`coinhive.min.js`) inside the `<script>` HTML tag. In this case, searching for keywords such as `CoinHive.Anonymous` or `coinhive.min.js` is enough to identify whether a website is using this particular drive-by mining service. We manually collected keywords for 13 well-known mining services (see Table 5.2) to identify the websites that are using them.

Identification of the mining payload. The orchestrator first checks whether the browser supports Wasm. If not, the browser loads the optimized hash generation mining payload in the web worker using asm.js, otherwise, the mining payload (Wasm module) is served to the client in one of the following three ways: (i) the code is stored in the orchestrator script in a text format, which is compiled at run time to create the Wasm module, (ii) the orchestrator script retrieves a pre-compiled Wasm module at run time from an external server, or (iii) the web worker itself directly downloads a compiled Wasm module from an exter-

nal server and executes it. For all three cases, we could have used the Chrome browser (which supports Wasm) with the `--dump-wasm-module` flag to dump the Wasm module that the JIT engine (V8) executes. However, this flag is not officially documented [210] and at the time of our large-scale analysis we were not aware of this feature. Hence, we detect the Wasm-based mining payload in the following way: First, we dump all the JavaScript code and search for keywords, such as `cryptonight_hash` and `CryptonightWasmWrapper`; the existence of these keywords in the JavaScript implies the mining payload is served in text format. We detect the second and third way of serving the payload by logging and analyzing all the network requests and responsens from and to the browser's web worker.

Code obfuscation. We noticed that many drive-by mining services obfuscate both the strings used in the orchestrator script and in the Wasm module to defeat such keyword-based detection. Hence, we also look for other indicators for cryptomining and store the Wasm module for further analysis. In this way, we can estimate the number of drive-by mining services that employ code obfuscation during our in-depth analysis in Section 5.4.3.3.

5.4.1.2 CPU Load as a Side Effect

A cryptominer is a CPU-intensive program; hence, execution of the mining payload usually results in a high CPU load. However, websites may also intentionally throttle their CPU usage, either to evade detection or an attempt to conserve a visitor's resources. As part of our analysis, we investigate how many websites keep the CPU usage lower than a certain threshold. To this end, we configured the web crawler to log the CPU usage of each core and aggregate the usage across cores.

5.4.1.3 Mining Pool Communication

Typically, a miner talks to a mining pool to fetch the block's headers to start computing hashes. Stratum is the most commonly used protocol to authenticate with the mining pool or the proxy server to receive the job that needs to be solved, and, if the correct hash is computed, to announce the result. Most drive-by mining websites use WebSockets for this type of communication. As processes running in a browser sandbox are not permitted to open system sockets, WebSockets were designed to allow full-duplex, asynchronous communication between code running on a webpage and servers. As a result of using WebSockets, the operators

Table 5.3. Stratum protocol commands and their keywords.

Command	Keywords
Authentication	type:auth | command:connect | identifier:handshake | command:info
Authentication accepted	type:authed | command:work
Fetch job	identifier:job | type:job | command:work | command:get_job | command:set_job
Submit solved hash	type:submit | command:share
Solution accepted	command:accepted
Set CPU limits	command:set_cpu_load

of drive-by mining services need to set up WebSocket servers to listen for connections from their miners, and either process this data themselves if they also operate their own mining pool or *unwrap* the traffic and forward it to a public pool.

Consequently, we log all the WebSocket frames which are sent and received by the browser, as well as the AJAX request/response from the webpage. Then, we analyze the logged data to detect any mining pool communication by searching for command and keywords that are used by the Stratum protocol (listed in Table 5.3). During this analysis, we also observed that some websites are obfuscating the communication with the mining pool to evade detection. Thus, if the logged data does not include any text but only binary content, we mark the WebSocket communication as *obfuscated*.

Extraction of pools, proxies and site keys. The communication between a cryptominer and the proxy server contains two interesting pieces of information: the proxy server address and the client identifier (also known as the *site key*). We also found several drive-by mining services that include the public mining pool and associated cryptocurrency wallet address that the proxy should use.

Clustering miners based on the proxy to which they connect gives us insights on the number of different drive-by mining services that are currently active. Additionally, clustering miners based on their site key can be used to identify campaigns. Finally, we can leverage information from public mining pool to estimate the profitability of different campaigns.

We extract this information by looking for keywords in each request sent from the cryptominer and its response. Table 5.3 lists the keywords commonly associated with each request/response pair in the Stratum protocol. For instance, if the request sent from the miner contains keywords related to *authentication*, we extract the site key from it.

5.4.1.4 Deployment and Dataset

We deployed our web crawler in Docker containers running on Kubernetes in an unfiltered network. We ran 50 Docker containers in parallel for one week mid-March 2018 to collect data from Alexa's Top 1 Million websites (as of February 28, 2018). Around 1% of the websites were offline or not responding, and we managed to crawl 991,513 of them. This process resulted in a total of 4.6 TB raw data, and a 550 MB database for the extracted information on identified miners, CPU load, and mining pool communication.

5.4.2 Data Analysis and Correlation

We first analyze the different artifacts produced by the data collection individually, i.e., the cryptomining code itself, the CPU load as a side effect, and the mining pool communication. We discuss how relying on each of these artifacts alone can lead to both false positives and false negatives, and therefore correlate our results across all three dimensions.

5.4.2.1 Cryptomining Code

We identified 13 well-known cryptomining services using the keywords listed in Table 5.2 and present our results in Table 5.4. We detected 866 websites (0.09%) that are using these 13 services without obfuscating the orchestrator code in the webpage. The majority of websites (59.35%) is using the Coinhive cryptomining service. We also found 65 websites using multiple cryptomining services.

We revisited this analysis after our data correlation (described in 5.4.2.4) and manually analysed part of the mining payloads of websites that we detected based on other signals. In this way, we extended our initial list of keywords for detecting unobfuscated payloads with `hash_cn`, `cryptonight`, `WASMWrapper`, and `crytenight`, and we were able to identify mining services that were not part of our initial dataset, but that are using CryptoNight-based payloads. In total, we could identify 1,627 websites based on either keywords in the orchestrator *or* in the mining payload.

However, similar to current blacklist-based approaches, keyword-based analysis alone suffers from false positives and false negatives. In terms of false positives, this approach does not consider user consent, i.e., whether a website waits for a user's consent before executing the mining code. In terms of false negatives, this approach cannot detect drive-by mining websites that use code obfuscation and URL randomization, which we detected being applied in some form or another by 82.14% of the services in our dataset (see Section 5.4.3.3).

Table 5.4. Distribution of well-known cryptomining services.

Mining Service	Number of Websites	Percentage
Coinhive	514	59.35%
CoinImp	94	10.85%
Mineralt	90	10.39%
JSECoin	50	5.77%
CryptoLoot	39	4.50%
CryptoNoter	31	3.58%
Coinhave	14	1.62%
Minr	13	1.50%
Webmine	8	0.92%
DeepMiner	5	0.58%
Cpufun	4	0.46%
Monerise	2	0.23%
NF WebMiner	2	0.23%
Total	866	100%

5.4.2.2 CPU Load as a Side Effect

Even though we logged the CPU load for each website during our crawl, we ultimately do not use these measurements to detect drive-by mining websites for the following reasons: First, since we were running the experiments in Docker containers, the other processes running on the same machine could affect and artificially inflate our CPU load measurement. Second, the crawler spends only four seconds on each webpage, thus, the page loading itself might lead to higher CPU loads.

We can, however, use these measurements to specifically look for drive-by mining websites with low CPU usage to give a lower bound for the pervasiveness of CPU throttling across miners and the false negatives that a detection approach solely relying on high CPU loads would cause.

5.4.2.3 Mining Pool Communication

Overall, 59,319 (5.39%) out of Alexa's Top 1 Million websites use WebSockets to communicate with external servers. Out of these, we identified 1,008 websites that are communicating with mining pool servers using the Stratum protocol based on the keywords shown in Table 5.3. We also found that 2,377 websites are encoding the data (as Hex code or salted Base64) that they send and receive

through the WebSocket, in which case we could not determine whether they are mining cryptocurrency.

Even though we successfully identified 1,008 drive-by mining websites using this method, this detection method suffers from the following two drawbacks, causing false negatives: drive-by mining services may use a custom communication protocol (that is, different keywords than the ones presented in Table 5.3), or they may be obfuscating their communication with the mining pool.

5.4.2.4 Data Correlation

In our preliminary analysis based on keyword search, we identified 866 websites using 13 well-known cryptomining services. To determine how many of these websites start mining without waiting for a user to give her consent, for example by clicking a button (which our web crawler was not equipped to do), we leverage the identification of the Stratum protocol: we identify 402 websites, based on both their cryptomining code and the communication with external pool servers, that initiate the mining process without requiring a user's input. The remaining 464 websites either wait for the user's consent, circumvent our Stratum protocol detection, or did not initiate the Stratum communication within the timeframe our web crawler spent on the website.

To extend our detection to miners that evade keyword-based detection, we combine the collected information from the following sources:

- *Mining payload*: Websites identified based on keywords found in the mining payload.
- *Orchestrator*: Websites identified based on keywords found in the orchestrator code.
- *Stratum*: Websites identified as using the Stratum communication protocol.
- *WebSocket communication*: Websites that potentially use an obfuscated communication protocol.
- *Number of web workers*: All the in-browser cryptominers use web worker threads to generate hashes, while only 1.6% of all websites in our dataset use more than two web worker threads.

We identify drive-by mining websites by taking the $union$ of all websites for which we identified the mining payload, orchestrator, *or* the Stratum protocol. We further add websites for which we identified WebSocket communication with an external server *and* more than two web worker threads.

As a result, we identify 1,735 websites as mining cryptocurrency, out of which 1,627 (93.78%) could be identified based on keywords in the cryptomining code, 1,008 (58.10%) use the Stratum protocol in plaintext, 174 (10.03%) obfuscate the communication protocol, and all the websites (100.00%) use Wasm for the cryptomining payload and open a WebSocket. Furthermore, at least 197 (11.36%) websites throttle their CPU usage to less than 50%, while for only 12 (0.69%) mining websites we observed a CPU load of less than 25%. In other words, relying on high CPU loads (e.g., $\geq$50%) for detection would result in 11.36% false negatives in this case (in addition to potentially causing false positives for other CPU-intensive loads, such as games and video codecs). Similarly, relying only on pattern matching on the payload would result in 6.23% false negatives.

Finally, in addition to the 13 well-known drive-by mining services that we started our analysis with (see Table 5.4), we also discovered 15 new drive-by mining services (see Section 5.4.3.6), for a total of 28 drive-by mining services in our dataset.

5.4.3 In-depth Analysis and Results

Based on the drive-by mining websites we detected during our data correlation, we now answer the questions posed at the beginning of this section.

5.4.3.1 User Notification and Consent

We consider cryptomining as abuse unless a user explicitly consents, e.g., by clicking a button. While one of the first court cases on in-browser mining suggests a more lenient definition of consent and only requires websites to provide a clear notification about the mining behavior to the user [165], we find that very few websites in our dataset do so.

To locate any notifications, we searched for mining-related keywords (such as `CPU`, `XMR`, `Coinhive`, `Crypto` and `Monero`) in the identified drive-by mining website's HTML content. In this way, we identified 67 out of 1,735 (3.86%) websites that inform their users about their use of cryptomining. These websites include 51 proxy servers to the Pirate Bay, as well as 16 unrelated websites, which, in some cases, justify the use of cryptomining as an alternative to advertisements.[3] We acknowledge that our findings only represent a lower bound of websites that notify their users, as the notifications could also be stored in other formats, for

[3]Examples: "If ads are blocked, a low percentage of your CPU's idle processing power is used to solve complex hashes, as a form of micro-payment for playing the game." (dogeminer2.com) and "This website uses some of your CPU resources to mine cryptocurrency in favor of the website owner. This is a some [sic] sort of donation to thank the website owner for the work done, as well as to reduce the amount of advertising on the website." (crypticrock.com)

example as images, or be part of a website's terms of service. However, locating and parsing these terms is out of scope for this work.

We also found a number of websites that include Coinhive's AuthedMine [107] in addition to drive-by mining. AuthedMine is not part of our threat model, as it requires user opt-in, and as such, we did not include websites using it in our analysis. Still, at least four websites (based on a simple string search) include the `authedmine.min.js` script, while starting to mine right away with a separate mining script that does not require user input: three of these websites include the miners on the *same* page, while the fourth (cnhv.co, a proxy to Coinhive), includes AuthedMine on the landing page, and a non-interactive miner on an internal page.

5.4.3.2 Mining from Internal Pages

We found 744 out of 1,735 websites (42.88%) stealing the visitor's computational power only when she visits one of their internal pages, validating our decision to not only crawl the landing page of a website but also some internal pages. From the manual analysis of these websites, we found that most of them are video streaming websites: the websites start cryptomining when the visitor starts watching a video by clicking the links displayed on the landing page.

5.4.3.3 Evasion Techniques

We have identified three evasion techniques, which are widely used by the drive-by mining services in our dataset.

Code obfuscation. For each of the 28 drive-by mining services in our dataset we manually analyzed some of the corresponding websites, which we identified as mining, but for which we could not find any of the keywords in their crypto-mining code. In this way, we identified 23 (82.14%) of drive-by mining services using one or more of the following obfuscation techniques in at least one of the websites that are using them:

- *Packed code*: The compressed and encoded orchestrator script is decoded using a chain of decoding functions at run time.
- *CharCode*: The orchestrator script is converted to charCode and embedded in the webpage. At run time, it is converted back to a string and executed using JavaScript's `eval()` function.

- *Name obfuscation*: Variable names and functions names are replaced with random strings.
- *Dead code injection*: Random blocks of code, which are never executed, are added to the script to make reverse engineering more difficult.
- *Filename and URL randomization*: The name of the JavaScript file is randomized or the URL it is loaded from is shortened to avoid detection based on pattern matching.

We mainly found these obfuscation techniques applied to the orchestrator code and not to the mining payload. Since the performance of the cryptomining payload is crucial to maximize the profit from browser-based mining, the only obfuscation currently performed on the mining payload is name obfuscation.

