Michael Schwarz

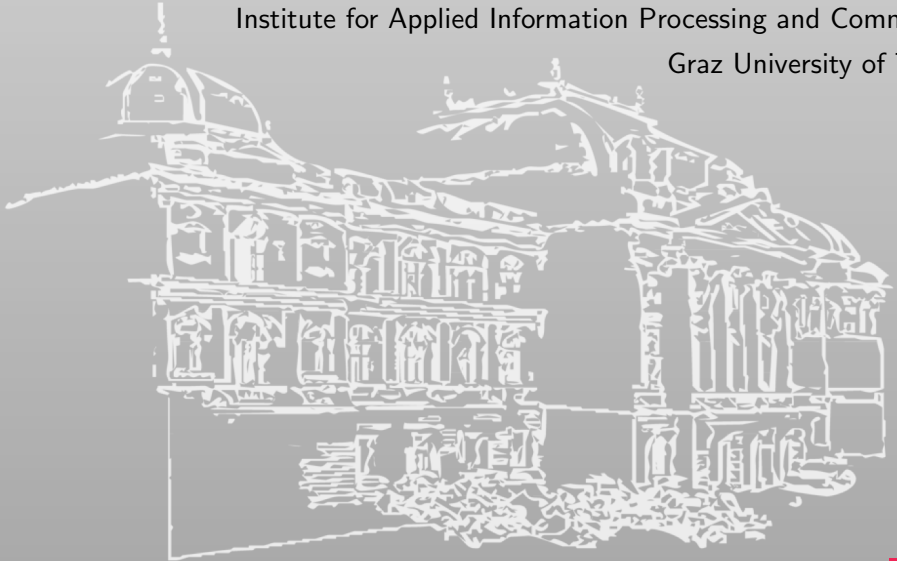
Software-based Side-Channel Attacks and Defenses in Restricted Environments

PhD Thesis

Assessors: Daniel Gruss, Frank Piessens

November 2019

Institute for Applied Information Processing and Communications
Graz University of Technology



SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Abstract

The main assumption of computer systems is that processed secrets are inaccessible for an attacker due to security measures in software and hardware. However, side-channel attacks allow an attacker to still deduce the secrets by observing certain side effects of a computation.

For software-based attacks, unprivileged code execution is often sufficient to exploit side-channel weaknesses in applications. More recently, it was also shown that native code execution is not strictly necessary for certain attacks. Software-based side-channel attacks are even possible in JavaScript, a sandboxed scripting language found in modern browsers.

In this thesis, we further investigate software-based side-channel attacks to develop effective countermeasures. We show that state-of-the-art countermeasures are not always effective, as the assumptions on which the countermeasures are based are not always correct. Our research resulted in novel side channels, reduction of requirements for existing attacks, and enabling attacks in environments which were considered too restricted before. This results in a better understanding of attack requirements and attack surface. As a consequence, we were able to propose better defenses against this class of attacks, both for native and restricted environments.

This thesis consists of two parts. In the first part, we introduce software-based side-channel and microarchitectural attacks. We provide the required background on side channels, microarchitecture, caches, sandboxing, and isolation. We then discuss state-of-the-art attacks and defenses. The second part consists of a selection of my peer-reviewed unmodified papers.[1] I was the main contributor and first author to these papers which have all been accepted and presented at renowned international security conferences.

[1]The content of the papers is unmodified from the conference-proceeding versions of the papers. Only the format of the papers was changed to fit the style and layout of this thesis.

Acknowledgments

First and foremost, I would like to thank my advisor Daniel Gruss. You introduced me to the fascinating world of microarchitectural attacks, and I could always count on you when I was stuck or required a different perspective on a problem. Without your ambitious goals for publishing high-quality papers in combination with the freedom to research whatever I thought was an interesting topic, my PhD would not have been the same great adventure. Moreover, thank you for allowing me to teach so many courses with you, I really enjoyed it.

I also want to thank Stefan Mangard for giving me the opportunity to start my PhD. You always supported me in both my research and my teaching.

My sincere thanks also goes to my assessor Frank Piessens for valuable comments, interesting discussions, and fruitful collaborations.

I thank my fellow colleagues at IAIK who were always there for discussions and an occasional cup of coffee. Thank you for making my time at IAIK so enjoyable.

A special thank goes to Moritz, who worked on so many projects with me. Thank you for discussing so many ideas, working on papers until late at night, going to conferences, and finding my bugs. I enjoyed every day working with you. Thank you, Clémentine, for helping me in the beginning of my PhD. I learned so much about writing papers from you. You really helped me at my first steps, and I always enjoyed working with you.

Finally, I want to thank my girlfriend Angela, my friends, and my family. Thank you, Angela, for supporting me with everything I do and tolerating that I often invested far too many hours into my work.

Table of Contents

Acknowledgements **v**

I Software-based Side-Channel Attacks and Defenses in Restricted Environments **ix**

1 Introduction and Motivation **1**
1.1 Main Contributions . . . 3
1.2 Other Contributions . . . 6
1.3 Outline . . . 9

Introduction and Contribution **1**

2 Background **11**
2.1 Virtual Memory . . . 11
2.2 Caches . . . 13
2.2.1 Cache Organization . . . 13
2.2.2 Set-associative Caches . . . 14
2.2.3 Cache-replacement Policies . . . 15
2.2.4 Caches on Intel x86 CPUs . . . 16
2.3 Side-Channel and Microarchitectural Attacks . . . 16
2.3.1 Side Channels . . . 17
2.3.2 Microarchitecture . . . 18
2.3.3 Microarchitectural Side-Channel Attacks . . . 19
2.4 Sandboxing and Isolation . . . 22
2.4.1 JavaScript . . . 22
2.4.2 Intel SGX . . . 23

3 State of the Art **25**
3.1 Software-based Microarchitectural Attacks . . . 25
3.1.1 Cache Attacks . . . 25
3.1.2 Attacks on Predictors . . . 31

3.1.3 DRAM Attacks 32
3.1.4 Transient-Execution Attacks 33
3.2 Defenses against Software-based Microarchitectural Attacks 36
3.2.1 Software Layer 36
3.2.2 System Layer 37
3.2.3 Hardware Layer 41

4 Conclusion 43

II Publications 65

5 Fantastic Timers 71

6 Malware Guard Extension 101

7 KeyDrown 131

8 JavaScript Zero 175

9 DECAF 219

10 JavaScrip Template Attacks 263

11 ZombieLoad 307

Affidavit 355

Part I

Software-based Side-Channel Attacks and Defenses in Restricted Environments

1 Introduction and Motivation

Computer systems are often used for complex computations with secret values, e.g., cryptographic operations. The central assumption is that processed secrets are inaccessible for an attacker due to security measures in software and hardware. However, side-channel attacks allow an attacker to observe certain side effects of computations to still deduce the secrets.

Hardware-based side-channel attacks have long been known as powerful attacks. However, until recently, little attention was paid to software-based side-channel attacks. Countermeasures have been developed to harden devices against physical attacks, e.g., ensuring that algorithms have a constant runtime. While these countermeasures can prevent certain hardware-based attacks, they are incomplete against software-based attacks. Kocher [106] was the first to describe how runtime differences introduced by computer caches can be exploited to break constant-time implementations of cryptographic algorithms. Since then, caches play a predominant role in software-based side-channel attacks, as they are present on most modern processors and are comparatively easy to exploit by an attacker. Cache-based attacks include Prime+Probe [132, 136], Evict+Time [132], Flush+Reload [201] and further variants, such as Flush+Flush [64] or Evict+Reload [63, 113]. These attack techniques have been used to, e.g., break cryptographic algorithms [26, 91, 93, 119, 132, 136, 201] or spy on user input [63, 113, 153]. Recently, cache attacks were also leveraged as building blocks for Meltdown [114] and Spectre [105],

two powerful attacks which break the security guarantees of modern processors. Meltdown effectively breaks the security boundary between user space and kernel space, allowing unprivileged attackers to read arbitrary memory. Spectre exploits the branch prediction of modern processors to achieve the same goal. While Meltdown exploited an implementation bug in out-of-order execution and Spectre exploited branch prediction, both attacks leveraged the cache as an intermediate medium to encode and later transmit the leaked data.

Compared to hardware-based side-channel attacks, software-based attacks have fewer requirements, e.g., they do not require full control over the device. Instead, unprivileged code execution is often sufficient to exploit side-channel weaknesses in applications. Side-channel attacks from unprivileged code cannot only attack other applications [26, 63, 91, 93, 113, 119, 132, 136, 201], but can also leak information from the underlying operating system [66, 68, 78, 96, 153, 157]. More recently, it was also shown that native code execution is not strictly necessary for certain attacks. Oren et al. [131] demonstrated the first cache attack, namely Prime+Probe, in JavaScript, a sandboxed scripting language found in modern browsers. Moreover, Gruss et al. [67] mounted a page-deduplication attack [167] in JavaScript, showing that even restricted environments allow mounting powerful attacks. Thus, while JavaScript is a language-level sandbox preventing classical memory-safety violations, such sandboxes do not necessarily prevent side-channel attacks. This is also true for other forms of isolation, such as virtual machines. Several side-channel attacks can be mounted across virtual machines, either to steal data [119, 209], or to covertly transmit data from one virtual machine to a different virtual machine [125, 197, 198]. In many scenarios, sandboxing or virtualization was considered an effective security measure. However, this is not always the case when taking side-channel attacks into account.

In this thesis, we further investigate the possibilities for mounting software-based side-channel and microarchitectural attacks from restricted environments, such as sandboxes, operating-system-level virtualization, and virtual machines. Such isolation mechanisms limit the impact of classical attacks. At first glance, some of these mechanisms also appear to mitigate certain side-channel attacks. However, we present novel attack primitives to enable known side-channel and microarchitectural attacks in these environments. Moreover, we show novel side channels which can even be exploited without requiring native code execution, e.g., in JavaScript.

We furthermore show that side-channel attacks are applicable to otherwise isolated environments, such as trusted-execution environments. We were the first to not only mount cache side-channel attacks on secure enclaves protected by Intel SGX but also from within such secure enclaves. These attacks are stealthy and not detectable or preventable by state-of-the-art detection and prevention mechanisms. This also introduces a novel threat model, where trusted-execution environments are not only the target of an attack but are abused to disguise attacks. Although trusted-execution environments are usually limited in functionality to prevent precisely such a scenario [83], we show that the provided functionality is still sufficient to mount powerful attacks.

In this thesis, we further investigate existing and novel side-channel attacks to develop effective countermeasures against software-based side-channel attacks. My research focused on *finding novel side channels*, *reducing the requirements of existing attacks*, and *enabling attacks in environments which were considered too restricted*. Such an overall picture of what is actually required to mount attacks allows reasoning about effective and efficient defenses which target the root cause of a problem, and not single instances of attacks. We show that for certain attack classes, such as inter-keystroke-timing attacks, as well as for certain environments, such as specific variants of JavaScript, it is indeed feasible to prevent all known attacks and possibly even future attacks. By first identifying the root cause, we ensure that our proposed solutions only result in minimal performance overhead.

Figure 1.1 shows all the papers in context. The y-axis shows the required level of control over the execution. It spans from native code execution (bottom), over sandboxes (e.g., Enclaves, VMs, Containers), script languages (e.g., JavaScript), to full remote attack (*i.e.*, only network access to the victim is required). The x-axis provides a classification of how the paper advanced the state of the art. Papers on the left introduced a new side channel or class of attacks/defenses. Papers on the right side use an existing side channel and show how to reduce the requirements to mount an attack, e.g., how to be independent from a specific functionality. As papers can be a combination of both these factors, the classification is not binary.