Listing 5.2. Anti-debugging trick used by 139 websites.

```
function check() {
    before = new Date().getTime();
    debugger;
    after = new Date().getTime();
    if (after-before > minimalUserResponseInMiliseconds) {
        document.write(" Dont open Developer Tools. ");
        self.location.replace('https:' +
            window.location.href.substring(window.
                    location.protocol.length));
    } else {
        before = null;
        after = null;
        delete before;
        delete after;
  }
  setTimeout(check, 100);
}
```

Obfuscated Stratum communication We only identified the Stratum protocol in plaintext (based on the keywords in Table 5.3) for 1,008 (58.10%) websites. We manually analyzed the WebSocket communication for the remaining 727 (41.90%) websites and found the following: (1) A common strategy to obfuscate the mining pool communication found in 174 (10.03%) websites is to encode the request, either as Hex code, or with salted Base64 encoding (i.e., adding a layer of encryption with the use of a pre-shared passphrase), before transmitting it through the WebSocket. (2) We could not identify any pool communication for the remaining 553 websites, either due to other encodings, or due to slow server connections, i.e., we were not able to observe any pool communication during the time our

Table 5.5. Identified campaigns based on site keys, number of participating websites (#), and estimated profit per month.

Site Key	#	Main Pool	Type	Profit (US$)
"428347349263284"	139	weline.info	Third party (video)	$31,060.80
OT1CI[..]oDWOri06	53	coinhive.com	Torrent portals	$8,343.18
ricewithchicken	32	datasecu.download	Advertisement-based	$1,078.27
jscustomkey2	27	207.246.88.253	Third party (counter12.com)	$86.98
CryptoNoter	27	minercry.pt	Advertisement-based	$20.35
489dj[..]c1X8ADsu	24	datasecu.download	Compromised websites	$142.40
first	23	cloudflane.com	Compromised websites	$120.02
vBaNY[..]8rnZEl00	20	hemnes.win	Third party (video)	$303.14
45CQj[..]23p5SkMN	17	rand.com.ru	Compromised websites	$306.60
Tumblr	14	count.im	Third party	$11.31
ClmAX[..]1uDYdJ8F	12	coinhive.com	Third party (night-skin.com)	$14.36

web crawler spent on a website, which could also be used by malicious websites as a tactic to evade detection by automated tools.

Anti-debugging tricks We found 139 websites (part of a campaign targeting video streaming websites) that employ the following anti-debugging trick (see Listing 5.2): The code periodically checks whether the user is analyzing the code served by the webpage using developer tools. If the developer tools are open in the browser, it stops executing any further code.

5.4.3.4 Private vs. Public Mining Pools

All the drive-by mining websites in our dataset connect to WebSocket proxy servers that listen for connections from their miners, and either process this data themselves (if they also operate their own mining pool), or *unwrap* the traffic and forward it to a public pool. That is, the proxy server could be connecting to a public mining or private mining pool. We identified 159 different WebSocket proxy servers being used by the 1,735 drive-by mining websites and only six of them are sending the public mining pool server address and the cryptocurrency wallet address (used by the pool administrator to reward the miner) associated with the website to the proxy server. These six websites use the following public mining pools: minexmr.com, supportxmr.com, moneroocean.stream, xmrpool.eu, minemonero.pro, and aeon.sumominer.com.

Table 5.6. Identified campaigns based on proxies, number of participating websites (#), and estimated profit per month.

WebSocket Proxy	#	Type	Profit (US$)
advisorstat.space	63	Advertisement-based	$321.71
zenoviaexchange.com	37	Advertisement-based	$1,516.08
stati.bid	20	Compromised websites	$34.94
staticsfs.host	20	Compromised websites	$384.91
webmetric.loan	17	Compromised websites	$181.32
insdrbot.com	7	Third party (video)	$1,689.26
1q2w3.website	5	Third party (video)	$2,012.90
streamplay.to	5	Third party (video)	$239.71
estream.to	4	Third party (video)	$872.72

5.4.3.5 Drive-by Mining Campaigns

To identify drive-by mining campaigns we rely on site keys and WebSocket proxy servers. If a campaign uses a public web mining service, the attacker uses the same site key and proxy server for all websites belonging to this campaign. If the campaign uses an attacker-controlled proxy server, the websites do not need to embed a site key, but the websites still connect to the same proxy. Hence, we use two approaches to find drive-by campaigns: First, we cluster websites that are using the same site key and proxy. We discovered 11 campaigns using this method (see Table 5.5). Second, we cluster the websites only based on the proxy and then manually verified websites from each cluster to see which mining code they are using and how they are including it. We identified nine campaigns using this method (see Table 5.6). In total, we identified 20 drive-by mining campaigns in our dataset. These campaigns include 566 websites (32.62%), for the remaining 1,169 (67.38%) websites we could not identify any connection.

We manually analyzed websites from each campaign to study their modus operandi. Based on this analysis, we classify the campaigns into the following categories based on their infection vector: miners injected through third-party services, miner injected through advertisement networks, and miners injected by compromising vulnerable websites. We also captured proxy servers to the Pirate Bay, which does not ask for users' explicit consent for mining cryptocurrency, but openly discusses this practice on its blog [193]. For each campaign, we estimate the number of visitors per month and their monthly profit (details on how we perform these estimations can be found in Section 5.4.3.7).

Third-party campaigns The biggest campaigns we found target video streaming websites: we identified nine third-party services that provide media players

that are embedded in other websites and which include a cryptomining script in their media player.

Video streaming websites usually present more than one link to a video, also known as *mirrors*. A click on such a link either loads the video in an embedded video player provided by the website, if it is hosting the video directly, or redirects the user to another website. We spotted suspicious requests originating from many such embedded video players which lead us to the discovery of eight third-party campaigns: Hqq.tv, Estream.to, Streamplay.to, Watchers.to, bitvid.sx, Speedvid.net, FlashX.tv and Vidzi.tv are the streaming websites that embed cryptomining scripts through embedded video players. The biggest campaign in our dataset is *Hqq player*, which we found on 139 websites through the proxy weline.info. We estimate that around 2,500 streaming websites are including the embedded video players from these eight services, attracting more than 250 million viewers per month. An independent study from AdGuard also reported similar campaigns in December 2017 [179], however, we could not find any indication that the video streaming websites they identified were still mining at the time of our analysis.

As part of third-party campaigns unrelated to video streaming, we found 14 pages on *Tumblr* under the domain tumblr[.]com mining cryptocurrency. The mining payload was introduced in the main page by the domain fontapis[.]com. We also found 39 websites were infected by using libraries provided by counter12.com and night-skin.com.

Advertisement-based campaigns We found four advertisement-based campaign in our dataset. In this case, attackers publish advertisements that include cryptomining scripts through legitimate advertisement networks. If a user visits the infected website and a malicious advertisement is displayed, the browser starts cryptomining. The *ricewithchicken* campaign was spreading through the AOL advertising platform, which was recently also reported in an independent study by TrendMicro [178]. We also identified three campaigns spreading through the oxcdn.com, zenoviaexchange.com and moradu.com advertisement networks.

Compromised websites. We also identified five campaigns that exploited web application vulnerabilities to inject miner code into the compromised website. For all of these campaigns, the same orchestrator code was embedded at the bottom of the main HTML page (and not loaded from any external libraries) in a similar fashion. Moreover, we could not find any relationship between the websites within the campaigns: they are hosted in different geographic locations and

Table 5.7. Additional cryptomining services we discovered, number of websites (#) using them, and whether they provide a private proxy *and* private mining pool (✓).

Mining Service	#	Main Pool	Private?
CoinPot	43	coinpot.co	
NeroHut	10	gnrdomimplementation.com	✓
Webminerpool	13	metamedia.host	
CoinNebula	6	1q2w3.website	✓
BatMine	6	whysoserius.club	✓
Adless	5	adless.io	✓
Moneromining	5	moneromining.online	✓
Afminer	3	afminer.com	✓
AJcryptominer	4	ajplugins.com	✓
Crypto Webminer	4	anisearch.ru	
Grindcash	2	ulnawoyyzbljc.ru	
Mining.Best	1	mining.best	✓
WebXMR	1	webxmr.com	✓
CortaCoin	1	cortacoin.com	✓
JSminer	1	jsminer.net	✓

registered to different organizations. One of the campaigns was using the public mining pool server minexmr.com.[4] We checked the status of the wallet address on the mining pool's website and found that the wallet address had already been blacklisted for malicious activity.

Torrent portals We found a campaign targeting 53 torrent portals, all but two of which are proxies to the Pirate Bay. We estimate that all together these websites attract 177 million users a month.

5.4.3.6 Drive-by Mining Services

We started our analysis with 13 drive-by mining services. By analyzing the clusters based on WebSocket proxy servers, we discovered 15 more *Coinhive*-like services (see Table 5.7). We classify these services into two categories: the first category only provides a private proxy; however, the client can specify the mining pool address that the proxy server should use as the mining pool. *Grindcash*, *Crypto Webminer*, and *Webminerpool* belong to this category. The second category provides a private proxy and a private mining pool. The remaining services listed in Table 5.7 belong to this category, except for *CoinPot*, which provides a private proxy but uses Coinhive's private mining pool.

[4] site key: *489djE22mdZ3j34vhES98tCzfVn57Wq4fA8JR6uzgHqYCfYE2nmaZxmjepwr3-GQAZd3qc3imFyGPHBy4PBWLb4tc1X8ADsu*

Table 5.8. Hash rate (H/s) on various mobile devices and laptops/desktops using Coinhive's in-browser miner.

Device Type		Hash Rate (H/s)
Mobile Device	Nokia 3	5
	iPhone 5s	5
	iPhone 6	7
	Wiko View 2	8
	Motorola Moto G6	10
	Google Pixel	10
	OnePlus 3	12
	Huawei P20	13
	Huawei Mate 10 Lite	13
	iPhone 6s	13
	iPhone SE	14
	iPhone 7	19
	OnePlus 5	21
	Sony Xperia	24
	Samsung Galaxy S9 Plus	28
	iPhone 8	31
	Mean	**14.56**
Laptop Desktop	Intel Core i3-5010U	16
	Intel Core i7-6700K	65
	Mean	**40.50**

5.4.3.7 Profit Estimation

All of the 1,735 drive-by mining websites in our dataset mine the CryptoNight-based Monero (XMR) cryptocurrency using mining pools. Almost all of them (1,729) use a site key and a WebSocket proxy server to connect to the mining pool; hence, we cannot determine their profit based on their wallet address and public mining pools.

Instead, we estimate the profit per month for all 1,735 drive-by mining websites in the following way: we first collect statistics on monthly visitors, the type of the device the visitor uses (laptop/desktop or mobile) and the time each visitor spends on each website on average from SimilarWeb [202]. We retrieved the average of these statistics for the time period from March 1, 2018 to May 31, 2018. SimilarWeb did not provide data for 30 websites in our dataset, hence, we consider only the remaining 1,705 websites.

We further need to estimate the average computing power, i.e., the hash rate per second (H/s), of each visitor. Since existing hash rate measurements [142] only consider native executables and are thus higher than the hash rates of in-browser miners—Coinhive states their Wasm-based miner achieves 65% of the

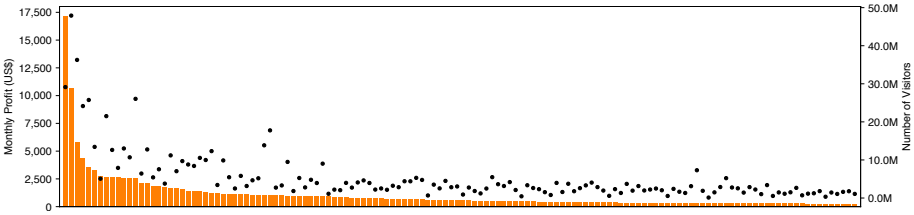

Figure 5.2. Profit estimation and visitor numbers for the 142 drive-by mining websites earning more than US$ 250 a month.

performance of their native miner [140],—we performed our own measurements. Table 5.8 shows our results: According to our experiments, an Intel Core i3 machine (laptop) is capable of at least 16 H/s, while an Intel Core i7 machine (desktop) is capable of at least 65 H/s using the CryptoNight-based in-browser miner from Coinhive. We use their hash rates (40.50 H/s) as the representative hash rate for laptops and desktops. For the mobile devices, we calculated the mean of the hash rates (14.56 H/s) that we observed on 16 different devices. Finally, we use the API provided by MineCryptoNight [183] to calculate the mining reward in US$ for these hash rates and estimate the profit based on SimilarWeb's visitor statistics.

When looking at the profit of individual websites (see Figure 5.2 for the most profitable ones), we estimate that the two most profitable websites are earning US$ 17,166.97 and US$ 10,667.82 a month from 29.13 million visitors (tumangaonline.com, average visit of 18.12 minutes) and 47.91 million visitors (xx1.me, average visit of 7.45 minutes), respectively. However, there is a long tail of websites with very low profits: on average, each of the 1,705 websites earned US$ 110.77 a month, and 900, around half of the websites in our dataset, earned less than US$ 10.

Still, drive-by mining can provide a steady income stream for cybercriminals, especially when considering that many of these websites are part of campaigns. We present the results, aggregated per campaign, in Table 5.5 and Table 5.6: the most profitable campaign, spread over 139 websites, potentially earned US$ 31,060.80 a month. In total, we estimate the profit of all 20 campaigns at US$ 48,741.12. However, almost 70% of websites in our dataset were not part of any campaign, and we estimate the total profit across all websites and campaigns at US$ 188,878.85.

Note that we only estimated the profit based on the websites and campaigns captured by crawling Alexa's Top 1 Million websites, and the same campaigns could make additional profit through websites not part of this list. As a point of reference, concurrent work [73] calculated the total monthly profit of *only*

the Coinhive service and *including legitimate mining*, i.e., user-approved mining through for example AuthedMine, at US$ 254,200.00 (at a market value of US$ 200) in May 2018. We base our estimations on Monero's market values on May 3, 2018 (1 XMR = US$ 253) [183]. The market value of Monero, as for any cryptocurrency, is highly volatile and fluctuated between US$ 488.80 and US$ 45.30 in the last year [141], and, thus profits may vary widely based on the current value of the currency.

5.4.4 Common Drive-by Mining Characteristics

Based on our analysis, we found the following common characteristics among all the identified drive-by mining services: (1) All services use CryptoNight-based cryptomining implementations. (2) All identified websites use a highly-optimized Wasm implementation of the CryptoNight algorithm to execute the mining code in the browser at native speed.[5] Moreover, our manual analysis of the Wasm implementation showed that the only obfuscation performed on Wasm modules is *name obfuscation* (all strings are stripped); any further code obfuscation applied to the Wasm module would degrade the performance (and hence negatively impact the profit). (3) All drive-by mining websites use WebSockets to communicate with the mining pool through a WebSocket proxy server.

We use our findings as the basis for MineSweeper, a detection system for Wasm-based drive-by mining websites, which we describe in the next section.

5.5 Drive-by Mining Detection

Building on the findings of our large-scale analysis, we propose MineSweeper, a novel technique for drive-by mining detection, which relies neither on blacklists nor on heuristics based on CPU usage. In the arms race between defenses trying to detect the miners and miners trying to evade the defenses, one of the few gainful ways forward for the defenders is to target properties of the mining code that would be impossible or very painful for the miners to remove. The more fundamental the properties, the better.

To this end, we characterize the key properties of the hashing algorithms used by miners for specific types of cryptocurrencies. For instance, some hashing algorithms, such as CryptoNight, are *fundamentally* memory-hard. Distilling the measurable properties from these algorithms allows us to detect not just one

[5] We also identified JSEminer in our dataset, which only supports asm.js; however, unlike the other services, the orchestrator code provided by this service always asks for a user's consent. For this reason, we do not classify the 50 websites using JSEminer as drive-by mining websites.

specific variant, but *all* variants, obfuscated or not. The idea is that the only way to bypass the detector is to cripple the algorithm.

MineSweeper takes the URL of a website as the input. It then employs three approaches for the detection of Wasm-based cryptominers, one for miners using mild variations or obfuscations of CryptoNight (Section 5.5.3.1), one for detecting cryptographic functions in a generic way (Section 5.5.3.2), and one for more heavily obfuscated (and performance-crippled) code (Section 5.5.3.3). For the first two approaches, MineSweeper statically analyses the Wasm module used by the website, for the third one it monitors the CPU cache events during the execution of the Wasm module. During the Wasm-based analysis, MineSweeper analyses the module for the core characteristics of specific classes of the algorithm. We use a coarse but effective measure to identify cryptographic functions in general, by measuring the number of cryptographic operations (as reflected by XOR, shift, and rotate operations). We focus on the CryptoNight algorithm and its variants, since it is used by all of the cryptominers we observed so far, but it is trivial to add other algorithms.

5.5.1 Cryptomining Hashing Code

The core component of drive-by miners, i.e., the hashing algorithm, is instantiated within the web workers responsible for solving the cryptographic puzzle. The corresponding Wasm module contains all the corresponding computationally-intensive hashing and cryptographic functions. As mentioned, all of the miners we observed mine CryptoNight-based cryptocurrencies. In this section, we discuss the key properties of this algorithm.

The original CryptoNight algorithm [200] was released in 2013 and represents, at heart, a memory-hard hashing function. The algorithm is explicitly amenable to cryptomining on ordinary CPUs, but inefficient on today's special purpose devices (ASICs). Figure 5.3 summarizes the three main components of the CryptoNight algorithm, which we describe below:

Scratchpad initialization. First, CryptoNight hashes the initial data with the Keccak algorithm (i.e., SHA-3), with the parameters $b = 1600$ and $c = 512$. Bytes 0–31 of the final state serve as an AES-256 key and expand to 10 round keys. Bytes 64–191 are split into 8 blocks of 16 bytes, each of which is encrypted in 10 AES rounds with the expanded keys. The result, a 128-byte block, is used to initialize a scratchpad placed in the L3 cache through several AES rounds of encryption.

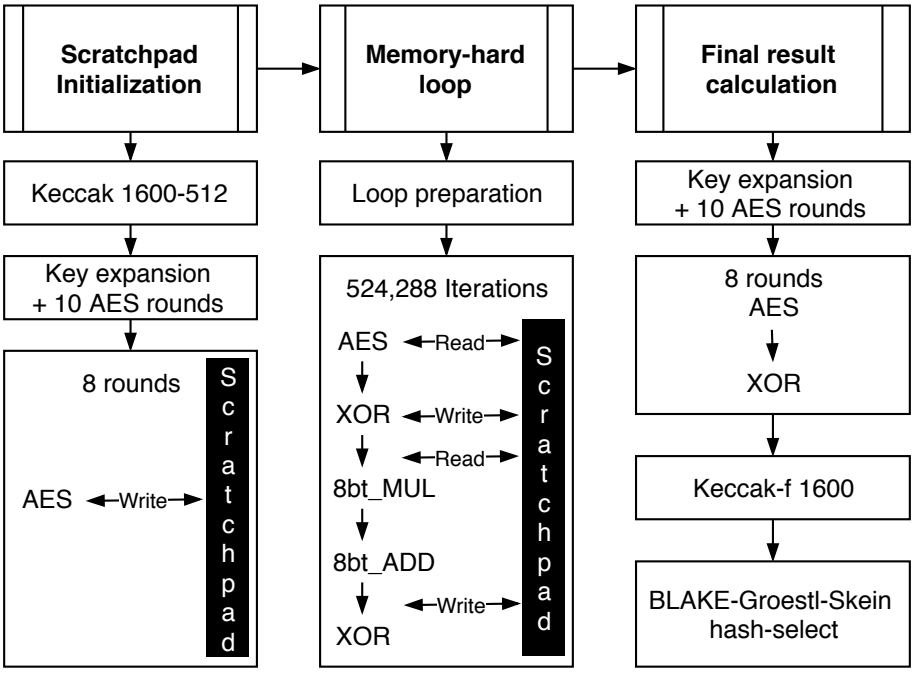

Figure 5.3. Components of the CryptoNight algorithm [200].

Memory-hard loop. Before the main loop, two variables are created from the XORed bytes 0–31 and 32–63 of Keccak's final state. The main loop is repeated 524,288 times and consists of a sequence of cryptographic and read and write operations from and to the scratchpad.

Final result calculation. The last step begins with the expansion of bytes 32–63 from the initial Keccak's final state into an AES-256 key. Bytes 64-191 are used in a sequence of operations that consists of an XOR with 128 scratchpad bytes and an AES encryption with the expanded key. The result is hashed with Keccak-f (which stands for Keccak permutation) with $b = 1600$. The lower 2 bits of the final state are then used to select a final hashing algorithm to be applied from the following: BLAKE-256, Groestl-256, and Skein-256.

There exist two CryptoNight variants made by Sumokoin and AEON, *cryptonight-heavy* and *cryptonight-light*, respectively. The main difference between these variants and the original design is the dimension of the scratchpad: the light version uses a scratchpad size of 1 MB, and the heavy version a scratchpad size of 4 MB.

5.5.2 Wasm Analysis

To prepare a Wasm module for analysis, we use the WebAssembly Binary Toolkit (WABT) debugger [234] to translate it into linear assembly bytecode. We then perform the following static analysis steps on the bytecode:

Function identification We first identify functions and create an internal representation of the code for each function. If the names of the functions are stripped, as part of common name obfuscation, we assign them an identifier with an increasing index.

Cryptographic operation count In the second step, we inspect the identified functions one by one in order to track the appearance of each relevant Wasm operation. More precisely, we first determine the structure of the control flow by identifying the control constructs and instructions. We then look for the presence of operations commonly used in cryptographic operations (XOR, shift and rotate instructions). In many cryptographic algorithms, these operations take place in loops, so we specifically use the knowledge of the control flow to track such operations in loops. However, doing so is not always enough. For instance, at compile time, the Wasm compiler unrolls some of the loops to increase the performance. Since we aim to detect all loops, including the unrolled ones, we identify repeated flexible-length sequences of code containing cryptographic operations and mark them as a loop if a sequence is repeated for more than five times.

5.5.3 Cryptographic Function Detection

Based on our static analysis of the Wasm modules, we now detect the CryptoNight's hashing algorithm. We describe three approaches: one for mild variations or obfuscations of CryptoNight, one for detecting any generic cryptographic function, and one for more heavily obfuscated code.

5.5.3.1 Detection Based on Primitive Identification

The CryptoNight algorithm uses five cryptographic primitives, which are all necessary for correctness: Keccak (Keccak 1600-512 and Keccak-f 1600), AES, BLAKE-256, Groestl-256, and Skein-256. MineSweeper identifies whether any of these primitives are present in the Wasm module by means of fingerprinting. It is important to note that the CryptoNight algorithm and its two variants must use *all* of these primitives in order to compute a correct hash; by detecting the

use of *any* of them, our approach can also detect payload implementation split across modules.

We create fingerprints of the primitives based on their specification, as well as the manual analysis of 13 different mining services (as presented in Table 5.2). The fingerprints essentially consist of the count of cryptographic operations in functions, and more specifically within regular and unrolled loops. We then look for the closest match of a candidate function in the bytecode to each of the primitive fingerprints based on the cryptographic operation count. To this end, we compare every function in the Wasm module one by one with the fingerprints and compute a "similarity score" of how many types of cryptographic instructions that are present in the fingerprint are also present in the function, and a "difference score" of discrepancies between the number of each of those instructions in the function and in the fingerprint. As an example, assume the fingerprint for BLAKE-256 has 80 XOR, 85 left shift, and 32 right shift instructions. Further assume, the function `foo()`, which is an implementation of BLAKE-256, that we want to match against this fingerprint, contains 86 XOR, 85 left shift, and 33 right shift instructions. In this case, the similarity score is 3, as all three types of instructions are present in `foo()`, and the difference score is 2, because `foo()` contains an extra XOR and an extra shift instruction.

Together, these scores tell us how close the function is to the fingerprint. Specifically, for a match we select the functions with the highest similarity score. If two candidates have the same similarity score, we pick the one with the lowest difference score. Based on the similarity score and difference score we calculated for each identified functions, we classify them in three categories: full match, good match, or no match. For a full match, all types of instructions from the fingerprint are also present in the function, and the difference score is 0. For a good match, we require at least 70% of the instruction types in the fingerprint to be contained in the function, and a difference score of less than three times the number of instruction types.

We then calculate the likelihood that the Wasm module contains a CryptoNight hashing function based on the number of primitives that successfully matched (either as a full or a good match). The presence of even one of these primitives can be used as an indicator for detecting potential mining payloads, but we can also set more conservative thresholds, such as flagging a Wasm module as a CryptoNight miner if only two or three out of the five cryptographic primitives are fully matched. We evaluate the number of primitives that we can match across different Wasm-based cryptominer implementations in Section 5.6.

5.5.3.2 Generic Cryptographic Function Detection

In addition to detecting the cryptographic primitives specific to the CryptoNight algorithm, our approach also detects the presence of cryptographic functions in a Wasm module in a more generic way. This is useful for detecting potential new CryptoNight variants, as well as other hashing algorithms. To this end, we count the number of cryptographic operations (XOR, shift, and rotate operations) inside loops in each function of the Wasm module, and flag a function as a cryptographic function if this number exceeds a certain threshold.

5.5.3.3 Detection Based on CPU Cache Events

While not yet an issue in practice, in the future, cybercriminals may well decide to sacrifice profits and highly obfuscate their cryptomining Wasm modules in order to evade detection. In that case, the previous algorithm is not sufficient. Therefore, as a last detection step, MineSweeper also attempts to detect cryptomining code by monitoring CPU cache events during the execution of a Wasm module—a fundamental property for any reasonably efficient hashing algorithm.

In particular, we make use of how CryptoNight explicitly targets mining on ordinary CPUs rather than on ASICs. To achieve this, it relies on random accesses to slow memory and emphasizes latency dependence. For efficient mining, the algorithm requires about 2 MB of fast memory per instance.
This is favorable for ordinary CPUs for the following reasons [200]:

1. Evidently, 2 MB do not fit in the L1 or L2 cache of modern processors. However, they fit in the L3 cache.
2. 1 MB of internal memory is unacceptable for today's ASICs.
3. Moreover, even GPUs do not help. While they may run hundreds of code instances concurrently, they are limited in their memory speeds. Specifically, their GDDR5 memory is much slower than the CPU L3 cache. Additionally, it optimizes pure bandwidth, but not random access speed.

MineSweeper uses this fundamental property of the CryptoNight algorithm to identify it based on its CPU cache usage. Monitoring L1 and L3 cache events using the Linux `perf` [114] tool during the execution of a Wasm module, MineSweeper looks for load and store events caused by random memory accesses. As our experiments in Section 5.6 demonstrate, we can observe a significantly higher load/store frequency during the execution of a cryptominer payload compared to other use cases, including video players and games, and thus detect cryptominers with high probability.

5.5.4 Deployment Considerations

While MineSweeper can be used for the profiling of websites as part of large-scale studies such as ours, we envision it as a tool that notifies users about a potential drive-by mining attack while browsing and gives them the option to opt-out, e.g., by not loading Wasm modules that trigger the detection of cryptographic primitives, or by suspending the execution of the Wasm module as soon as suspicious cache events are detected.

Our defense based on the identification of cryptographic primitives could be easily integrated into browsers, which, so far, mainly rely on blacklists and CPU throttling of background scripts as a last line of defense [138, 139, 159]. As our approach is based on static analysis, browsers could use our techniques to profile Wasm modules as they are loaded and ask the user for permission before executing them. As an alternative and browser-agnostic deployment strategy, SEISMIC [83] instruments Wasm modules to profile their use of cryptographic operations during execution, although this approach comes with considerable run-time overhead.

Integrating our defense based on monitoring cache events, unfortunately, is not so straightforward: access to performance counters requires root privileges and would need to be implemented by the operating system itself.

5.6 Evaluation

In this section, we evaluate the effectiveness of MineSweeper's components based on static analysis of the Wasm code and CPU cache event monitoring for the detection of the cryptomining code currently used by drive-by mining websites in the wild. We further compare MineSweeper to a state-of-the-art detection approach based on blacklisting. Finally, we discuss the penalty in terms of performance, and thus profits, evasion attempts against MineSweeper would incur.

Dataset. To test our Wasm-based analysis we crawled Alexa's Top 1 Million websites a second time over the period of one week in the beginning of April 2018 with the sole purpose of collecting Wasm-based mining payloads. This time we configured the crawler to visit only the landing page of each website for a period of four seconds. The crawl successfully captured 748 Wasm modules served by 776 websites. For the remaining 28 modules, the crawler was killed before it was able to dump the Wasm module completely.

Table 5.9. Results of our cryptographic primitive identification. MineSweeper detected at least two of CryptoNight's primitives in *all* mining samples with no false positives.