1.1 Main Contributions

As one of the goals in the thesis, we strive to reduce the requirements of existing side-channel attacks further, allowing them to be mounted

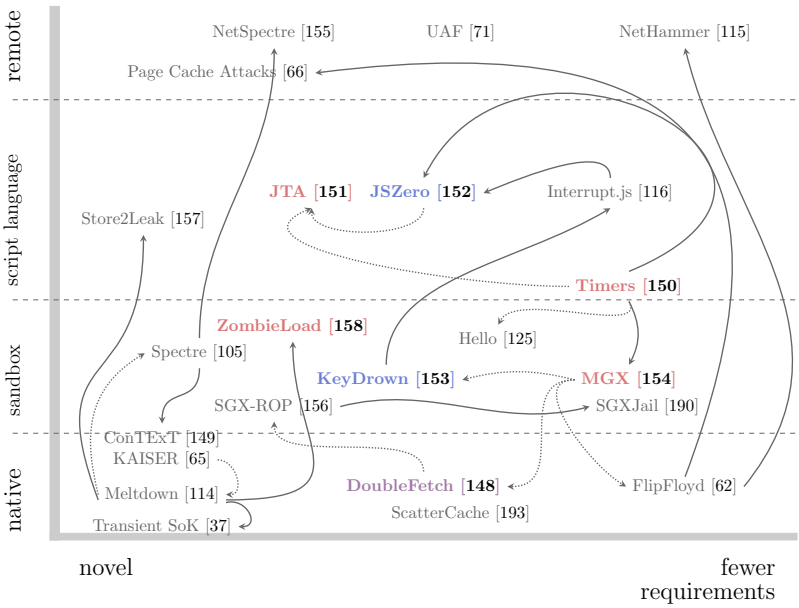


Figure 1.1: A relation between all the papers. Red and blue papers represent papers which are in the thesis, gray papers are co-authored papers which will not appear in the thesis. Blue represents defenses, and red represents attacks. Solid arrows indicate that ideas and techniques directly influenced a paper, whereas dotted arrows indicate an indirect or loose connection between papers.

from more restricted environments. We started working on reducing the requirements for the DRAM side channel [139]. Pessl et al. [139] showed that the DRAM can be used in a similar manner as the cache to transmit data covertly. By building on techniques from Gruss et al. [69], we managed to implement this covert channel in JavaScript. This enabled exploitation of the side channel directly from the browser. The paper was published at FC 2017 [150] in collaboration with Clémentine Maurice, Daniel Gruss, and Stefan Mangard.

The experience with cache eviction and sandboxes allowed us to implement Prime+Probe in the sandbox-like environment of Intel SGX. Intel SGX is designed to protect applications and their data in hostile environments using so-called enclaves. These enclaves run unprivileged code with further restrictions to prevent them from running harmful code. We were the first to show that this guarantee does not hold and that it is possible to mount cache attacks from within SGX [154]. To enable such

attacks, we combined the DRAM side channel [139] with our previously developed timing primitives [150]. Moreover, we showed that Intel SGX does not protect against software-based side channel attacks. The paper was published at DIMVA 2017 [154] in collaboration with Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard.

Due to the exact measurement methods used for Prime+Probe in SGX [154], we observed spikes in our timing measurements caused by keystrokes. We showed that these spikes can also be generated using artificial interrupts, which allowed us to build the first effective countermeasure against keystroke-timing attacks. By hiding the actual keystrokes among specially crafted noise, we prevented keystroke-timing attacks on both x86 and ARM. This paper was published at NDSS 2018 [153] in collaboration with Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard.

As a consequence of the novel attacks in JavaScript [116, 150], we presented a generic countermeasure to prevent side-channel attacks from the browser. With JavaScript Zero [152], we developed a framework which transparently modifies or replaces functionality in the JavaScript language, effectively preventing all known side-channel attacks without noticeable side effects. The paper provides a classification of all known side-channel attacks in JavaScript and identifies the required actions to prevent all of them and possibly even future attacks. The research was published at NDSS 2018 [152] in collaboration with Moritz Lipp and Daniel Gruss.

Double-fetch vulnerabilities are a special kind of race condition, where a privileged environment fetches data multiple times from an unprivileged environment, exposing the privileged environment to the risk of working on inconsistent data. We were the first to show that Flush+Reload can be leveraged to detect vulnerable code inside SGX enclaves and the kernel. We showed that Flush+Reload cannot only be used to detect double-fetch vulnerabilities, but also to reliably exploit them, which significantly advanced the state of the art for such exploits. Moreover, we showed that Intel TSX can be used to automatically prevent the exploitation of such bugs due to the properties inherent to transactional memory. The paper was published at AsiaCCS 2018 [148] in collaboration with Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard.

We investigated how much information JavaScript engines leak about the environment by developing a template attack which automatically finds differences in JavaScript properties influenced by the environment. In addition with 2 new side-channel attacks on the JavaScript engine, we

showed that attackers can in most cases detect the exact browser version and environment. This information can be used for fingerprinting, targeted exploitation, and tailoring side-channel attacks to the specific environment. The paper was published at NDSS 2019 [151] in collaboration with Florian Lackner, and Daniel Gruss.

1.2 Other Contributions

While working on covert channels, we came up with techniques to ensure error-free transmission for such channels, advancing the state of the art in reliability [125]. We showed that with techniques from wireless communication, cache-based covert channels can transmit data without errors. We demonstrated that it is possible to establish an SSH connection between two Amazon virtual machines by solely using the cache as the transmission medium. The paper was published at NDSS 2017 [125] in collaboration with Clémentine Maurice, Manuel Weber, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard.

In response to multiple side-channel attacks against KASLR [68, 78, 96], we designed and implemented a new technique for operating systems to manage virtual address spaces such that the kernel is better protected against these and similar attacks. Our design unintentionally mitigated Meltdown [114] as well and is now part of every modern operating system. The paper was published at ESSoS 2017 [65] in collaboration with Daniel Gruss, Moritz Lipp, Richard Fellner, Clémentine Maurice, and Stefan Mangard.

We showed that the keystroke-timing attack we prevented [153] is not only exploitable from native code, but also from JavaScript. By using previously discovered timing primitives [150], we were able to mount keystroke-timing attacks from the browser. The paper was published at ESORICS 2018 [116] in collaboration with Moritz Lipp, Daniel Gruss, David Bidner, Clémentine Maurice, and Stefan Mangard.

Our effort to show that Intel SGX can be abused to hide malicious software resulted in a novel Rowhammer variant which can be hidden inside SGX enclaves [62]. We were able to circumvent all published countermeasures against Rowhammer, showing that more research is still required to find effective countermeasures. Moreover, we showed that the memory encryption of Intel SGX can also be exploited for a denial-of-service attack. In a collaboration with Daniel Gruss, Moritz Lipp, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom, we presented several new techniques to make Rowhammer

attacks more stable and stealthy in the paper, which was published at S&P 2018 [62].

Use-after-free attacks are a class of attacks known for a long time in many programming languages. We generalized these attacks, showing that they apply to different scenarios. Moreover, we showed that email addresses can also introduce use-after-free attacks. The paper was published at AsiaCCS 2018 [71] in collaboration with Daniel Gruss, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp.

After our stronger kernel protection [65], we further investigated attacks on the kernel space. This led to the discovery that out-of-order execution on Intel CPUs allows circumventing the privilege definition in page-table entries, ultimately allowing an attacker to read arbitrary memory. The class of attacks we introduced with this paper is now widely known as Meltdown attacks. The paper was published at USENIX Security 2018 [114] in collaboration with Moritz Lipp, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg.

While working on Meltdown [114], we also identified security problems caused by branch prediction and the subsequent speculative execution. Leveraging these design problems also allowed reading the memory of other applications. The class of attacks we introduced with this paper is now widely known as Spectre attacks. This paper was published at IEEE S&P 2019 [105] in collaboration with Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, and Yuval Yarom.

With NetSpectre [155], we reduced the requirements for Spectre attacks [105], such that no local code execution is required anymore. We show that we can mount a variant of Evict+Reload over the network, and exploit Spectre gadgets in a remote attack. Furthermore, we show that Spectre does not require the cache to leak information by introducing a new covert channel leveraging SIMD instructions. This paper was a collaboration with Martin Schwarzl, Moritz Lipp, and Daniel Gruss and was published at ESORICS 2019 [155].

In addition to reducing the requirements for Spectre attacks [155], we also reduced the requirements for Rowhammer attacks, making it possible to mount them remotely [115]. This paper is currently in submission and is a collaboration with Moritz Lipp, Misiker Tadesse Aga, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster.

We showed that techniques from microarchitectural side-channel attacks are also applicable to the operating-system layer. As operating

systems also provide caches for pages loaded from the hard disk, we showed that we can mount cache attacks on these software caches. This paper was a collaboration with Daniel Gruss, Erik Kraft, Trishita Tiwari, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh and was published at CCS 2019 [66].

With Malware Guard Extension [154], we showed that side-channel attacks can be mounted from SGX. SGX ROP [156] continues with the idea of malicious enclaves and demonstrates that traditional malware can be hidden inside SGX enclaves. Due to the asymmetry in the memory-access model, SGX enclaves can mount return-oriented programming (ROP) attacks on the host to execute arbitrary code. This paper was a collaboration with Samuel Weiser and Daniel Gruss and was published at DIMVA 2019 [156].

After demonstrating malicious SGX enclaves [156], we proposed a generic defense similar to site isolation [171] and KAISER [65] to prevent attacks from a large class of malicious enclaves. Our defense, SGXJail, does not require any changes to existing enclaves. The paper was a collaboration with Samuel Weiser, Luca Mayr, and Daniel Gruss and was published at RAID 2019 [190].

With ScatterCache [193], we proposed a novel cache design which breaks the direct mapping between addresses and cache sets, making eviction-based attacks infeasible. Our cache design does not only prevent Prime+Probe, Evict+Time, and Evict+Reload, but also outperforms state-of-the-art caches for certain realistic workloads. The paper was published at USENIX Security 2019 [193] in collaboration with Mario Werner, Thomas Unterluggauer, Lukas Giner, Daniel Gruss, and Stefan Mangard.

Since our discovery of Meltdown [114] and Spectre [105], many transient-execution attacks were presented. We unified the naming scheme and classified all existing attacks, which led to the discovery of new attacks which were overlooked so far. We also classified and evaluated proposed and existing defenses against transient-execution attacks. The paper was published at USENIX Security 2019 [37] in collaboration with Claudio Canella, Jo Van Bulck, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss.

As most defenses against transient-execution attacks turned out to be incomplete, we proposed a novel generic defense based on annotating secrets and taint tracking. Our defense tackles the root cause by preventing secrets and values derived from secrets to be used in a transient execution. The paper is currently in submission [149] and is a collaboration with

Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss.

After showing that Meltdown can leak memory from the line-fill buffer [114], we showed with ZombieLoad [158] that this enables powerful attacks allowing to leak data across all privilege boundaries, such as processes, the kernel, SGX enclaves, and even virtual machines. The paper was published at CCS 2019 [158] and is a collaboration with Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss.

We showed another transient-execution attack exploiting store-to-load forwarding of the store buffer and its effects on the TLB. Exploiting a missing permission check on Intel CPUs, we can abuse the store-to-load-forwarding logic to spy on the TLB state of any address and break KASLR within a few milliseconds. The paper is currently in submission [157] and is a collaboration with Claudio Canella, Lukas Giner, and Daniel Gruss.

In another transient-execution attack, we showed that the store buffer can be exploited to leak previously written data. This enabled us to leak data written by the kernel such as AES keys. The paper is currently in submission [126] and is a collaboration with Marina Minkin, Daniel Moghimi, Moritz Lipp, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom.

1.3 Outline

This thesis consists of two parts. In the first part (Chapters 2 to 4), we present on overview of the topic of this thesis, consisting of background, state of the art, and conclusions.

Chapter 2 provides the background required for this thesis. Section 2.3 introduces side channels and microarchitectural side-channel attacks. Section 2.1 explains virtual memory. Section 2.2 explains various types of caches and how they work. Finally, Section 2.4 gives an overview of sandboxing and isolation.