Detected Primitives	Number of Wasm Samples	Number of Cryptominers	Missing Primitives
5	30	30	-
4	3	3	AES
3	-	-	-
2	3	3	Skein, Keccak, AES
1	-	-	-
0	4	0	All

Evaluation of cryptographic primitive identification Even though we were able to collect 748 valid Wasm modules, only 40 among them are, in fact, unique. This is because many websites use the same cryptomining services. We also found that some of these cryptomining services are providing different versions of their mining payload. Table 5.9 shows our results for the CryptoNight function detection on these 40 unique Wasm samples. We were able to identify all five cryptographic primitives of CryptoNight in 30 samples, four primitives in three samples and two primitives in another three samples. In these last three samples, we could only detect the Groestl and BLAKE primitives, which suggests that these are the most reliable primitives for this detection. As part of an in-depth analysis, we identified these samples as being part of the mining services BatMine and Webminerpool (two of the samples are a different version of the latter), which were not part of our dataset of mining services that we used for the fingerprint generation, but rather services we discovered during our large-scale analysis.

However, our approach did not produce any false positives and the four samples in which MineSweeper did not detect any cryptographic primitive were, in fact, benign: an online magazine reader, a videoplayer, a node library to represent a 64-bit two's-complement integer value, and a library for hyphenation. Furthermore, the generic cryptographic function detection successfully flagged all 36 mining samples as positives and all four benign cases as negatives.

Evaluation of CPU cache event monitoring For this evaluation we used `perf` to capture L1 and L3 cache events when executing various types of web applications. We conducted all experiments on an Intel Core i7-930 machine running Ubuntu 16.04 (baseline). We captured the number of L1 data cache loads, L1 data cache stores, L3 cache stores, and L3 cache loads within 10 seconds when visiting four categories of web applications: cryptominers (Coinhive and NFWebMiner,

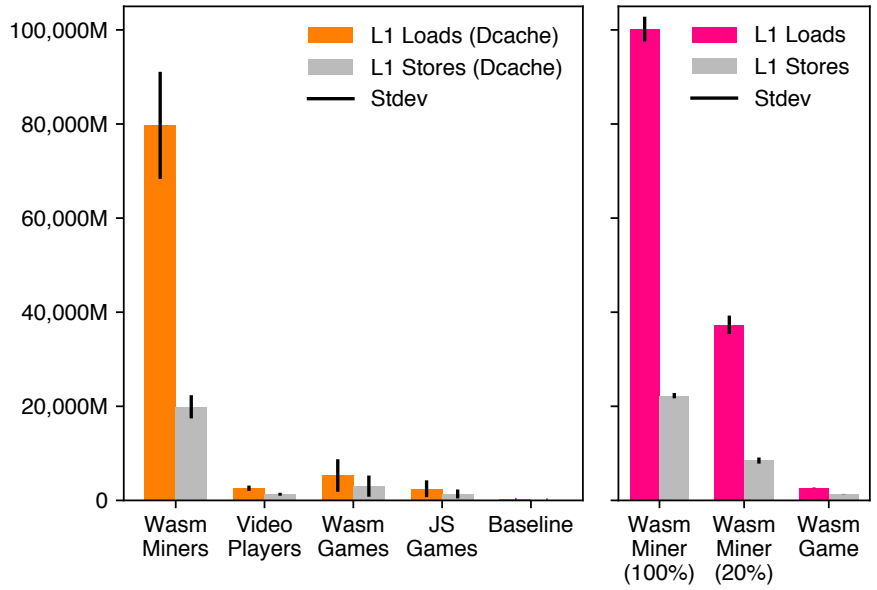

Figure 5.4. Performance counter measurements for the L1 data cache for miners and other web applications on two different machines (# of operations per 10 seconds, M=million).

both with 100% CPU usage), video players, Wasm-based games, and JavaScript (JS) games. We visited seven websites from each category and calculated the mean and standard deviation (stdev) of all the measurements for each category.

As Figure 5.4 (left) and Figure 5.5 (left) show, that L1 and L3 cache events are very high for the web applications that are mining cryptocurrency, but considerably lower for the other types of web applications. Compared to the second most cache-intensive applications, Wasm-based games, the Wasm-based miners perform on average 15.05x as many L1 data cache loads, and 6.55x as many L1 data cache stores. The difference for the L3 cache is less severe, but still noticeable: here on average the miners perform 5.50x and 2.93x as many cache loads and stores, respectively, compared to the games.

We performed a second round of experiments on a different machine (Intel Core i7-6700K), which has a slightly different cache architecture, to verify the reliability of the CPU cache events. We also used these experiments to investigate the effect of CPU throttling on the number of cache events. Coinhive's Wasm-based miner allows throttling in increments of 10% intervals. We configured it to use 100% CPU and 20% CPU and compared it against a Wasm-based game. We executed the experiments 20 times and calculated the mean and standard deviation (stdev). As Figure 5.4 (right) and Figure 5.5 (right) show, on this machine L3 cache store events cannot be used for the detection of miners: we observed only

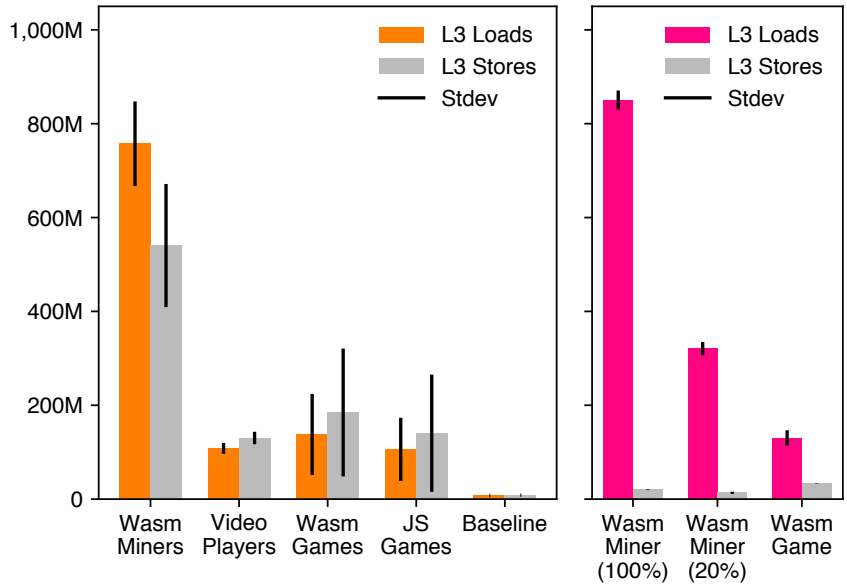

Figure 5.5. Performance counter measurements for the L3 cache for miners and other web applications on two different machines (# of operations per 10 seconds, M=million).

a low number of L3 cache stores overall, and on average more stores for the game than for the miners. However, L3 cache loads, as well as L1 data cache loads and stores are a reliable indicator for mining. When using only 20% of the CPU, we still observed 37.25%, 38.05%, and 37.71% of the average number of events compared to 100% CPU usage for L1 data cache loads, L1 data cache stores, and L3 cache loads, respectively. Compared to the game, the miner performed $13.96x$ and $6.29x$ as many L1 data cache loads and stores, and $2.46x$ as many L3 cache loads even when utilizing only 20% of the CPU.

Comparison to blacklisting approaches To compare our approach against existing blacklisting-based defenses we evaluate MineSweeper against Dr. Mine [231]. Dr. Mine uses CoinBlockerLists [227] as the basis to detect mining websites. For the comparison we visited the 1,735 websites that were mining during our first crawl for the large-scale analysis in mid-March 2018 (see Section 5.4) with both tools. We made sure to use updated CoinBlockerLists and executed Dr. Mine and MineSweeper in parallel to maximize the chance that the same drive-by mining websites would be active. During this evaluation, on May 9, 2018, Dr. Mine could only find 272 websites, while MineSweeper found 785 websites that were still actively mining cryptocurrency. Furthermore, all the 272 websites identified by Dr. Mine are also identified by MineSweeper.

Impact of evasion techniques In order to evade our identification of cryptographic primitives, attackers could heavily obfuscate their code, or implement the CryptoNight functions completely in asm.js or JavaScript. In both cases, MineSweeper would still be able to detect the cryptomining based on the CPU cache event monitoring. To evade this type of defense, and since we are only monitoring unusually high cache load and stores that are typical for cryptomining payloads, attackers would need to slow down their hash rate, for example by interleaving their code with additional computations that have no effect on the monitored performance counters.

Table 5.10. Decrease in the hash rate (H/s), and thus profit, *compared to the best-case scenario* (∗) using Wasm with 100% CPU utilization if asm.js is being used and the CPU is throttled on an Intel Core i7-6700K and an Intel Core i3-5010U machine.

	Baseline		100% CPU			75% CPU				
	H/s Wasm	Profit US$	H/s asm.js	H/s Loss	Profit US$	H/s Wasm	Profit US$	H/s asm.js	H/s Loss	Profit US$
i7	*65∗*	100.00	39	40.00%	60.00	48.75	75.00	29.25	55.00%	45.00
i3	*16∗*	24.62	9	43.75%	13.85	12	18.46	6.75	57.81%	10.38
	50% CPU					25% CPU				
	H/s Wasm	Profit US$	H/s asm.js	H/s Loss	Profit US$	H/s Wasm	Profit US$	H/s asm.js	H/s Loss	Profit US$
i7	32.5	50.00	19.5	70.00%	30.00	16.25	25.00	9.75	85.00%	15.00
i3	8	12.31	4.5	71.88%	6.92	4	6.15	2.25	85.94%	3.46

In the following, we discuss the performance hit (and thus loss of profit) that alternative implementations of the mining code in asm.js, and an intentional sacrifice of the hash rate, in this case by throttling the CPU usage, would incur. Table 5.10 show our estimation for the potential performance and profit losses on a high-end (Intel Core i7-6700K) and a low-end (Intel Core i3-5010U) machine. As an illustrative example, we assume that in the best case an attacker is able to make a profit of US$ 100 with the maximum hash rate of 65 H/s on the i7 machine. Just falling back to asm.js would cost an attacker 40.00%–43.75% of her profits (with a CPU usage of 100%). Moreover, throttling the CPU speed to 25% on top of falling back to asm.js would cost her 85.00%–85.94% of her profits, leaving her with only US$ 15.00 on a high-end and US$ 3.46 on a low-end machine. In more concrete numbers from our large-scale analysis of drive-by mining campaigns in the wild (see Section 5.4.3), the most profitable campaign, which is potentially earning US$ 31,060.80 a month (see Table 5.5), would only earn US$ 4,367.15 a month.

5.7 Limitations and Future Work

Our large-scale analysis of drive-by mining in the wild likely missed active cryptomining websites due to limitations of our crawler. We only spend four seconds on each webpage; hence we could have missed websites that wait for a certain amount of time before serving the mining payload. Similarly, we are not able to capture the mining pool communication for websites that implement mining delays, and in some cases due to slow server connections, which exceed the timeout of our crawler. Moreover, we only visit each webpage once, but, some cryptomining payloads, especially the ones that spread through advertisement networks, are not served on every visit. Our crawler also did not capture the cases in which cryptominers are loaded as part of "pop-under" windows. Furthermore, the crawler visited each website with the User Agent String of the Chrome browser on a standard desktop PC. We leave the study of campaigns specifically targeting other devices, such as Android phones, for future work. Another avenue for future work is studying the longevity of the identified campaigns. We based our profit estimations on the assumption that they stayed active for at least a month, but they might have been disrupted earlier.

Our defense based on static analysis is similarly prone to obfuscation as any related static analysis approach. However, even if attackers decide to sacrifice performance (and profits) for evading our defense through obfuscation of the cryptomining payload, we would still be able to detect the mining based on monitoring the CPU cache. Trying to evade this detection technique by adding additional computations would severely degrade the mining performance—to a point that it is not profitable anymore.

Furthermore, currently all drive-by mining services use Wasm-based cryptomining code, and hence, we implemented our defense only for this type of payload. Nevertheless, we could implement our approach also for the analysis of asm.js in future work. Finally, our defense is tailored for detecting cryptocurrencies using the CryptoNight algorithm, as these are currently the only cryptocurrencies that can profitably be mined using regular CPUs [183]. Even though our generic cryptographic function detection did not produce any false positives in our evaluation, we still can imagine many benign Wasm modules using cryptographic functions for other purposes. However, Wasm is not widely adopted yet for other use cases besides drive-by mining and we therefore could not evaluate our approach on a larger dataset of benign applications.

5.8 Related Work

Related work has extensively studied how and why attackers compromise websites through the exploitation of software vulnerabilities [7, 15], misconfigurations [19], inclusion of third-party scripts [61], and advertisements [90]. Traditionally, the attackers' goals ranged from website defacements [8, 51], over enlisting the website's visitors into distributed denial-of-service (DDoS) attacks [64], to the installation of exploit kits for drive-by download attacks [29, 68, 69], which infect visitors with malicious executables. In comparison, the abuse of the visitors' resources for cryptomining is a relatively new trend.

Previous work on cryptomining focused on botnets that were used to mine Bitcoin during the year 2011–2013 [36]. The authors found that while mining is less profitable than other malicious activities, such as spamming or click fraud, it is attractive as a secondary monetizing scheme, as it does not interfere with other revenue-generating activities. In contrast, we focused our analysis on drive-by mining attacks, which serve the cryptomining payload as part of infected websites, and not malicious executables. The first other study in this direction was recently performed by Eskandari et al. [23]. However, they based their analysis solely on looking for the `coinhive.min.js` script within the body of each website indexed by Zmap and PublicWWW [185]. In this way, they were only able to identify the Coinhive service. Furthermore, contrary to the observations made in their study, we found that attackers have found valuable targets, such as online video streaming, to maximize the time users spend online, and consequently the revenue earned from drive-by mining. Concurrently to our work, Papadopoulos et al. [102] compared the potential profits from drive-by mining to advertisement revenue by checking websites indexed by PublicWWW against blacklists from popular browser extensions. They concluded that mining is only more profitable than advertisements when users stay on a website for longer periods of time. In another concurrent work, Rüth et al. [73] studied the prevalence of drive-by miners in Alexa's Top 1 Million websites based on JavaScript code patterns from a blacklist, as well as based on signatures generated from SHA-255 hashes of the Wasm code's functions. They further calculated the Coinhive's overall monthly profit, which includes legitimate mining as well. In contrast, we focus on the profit of individual campaigns that perform mining without their user's explicit consent. Furthermore, with MineSweeper, we also present a defense against drive-by mining that could replace current blacklisting-based approaches.

The first part of our defense, which is based on the identification of cryp-

tographic primitives is inspired by related work on identifying cryptographic functionality in desktop malware, which frequently uses encryption to evade detection and secure the communication with its command-and-control servers. Gröbert et al. [30] attempt to identify cryptographic code and extract keys based on dynamic analysis. Aligot [48] identifies cryptographic functions based on their input-output (I/O) characteristics. Most recently, CryptoHunt [88] proposed to use symbolic execution to find cryptographic functions in obfuscated binaries. In contrast to the heavy use of obfuscation in binary malware, obfuscation of the cryptographic functions in drive-by miners is much less favorable for attackers. Should they start to sacrifice profits in favor of evading defenses in the future, we can explore the aforementioned more sophisticated detection techniques for detecting cryptomining code. For the time being, relatively simple fingerprints of instructions that are commonly used by cryptographic operations are enough to reliably detect cryptomining payloads, as also observed by Wang et al. [83] in concurrent work. Their approach, SEISMIC, generates signatures based on counting the execution of five arithmetic instructions that are commonly used by Wasm-based miners. In contrast to profiling whole Wasm modules, we detect the individual cryptographic primitives of the cryptominers' hashing algorithms, and also supplement our approach by looking for suspicious memory access patterns.

This second part of our defense, which is based on monitoring CPU cache events, is related to CloudRadar [92], which uses performance counters to detect the execution of cryptographic applications and to defend against cache-based side-channel attacks in the cloud. Finally, the most closely related work in this regard is MineGuard [77], also a hypervisor tool, which uses signatures bases on performance counters to detect both CPU- and GPU-based mining executables on cloud platforms. Similar to our work, the authors argue that the evasion of this type of detection would make mining unprofitable—or at least less of a nuisance to cloud operators and users by consuming fewer resources.

5.9 Conclusion

In this chapter, we examined the phenomenon of drive-by mining. The rise of mineable alternative coins (altcoins) and the performance boost provided to in-browser scripting code by WebAssembly, have made such activities quite profitable to cybercriminals: rather than being a one-time heist, this type of attack provides continuous income to an attacker.

Detecting miners by means of blacklists, string patterns, or CPU utilization

alone is an ineffective strategy, because of both false positives and false negatives. Already, drive-by mining solutions are actively using obfuscation to evade detection. Instead of the current inadequate measures, we proposed MineSweeper, a new detection technique tailored to the algorithms that are fundamental to the drive-by mining operations—the cryptographic computations required to produce valid hashes for transactions.

The current security model (CIA triad) and security principles are inadequate to offer protection from this new class of cyber threat. This is the root cause of this cyber threat. Hence, we propose to update current design principles by adding *least required resource* to it to prevent this class of cyber threat. The computing systems have to be designed not only to provide Confidentiality, Integrity and Availability but also to protect the user's computational resources. For instance, the applications like Browser should be designed by incorporating this design principle to prevent similar cyber threats that focus on *resource stealing/-exploitation*. These applications should at least give users an option to control the amount of resources they are allowed to use.

6 Conclusion

The goal of this work is to study and advance computer defenses that primarily focus on preventing exploitation based on design flaws. First, we have focused on the new cyber threats that emerged from design flaws at both the software and hardware level. Then, we have studied whether the current set of design principles is comprehensive enough to prevent today's cyber threats, focusing especially on cryptojacking.

In Chapter 1, we formulated four research questions around these key goals. We now recapitulate the conclusions we have reached for each of the research questions. Finally, we provide a discussion of the limitations of our work and the possibilities for future research.

Results

In this subsection, we briefly recall each of the research questions that we formulated in Chapter 1 and summarize our main conclusions. Our main conclusions for each of the research questions are as follows.

Question (1): *Given that exploitable design flaws exist in hardware, can we also discover them in software that deals with highly sensitive operations such as financial transactions?*

In Chapter 2 of this thesis, we discussed the *remote-install* feature, and various *synchronization* features introduced by vendors to enhance usability. These usability features are blurring boundaries between platforms, thus violating the design principle called *compartmentalize* (see Table 1.2). The basic idea behind the *compartmentalization* is to segment a system into multiple compartments or units that are protected independently so that a vulnerability in one unit will not jeopardize the security of the other units. Note that these usability features also violate the design principle called *separation of privileges* (see Table 1.1).

In the chapter, we showed that an attacker could bypass mobile-based 2FA (used by a wide range of financial web services) by exploiting these usability features. Despite our efforts, this design flaw still exists because, unfortunately, sometimes, when given a choice, vendors choose usability over security. We explored what can be done about this when answering our second research question.

Question (2): *Given that it is harder to fix such a design/logical flaw when compared to patching a typical software bug, can we still mitigate the cyber threat stemming from the design flaw that we identified as part of our first research question under the assumption that the flaw itself cannot be fixed for practical reasons? What will be the cost of such a solution?*

We address this question in Chapter 3 by building a software-based solution for the Bandroid attack. In this chapter, we showed that it is possible to mitigate a design-flaw-based attack at the software level without sacrificing the usability feature. However, note that an attacker can still spread the damage to other devices of the victim from a single compromised device exploiting the *remote-install* feature. To prevent this, we recommend the vendors to implement an extra authentication mechanism on the remote device to verify the *remote-install* request.

Question (3): *Given that it is often more complex to patch a design/logical flaw in a hardware component, can we still mitigate the cyber threat originated from the Rowhammer bug using a software-based solution? What will be the cost of such a solution?*

We address this question in Chapter 4 by building a novel and comprehensive software-based protection against the Rowhammer attack which exploits a design flaw in the memory hardware component (commonly called DRAM). The root cause of the Rowhammer bug is the vendors' design choice to optimize the cost-per-bit by cramming bits very close together. Since the bit flips/memory errors induced by Rowhammer happen without any indication, the system fails to detect it. Clearly, this design of memory chips violates the design principle called *Fail securely* (see Table 1.2). The basic idea behind the *Fail securely* is that when a system fails, it should do so securely; the confidentiality and integrity of a system should remain even though availability has been lost. The attackers must not be permitted to gain access rights to privileged objects that are normally inaccessible during a failure.

In Chapter 4, we showed that it is possible to provide comprehensive software-based protection against the Rowhammer attacks. Our evaluation shows ZebRAM to be a strong defense able to use all available memory at a performance cost that is a function of the active working set in DRAM.

Question (4): *Given that cyber attacks have evolved over time to become more stealthy and complex, is the current set of design principles comprehensive enough to prevent today's cyber threats?*

In Chapter 5, we answered this question by researching a new class of cyber threat called *cryptojacking*. First, we showed that the current measures like blacklisting and monitoring the CPU are inadequate to detect this cyber attack. Then we proposed MineSweeper, a new detection technique tailored to the algorithms that are fundamental to the drive-by mining operations — the cryptographic computations required to produce valid hashes for transactions. Finally, we concluded that our current security model (CIA triad) and security principles are inadequate to offer protection from this new class of cyber threat. We propose to update current design principles by adding *least required resource* to it to prevent this class of cyber threat. The computing systems have to be designed not only to provide Confidentiality, Integrity and Availability, but also to protect the user's computational resources. For instance, applications like the browser should be designed by incorporating this design principle to prevent similar cyber threats that focus on *resource stealing/exploitation*.

Future Directions

In this thesis, our primary focus was on mitigating design flaws or logical flaws — though identifying such flaws is sometimes rather difficult. There has been a lot of research on building fuzzers to find software bugs, but not much on the automation of finding logical flaws. This remains a promising direction for future research.

Chapter 3 shows that it is possible to build software-based solutions for the Bandroid attack without sacrificing the problematic feature. However, SecurePay does not eradicate all the cyber threats introduced by synchronization features. An attacker can still spread the damage to other devices of the victim from a single compromised device exploiting the *remote-install* feature. To protect from such attacks, we need to enforce access control between different trust domains/platforms. For instance, an extra authentication mechanism on the remote device to verify the *remote-install* request. More research is required to evaluate

whether proper access control can be enforced between different platforms (trust domains). This remains a promising direction for future research.

Even though the ZebRAM defense sounds promising compared to previous research, the worst-case overhead of ZebRAM (measured in the dissertation with a uniformly random and hence pessimistic synthetic distribution) is high. The root cause of the Rowhammer bug is the vendors' design choice to optimize the cost-per-bit by cramming bits so close together that an access can flip bits in adjacent cells; no reliable solution that can undo such effects will be free of cost. Nevertheless, there is a need for more research on the ZebRAM design that focuses on its performance improvement.

The ZebRAM defense requires correct physical addresses to DRAM addresses mapping to partition the memory in zebra pattern. Currently, we need to reverse engineer this mapping for different architectures because hardware vendors do not publish this mapping. We recommend every hardware vendor to expose this mapping to the software so that ZebRAM can easily support any underlying architecture. We believe that the only way to provide complete protection from various Rowhammer attacks is by building a system-level defense like ZebRAM where software and hardware collaborate to detect and fix DRAM errors. For instance, a memory controller can be set to use one bit of the physical address to differentiate between physical addresses that fall into odd and even DRAM row. With this information available to software, a system-level defense can utilize unsafe memory more efficiently and safely. Furthermore, currently, software only takes action when the system encounters uncorrectable errors. It remains a promising direction for future research to figure out whether hardware and software can collaborate better to detect ongoing Rowhammer attacks. Hence, enhancing the performance of ZebRAM by improving the software and hardware collaboration remains a promising direction for future research.

Chapter 5 provides a novel detection technique that is based on the intrinsic characteristics of the cryptomining code. Currently, all drive-by mining services use *Wasm*-based cryptomining code, and hence, we implemented our defense only for this type of payload. In the future, the attacker could use *asm.js* for drive-by mining attacks. Hence, implementing our detection approach for the analysis of *asm.js* remains a promising direction for future research. Note that this does not prevent such attacks. For prevention, an interesting direction for future work is to re-design applications like the browser by considering the *least required resource* design principle. These applications should at least give users an option to control the amount of resources they are allowed to use.

Cyber attacks are evolving to become more stealthy and complex. There is

a need for periodic research to update design principles regularly to keep the emerging cyber threats in control. We hope that this work shall be a motivation for future research.

References

The references in this thesis are organized in different sections: conference proceedings, (journal) articles, books, technical reports and documentation, online articles, talks, and source code. All online references were archived and are available in the *Internet Archive Wayback Machine*.[1] They were all accessed on April 1, 2020, unless stated otherwise.

Conference Proceedings

[1] B. Aichinger. **DDR memory errors caused by row hammer**. In *Proceedings of the* 19^{th} *IEEE High Performance Extreme Computing Conference (HPEC)*. Sep. 2015.

[2] F. Aloul, S. Zahidi, and W. E. Hajj. **Two Factor Authentication Using Mobile Phones**. In *Proceedings on the* 7^{th} *ACS/IEEE International Conference on Computer Systems and Applications (AICCA)*. May 2009.

[3] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. **ANVIL: Software-based protection against next-generation rowhammer attacks**. In *Proceedings of the* 21^{st} *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Apr. 2016.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Martin, and W. Shen. **Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world**. In *Proceedings of the* 21^{st} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2014.

[5] S. Bhattacharya and D. Mukhopadhyay. **Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis**. In *Proceedings of the* 18^{th} *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Aug. 2016.

[6] A. Blom, G. de Koning Gans, E. Poll, J. de Ruiter, and R. Verdult. **Designed to fail: A usb-connected reader for online banking**. In *Proceedings of the* 17^{th} *Nordic conference on Secure IT Systems (NordSec)*. Nov. 2012.

[7] K. Borgolte, C. Kruegel, and G. Vigna. **Delta: Automatic identification of unknown web-based infection campaigns**. In *Proceedings of the* 20^{th} *ACM Conference on Computer and Communications Security (CCS)*. Nov. 2013.

[1] https://web.archive.org

[8] K. Borgolte, C. Kruegel, and G. Vigna. **Meerkat: Detecting website defacements through image-based object recognition**. In *Proceedings of the* 24^{th} *USENIX Conference on Security Symposium (USENIX SEC)*. Aug. 2015.

[9] E. Bosman and H. Bos. **Framing signals—a return to portable shellcode**. In *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2014.

[10] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. **Dedup est machina: Memory deduplication as an advanced exploitation vector**. In *Proceedings of the* 37^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2015.

[11] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi. **CAn't Touch This: Software-only mitigation against rowhammer attacks targeting kernel memory**. In *Proceedings of the* 26^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[12] A. Buescher, F. Leder, and T. Siebert. **Banksafe information stealer detection inside the web browser**. In *Proceedings on the* 14^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*. Sep. 2011.

[13] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. **Measuring pay-per-install: The commoditization of malware distribution**. In *Proceedings of the* 20^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2011.

[14] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch. **Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques**. In *Proceedings of the* 23^{rd} *International on High-Performance Computer Architecture (HPCA)*. Feb. 2017.

[15] D. Canali and D. Balzarotti. **Behind the scenes of online attacks: An analysis of exploitation behaviors on the web**. In *Proceedings of the* 20^{th} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2013.

[16] J. M. Carrascosa, J. Mikians, R. Cuevas, V. Erramilli, and N. Laoutaris. **I always feel like somebody's watching me: Measuring online behavioural advertising**. In *Proceedings of the* 13^{th} *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. Dec. 2015.

[17] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos. **Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks**. In *Proceedings of the* 40^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2019.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. **Benchmarking cloud serving systems with ycsb**. In *Proceedings of the* 1^{st} *ACM Symposium on Cloud Computing (SOCC)*. Jun. 2010.

[19] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig. **Investigating operators' perspective on security misconfigurations**. In *Proceedings of the* 25^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2018.

[20] A. Dmitrienko, C. Liebchen, C. Rossow, and A. Sadeghi. **On the (in)security of mobile two-factor authentication**. In *Proceedings of the* 18^{th} *International Conference on Financial Cryptography and Data Security (FC)*. Mar. 2014.

[21] D. Dolev and A. C.-C. Yao. **On the security of public key protocols**. In *Proceedings of the* 22^{nd} *Symposium on Foundations of Computer Science (SFCS)*. Oct. 1981.

[22] J.-E. Ekberg, K. Kostiainen, and N. Asokan. **The untapped potential of trusted execution environments on mobile devices**. In *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2014.

[23] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark. **A First Look at Browser-based Cryptojacking**. In *Proceedings of the* 3^{rd} *European Symposium on Security and Privacy Workshops (EuroS&PW)*. Oct. 2010.

[24] A. Feder, N. Gandal, J. Hamrick, T. Moore, and M. Vasek. **The rise and fall of cryptocurrencies**. In *Proceedings of the* 17^{th} *Annual Workshop on the Economics of Information Security (WEIS)*. Jun. 2018.