Chapter 3 gives an overview on the state of the art. In Section 3.1, we discuss state-of-the-art microarchitectural attacks, including side-channel attacks and transient-execution attacks. In Section 3.2, we discuss proposed and implemented defenses against microarchitectural attacks.

In Chapter 4, we draw conclusions from our work and discuss future work.

In the second part (Chapters 5 to 10), we present the main contributions, *i.e.*, the publications which comprise this thesis.

In Chapter 5, we present new timing techniques for JavaScript, which were published as a conference paper at Financial Crypto 2017 [150]. In Chapter 6, we show how Intel SGX can be abused to hide cache attacks completely, which is a conference paper published at DIMVA 2017 [154]. In Chapter 7, we demonstrate an efficient technique to prevent keystroke-timing attacks, which is an NDSS 2018 conference paper [153]. In Chapter 8, we introduce a browser defense against all known microarchitectural and side-channel attacks in JavaScript, which is published as an NDSS 2018 conference paper [152]. In Chapter 9, we are the first to show benign cache attacks to detect double-fetch vulnerabilities, which was published as a conference paper at AsiaCCS 2018 [154]. In Chapter 10, we show how to automatically find side-channel information in browsers exploitable for attacks, which is a conference paper published at NDSS 2019 [151]. In Chapter 11, we show a transient-execution attack leaking data across all privilege boundaries, which is published at CCS 2019 [158].

2 Background

In this chapter, we provide the required background for this thesis. In Section 2.1, we first explain the concept of virtual memory. In Section 2.2, we explain caches in more detail, as cache attacks play a predominant role in microarchitectural attacks, either as an attack primitive itself, or as part of an attack. We explain the cache organization (Section 2.2.1), how data is managed in caches (Section 2.2.2 and Section 2.2.3) and describe caches of Intel x86 CPUs in detail (Section 2.2.4). In Section 2.3, we then define side channels (Section 2.3.1) and microarchitecture (Section 2.3.2), and discuss the general idea of side-channel and microarchitectural side-channel attacks (Section 2.3.3). Finally, Section 2.4 provides background on restricted environments, such as sandboxes and trusted-execution environments.

2.1 Virtual Memory

With the rise of multi-processing, it became necessary to isolate different processes, both from each other as well as from the operating system. Thus, processes nowadays do not work directly on physical addresses, *i.e.*, addresses directly referring to the main memory, but instead on virtual addresses. With virtual addresses, each process has its own virtual address space which does not interfere with other processes. The processor

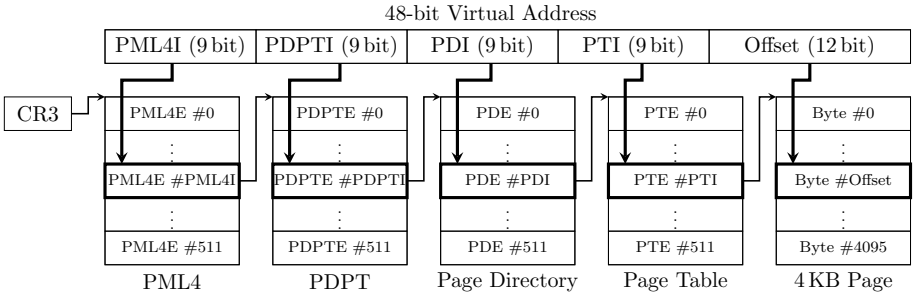


Figure 2.1: On x86_64, every process has a 4-level page-table hierarchy used for the translation from virtual to physical addresses by the MMU. The `CR3` register points to the first level of the page tables. The virtual address is split into parts which are used to index the page tables.

translates every virtual address of a process to a corresponding physical access.

The translation from virtual to physical addresses relies on multi-level page tables, which are defined per process. These page tables define a process-specific mapping with a granularity of typically 4 KB, mapping virtual pages to physical pages. In addition to this mapping, the page tables also define permissions for every virtual page.

On 64-bit x86 CPUs, a virtual address has 48 bit and the CPU uses 4 levels of page tables for the translation from virtual to physical addresses. An extension to 57-bit virtual addresses and 5 levels of page tables is specified for newer processors and already supported by Linux. Figure 2.1 illustrates the multi-level page-table hierarchy. The first level, the Page-Map Level 4 (PML4) is referenced by the process-specific processor register `CR3`. The PML4 is basically an array consisting of 512 PML4 entries, each of them 64 bit wide. Every entry contains several flags, e.g., whether the entry is valid and thus refers to the next page-table level, the Page-Directory Pointer Table (PDPT). The index into the PML4, *i.e.*, which PML4 entry is used, is determined by bits 47 to 39 of the virtual address. The PDPT follows the same structure as the PML4, with 512 PDPT entries. Starting from the PDPT, each entry specifies whether it directly maps a physical page or whether it points to the next level of the page tables. The PDPT entry used for the translation is determined by bits 38 to 30 of the virtual address. If the PDPT entry directly maps a physical page, the size of the corresponding page is 1 GB, and the remaining 30

bits of the virtual address are used as an offset into this page. Otherwise, bits 21 to 29 of the virtual address are used to select the entry in the next level, the Page Directory (PD). If the PD entry directly maps a physical page, the size of the corresponding page is 2 MB, and the remaining 21 bits of the virtual address are used as an offset into this page. Otherwise, bits 20 to 12 of the virtual address are used to select the entry in the next level, the Page Table (PT). The remaining 12 bits of the virtual address are used as an offset into the 4 KB page referenced by the PT entry.

As all paging structures are stored in memory, caches (cf. Section 2.2) are used to reduce the number of memory accesses. Furthermore, the Translation Lookaside Buffer (TLB) is a separate cache which caches the result of recent translations from virtual to physical addresses.

2.2 Caches

In this section, we discuss how caches work. In Section 2.2.1, we describe how caches are organized. Section 2.2.2 gives more details about how data is stored in caches. Section 2.2.3 describes cache-replacement policies of modern CPUs. Finally, Section 2.2.4 looks explicitly at caches in Intel x86 CPUs.

2.2.1 Cache Organization

While the performance of CPUs increased, the performance of the main memory (DRAM) did not increase with the same rate. Thus, DRAM is the bottleneck for computation. As a consequence, caches were introduced to get rid of this bottleneck.

Caches are small and fast buffers between the CPU and the DRAM. Thus, all memory accesses go through the cache. Caches keep copies of recently used data. If a memory access can be served from the cache, this is called a *cache hit*. If the data for the memory access is not in the cache, this is a *cache miss*, and it has to be served from DRAM.

Caches are usually organized in a *cache hierarchy* as illustrated in Figure 2.2 for Intel CPUs. The fastest and smallest caches are directly connected to the CPU core and are usually private to the core. The further away caches are from the CPU, the larger and slower they are. Caches are typically grouped into *cache levels*. The cache closest to the CPU is called first-level (L1) cache, and it is usually followed by a second-level (L2) cache. The last-level cache (LLC) is the slowest and largest cache level and it is often shared among CPU cores.

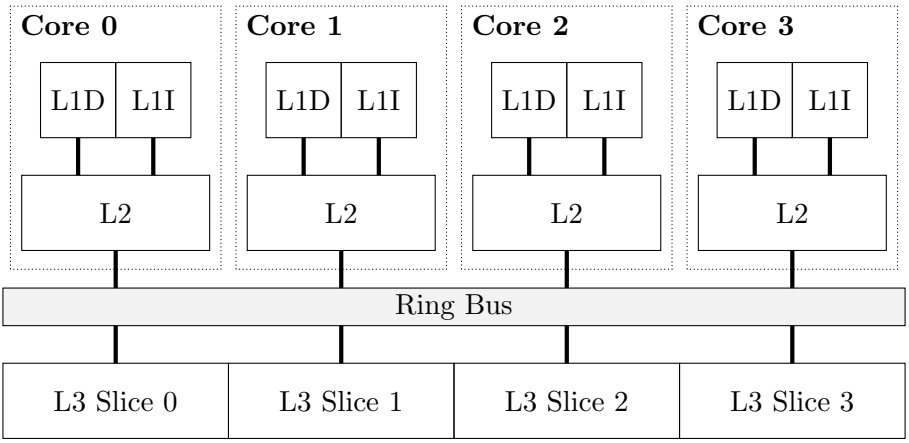


Figure 2.2: The cache hierarchy on modern Intel CPUs. Every CPU core has a private L1 cache which is statically split into an L1 instruction (L1I) and L1 data (L1D) cache, and a private unified L2 cache. The L3 cache (LLC) is split into slices. Every core can access one slice directly and the other slices via a ring bus.

Inclusiveness. If a hierarchy of caches exists, these caches can either be *inclusive*, *non-inclusive* or *exclusive* with respect to other cache levels. An inclusive cache includes all data of the other cache levels to which the inclusiveness property refers to. For example, if the L3 is inclusive to the L1 cache, all data stored in the L1 cache must also be stored in the L3 cache. Exclusive caches ensure that data is always stored in exactly one of the mutually exclusive cache levels. Non-inclusive caches do not provide such strict guarantees. Data in non-inclusive caches might also be in other cache levels.

2.2.2 Set-associative Caches

Most modern processors use set-associative caches as illustrated in Figure 2.3. There, the cache is divided into *cache sets* which are further divided into *cache ways*. Each cache way in a cache set is called a *cache line*.

The actual data is stored within the cache line which is also *tagged*. The tag is used to check whether a cache way contains the requested data and not unrelated data mapping to the same cache set.

Both the cache set and the tag are derived from the address of the data, whereas the cache way is determined by the cache-replacement policy

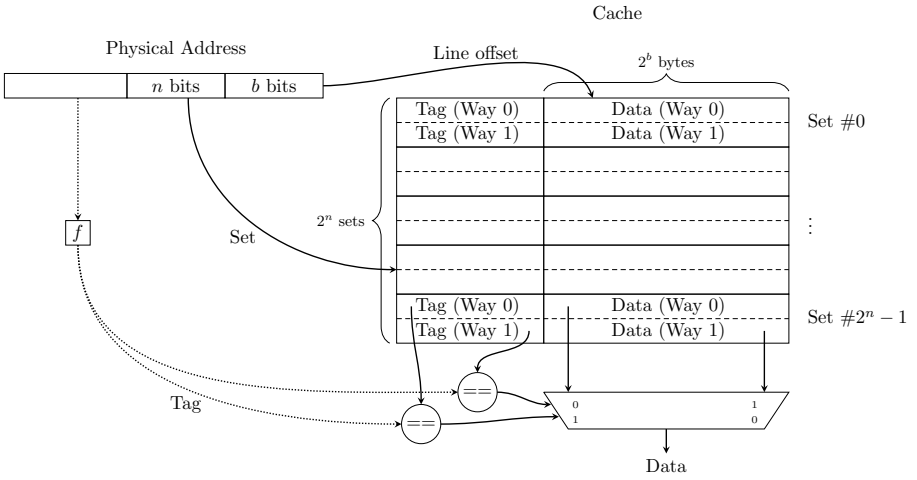


Figure 2.3: A simple example of a two-way set-associative cache. The cache set is determined by n bits of the memory address, the cache way is determined by the cache-replacement policy. The lowest b bits of the address are only used to address the byte inside a cache line. The highest bits of the address are used as the tag.

(cf. Section 2.2.3). Different cache designs use different combinations of virtual and physical address to calculate the set and tag. Mainly, there are 2 types of caches: virtually indexed and physically tagged (VIPT), as well as physically indexed and physically tagged (PIPT).

2.2.3 Cache-replacement Policies

As the size of the cache is limited and compared to the DRAM extremely small, data in the cache has to be regularly replaced with data from the DRAM. This is automatically done by the cache, transparent to the applications.

For set-associative caches, the cache set is determined by a fixed set of bits of the address of the memory access. For addresses mapping to the same set, the cache-replacement policy decides in which cache way the data is stored, and which cache way is replaced with the newly fetched data.