[25] R. Fedler, M. Kulicke, and J. Schutte. **An antivirus API for Android malware recognition**. In *Proceedings of the* 8^{th} *International Conference on Malicious and Unwanted Software (MALWARE)*. Oct. 2013.

[26] N. Feske and C. Helmuth. **A nitpicker's guide to a minimal-complexity secure gui**. In *Proceedings of the* 21^{nd} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2005.

[27] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. **Grand pwning unit: Accelerating microarchitectural attacks with the gpu**. In *Proceedings of the* 39^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2018.

[28] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi. **TRRespass: Exploiting the Many Sides of Target Row Refresh**. In *Proceedings of the* 41^{st} *IEEE Symposium on Security and Privacy (S&P)*. May 2020.

[29] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker. **Manufacturing compromise: The emergence of exploit-as-a-service**. In *Proceedings of the* 19^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2012.

[30] F. Gröbert, C. Willems, and T. Holz. **Automated identification of cryptographic primitives in binary programs**. In *Proceedings on the* 14^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*. Sep. 2011.

[31] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. **Another flip in the wall of rowhammer defenses**. In *Proceedings of the* 39^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2018.

[32] D. Gruss, C. Maurice, Stefan, and Mangard. **Rowhammer.js: A remote software-induced fault attack in javascript**. In *Proceedings of the* 13^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jul. 2016.

[33] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. **Bringing the web up to speed with webassembly**. In *Proceedings of the* 38^{th} *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Jun. 2017.

[34] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. **Lest we remember: Cold boot attacks on encryption keys**. In *Proceedings of the* 17^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2008.

[35] C. Herley and D. Florencio. **How to login from an internet café without worrying about keyloggers**. In *Proceedings of the* 2^{nd} *Symposium on Usable Privacy and Security (SOUPS)*. Jul. 2006.

[36] D. Y. Huang, H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. Mccoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko. **Botcoin: Monetizing stolen cycles**. In

Proceedings of the 21^{st} Annual Network and Distributed System Security Symposium (NDSS). Feb. 2014.

[37] Y. Jang, J. Lee, S. Lee, and T. Kim. **Sgx-bomb: Locking down the processor via rowhammer attack**. In *Proceedings of the 2^{nd} Workshop on System Software for Trusted Execution (SysTEX)*. Oct. 2017.

[38] A. Kellner, M. Horlboge, K. Rieck, and C. Wressnegger. **False sense of security: A study on the effectivity of jailbreak detection in banking apps**. In *Proceedings of the 4^{th} IEEE European Symposium on Security and Privacy (EuroS&P)*. Jun. 2019.

[39] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. **Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors**. In *Proceedings of the 41^{st} International Symposium on Computer Architecture (ISCA)*. Jun. 2014.

[40] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. **SeL4: Formal Verification of an OS Kernel**. In *Proceedings of the 22^{nd} ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2009.

[41] B. Kollenda, E. Göktaş, T. Blazytko, P. Koppe, R. Gawlik, R. K. Konoth, C. Giuffrida, H. Bos, and T. Holz. **Towards automated discovery of crash-resistant primitives in binary executables**. In *Proceedings of the 47^{th} Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Jun. 2017.

[42] R. K. Konoth, B. Fischer, W. Fokkink, E. Athanasopoulos, K. Razavi, and H. Bos. **SecurePay: Strengthening two-factor authentication for arbitrary transactions**. In *Proceedings of the 5^{th} IEEE European Symposium on Security and Privacy (EuroS&P)*. Sep. 2020.

[43] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi. **ZebRAM: Comprehensive and compatible software protection against rowhammer attacks**. In *Proceedings of the 13^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2018.

[44] R. K. Konoth, V. van der Veen, and H. Bos. **How anywhere computing just killed your phone-based two-factor authentication**. In *Proceedings of the 20^{th} International Conference on Financial Cryptography and Data Security (FC)*. Feb. 2016.

[45] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna. **Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense**. In *Proceedings of the 25^{th} ACM Conference on Computer and Communications Security (CCS)*. Oct. 2018.

[46] R. Krishnan and R. Kumar. **Securing user input as a defense against MitB**. In *Proceedings of the International Conference on Interdisciplinary Advances in Applied Computing (ICONIAAC)*. Oct. 2014.

[47] A. Kurmus, N. Ioannou, N. Papandreou, and T. Parnell. **From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks**. In *Proceedings of the 11^{th} USENIX Workshop on Offensive Technologies (WOOT)*. Aug. 2017.

[48] P. Lestringant, F. Guihéry, and P.-A. Fouque. **Aligot: Cryptographic function identification in obfuscated binary programs**. In *Proceedings of the 10^{th} ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Apr. 2015.

[49] W. Li, H. Li, H. Chen, and Y. Xia. **AdAttester: Secure online mobile advertisement attestation using trustzone**. In *Proceedings of the* 13^{th} *Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. May 2015.

[50] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. **VBtton: Practical attestation of user-driven operations in mobile apps**. In *Proceedings of the* 16^{th} *International Conference on Mobile Systems, Applications, and Services (MobiSys)*. Jun. 2018.

[51] F. Maggi, M. Balduzzi, R. Flores, L. Gu, and V. Ciancaglini. **Investigating web defacement campaigns at large**. In *Proceedings of the* 13^{th} *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. May 2018.

[52] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun. **Smartphones as practical and secure location verification tokens for payments**. In *Proceedings of the* 21^{st} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2014.

[53] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Čapkun. **Evaluation of Personalized Security Indicators as an Anti-Phishing Mechanism for Smartphone Applications**. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. May 2016.

[54] C. Marforio, R. J. Masti, C. Soriente, K. Kostiainen, and S. Čapkun. **Hardened Setup of Personalized Security Indicators to Counter Phishing Attacks in Mobile Banking**. In *Proceedings of the* 6^{th} *Workshop on Security and Privacy in Smartphones and Mobile Devices*. Oct. 2016.

[55] A. M. McDonald and L. F. Cranor. **Americans' Attitudes About Internet Behavioral Advertising Practices**. In *Proceedings of the* 9^{th} *ACM Workshop on Privacy in the Electronic Society (WPES)*. Oct. 2010.

[56] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. **The tamarin prover for the symbolic analysis of security protocols**. In *Proceedings of the* 25^{th} *International Conference on Computer Aided Verification (CAV)*. Jul. 2013.

[57] T. Müller and M. Spreitzenbarth. **FROST: Forensic recovery of scrambled telephones**. In *Proceedings of the* 11^{th} *International Conference on Applied Cryptography and Network Security (ACNS)*. Feb. 2013.

[58] C. Mulliner, R. Borgaonkar, P. Stewin, and J. P. Seifert. **SMS-based one-time passwords: Attacks and defense**. In *Proceedings of the* 10^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jul. 2013.

[59] O. Mutlu. **The rowhammer problem and other issues we may face as memory becomes denser**. In *Proceedings of the* 20^{th} *Conference on Design, Automation & Test in Europe (DATE)*. Mar. 2017.

[60] M. Neugschwandtner, M. Lindorfer, and C. Platzer. **A view to a kill: Webview exploitation**. In *Proceedings of the* 6^{th} *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. Aug. 2013.

[61] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. **You are what you include: Large-scale evaluation of remote javascript inclusions**. In *Proceedings of the* 19^{th} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2012.

[62] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida. **Secure page fusion with vusion**. In *Proceedings of the* 26^{th} *ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2017.

[63] P. Papadopoulos, N. Kourtellis, and E. P. Markatos. **The cost of digital advertisement: Comparing user and advertiser views**. In *Proceedings of the* 27^{th} *International World Wide Web Conference (WWW)*. Oct. 2018.

[64] G. Pellegrino, C. Rossow, F. J. Ryba, T. C. Schmidt, and M. Wählisch. **Cashing out the great cannon? on browser-based ddos attacks and economics**. In *Proceedings of the* 9^{th} *USENIX Workshop on Offensive Technologies (WOOT)*. Aug. 2015.

[65] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. **DRAMA: Exploiting DRAM addressing for cross-CPU attacks**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[66] M. Pirker and D. Slamanig. **A framework for privacy-preserving mobile payment on security enhanced ARM TrustZone platforms**. In *Proceedings of the* 11^{th} *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. Jun. 2012.

[67] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. **Execute this! Analyzing unsafe and malicious dynamic code loading in android applications**. In *Proceedings of the* 21^{st} *Annual Network and Distributed System Security Symposium (NDSS)*. Feb. 2014.

[68] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. **All your iframes point to us**. In *Proceedings of the* 17^{th} *USENIX Conference on Security Symposium (USENIX SEC)*. Jul. 2008.

[69] N. Provos, D. McNamee, K. Wang, and N. Modadugu. **Bringing the web up to speed with webassembly**. In *Proceedings of the* 1^{st} *conference on First Workshop on Hot Topics in Understanding Botnets (HotBots)*. Apr. 2007.

[70] R. Qiao and M. Seaborn. **A new approach for rowhammer attacks**. In *Proceedings of the* 9^{th} *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. May 2016.

[71] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. **Flip Feng Shui: Hammering a needle in the software stack**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[72] R. van Rijswijk-Deij and E. Poll. **Using trusted execution environments in two-factor authentication: Comparing approaches**. In *Proceedings of the* 1^{st} *Open Identity Summit (OID)*. Sep. 2013.

[73] J. Rüth, T. Zimmermann, K. Wolsing, and O. Hohlfeld. **Digging into browser-based crypto mining**. In *Proceedings of the* 18^{th} *ACM Internet Measurement Conference (IMC)*. Oct. 2018.

[74] N. Santos, H. Raj, S. Saroiu, and A. Wolman. **Using ARM TrustZone to build a trusted language runtime for mobile applications**. In *Proceedings of the* 19^{th} *international conference on Architectural support for programming languages and operating systems (ASPLOS)*. Mar. 2014.

[75] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. **Reducing tcb complexity for security-sensitive applications: Three case studies**. In *Proceedings of the* 1^{st} *ACM European Conference on Computer Systems (EuroSys)*. Apr. 2006.

[76] H. Sun, K. Sun, Y. Wang, and J. Jing. **Trustotp: Transforming smartphones into secure one-time password tokens**. In *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2015.

[77] R. Tahir, M. Huzaifa, A. Das, M. Ahmad, C. Gunter, F. Zaffar, M. Caesar, and N. Borisov. **Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises**. In *Proceedings of the* 20^{th} *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Oct. 2017.

[78] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi. **Defeating software mitigations against Rowhammer: A surgical precision hammer**. In *Proceedings of the* 21^{st} *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Sep. 2018.

[79] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi. **Throwhammer: Rowhammer attacks over the network and defenses**. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*. Jul. 2018.

[80] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. **Drammer: Deterministic rowhammer attacks on mobile platforms**. In *Proceedings of the* 23^{rd} *ACM Conference on Computer and Communications Security (CCS)*. Oct. 2016.

[81] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi. **GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM**. In *Proceedings of the* 15^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Jun. 2018.

[82] T. Wang, K. Lu, L. Lu, S. P. Chung, and W. Lee. **Jekyll on iOS: When benign apps become evil**. In *Proceedings of the* 22^{nd} *USENIX Security Symposium (USENIX SEC)*. Aug. 2013.

[83] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao. **SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks**. In *Proceedings of the* 23^{rd} *European Symposium on Research in Computer Security (ESORICS)*. Aug. 2018.

[84] Z. Wang and A. Stavrou. **Exploiting smart-phone USB connectivity for fun and profit**. In *Proceedings of the* 26^{th} *Annual Computer Security Applications Conference (ACSAC)*. Dec. 2010.

[85] T. Weigold, T. Kramp, R. Hermann, F. Höring, P. Buhler, and M. Baentsch. **The Zurich trusted information channel – an efficient defence against man-in-the-middle and malicious software attacks**. In *Proceedings on the* 1^{st} *International Conference on Trusted Computing - Challenges and Applications (TRUST)*. Mar. 2008.

[86] J. Winter. **Experimenting with ARM TrustZone – or: How i met friendly piece of trusted hardware**. In *Proceedings of the* 11^{th} *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. Jun. 2012.

[87] Y. Xiao, X. Zhang, Y. Zhang, and M.-R. Teodorescu. **One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation**. In *Proceedings of the* 25^{th} *USENIX Security Symposium (USENIX SEC)*. Aug. 2016.

[88] D. Xu, J. Ming, and D. Wu. **Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping**. In *Proceedings of the* 38^{th} *IEEE Symposium on Security and Privacy (S&P)*. May 2017.

[89] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. **TruZ-Droid: Integrating trustzone with mobile operating system**. In *Proceedings of the* 16^{th} *International Conference on Mobile Systems, Applications, and Services (MobiSys)*. Jun. 2018.

[90] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna. **The dark alleys of madison avenue: Understanding malicious advertisements**. In *Proceedings of the* 14^{th} *ACM Internet Measurement Conference (IMC)*. Nov. 2014.

[91] F. Zhang and H. Zhang. **SoK: A study of using hardware-assisted isolated execution environments for security**. In *Proceedings of the* 5^{th} *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. Jun. 2016.

[92] T. Zhang, Y. Zhang, and R. B. Lee. **Cloudradar: A real-time side-channel attack detection system in clouds**. In *Proceedings of the* 19^{th} *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Sep. 2016.

[93] X. Zheng, L. Yang, J. Ma, G. Shi, and D. Meng. **Trustpay: Trusted mobile payment on security enhanced ARM TrustZone platforms**. In *Proceedings of the* 21^{st} *IEEE Symposium on Computers and Communication (ISCC)*. Jun. 2016.

Articles

[94] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi. **CAn't touch this: Practical and generic software-only defenses against rowhammer attacks**. *arXiv Computing Research Repository*. vol. abs/1611.08396. Nov. 2016.

[95] A. ElBahrawy, L. Alessandretti, A. Kandler, R. Pastor-Satorras, and A. Baronchelli. **Bitcoin ecology: Quantifying and modelling the long-term dynamics of the cryptocurrency market**. *arXiv Computing Research Repository*. vol. abs/1705.05334v2. May 2017.

[96] G. Gogniat, T. Wolf, W. Burleson, J. P. Diguet, L. Bossuet, and R. Vaslin. **Reconfigurable hardware for high-security/ high-performance embedded systems: The SAFES perspective**. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. vol. 16. no. 2, pp. 144–155. Feb. 2008.

[97] R. W. Hamming. **Error detecting and error correcting codes**. *The Bell System Technical Journal*. vol. 29. no. 2, pp. 147–160. Apr. 1950.

[98] J. Hong. **The state of phishing attacks**. *Communications of the ACM*. vol. 55. no. 1, pp. 74–81. Jan. 2012.

[99] H. Meng, V. L. L. Thing, Y. Cheng, Z. Dai, and L. Zhang. **A survey of android exploits in the wild**. *Computers & Security*. vol. 76. no. 1, pp. 71–91. Feb. 2018.

[100] G. A. Miller. **The magical number seven, plus or minus two: Some limits on our capacity for processing information**. *Psychological review*. vol. 101. no. 2, p. 343. Apr. 1994.

[101] D. A. Ortiz-Yepes, R. J. Hermann, H. Steinauer, and P. Buhlerr. **Bringing strong authentication and transaction security to the realm of mobile devices**. *IBM Journal of Research and Development*. vol. 58. no. 1, 4:1–4:11. Jan. 2014.

[102] P. Papadopoulos, P. Ilia, and E. P. Markatos. **Truth in Web Mining: Measuring the Profitability and Cost of Cryptominers as a Web Monetization Model**. *arXiv Computing Research Repository*. vol. abs/1806.01994v1. Jun. 2018.

[103] J. H. Saltzer and M. Schroeder. **The protection of information in computer systems**. *Proceedings of the IEEE*. vol. 63. no. 9, pp. 1278–1308. Sep. 1975.

[104] L. Wagner. **Turbocharging the web**. *IEEE Spectrum*. vol. 54. no. 1, pp. 48–53. Nov. 2017.

Books

[105] G. McGraw. **Software security: building security in**. Addison-Wesley Professional, 2006.

[106] J. Viega and G. McGraw. **Building Secure Software: How to Avoid Security Problems the Right Way**. Addison-Wesley Professional, 2001.

Technical Reports and Documentation

[107] Coinhive. **Coinhive AuthedMine - A Non-Adblocked Miner**. https://coinhive.com/documentation/authedmine. 2018.

[108] FuturePlus System. **DDR2 800 bus analysis probe**. https://www.yumpu.com/en/document/view/41592980/fs2334-ddr2-800-mt-s-dimm-analysis-probe-futureplus-systems. 2016.

[109] GlobalPlatform. **TEE Client API Specification Version 1.0**. GPD_SPE_007. 2010.

[110] . GlobalPlatform. **TEE Internal API Specification Version 1.0**. GPD_EPR_017. 2014.

[111] GlobalPlatform. **TEE System Architecture Version 1.1.0.10**. GPD_SPE_009. 2018.

[112] JEDEC Solid State Technology Association. **Low power double data 4 (LPDDR4)**. JESD209-4A. 2015.

[113] JEDEC Solid State Technology Association. **Low power double data 4 (LPDDR4)**. JESD79-4B. 2017.

[114] Perf Wiki. **Perf: Linux profiling with performance counters**. https://perf.wiki.kernel.org/index.php/Main_Page.

[115] Samsung. **Trusted boot**. https://docs.samsungknox.com/whitepapers/knox-platform/trusted-boot.htm.

[116] Slushpool. **Stratum Mining Protocol**. https://slushpool.com/help/manual/stratum-protocol. 2016.

[117] Web Hypertext Application Technology Working Group. **HTML Living Standard: Web workers**. https://html.spec.whatwg.org/multipage/workers.html. 2018.

[118] T. Alves and D. Felton. **TrustZone: Integrated Hardware and Software Security**. Technical Report. Jul. 2004.

[119] ARM. **ARM security technology: Building a secure system using TrustZone technology**. Technical Report. Apr. 2009.

[120] S. A. Bailey, D. Felton, V. Galindo, F. Hauswirth, J. Hirvimies, M. Fokle, F. Morenius, C. Colas, J.-P. Galvan, G. Bernabeu, *et al.* **The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market**. GlobalPlatform. Technical Report. Feb. 2011.

[121] Dick O'Brien. **Dridex: Tidal waves of spam pushing dangerous financial trojan**. Symantec. Technical Report. Feb. 2016.

[122] ENSILO. **Vulnerable by design: Why destructive exploits keep on coming**. Technical Report. Mar. 2016.

[123] M. Lanteigne. **How rowhammer could be used to exploit weaknesses in computer hardware**. Third I/O Inc. Technical Report. Mar. 2016.

[124] P. Schartner and S. Bürger. **Attacking mTAN-applications like e-banking and mobile signatures**. Univeristy of Klagenfurt. Technical Report. Dec. 2011.

[125] Symantec. **Dyre: Emerging threat on financial fraud landscape**. Technical Report. Jun. 2015.

Online

[126] ABN AMRO. **E.dentifier2**. [Online]. Available: https://www.abnamro.nl/en/images/Generiek/PDFs/Overig/edentifier2_usermanual_english.pdf.

[127] Alexa. **SEO and Competitive Analysis Software**. [Online]. Available: https://www.alexa.com.

[128] Apple. **iCloud: Erase your device**. Mar. 2015. [Online]. Available: https://support.apple.com/kb/PH2701?locale=en_US.

[129] Apple. **Use continuity to connect your iPhone, iPad, iPod touch, and Mac**. Nov. 2015. [Online]. Available: https://support.apple.com/en-gb/HT204681.

[130] F. Assolini. **SMiShing and the rise of mobile banking attacks**. Aug. 2016. [Online]. Available: https://securelist.com/smishing-and-the-rise-of-mobile-banking-attacks/75575/.

[131] N. Avital, M. Lion, and R. Masas. **Crypto Me0wing Attacks: Kitty Cashes in on Monero**. May 2018. [Online]. Available: https://www.incapsula.com/blog/crypto-me0wing-attacks-kitty-cashes-in-on-monero.html.

[132] B. Barrett. **How to protect yourself against a sim swap attack**. Aug. 2018. [Online]. Available: https://www.wired.com/story/sim-swap-attack-defend-phone/.

[133] Board of Governors of the Federal Reserve System. **Consumers and mobile financial services**. Mar. 2016. [Online]. Available: http://www.federalreserve.gov/econresdata/consumers-and-mobile-financial-services-report-201603.pdf.

[134] J. I. Boutin. **The evolution of webinjects**. Sep. 2014. [Online]. Available: https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Boutin.pdf.

[135] C. Castillo. **Phishing attack replaces android banking apps with malware**. Jun. 2013. [Online]. Available: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/phishing-attack-replaces-android-banking-apps-with-malware/.

[136] F. Chytry. **Apps on Google Play pose as games and infect millions of users with adware**. Feb. 2015. [Online]. Available: https://blog.avast.com/2015/02/03/apps-on-google-play-pose-as-games-and-infect-millions-of-users-with-adware/.

[137] C. Cimpanu. **Cryptojackers found on starbucks wifi network, github, pirate streaming sites**. Dec. 2017. [Online]. Available: https://www.bleepingcomputer.com/news/security/cryptojackers-found-on-starbucks-wifi-network-github-pirate-streaming-sites/.

[138] C. Cimpanu. **Firefox Working on Protection Against In-Browser Cryptojacking Scripts**. Mar. 2018. [Online]. Available: https://www.bleepingcomputer.com/news/security/tweak-to-chrome-performance-will-indirectly-stifle-cryptojacking-scripts/.

[139] C. Cimpanu. **Tweak to Chrome Performance Will Indirectly Stifle Cryptojacking Scripts**. Feb. 2018. [Online]. Available: https://www.bleepingcomputer.com/news/security/tweak-to-chrome-performance-will-indirectly-stifle-cryptojacking-scripts/.

[140] Coinhive. **Coinhive monetize your business with your users cpu power**. [Online]. Available: https://coinhive.com/.

[141] CryptoCompare. **Monero**. Jul. 2015. [Online]. Available: https://www.cryptocompare.com/coins/xmr/.

[142] Cryptomining24.net. **CPU for Monero**. Nov. 2017. [Online]. Available: https://cryptomining24.net/cpu-for-monero/.

[143] J. J. Drake. **Google Android - "Stagefright" Remote Code Execution (CVE-2015-1538)**. Sep. 2015. [Online]. Available: https://www.exploit-db.com/exploits/38124.

[144] eMarketer. **Smartphone users worldwide will total 1.75 billion in 2014**. Jan. 2014. [Online]. Available: https://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536.

[145] EMC^2. **RSA SecureID Hardware Tokens**. [Online]. Available: https://www.rsa.com/en-us/products/identity-and-access-management/securid-hardware-tokens.

[146] ENCAP Security. **SMiShing and the rise of mobile banking attacks**. Aug. 2016. [Online]. Available: https://securelist.com/smishing-and-the-rise-of-mobile-banking-attacks/75575/.

[147] J. Evers. **Virus makes leap from PC to PDA**. Mar. 2006. [Online]. Available: https://www.cnet.com/news/virus-makes-leap-from-pc-to-pda/.

[148] Exact Target. **2014 mobile behavior report**. Feb. 2014. [Online]. Available: https://brandcdn.exacttarget.com/sites/exacttarget/files/deliverables/etmc-2014mobilebehaviorreport.pdf.

[149] Federal Financial Institutions Examination Council. **Authentication in an internet banking environment**. Oct. 2005. [Online]. Available: https://www.ffiec.gov/pdf/authentication_guidance.pdf.

[150] L. Franceschi-Bicchierai. **How criminals recruit telecom employees to help them hijack sim cards**. Aug. 2018. [Online]. Available: https://www.vice.com/en_us/article/3ky5a5/criminals-recruit-telecom-employeessim-swapping-port-out-scam.

[151] Giesecke and Devrient. **MobiCore**. [Online]. Available: https://www.gi-de.com/.

[152] Github. **The world's leading software development platform**. [Online]. Available: https://github.com/.

[153] GlobalPlatform. **Certified products**. [Online]. Available: https://globalplatform.org/certified-products/.

[154] D. Goodin. **Websites use your CPU to mine cryptocurrency even when you close your browser**. Nov. 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/11/sneakier-more-persistent-drive-by-cryptomining-comes-to-a-browser-near-you/.

[155] D. Goodin. **Now even YouTube serves ads with CPU-draining cryptocurrency miners**. Jan. 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/01/now-even-youtube-serves-ads-with-cpu-draining-cryptocurrency-miners/.

[156] D. Goodin. **Attackers exploit 0-day vulnerability that gives full control of android phones**. Oct. 2019. [Online]. Available: https://arstechnica.com/information-technology/2019/10/attackers-exploit-0day-vulnerability-that-gives-full-control-of-android-phones/.

[157] Google. **Android Intents with Chrome**. Feb. 2015. [Online]. Available: https://developer.chrome.com/multidevice/android/intents.

[158] Google. **Remotely ring, lock or erase a lost device**. Oct. 2015. [Online]. Available: https://support.google.com/accounts/answer/6160500?hl=en.

[159] Google. **Chromium Issue 766068: Please consider intervention for high cpu usage js**. Sep. 2017. [Online]. Available: https://blog.malwarebytes.com/cybercrime/2018/02/state-malicious-cryptomining/.

[160] Google. **Get your bookmarks, passwords & more on all your devices**. Jun. 2018. [Online]. Available: https://support.google.com/chrome/answer/165139?visit_id=1-636634877196601807-123781004.

[161] Google. **Google Authenticator**. [Online]. Available: https://github.com/google/google-authenticator.

[162] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, and A. R. Rege. **Digital identity guidelines – authentication and lifecycle management**. Jun. 2017. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-63b.

[163] J. Gu, V. Zhang, and S. Shen. **ZNIU: First android malware to exploit dirty cow vulnerability**. Sep. 2017. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/zniu-first-android-malware-exploit-dirty-cow-vulnerability/.

[164] P. Gühring. **Concepts against man-in-the-browser attacks**. Sep. 2006. [Online]. Available: http://www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf.

[165] J. J. Hoffman, S. C. Lee, and J. S. Jacobson. **New jersey division of consumer affairs obtains settlement with developer of bitcoin-mining software found to have accessed new jersey computers without users' knowledge or consent**. May 2015. [Online]. Available: https://nj.gov/oag/newsreleases15/pr20150526b.html.

[166] Inazaruk. **"activating" Android applications**. Dec. 2011. [Online]. Available: https://devmaze.wordpress.com/2011/12/05/activating-applications/.

[167] N. Johnson. **Qualcomm's chain of trust**. Sep. 2018. [Online]. Available: https://lineageos.org/engineering/Qualcomm-Firmware/.

[168] D. Kawamoto. **Cell phone virus tries leaping to PCs**. Sep. 2005. [Online]. Available: https://www.cnet.com/news/cell-phone-virus-tries-leaping-to-pcs/.

[169] S. Kenin. **Mass MikroTik Router Infection – First we cryptojack Brazil, then we take the World?** Aug. 2018. [Online]. Available: https://www.trustwave.com/Resources/SpiderLabs-Blog/Mass-MikroTik-Router-Infection---First-we-cryptojack-Brazil,-then-we-take-the-World-/.

[170] L. Kharouni. **Automating online banking fraud**. Jun. 2012. [Online]. Available: https://www.trendmicro.de/cloud-content/us/pdfs/security-intelligence/white-papers/wp_automating_online_banking_fraud.pdf.

[171] B. Krebs. **Busting sim swappers and sim swap myths**. Nov. 2018. [Online]. Available: https://krebsonsecurity.com/2018/11/busting-sim-swappers-and-sim-swap-myths/.

[172] B. Krebs. **Who and what is coinhive?** Mar. 2018. [Online]. Available: https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/.

[173] M. Labs. **Mcafee labs threats report**. Jun. 2014. [Online]. Available: https://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2014.pdf.

[174] Lenovo. **Row hammer privilege escalation**. Mar. 2015. [Online]. Available: https://support.lenovo.com/us/en/product_security/row_hammer.

[175] S. Liao. **Showtime websites secretly mined user cpu for cryptocurrency**. Sep. 2017. [Online]. Available: https://www.theverge.com/2017/9/26/16367620/showtime-cpu-cryptocurrency-monero-coinhive/.

[176] S. Liao. **UNICEF wants you to mine cryptocurrency for charity**. Apr. 2018. [Online]. Available: https://www.theverge.com/2018/4/30/17303624/unicef-mining-cryptocurrency-charity-monero/.

[177] Linaro. **OP-TEE**. [Online]. Available: https://www.op-tee.org/.

[178] C. Liu and J. C. Chen. **Cryptocurrency Web Miner Script Injected into AOL Advertising Platform**. Apr. 2018. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/cryptocurrency-web-miner-script-injected-into-aol-advertising-platform/.

[179] A. Meshkov. **Crypto-Streaming Strikes Back**. Dec. 2017. [Online]. Available: https://blog.adguard.com/en/crypto-streaming-strikes-back/.