As the cache-replacement policy has a significant impact on the overall system performance, it is usually considered intellectual property and thus not disclosed by CPU vendors. Common cache-replacement policies are

least-recently used (LRU) and pseudo-random. These were used in older Intel microarchitectures and ARM CPUs respectively.

Least-recently used keeps track of when the data in a cache way was last accessed, and always replaces the cache way that has not been accessed for the longest time. While this policy is relatively straightforward, it is not so easy to implement. It also requires additional metadata to keep track of the last access time. A pseudo-random policy is much simpler to implement, as no additional metadata is required, and the performance is not much worse.

Newer Intel processors use more sophisticated policies, which only degrade to LRU in a worst-case scenario. Such adaptive policies are Bimodal Insertion Policy (BIP) [142] or Quad-age LRU [95], a pseudo-LRU variant which is easier to implement than LRU.

2.2.4 Caches on Intel x86 CPUs

The cache hierarchy of most Intel x86 CPUs is comprised of three levels, as illustrated in Figure 2.2. The L1 cache is statically split in half into an L1 data cache and an L1 instruction cache. The L1 cache is private to one physical core and hence only shared among sibling hyperthreads. The L2 cache is a unified cache that contains both data and instructions. The L2 is also private to one physical core.

The LLC is split into slices and shared among all cores of the CPU. Every physical core has direct access to one of the LLC slices, and access to all other slices through the bus. To balance the load on the cache slices, the CPU uses a simple hash function [124] to calculate the slice from the physical address.

Until Skylake-X, the LLC is also an inclusive cache, meaning that all data stored in the L1 and L2 is also stored in the LLC. The cache design of the LLC changed with Skylake-X to a non-inclusive cache.

2.3 Side-Channel and Microarchitectural Attacks

Side-channel attacks are a class of attacks which do not directly attack an application. Such attacks rather observe side effects of applications to infer the processed data. While hardware-based side-channel attacks, e.g., power-analysis attacks, were long known, they were long not seen as a threat to general-purpose commodity systems. The reason for this is that they require an attacker to have physical access to the target as well as sophisticated measurement equipment. However, recently, it was shown

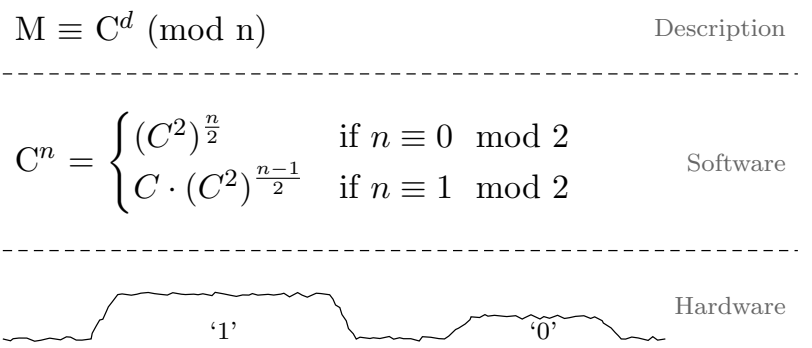


Figure 2.4: Different abstraction layers of a cryptographic algorithm (RSA decryption). The mathematical description is proven to be secure, however the software leaks timing information when e.g., implemented using the non-constant-time square-and-multiply algorithm. On the hardware level, there can even be leakage for constant-time algorithms due to electrical properties of the hardware.

that side-channel attacks can also be mounted purely with software. This reduced the requirements to local execution of unprivileged code.

In this section, we first define what we consider a side channel. Then, we describe why the implementation of modern processors makes them vulnerable to side-channel attacks, which are exploitable from software.

2.3.1 Side Channels

Side channels arise from implementation details, which are often caused by *abstraction layers*. A simple example of an abstraction layer which introduces side-channel leakage is illustrated in Figure 2.4 using a cryptographic algorithm. The algorithm is mathematically proven to be secure, and the software implementing the algorithm is free of software errors. However, when implementing the algorithm in software, the runtime of the algorithm depends on which bits in the secret keys are set. This arises from the property that not all (mathematical) operations have the same execution time on modern CPUs. The mathematical description is only an abstraction of the actual software implementation, and this abstraction introduces the side-channel information.

In this example, a side channel can also exist in a different layer. Specifically, the abstraction of the instruction-set architecture (ISA) and the actual hardware implementation. Even if the same instructions are

used for different key bits, the power consumption can be correlated with the number of bits set in the operand of the instructions.

Side-channel attacks exploit the leakage of data from such side channels. In contrast to traditional attacks, which target, e.g., algorithms, protocols, or implementation errors, side-channel attacks assume bug-free and correct implementations. They often attack a layer which exists due to the combination of abstraction layers and is thus not directly considered in any specification.

2.3.2 Microarchitecture

The continuous performance optimizations of modern CPUs also lead to increasing complexity of CPUs. While the ISA defines an abstraction of the CPU, *i.e.*, the CPU architecture, it does not define the actual implementation. The implementation details of an ISA are defined by the *microarchitecture* and vary between different CPUs which implement the same ISA.

The microarchitecture is typically not visible to the programmer and thus often not fully documented in detail. The same ISA can be implemented in different ways, leading to different microarchitectures. For example, Intel's implementation details, *i.e.*, the microarchitecture, of the x86 ISA differ significantly between, e.g., their Core, Xeon, and Atom CPUs. Similarly, AMD has different implementations, e.g., Bulldozer, Zen, or Bobcat. Moreover, microarchitectural changes do not only exist between different CPU types (e.g., desktop, server) but also between different versions of the same CPU type. Thus, even if architectural parameters of a CPU are the same (e.g., the same number of cores and same clock speed), the performance can differ significantly due to microarchitectural optimizations, e.g., branch prediction or out-of-order execution.

Although the microarchitecture is implementation specific, there are various *microarchitectural elements* which are widely deployed as they improve the performance of CPUs. Such elements include pipelines with out-of-order execution to parallelize execution, caches to reduce the latency of repeatedly used data (cf. Section 2.2), and various predictors to reduce the stalling time of CPUs. Some of these elements have such a significant impact on the performance that their parameters are explicitly stated for the CPU, e.g., the number and respective sizes of caches. While improving the performance of CPUs, they again introduce an abstraction layer in CPUs, as their side effects are not specified in the ISA.

A programmer as well as a compiler, however, are supposed to only adhere to the specification of the ISA, and should not care about the microarchitectural details. Taking the microarchitecture into account when writing programs does not only impair the portability of the applications. The ISA defines the guaranteed functionality of the underlying CPU and is thus an abstraction of the underlying microarchitecture. Both a developer as well as a user can rely on the stability of the ISA specification, making programs compatible with all CPUs correctly implementing the ISA. The microarchitecture can and does change without notice, thus relying on specific microarchitectural properties reduces the portability of an application. The same is true for microarchitectural elements – developers cannot rely on the existence of specific microarchitectural elements, such as the cache.

Hence, the microarchitecture has to be as transparent as possible, *i.e.*, it has to perform all optimizations without explicit help and even knowledge of the developer. While this is beneficial for the performance of CPUs, it again introduces side channels. For any optimization with a visible performance impact, there is information leakage if the optimization depends on data or meta data. That is, if any operation uses fewer resources (e.g., power, time) for specific inputs, the observation of these optimizations allows an attacker to infer information about the input. Such information leakage is exploited in microarchitectural attacks.

2.3.3 Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks exploit information leakage, which is specifically introduced by the microarchitectural implementation of a CPU. In contrast to traditional side-channel attacks, microarchitectural side-channel attacks are typically exploited solely from software.

The microarchitecture is designed to improve the performance of applications. As performance is an essential factor for many applications, there are several functionalities built into modern CPUs to measure the performance of applications. There are both architecturally defined measurement methods as well as microarchitecture-specific measurement methods. Architecturally-defined methods include high-resolution time-stamp counters which are exposed via the unprivileged `rdtsc` instruction on x86. Microarchitectural methods include performance counters which are exposed via special CPU registers, so-called model-specific registers (MSRs). The granularity of microarchitectural measurement methods is often even below the instruction level, allowing developers to measure

performance bottlenecks caused by the microarchitecture. The granularity of architectural measurement methods is usually limited by the CPU speed (e.g., time-stamp counters), or the instruction level (e.g., exceptions).

While performance-measurement methods are obviously necessary to measure the performance, they can be abused for microarchitectural side-channel attacks. Specifically, in a microarchitectural side-channel attack, an attacker application tries to infer information from a victim application by monitoring side channels.

Microarchitectural side-channel attacks typically rely on the existence of a microarchitectural element with the following properties:

$\mathcal{P}$1 It is shared between attacker and victim application.

$\mathcal{P}$2 Its state changes based on the processed data.

$\mathcal{P}$3 The state can be inferred from (a combination of) side channels.

Microarchitectural elements only affect applications which use the microarchitectural element. Thus, if the attacker and victim application do not use the same microarchitectural element (**$\mathcal{P}$1**), the attacker application cannot infer anything about the victim application via the state of the microarchitectural element.

Property **$\mathcal{P}$1** generally defines the scope of a microarchitectural side-channel attack as illustrated in Figure 2.5 for a modern Intel x86 CPU. Some microarchitectural elements are private to the CPU core (e.g., registers), shared among hyperthreads (e.g., L1 caches), shared among all CPU cores (e.g., some last-level caches), or even shared among all CPUs (e.g., main memory). Thus, the domain in which the element is shared defines the domain in which the attacker can mount an attack.

Some microarchitectural elements optimize the performance of an application independently of the processed data. An example for this is a microarchitectural element which implements constant-time AES encryption and decryption. However, many microarchitectural elements achieve different performance optimizations depending on the processed data (**$\mathcal{P}$2**). For example, a data cache reduces the access latency for recently used data, *i.e.*, data the application used before. For these elements, the current internal state is influenced by previously processed meta data and data, and observing the internal states leaks information.

As microarchitectural elements usually do not provide an interface to query their state, an attacker can only observe the internal state via side channels (**$\mathcal{P}$3**). In many microarchitectural side-channel attacks, the timing is used as a side channel. That is, based on the execution time

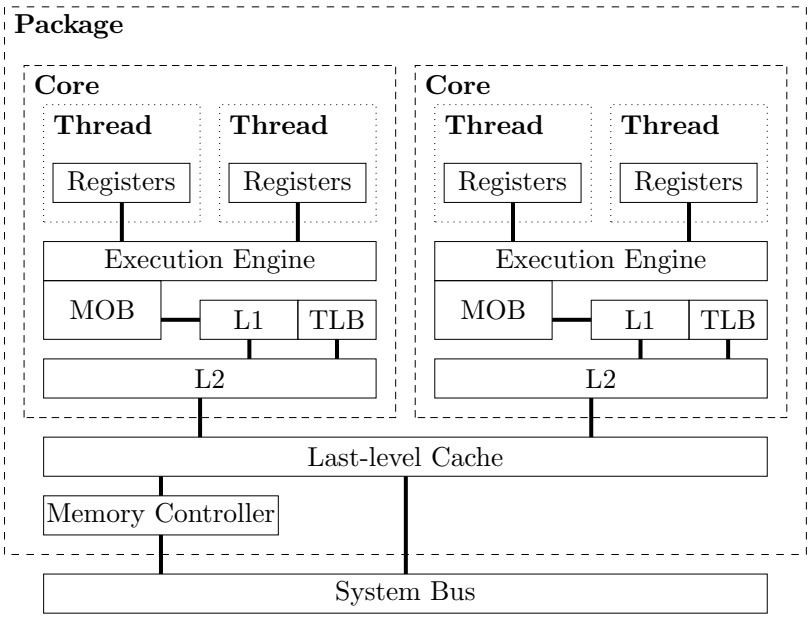


Figure 2.5: The multiple scopes of shared resources: private per thread, shared among threads, shared across cores, and global resources visible to the entire system.

of an operation, an attacker can infer information on the internal state of a microarchitectural element. The execution time often depends on whether the element can optimize a specific operation, or cannot optimize a specific operation anymore due to actions of the victim application (e.g., memory access, control-flow prediction).