[180] Microsoft. **Find a lost phone**. May 2016. [Online]. Available: https://support.microsoft.com/en-gb/help/11585/windows-phone-find-a-lost-phone.

[181] Microsoft. **Microsoft security bulletin ms16-092 - important**. Jul. 2016. [Online]. Available: https://docs.microsoft.com/en-us/security-updates/securitybulletins/2016/ms16-092.

[182] Microsoft. **Microsoft Authenticator**. [Online]. Available: https://www.microsoft.com/en-us/store/p/microsoft-authenticator/9nblgggzmcj6.

[183] MineCryptoNight. **Enter your CryptoNight original/v1 hash rate and get the most profitable coins**. 2018. [Online]. Available: https://minecryptonight.net.

[184] Mozilla. **How do I set up Sync on my computer?** Jun. 2018. [Online]. Available: https://support.mozilla.org/en-US/kb/how-do-i-set-sync-my-computer.

[185] T. Mursch. **Cryptojacking malware coinhive found on 30,000+ websites**. Nov. 2017. [Online]. Available: https://badpackets.net/cryptojacking-malware-coinhive-found-on-30000-websites/.

[186] T. Mursch. **How to find cryptojacking malware**. Feb. 2018. [Online]. Available: https://badpackets.net/how-to-find-cryptojacking-malware/.

[187] S. Nakamoto. **Bitcoin: A peer-to-peer electronic cash system**. Oct. 2008. [Online]. Available: https://www.bitcoin.org/bitcoin.pdf.

[188] NetMarketShare. **Operating system market share**. Jun. 2018. [Online]. Available: https://www.netmarketshare.com/operating-system-market-share.aspx.

[189] L. H. Newman. **The hidden toll of fixing meltdown and spectre**. Dec. 2018. [Online]. Available: https://www.wired.com/story/meltdown-and-spectre-patches-take-toll/.

[190] L. O'Donnell. **Cryptojacking Attack Found on Los Angeles Times Website**. Feb. 2018. [Online]. Available: https://threatpost.com/cryptojacking-attack-found-on-los-angeles-times-website/130041/.

[191] L. O'Donnell. **Cryptojacking Campaign Exploits Drupal Bug, Over 400 Websites Attacked**. May 2018. [Online]. Available: https://threatpost.com/cryptojacking-campaign-exploits-drupal-bug-over-400-websites-attacked/131733/.

[192] P. Oester. **A privilege escalation vulnerability in the linux kernel (cve-2016-5195)**. Jul. 2019. [Online]. Available: https://dirtycow.ninja/.

[193] Pirate Bay. **Miner**. Sep. 2017. [Online]. Available: https://thepiratebay.org/blog/242.

[194] PublicWWW. **Source Code Search Engine**. [Online]. Available: https://publicwww.com.

[195] Qualcomm. **Qualcomm Trusted Execution Environment (QSEE)**. [Online]. Available: https://www.qualcomm.com/.

[196] Salon. **FAQ: What happens when I choose to "Suppress Ads" on Salon?** [Online]. Available: https://www.theverge.com/2018/4/30/17303624/unicef-mining-cryptocurrency-charity-monero/.

[197] B. Sams. **Microsoft confirms Edge will sync passwords, bookmarks, tabs, and more**. May 2015. [Online]. Available: https://www.neowin.net/news/microsoft-confirms-edge-will-sync-passwords-bookmarks-tabs-and-more.

[198] J. Segura. **Malicious cryptomining and the blacklist conundrum**. Mar. 2018. [Online]. Available: https://blog.malwarebytes.com/threat-analysis/2018/03/malicious-cryptomining-and-the-blacklist-conundrum/.

[199] J. Segura. **The state of malicious cryptomining**. Mar. 2018. [Online]. Available: https://blog.malwarebytes.com/cybercrime/2018/02/state-malicious-cryptomining/.

[200] Seigen, M. Jameson, T. Nieminen, Neocortex, and A. M. Juarez. **Cryptonight hash function**. Mar. 2013. [Online]. Available: https://cryptonote.org/cns/cns008.txt.

[201] Sierraware. **SierraTEE Trusted Execution Environment**. [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html.

[202] SimilarWeb. **Website Traffic Statistics & Market Intelligence**. [Online]. Available: https://www.similarweb.com.

[203] D. Sinegubko. **Hacked Websites Mine Cryptocurrencies**. Sep. 2017. [Online]. Available: https://blog.sucuri.net/2017/09/hacked-websites-mine-crypocurrencies.html.

[204] J. Smith and creker. **Get SMS broadcast with text body without jailbreak but private frameworks in iOS**. Oct. 2014. [Online]. Available: http://stackoverflow.com/questions/26642770/get-sms-broadcast-with-text-body-without-jailbreak-but-private-frameworks-in-ios.

[205] A. K. Sood, R. J. Enbody, and R. Bansal. **The art of stealing banking information — form grabbing on fire**. Nov. 2011. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2011/11/art-stealing-banking-information-form-grabbing-fire.

[206] Statista. **Global smartphone sales to end users 2007–2014**. Mar. 2015. [Online]. Available: http://www.statista.com:80/statistics/263437/global-smartphone-sales-to-end-users-since-2007.

[207] The Guardian. **'criminal mastermind' of $4bn bitcoin laundering scheme arrested**. Sep. 2017. [Online]. Available: https://www.theguardian.com/technology/

2017/jul/27/russian-criminal-mastermind-4bn-bitcoin-laundering-scheme-arrested-mt-gox-exchange-alexander-vinnik.

[208] I. Thomson. **Pulitzer-winning website politifact hacked to mine crypto-coins in browsers**. Oct. 2017. [Online]. Available: https://www.theregister.co.uk/2017/10/13/politifact_mining_cryptocurrency/.

[209] P. Tjin. **Android-7.1.0_r7 (disable ion_heap_type_system _contig)**. Oct. 2016. [Online]. Available: https://android.googlesource.com/device/google/marlin-kernel/+/android-7.1.0%5C_r7.

[210] M. Trofin. **Chromium code reviews issue 2656103003: [wasm] flag for asm-wasm investigations**. Jan. 2017. [Online]. Available: https://codereview.chromium.org/2656103003/.

[211] TrustKernel. **T6**. [Online]. Available: https://www.trustkernel.com/products/tee/t6.html.

[212] Trustonic. **What is a Trusted Execution Environment (TEE)?** [Online]. Available: https://www.trustonic.com/news/technology/what-is-a-trusted-execution-environment-tee/.

[213] A. Viquez. **Opera introduces bitcoin mining protection in all mobile browsers – here's how we did it**. Jan. 2018. [Online]. Available: https://blogs.opera.com/mobile/2018/01/opera-introduces-bitcoin-mining-protection-mobile-browsers/.

[214] D. Vyukov. **Use-After-Free Remote Code Execution Vulnerability (CVE-2016-7117)**. Mar. 2016. [Online]. Available: https://www.exploit-db.com/exploits/38124.

[215] C. Williams. **UK ICO, USCourts.gov... Thousands of websites hijacked by hidden crypto-mining code after popular plugin pwned**. Feb. 2018. [Online]. Available: http://www.theregister.co.uk/2018/02/11/browsealoud_compromised_coinhive/.

[216] J. Wyke. **What is Zeus?** May 2011. [Online]. Available: https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/Sophos%20what%20is%20zeus%20tp.pdf.

[217] E. Xu and J. C. Chen. **First active attack exploiting cve-2019-2215 found on google play, linked to sidewinder apt group**. Jan. 2020. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/first-active-attack-exploiting-cve-2019-2215-found-on-google-play-linked-to-sidewinder-apt-group/.

[218] Yandex. **Yandex Browser Strengthens Cryptocurrency Mining Protection**. Mar. 2018. [Online]. Available: https://yandex.com/company/blog/yandex-browser-strengthens-cryptocurrency-mining-protection/.

[219] Yubico. **Yubikey**. [Online]. Available: https://www.yubico.com/.

[220] Z. Zaifeng. **Who is stealing my power iii: An adnetwork company case study**. Feb. 2018. [Online]. Available: https://blog.netlab.360.com/who-is-stealing-my-power-iii-an-adnetwork-company-case-study-en/.

[221] Z. Zorz. **How a URL shortener allows malicious actors to hijack visitors' CPU power**. May 2018. [Online]. Available: https://www.helpnetsecurity.com/2018/05/23/url-shortener-cryptojacking/.

Talks

[222] P. Gullberg. **Trusted execution environment, trustzone and mobile security**. *OWASP göteborg: security tapas*. Oct. 2015.

[223] J. Oberheide and C. Miller. **Dissecting the Android Bouncer**. *Summercon*. Jun. 2012.

[224] M. Schwarz, D. Gruss, and M. Lipp. **Another Flip in the Row**. *Black Hat USA*. Aug. 2018.

[225] M. Seaborn and T. Dullien. **Exploiting the DRAM rowhammer bug to gain kernel privileges**. *Black Hat USA*. Aug. 2015.

[226] V. van der Veen. **Drammer: Deterministic Rowhammer Attacks on Mobile Platforms**. *CCS*. Oct. 2016.

Source Code

[227] GitHub. **CoinBlockerLists Homepage**. Apr. 2018. [Online]. Available: https://github.com/ZeroDot1/CoinBlockerListsWeb.

[228] Github Repository. **Open source Android apps in Github**. Sep. 2016. [Online]. Available: https://github.com/pcqpcq/open-source-android-apps.

[229] MinerBlock. **An efficient browser extension to block browser-based cryptocurrency miners all over the web.** Jan. 2018. [Online]. Available: https://github.com/xd4rker/MinerBlock.

[230] M. F. Oberhumer. **LZO**. Mar. 2017. [Online]. Available: http://www.oberhumer.com/opensource/lzo.

[231] Prakash. **Dr. Mine**. Feb. 2018. [Online]. Available: https://github.com/1lastBr3ath/drmine.

[232] Rafael K. **No Coin is a tiny browser extension aiming to block coin miners such as Coinhive.** Oct. 2017. [Online]. Available: https://github.com/keraf/NoCoin.

[233] G. Tene. **WRK2 - a HTTP Benchmarking Tool**. Sep. 2018. [Online]. Available: https://github.com/giltene/wrk2.

[234] WebAssembly. **WABT: The WebAssembly Binary Toolkit**. Apr. 2018. [Online]. Available: https://github.com/WebAssembly/wabt.

Summary

Design flaws and implementation bugs are two different types of security defects. Design flaws are mistakes/errors that occur in the design phase, while implementation bugs are errors that occur in the implementation phase of the product development lifecycle. Unfortunately, the current focus of the systems security community is more on common implementation bugs than on design flaws even though design flaws constitute 50% of the security defects.

To enhance usability and performance, both application developers and platform vendors are constantly introducing new features (like *synchronization features*), and often such desires for increased usability/performance results in a violation of secure design principles. This is the reason why most design flaws hide in plain sight as product features. Attackers can take advantage of the unintended consequence of such features to compromise the whole system. This is a different way of exploitation when compared to typical memory corruption bug exploitation. Hence, it is typically difficult to detect, and complex to patch a design flaw compared to an implementation bug — often requiring solutions that are unique to each attack.

This thesis explores design flaws that may occur at the software- and hardware-level of computing systems, and the cyber threats stemming from them. We build novel software-level computer defenses to protect from these identified cyber threats and discuss the cost associated with them. Futhermore, this research is broadened by identifying whether the current set of design principles is comprehensive enough to prevent today's cyber threats. We achieve this goal by performing an in-depth analysis of a new cyber attack called *cryptojacking* which does not break today's widely-accepted security model called CIA triad (standing for Confidentiality, Integrity, and Availability; yet, a very practical and stealthy cyberattack that monetizes off a victim's computational resources. Based on this study, we propose to include *least required resource* as a new principle to the current set of design principles to offer protection from such futuristic attacks.

Samenvatting

Ontwerpfouten en implementatiefouten zijn twee verschillende soorten beveiligingsgebreken. Ontwerpfouten zijn fouten die zich voordoen in de ontwerpfase, terwijl implementatiefouten fouten zijn die zich voordoen in de implementatiefase van de productontwikkelingscyclus. Helaas is de huidige focus van de systeembeveiligingsgemeenschap meer gericht op veelvoorkomende implementatiefouten dan op ontwerpfouten, ook al vormen ontwerpfouten 50% van de beveiligingsgebreken.

Om de gebruiksvriendelijkheid en prestaties te verbeteren, introduceren zowel applicatieontwikkelaars als platformverkopers voortdurend nieuwe functies (zoals *synchronisatiefuncties*), en vaak resulteert een dergelijke wens tot verhoogde gebruiksvriendelijkheid/prestatie in een schending van de principes van veilig ontwerp. Dit is de reden waarom de meeste ontwerpfouten zichtbaar zijn als producteigenschappen maar toch niet gezien worden. Aanvallers kunnen profiteren van de onbedoelde gevolgen van dergelijke functies om het hele systeem in gevaar te brengen. Dit is een andere manier van misbruik in vergelijking met typische manieren om geheugenfouten te misbruiken. Om deze reden is het normaal gesproken moeilijk om dergelijke fouten te ontdekken, en in vergelijking met een implementatiefout is het complex om een ontwerpfout te herstellen. Dit vereist vaak oplossingen die voor elke aanval uniek zijn.

In deze dissertatie onderzoeken wij ontwerpfouten die kunnen voorkomen op het software- en hardwareniveau van computersystemen, en de cyberbedreigingen die daaruit voortvloeien. We bouwen nieuwe softwarematige computerbeveiliging om te beschermen tegen de geïdentificeerde cyberbedreigingen en bespreken de kosten die ermee gepaard gaan. Bovendien wordt dit onderzoek verbreed door na te gaan of de huidige verzameling van ontwerpprincipes uitgebreid genoeg is om de huidige cyberdreigingen te voorkomen. We bereiken dit doel door een diepgaande analyse uit te voeren van een nieuwe cyberaanval, *cryptojacking* genaamd, die het algemeen aanvaarde veiligheidsmodel van van-

daag, de CIA-triade, niet doorbreekt (dat staat voor Condentiality, Integrity en Availability) maar wel een zeer praktische en moelijk te ontdekken cyberaanval die de rekenkracht van een slachtoffer in geld omzet. Op basis van deze studie stellen we voor om *least required resource* op te nemen als een nieuw principe in de huidige set van ontwerpprincipes om bescherming te bieden tegen dergelijke futuristische aanvallen.