Sometimes the timing cannot directly be measured, but an indirection is required to observe side-channel information. For example, an attacker can infer information about the internal state of a control-flow predicting mechanism such as the branch predictor by observing the memory access time of subsequent instructions.

Summarizing, for microarchitectural side-channel attacks, an attacker relies on a microarchitectural element fulfilling $\boldsymbol{\mathcal{P}1}$, $\boldsymbol{\mathcal{P}2}$, and $\boldsymbol{\mathcal{P}3}$. Then, the observed leakage can be used to infer information about a victim application.

2.4 Sandboxing and Isolation

Sandboxes provide a restricted environment for executing typically untrusted code. A sandbox strictly controls the resources available to the code running inside the sandbox. This control can be enforced by the hardware, operating system, runtime environment, or design of the language itself. While there are multiple different types of sandboxes, we focus mostly on JavaScript in this thesis.

Isolation is orthogonal to sandboxing. Sandboxes solve the problem of running untrusted code inside a trusted environment. Isolation, however, solves the problem of running trusted code inside an untrusted environment. As the hardware is usually assumed to be trusted, isolation is often enforced by the hardware, e.g., process and kernel isolation, trusted execution environments, or hardware security modules. In this thesis, we focus mostly on Intel SGX as an isolation technique as it is commonly available on modern Intel CPUs.

2.4.1 JavaScript

JavaScript is a scripting language extensively used on the web and thus supported by most modern browsers [183]. JavaScript programs are generally untrusted and executed automatically when opening a website.

To prevent any harm to the system, scripts are only run inside the JavaScript sandbox. Furthermore, the language itself already provides good sandboxing by only providing limited functionality. JavaScript

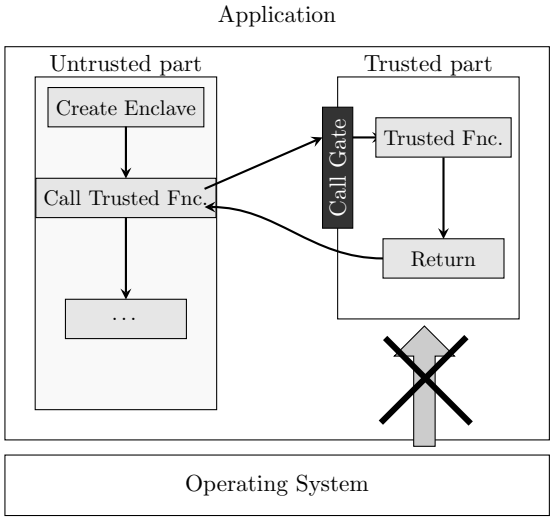


Figure 2.6: In the SGX model, applications are split into a trusted (enclave) and an untrusted (host) part. The hardware prevents any access to the trusted part. The only communication between enclave and host is via predefined ECALLs and OCALLs.

does not expose the concept of memory addresses and pointers to the programmer, and there is no interface to the operating system, e.g., syscalls.

Hence, while JavaScript is designed to prevent traditional exploits, its limitations also impede microarchitectural attacks. In most browsers, there are no high-resolution timers available anymore [150]. Thus, observing timing differences in JavaScript is not straightforward.

2.4.2 Intel SGX

Intel Software Guard Extension (SGX) is an x86 instruction-set extension introduced with the Skylake microarchitecture for isolating trusted code [83]. With Intel SGX, applications are split into a trusted enclave and an untrusted application (cf. Figure 2.6). The CPU fully isolates the trusted enclave, and neither the application nor the operating system can access the enclave's memory. Furthermore, to protect against bus-probing attacks on the DRAM bus and cold-boot attacks, the memory range used by SGX is encrypted via transparent memory encryption.

The application and enclave can only communicate through a well-defined interface. Using the `eenter` function, applications can call func-

tions provided by the enclave. The hardware prevents any other access to the enclave. In the attacker model of Intel SGX, only the hardware is trusted. All software, including the operating system, is assumed to be compromised and, therefore, untrusted.

Although SGX enclaves run native code, there are several restrictions for enclaves to reduce the attack surface [83]. Enclave code cannot use any I/O operations, including syscalls. Thus, any communication with the operating system is only possible by using the untrusted application as a proxy. Moreover, certain other instructions are not supported, such as `rdtsc`. Hence, Intel SGX also impedes microarchitectural attacks.

3 State of the Art

In this chapter, we discuss state-of-the-art microarchitectural attacks and defenses. In Section 3.1, we discuss microarchitectural side-channel attacks and transient-execution attacks. In Section 3.2, we discuss the defenses against microarchitectural side-channel attacks and transient-execution attacks.

3.1 Software-based Microarchitectural Attacks

Software-based microarchitectural side-channel attacks exploit leakage from the state of microarchitectural elements. The predominant microarchitectural element for attacks is the cache, as caches are well documented, encode a large state, and their state is comparably easy to observe in software. However, there are also state-of-the-art attacks on different microarchitectural elements, namely predictors and DRAM. Moreover, transient-execution attacks are a novel class of extremely powerful microarchitectural attacks using microarchitectural side channels as a building block.

3.1.1 Cache Attacks

Cache attacks exploit the fundamental property of caches that data residing in the cache can be accessed faster than data not residing in the cache.

There are different methods of how an attacker can abuse this property to infer whether a specific memory location resides in the cache. Cache attacks exploit the observation that the information whether a specific memory location is in the cache often correlates with the activity of the victim application, thus leaking information about the victim application.

Cache attacks can be divided into three main categories. For the first type of cache attacks (Evict+Time), the attacker modifies the cache state and monitors the runtime of the victim. In the second type of cache attacks (Prime+Probe), the attacker brings the cache into a known state and monitors whether the victim execution influenced this known state without directly observing memory accesses of the victim. In the third type of cache attacks (Flush+Reload), the attacker measures cache-state changes directly on memory which is shared with the victim, e.g., shared libraries.

Evict+Time

The first cache attacks [27, 136] targeted cryptographic algorithms and were generalized as Evict+Time by Osvik et al. [132]. With Evict+Time, an attacker manipulates the cache state by evicting a specific cache set of the victim from the cache. Then, the attacker monitors the runtime of the victim, trying to detect timing differences in the execution time. If the attacker measures a difference in the runtime, the attacker can conclude that the victim accessed data which maps to the evicted cache set.

Evict+Time has been used to attack OpenSSL AES on x86 [132, 174], as well as on mobile ARM platforms [113, 163]. Hund et al. [78] exploited Evict+Time to de-randomize the kernel address space, and Gras et al. [60] exploited it to break ASLR from JavaScript.

Eviction sets To target a certain cache set for the eviction, eviction-based attacks require an eviction set and an eviction strategy. By accessing the addresses of the eviction set with a specific strategy, the current data of the targeted cache set is evicted from the cache.

The cache set is typically directly determined by several bits of the physical address. On CPUs without cache slices, this makes it easy to generate the eviction set [113, 132, 163]. Modern x86 CPUs[1] additionally partition the last-level cache using cache slices. The mapping from physical addresses to cache slices is not documented. However, it has been reverse

[1]Intel introduced cache slices with the Sandy Bridge microarchitecture

engineered for multiple CPUs using performance counters [124] and timing measurements [64, 78, 81, 92, 119, 202].

Nowadays, physical addresses are not exposed to an unprivileged attacker anymore. Hence, creating eviction sets is not straightforward. State-of-the-art approaches rely on a combination of static and dynamic approaches to find eviction sets [181]. Several attacks exploit large 2 MB pages [69, 91, 119, 125, 150]. There, the least-significant 21 bits of the physical and virtual address are the same, which is sufficient to determine the cache set. More generic variants only rely on timing and can even be mounted from JavaScript [32, 60, 131, 181]. We show that the time to generate an eviction set can be further improved by combining timing information with side-channel information from the DRAM [154].

For older CPUs, it is sufficient to simply access all addresses in the eviction set. However, modern CPUs use more complex cache-replacement policies, requiring different strategies for accessing eviction sets. Gruss et al. [69] and Briongos et al. [36] present different eviction strategies for multiple CPUs and methods to find and evaluate such strategies.

Prime+Probe

Prime+Probe was also first used on cryptographic algorithms [27, 136] and generalized by Osvik et al. [132]. It is a more generic, and thus more powerful, cache attack, as it does not require the indirection of measuring the victim's execution time. Similarly to Evict+Time, Prime+Probe also requires eviction of a specific cache set.

Figure 3.1 illustrates the steps of a Prime+Probe attack. The basic idea of Prime+Probe is to populate ("prime") a cache set with attacker-controlled data (①). If the victim accesses data that falls into this cache set, the CPU has to evict the attacker data and replace it by the victim data (②). Then, the attacker probes the cache state by priming it again and measuring how long it takes (③). If the victim caused any attacker-controlled data to be evicted from the cache set, this step takes longer as compared to the case where the cache set is still populated with only attacker-controlled data.

Prime+Probe attacks have first been demonstrated on the L1 cache [136]. Most Prime+Probe attacks on the L1 cache target cryptographic algorithms [56]. Attacks have been demonstrated on both the L1 instruction cache [2, 5, 7, 209] as well as the L1 data cache [1, 3, 4, 30, 174].

With the reverse engineering of the cache slicing functions, in addition to the inclusiveness of the last-level cache, Prime+Probe attacks became

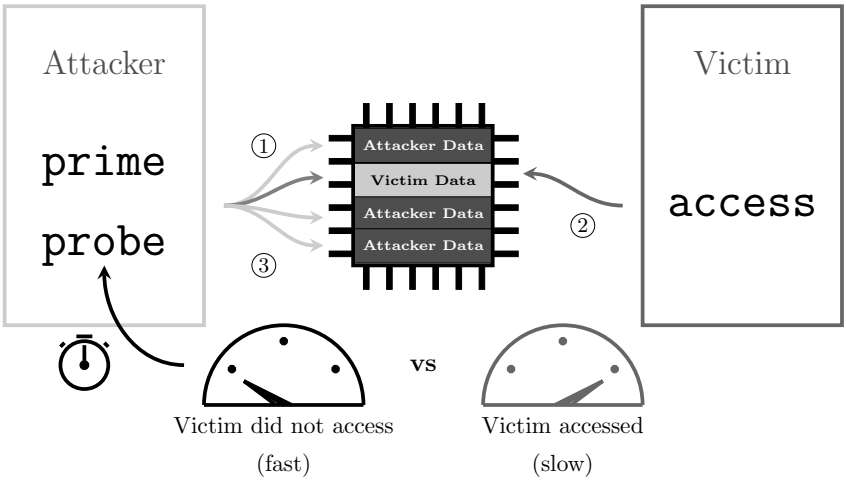


Figure 3.1: In a Prime+Probe attack, the attacker fills a target cache set with data ("prime"). If the victim accesses data in this cache set, the attacker data is evicted. The next time the attacker fills the cache set ("probe"), the time depends on whether the victim evicted the attacker data.

possible on the last-level cache as well. Ristenpart et al. [144] were the first to mount a Prime+Probe attack on the last-level cache on older CPUs. Maurice et al. [123] presented a Prime+Probe covert channel on modern Intel CPUs. Similar to attacks on the L1 cache, most Prime+Probe attacks exploiting the last-level cache target cryptographic algorithms, such as AES [91, 97, 113] or ElGamal [80, 119]. Oren et al. [131] presented the first Prime+Probe attack from JavaScript to spy on user behavior. Genkin et al. [57] showed that Prime+Probe attacks are also possible from PNaCl and WebAssembly. With Multi-Prime+Probe [153], we improved state-of-the-art Prime+Probe-attacks by spying on multiple cache sets in parallel, enabling reliable attacks on user input.

Prime+Probe attacks on the last-level cache have also been studied in cloud scenarios, both as side channels [209] as well as covert channels [125, 197, 198].

As Prime+Probe attacks do not require any form of shared memory or access to the victim, they have also been used to attack trusted execution environments such as Intel SGX. In concurrent work, Brasser et al. [33], Moghimi et al. [127] and Götzfried et al. [59] showed how a malicious operating system can leverage Prime+Probe to leak secrets from SGX. In

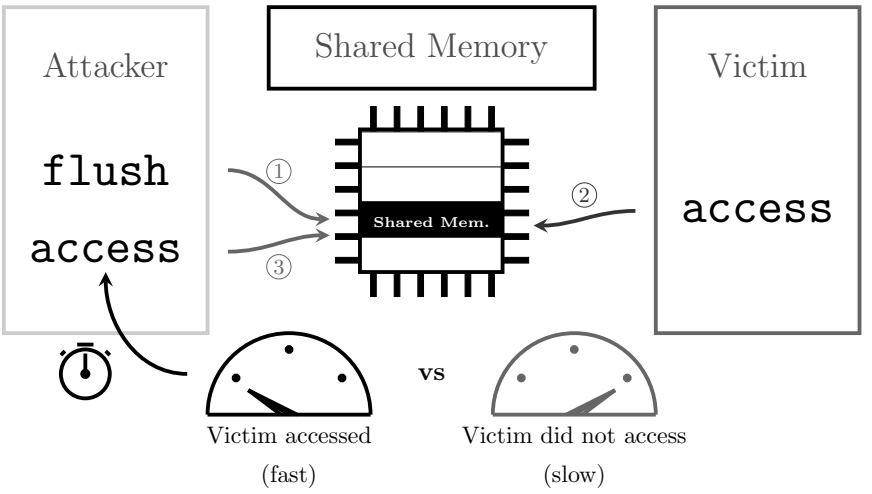


Figure 3.2: In a Flush+Reload attack, the attacker flushes a shared cache line out of the cache ("flush"). If the victim accesses the cache line, it is brought back into the cache. A subsequent access of the attacker ("reload") is then faster.

addition, we also showed that Prime+Probe can be mounted from inside an SGX enclave, showing the first malicious enclave [154].

In addition to Prime+Probe as an attack by itself, it has also been used as a building block for transient-execution attacks [173].

Flush+Reload

Gullasch et al. [73] described the first flush-based attack, which led to the generic Flush+Reload attack introduced by Yarom et al. [201]. Flush+Reload is considered an extremely powerful cache attack, as it is basically noise-free and has cache-line granularity.

Flush+Reload exploits the fact that on x86, the `clflush` instruction is an unprivileged instruction which allows flushing a specific cache line from the cache. Moreover, as the cache works with physical addresses, shared memory is only once in the cache. Thus, flushing it in one process, flushes it for all processes.

Figure 3.2 illustrates the Flush+Reload attack. The attacker targets a memory region shared with the victim, e.g., a shared-memory segment. First, the attacker flushes the shared memory location from the cache using the `clflush` instruction (①). If the victim accesses the shared

memory location, it is cached again (②). Then, the attacker measures how long it takes to access the shared memory location (③). If it is fast, the attacker learns that the victim has accessed it. Otherwise, the attacker knows that the victim has not accessed the shared memory location.

Flush+Reload does not require knowledge of physical addresses, as `clflush` uses virtual addresses. Moreover, there is no eviction required. The target can be directly flushed from the cache. As both the access and the flushing are reliable operations, and data is unlikely to be cached by accident (e.g., due to hardware prefetching or misspeculation), Flush+Reload is an extremely reliable cache attack.

As Prime+Probe and Evict+Time, Flush+Reload has been used for attacks on cryptographic algorithms [10, 12, 26, 55, 63, 73, 89, 93, 137, 140, 200, 201] as well as on user behavior [63, 113, 185, 209].

Due to its robustness, Flush+Reload is also used as a building block for other attacks, mainly as the covert channel for transient-execution attacks [37]. We showed that Flush+Reload can also be used for benign use cases, such as detecting double-fetch bugs [148].

Flush+Flush Gruss et al. [64] demonstrated a variant of Flush+Reload which exploits timing differences of the `clflush` instruction. Hence, instead of measuring how long it takes to reload the data (cf. Figure 3.2, ③), the attacker measures how long it takes to flush the data.

Flush+Flush has been used to build attacks on cryptographic algorithms [35, 64], user behavior [64], and page tables [177].

Evict+Reload In certain environments, e.g., in JavaScript, there is no instruction to flush a specific cache line. Evict+Reload is a variant of Flush+Reload, where the flush (cf. Figure 3.2, ①) is replaced by eviction [63, 113].

Evict+Reload enabled Flush+Reload-type attacks on ARM processors without a flush instruction [113]. Moreover, Evict+Reload is used to mount cache attacks on the own process from JavaScript, which does not provide access to the flush function [60, 105, 146, 157].

TLB Attacks

Besides attacks on data and instruction caches, there are also attacks exploiting timing differences in the page-translation caches, *i.e.*, TLBs. As with data and instructions caches, TLBs also expose measurable timing differences for cached and uncached translations. Hence, an attacker can

deduce from the time it takes to read from an address whether the address translation is present in the TLB.

These timing difference have been exploited to de-randomize the kernel address space [68, 78, 96, 157], attack cryptographic algorithms [61], and spy on user behavior [157]. Van Schaik et al. [178] leverage the MMU for building cache-eviction sets by exploiting that page tables are cached, and that the MMU accesses these cached page tables in a deterministic way.

3.1.2 Attacks on Predictors

Modern CPUs use several prediction mechanisms to avoid pipeline stalls. These predictors include branch predictors, prefetchers, and memory-aliasing predictors. As predictions are based on previously observed data, an attacker can often infer information from observing the predictions.

Branch predictors were first exploited by Acıiçmez et al. [6, 8] by measuring differences in the execution time of a victim on branch mispredictions. They also presented a variant in which a victim's update to the branch predictor evicts one of the spy branches, leading to a misprediction in the spy. This is even the case for victims running inside SGX enclaves [51, 111]. Cock et al. [44] observed that the number of branch mispredictions can be monitored through a cycle counter on ARM. Evtyushkin et al. used the branch-prediction side channel to build a covert channel [52] and to break ASLR [53]. With Spectre [105], we showed that branch predictors can also be leveraged for transient-execution attacks to leak actual data.

In addition to branches, CPUs also predict future memory accesses and already cache the predicted memory locations ahead of time. Wang et al. [184] reverse engineered the prefetcher on the Intel Atom in-order CPUs to reduce the prefetcher interference when mounting Prime+Probe attacks. Prefetchers have also been reverse engineered on out-of-order Intel CPUs and exploited to attack cryptographic algorithms [28, 161]

Another prediction mechanism used for microarchitectural attacks is the memory-aliasing prediction, also known as memory disambiguation. This prediction mechanism in out-of-order CPUs tries to ensure that a load (partially) depending on a previous store gets its value from the most recent store, and not a stale value from the cache. Modern CPUs use store-to-load forwarding, where the the store buffer is used to directly forwarded a store to the respective load. Mispredictions in the store-to-load forwarding logic were exploited to build a covert channel [166] and to learn physical addresses [94]. We showed that the store-to-load forwarding

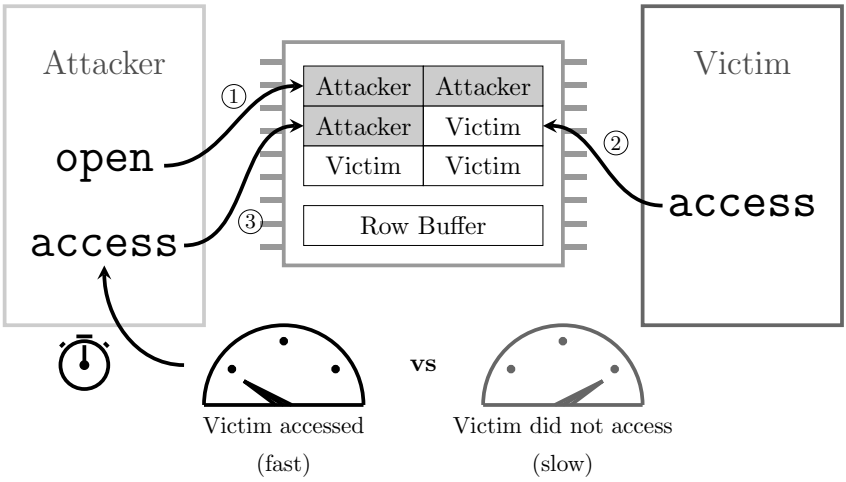


Figure 3.3: In a DRAMA attack, the attacker exploits that DRAM banks are shared among attacker and victim, and that the row buffer acts as a cache.

misses permission checks, allowing to de-randomize the kernel address space and break ASLR from JavaScript [157].

3.1.3 DRAM Attacks

The DRAM is another microarchitectural element which can be abused for microarchitectural side-channel attacks. Especially as the DRAM is shared among multiple CPUs, DRAM-based attacks can be mounted across CPUs. DRAM modules contain row buffers which act as caches for the rows in the DRAM. Every read requires the data to be copied from the destination row to this buffer. As with CPU caches, accessing data which is already in the row buffer results in faster access times.

Pessl et al. [139] introduced the DRAMA attack, a side-channel attack which exploits the timing differences caused by the row buffer. Figure 3.3 illustrates the basic concept of a DRAMA attack. First, the attacker accesses attacker-controlled data residing in a DRAM row (①). If the attacker accesses data which falls into a conflicting DRAM row (②), *i.e.*, a different row in the same DRAM bank [139], the row-buffer content is replaced with this row. Finally, the attacker accesses attacker-controlled data, which are in the same row as the victim data and measures the access time (③). If the access is fast, the attacker knows that the victim

accessed data in this row, as the row is in the row buffer. Otherwise, the row buffer content has to be replaced for the current access, which results in a slower access.

We showed that DRAMA attacks are even possible in JavaScript by building a DRAM-based covert channel [150]. We also demonstrated that the DRAM side channel can be used to learn parts of the physical address in Intel SGX enclaves [154].

3.1.4 Transient-Execution Attacks

Transient-execution attacks are a class of microarchitectural attacks exploiting out-of-order and speculative execution of modern CPUs to leak data. Transient-execution attacks rely on computations which were never intended in an application's control flow. Such computations by transient instructions can be a result of mispredictions in the control or data flow, or out-of-order execution after an exception. While these transient instructions are never committed to the architectural state, they may show side effects in the microarchitectural state. These side effects can then be made visible in the architectural domain using traditional side-channel attacks.

With Meltdown [114] and Spectre [105], we presented the first transient-execution attacks, exploiting out-of-order and speculative execution respectively.

Spectre

Spectre is a class of transient-execution attacks exploiting control- and data-flow mispredictions of CPUs. By triggering such a misprediction, Spectre attacks transiently execute code to access data that is architecturally accessible but never reached. Subsequently, Spectre attacks encode the accessed data in the microarchitectural state, e.g., in the cache. An attacker can then rely on traditional side-channel attacks to transfer the microarchitectural state to the architectural state.

Figure 3.4 illustrates the basic idea of a Spectre attack using Spectre-PHT [105] (also known as Spectre v1) as an example. First, the attacker mistrains a conditional branch used for an out-of-bounds check, e.g., by providing multiple valid values. Then, when the attacker provides an out-of-bounds value (①), the CPU mispredicts the conditional jump (②). This leads to a transient out-of-bounds access to the data (③). The accessed data can then be encoded into a microarchitectural state, e.g., by caching a shared memory location corresponding to the value of the

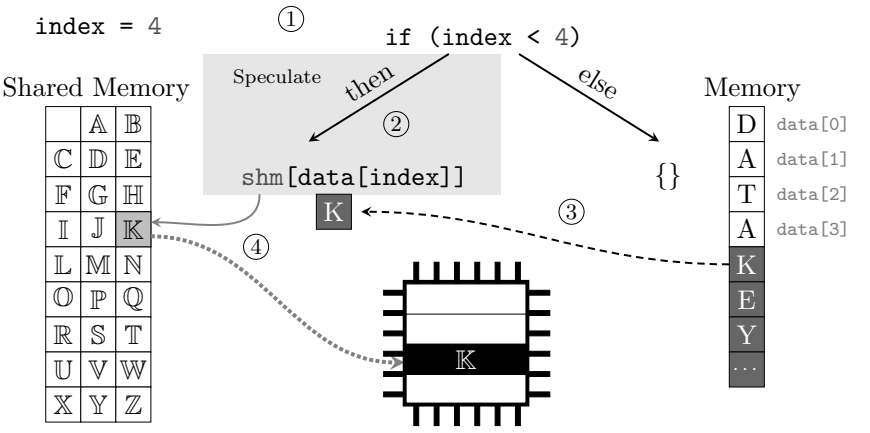


Figure 3.4: In a Spectre attack, an attacker manipulates a branch predictor such that the victim accesses and encodes data which is normally not accessed. Using a side-channel attack, the attacker recovers the encoded data.

leaked data (④). Finally, the attacker can use a side-channel attack to probe the microarchitectural state and infer the leaked data value.

We showed the first Spectre attacks exploiting the branch-target buffer (BTB) and the pattern-history table (PHT) [105]. Further Spectre variants also exploit the PHT [101] as well as the return-stack buffer (RSB) [110, 121] and memory-aliasing predictor [75]. As a side channel for recovering the leaked data value, we used Flush+Reload [105], Evict+Reload [155] as well as timing differences caused by the AVX unit [155]. Other side channels that have been shown to work are Prime+Probe [173] and port contention [29].

Spectre attacks have also been demonstrated to leak values from SGX [40, 129] and the system management mode [50]. We showed that Spectre attacks can be mounted from JavaScript [105] and even remotely [155].

Meltdown

Meltdown is a class of transient-execution attacks exploiting transient instructions caused by out-of-order execution after an exception. On affected CPUs, memory loads triggering an exception still return data which can be used in the transient execution to encode it in a microarchitectural

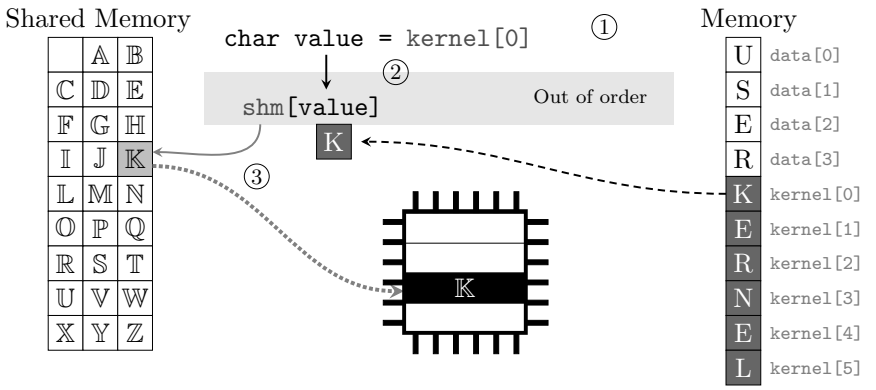


Figure 3.5: In a Meltdown attack, an attacker exploits that the CPU forwards inaccessible data to transient instructions caused by an exception. The data can then be encoded in a microarchitectural state, from which the attacker can recover it using a side-channel attack.

state. After the exception is handled (or suppressed), an attacker can again use a side channel to transfer the microarchitectural state into the architectural state.

Figure 3.5 illustrates the basic idea of a Meltdown attack based on Meltdown-US, *i.e.*, the original Meltdown attack [114]. The attacker accesses a memory location which results in a fault (①), e.g., a kernel address. Although the fault ensures that subsequent instructions are not executed architecturally, they are still executed out of order (②). Moreover, on affected CPUs, the loaded data is forwarded to these transient instructions and can thus be encoded in a microarchitectural element, such as the cache (③). Finally, the attacker can use, e.g., Flush+Reload to recover the leaked values.

With Meltdown [114], we showed the first Meltdown attack which broke the process isolation, allowing an attacker to read arbitrary kernel-memory locations. This attack exploited the lazy enforcement of the user-accessible permission in the page table. Van Bulck et al. [176] exploited the present bit in the page table to attack Intel SGX enclaves. Their attack can also be mounted from virtual machines to leak hypervisor data [192]. Meltdown has also been shown using other bits in the page table causing exceptions, such as the read-only bit [101], and memory-protection keys [37]. Other exceptions which have been used for Meltdown attacks are the device-

not-present exception to leak from floating-point registers [164], and bounds-checking exceptions to read out-of-bounds values [37].

Transient exceptions, such as microcode assists [158] have also been exploited for Meltdown attacks. Van Schaik et al. [146] and we [158] demonstrated Meltdown attacks on the line-fill buffer and load port, leaking data currently used by the current and sibling hyperthread. We also showed a Meltdown attack on the store buffer to leak values recently written by the current hyperthread [126].

3.2 Defenses against Software-based Microarchitectural Attacks

While side-channel leakage can be reduced by reducing the sharing of resources, this is in most cases not a practical solution. Hence, there are various proposals to reduce or even altogether remove side-channel leakage using various defense strategies.

We can classify defenses against software-based microarchitectural attacks based on where they are implemented: in the software layer, system layer, or in the hardware. Several countermeasures also require the interaction of multiple layers, *i.e.*, software and operating support for modified hardware. Due to their severity, generic countermeasures against transient-execution attacks are widely deployed on all of the three layers. In contrast, generic countermeasures against traditional side-channel attacks are not that widespread and mainly implemented in cryptographic libraries and browsers.

3.2.1 Software Layer

Software-layer countermeasures can be applied by the application itself to protect against microarchitectural side-channel attacks.

Constant Time. In addition to introducing cache-timing attacks, Bernstein [27] also emphasized to implement cryptographic algorithms that run in constant time. Constant-time implementations were also shown to mitigate side-channel leakage in later works [45, 209]. Agosta et al. [9] argued that side-channel leakage can be eliminated if there are no secret-dependent memory accesses or branches. Andrysco et al. [18] presented a side-channel free library for floating-point operations.

While algorithms without secret-dependent memory accesses and branches protect against side-channel attacks, they are not easy to write.

Several tools have been proposed to verify whether an implementation is constant time [13, 49, 63, 109, 143, 189, 195, 204]. Still, several proclaimed constant-time implementations turned out to be not completely constant time [55, 137, 189, 191].

Detection. Different proposals suggest to detect side-channel attacks and abort the computation in such a case. The advantage of these approaches is that they are more generic, as it is not necessary to find a constant-time variant of an algorithm.

To reliably detect cache attacks, Gruss et al. [70] leverage Intel TSX to automatically abort cryptographic operations if data is evicted from the cache. SGX enclaves can also leverage TSX to detect side-channel or controlled-channel attacks on page tables, and consequently abort any ongoing operation [160, 165].

Confining Speculation. Spectre attacks trick the victim application into accessing memory, which should not be accessed in a normal control flow. Hence, if an application wants to protect itself against Spectre attacks, it has to ensure that misspeculations do not leak secrets.

One possibility for Spectre-PHT is to stop speculation using memory fences after vulnerable branches [16, 24, 37, 82]. For Spectre-BTB, the branch predictor can be tricked into always misspeculating to a safe location [175]. Another possibility is to confine the speculation target to only safe locations by arithmetically applying bitmasks to array indices [39]. If enabled, memory fences are automatically generated by the Microsoft compiler [104, 105]. Both GCC and LLVM support the confinement using bitmasks [39, 120].

3.2.2 System Layer

System-layer countermeasures are provided by the environment in which vulnerable applications are executed. This can be the operating system, hypervisor, or a runtime environment.

Impair Timing Measurement. To impair side-channel attacks relying on timing differences, a system can introduce noise into the measurements of an attacker. Hu [76] proposed fuzzy time, which introduces noise into any event measurable by an attacker. This concept was later implemented in the Xen hypervisor [180] and proposed as a hardware modification [122].

The concept of fuzzy time was also proposed for browsers [107, 152] and is implemented in most major browsers [11, 31, 43, 138, 182].

However, even if the environment does not provide a high-resolution timer, an attacker can build a timer using shared data and concurrency [44, 194]. Such self-built timers have been used for attacks on ARM devices without access to a high-resolution timer [113]. We also showed that a variable, which is continuously incremented by a thread, can be used as a timing primitive in Intel SGX where no other high-resolution timer is available [154]. In concurrent work, Gras et al. [60] and we [150] showed that such a timer can also be built in JavaScript, which is now state of the art for attacks in JavaScript.

There are multiple proposed solutions to limit the theoretic leakage rate by hiding timing differences from an external observer. This can be accomplished by bucketing, *i.e.*, always padding times to multiples of a pre-defined bucket size [108, 205]. Li et al. [112] proposed to run three replicas of the system and return the average execution time to an external observer. Wu et al. [196] extended the virtual-time-based deterministic execution frameworks from Aviram et al. [25] and Ford et al. [54], limiting the theoretical leakage rate to 1 Kb/s. Kohlbrenner and Shacham [107] applied the concepts to browsers to ensure that all operations appear deterministic.

Adding Noise. Instead of adding noise to the timing primitives, noise can also be added to the observed event. Brickel et al. [34] proposed to randomize and prefetch lookup tables for AES computation to make attacks harder. However, for events which can be repeatedly observed by an attacker, adding noise only increases the number of required measurements. Using statistical methods, statistically independent noise can be averaged out by combining multiple traces [119, 155].

For one-time events, however, adding noise can make attacks infeasible. We showed that by injecting artificial keyboard interrupts, we can ensure that the interrupt density is uniform over time, making side-channel attacks on keystroke timings infeasible [153]. Similarly, adding noise to other user inputs have also been shown to make side-channel attacks infeasible [162].

State Flushing. One possibility to prevent information leakage through a microarchitectural state is to ensure that the state is flushed before the attacker can exploit it [208]. Exiting Intel SGX enclaves flushes the TLB to not leak information about enclave memory accesses [47]. Similarly,

switches into the kernel flush the return-stack buffer and the branch-target buffer to prevent Spectre attacks from the user space [19, 46, 87, 100].

Godfrey and Zulkernine [58] proposed that cloud providers flush the entire cache hierarchy if the CPU switches security domains. To prevent Foreshadow-VMM [192], virtual machines flush the L1 cache on VM exit [84]. Additionally, to prevent RIDL [146], ZombieLoad [158] and Fallout [126], every context switch has to also flush the store buffer, load ports, and line-fill buffer [85]. However, all these mitigations are only sufficient if an attacker cannot mount an attack in parallel, e.g., on a hyperthread. Hence, the system has to ensure that only mutually trusted processes are scheduled on the same physical core [85], or that time slices are long enough that the attacker cannot interrupt the victim computation [179].

Partitioning. Reducing the sharing of resources can also reduce the leakage observable through side-channel attacks. To prevent the sharing of cache sets, and thus mitigate Prime+Probe, Shi et al. [159] proposed cache coloring. With cache coloring, mutually untrusted applications do not share cache sets with each other. Costan et al. [47] showed that this also helps to protect the trusted-execution environment from cache attacks. Kim et al. [99] and Cock et al. [44] demonstrated that cache coloring has only a small performance overhead. However, it has a large memory overhead, as cache coloring statically splits the cache into partitions.

Instead of partitioning the cache by cache sets, Intel CAT allows partitioning the cache by cache way. Liu et al. [118] leveraged Intel CAT to mitigate cache-based side-channel attacks in the cloud. Zhou et al. [211] showed that cache partitioning can also prevent cache-line sharing in the cloud.

To prevent attacks from hyperthreads, processes can be scheduled to ensure mutually untrusted applications are not running on the two hyperthreads of the same physical core [85, 133, 147]. This was also demonstrated for Intel SGX [130].

To mitigate transient-execution attacks, the system can ensure that secrets are not mapped into the attacker's address space. We proposed KAISER [65] to split the kernel and the user space into two different address spaces. KAISER is implemented in all major operating systems [37] to prevent Meltdown-US [114]. A similar approach has been shown for virtual machines [77]. Instead of preventing Spectre attacks, Chromium relies on site isolation, which ensures that no secrets are mapped into the attacker's address space [171].

Blocking Functionality. Certain functions which are used for attacks can be blocked by the system. On ARM, the operating system can decide to prevent unprivileged programs from using the flush instruction [113]. This prevents Flush+Reload attacks, and an attacker has to resort to eviction-based attacks such as Evict+Reload.

Linux removed the unprivileged access to the pagemap [103] which is used by applications to translate virtual addresses to physical addresses. This impairs attacks requiring knowledge of physical addresses, such as Evict+Reload or Prime+Probe.

Vattikonda et al. [180] blocked direct access to the timestamp counter in the Xen hypervisor. We showed that blocking direct access to timing sources and other functions commonly used for side-channel attacks is also a possibility in JavaScript [152].

Safe Speculation. In theory, Spectre attacks can be mitigated by turning off all speculation features in CPUs. However, in practice, this is neither possible nor desirable for performance reasons. We showed that marking secret values as uncachable prevents transient execution from accessing and hence leaking them [149].

Detection. An alternative to preventing attacks is to detect attacks. Irazoqui et al. [90], proposed MASCAT, a static-analysis framework to scan binaries for side-channel attacks similar to antivirus software.

A dynamic approach is to mount dummy attacks and see whether there is any interference from real attacks [63, 79, 210]. For the detection of ongoing attacks, performance counters can provide useful data, especially the number of cache hits and misses [41, 64, 74, 128, 208]. Payer et al. [135] presented a framework to combine multiple performance counters for detecting ongoing attacks. Demme et al. [48] used these performance counters to detect malware, which was later improved by leveraging machine learning [169].

Performance counters can also be used to detect cross-VM attacks. Cardenas et al. [38] and Zhang et al. [207] leveraged performance counters to detect denial-of-service attacks. Zhang et al. [206] and Chouhan and Hasbullah [42] leveraged performance counters to detect cross-VM side-channel attacks. Paundu et al. [134] proposed to use hypervisor events in combination with machine learning to detect cache attacks.

However, these detection methods suffer from false positives and false negatives [56]. Moreover, attackers can adapt their attacks or find new attacks which are not easily detectable through performance counters [64].

We also showed that trusted-execution environments, such as Intel SGX, can be abused to protect attacks from being detected [154].

3.2.3 Hardware Layer

Various countermeasures proposed on the hardware layer try to either get rid of the root cause of a microarchitectural side channel or make it infeasible to exploit it.

Constant Time. Microarchitectural side-channel attacks exploiting runtime differences in certain instructions can be prevented by making the instructions execute in constant time. Gruss et al. proposed this for the `clflush` instruction to mitigate the Flush+Flush attack [64] and for the software-prefetch instructions to mitigate prefetch-based attacks [68].

With ARMv8.5, ARM supports a Data-Independent-Timing (DIT) bit in the processor-state register to make supported instructions run in constant time [21]. If enabled, the runtime of instructions does not depend on the data operated on. With the conditional-move instruction family (`CMOVxx`), Intel also provides a constant-time operation which can be used for implementing side-channel-resistant cryptographic algorithms [86].

Wang et al. [186] proposed to change the row-buffer policy of the DRAM controller to always close a DRAM row. This might eliminate the timing differences exploited in DRAMA attacks [139].

Cache Designs. To thwart cache attacks, various proposals for alternative or adapted cache designs exist. Wang and Lee [188] proposed PLcache, which allows to lock cache lines in the cache temporarily. This ensures that an attacker cannot evict the cache lines in a cache attack. Certain ARM cache controllers support such a cache lockdown by cache way and cache line [20].

Tan et al. [168] suggested such a locking mechanism for the BTB to defend against branch-prediction attacks.

RPcache [188] and NewCache [187] use a random per-process mapping from addresses to cache sets. With this design, an attacker cannot construct an eviction set for a target cache set. Liu and Lee [117] proposed random fill caches, where data on a cache miss is sent directly to the processor and not cached. Instead, they cache a random address in the neighborhood of the data.

The time-secure cache [172] uses a cache-set-indexing function based on the process ID. However, we have shown that the used indexing function

is weak [193]. We generalized the concept of the time-secure cache and used a stronger indexing function [193]. CEASER [141] relies on a similar principle. However, due to its design, the inter-process cache interference is predictable for an attacker.

Saileshwar et al. [145] proposed to add a "zombie bit" to every cache line on an explicit flush. Cache hits to zombie lines suffer an additional delay making them indistinguishable from cache misses, consequently thwarting Flush+Reload attacks.

Instruction Set Extensions. While constant-time cryptographic algorithms can be implemented in software, most processors provide a constant-time instruction-set extension for computing at least AES [14, 22, 23, 72]. The hardware AES implementation is not only faster but also resistant against software-based side-channel attacks.

Strackx et al. [165] proposed small changes to the SGX instruction-set extension to protect against page-table side-channel attacks.

Intel and AMD extended the instruction set with functionality to have more control over branch prediction [15, 17, 88]. With these extensions, the branch prediction for indirect branches can be limited to privilege levels and hyperthreads [37], and the speculative store bypass can be disabled. ARM introduced speculative barriers as well as control registers to restrict speculation in ARMv8.5 [21].

Safe Transient Execution. As Meltdown attacks are vulnerabilities in the CPU, they ultimately require fixes in hardware. While this often requires a new hardware revision, some Meltdown attacks can be fixed by changing the hardware behavior through microcode updates [37]. Meltdown-GP [24, 82] on Intel CPUs has been fixed using a microcode update [82]. For Meltdown-P [176, 192], Meltdown-MCA [146, 158] (also known as ZombieLoad or RIDL), and Meltdown-STL [126] (also known as Fallout), Intel released microcode updates which expose new flushing functionalities for the L1, store buffer, line-fill buffer, and the load ports.

For Spectre attacks, this is more difficult, as their root cause is intended and cannot directly be fixed. Most countermeasures try to mitigate Spectre by preventing extraction of the leaked data through the cache [98, 102, 199]. However, the cache is not the only covert channel which can be used to extract data [29, 105, 155]. Hence, these countermeasures are incomplete [37].

We proposed a different approach using taint tracking of secrets [105, 149]. By annotating and tracking secrets, our approach ensures that

secrets can never be used in transient execution, thus preventing any leakage of secret data. A similar approach was also proposed by Yu et al. [203].

4 Conclusion

In this thesis, we researched the requirements for software-based microarchitectural attacks, showing that several wrong assumptions about their requirements existed. From our results, we can conclude several things.

Few Requirements. Mounting software-based microarchitectural attacks does not require many primitives. For many attacks it is sufficient to have read access to the memory, compute on these values, and measure time. While it is hardly possible to restrict memory access and general-purpose computation, it is even difficult to prevent timing measurement. We showed that the lack of a timer can be overcome as long as shared resources and concurrent execution is possible in a language [116, 150, 151, 154].

Based on these results, we showed software-based microarchitectural attacks in environments which were assumed to be too restricted, such as Intel SGX or JavaScript [150, 151, 154]. Moreover, we demonstrated that the techniques learned from the past years of microarchitectural side-channel attacks can also be applied to different abstraction layers, such as the operating system [66], which makes the attacks even hardware-agnostic.

Code Execution. Many countermeasures built upon the assumption that attack code runs natively and can thus be inspected or detected.

However, by moving attacks inside sandboxes [150, 151, 154] this is only partly true. Moreover, we were the first to show a fully remote Spectre attack [155], and a remote Rowhammer attack [115] also shown in concurrent work [170].

These results show a trend to move attacks from native code to more restricted environments and even allow attackers to mount attacks remotely. While the performance of these attacks can still be improved significantly, it shows that current threat models might not be complete.

New Side Channels. During this thesis, we discovered several novel side channels [139, 151, 153, 155, 157]. From that, we can conclude that there are still many undiscovered side channels in modern CPUs. Moreover, we will see more sophisticated side channels in the future which combine multiple effects, as we have shown with Store-to-leak forwarding [157].

We have seen that performance optimizations often introduce new side channels. Hence, we assume that as CPUs are mainly optimized for performance and not for security, there will be more side channels in the future.

Hardware Vulnerabilities. With transient-execution attacks [37, 105, 114], we have shown that side-channel attacks are even more powerful than assumed. Side-channel attacks are a vital part of transient-execution attacks to leak secrets from the transient to the architectural domain. Hence, side-channel attacks are a tool to look into the microarchitectural state of the CPU.

In addition to the hardware vulnerabilities we have already discovered using side-channel attacks [37, 105, 114, 126, 157, 158], we can expect to see more hardware vulnerabilities which can be exploited. Especially as transient-execution attacks have so far mainly exploited the low-hanging fruit, we can expect even more sophisticated transient-execution attacks.

Effective Defenses. To build effective defenses against attacks, it is extremely important to first understand the attack surface and requirements. Only then, it is possible to build defenses which mitigate entire classes of attacks [65, 148, 149, 152], or fully prevent leakage of specific data [153]. We have shown that otherwise, defenses can be bypassed by adapting attacks [62, 116, 150]. Hence, it is necessary to further research attacks to be able to build complete defenses.

We assume that many defenses cannot simply be retrofitted to the existing architectures and software infrastructure [149, 153]. Instead,

cooperation between hardware and software will be required to ensure efficient and effective defenses. While we see this as a promising direction, it requires changes in all layers, *i.e.*, in hardware, operating systems, and toolchains. Thus, we have to move from CPUs designed as black boxes that run any software to hardware-software co-design [149]. Side-channel-aware CPU designs potentially reduce the difficulty of writing side-channel-resistant applications. In general, it is not realistic to eliminate all side channels in all scenarios [105]. However, tighter integration of software and hardware gives the hardware the possibility to reduce information leakage for sensitive data while still providing performance optimizations for other data [149, 152]. This might also require developers to potentially provide metadata for data [105, 149, 203]. With additional metadata, the CPU can then selectively disable specific optimizations when working with sensitive data [149]. However, this does not only require changes to software and hardware, there also needs to be an awareness of side-channel leakage among software developers, which might take more time.

References

[1] Onur Acıiçmez. "Advances in Side-Channel Cryptanalysis: MicroArchitectural Attacks." PhD thesis. Oregon State University, 2007.

[2] Onur Acıiçmez. "Yet Another MicroArchitectural Attack: Exploiting I-cache." In: *ACM Computer Security Architecture Workshop (CSAW)*. 2007.

[3] Onur Acıiçmez and Çetin Kaya Koç. "Trace-Driven Cache Attacks on AES." In: *IACR Cryptology ePrint Archive* (2006). URL: `http://eprint.iacr.org/2006/138`.

[4] Onur Acıiçmez and Çetin Kaya Koç. "Trace-Driven Cache Attacks on AES (Short Paper)." In: *Information and Communications Security*. 2006.

[5] Onur Acıiçmez and Werner Schindler. "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL." In: *CT-RSA*. 2008.

[6] Onur Acıiçmez, Shay Gueron, and Jean-pierre Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures." In: *IMA International Conference on Cryptography and Coding*. 2007.

[7] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. "New Results on Instruction Cache Attacks." In: *CHES*. 2010.

[8] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. "Predicting secret keys via branch prediction." In: *CT-RSA*. 2007.

[9] Giovanni Agosta, Luca Breveglieri, Gerardo Pelosi, and Israel Koren. "Countermeasures against branch target buffer attacks." In: *Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2007.

[10] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. "Cache-Timing Attacks on RSA Key Generation." In: *IACR Cryptology ePrint Archive* 2018 (2018).

[11] Alex Christensen. *Reduce resolution of performance.now*. 2015. URL: `https://bugs.webkit.org/show_bug.cgi?id=146531`.

[12] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van De Pol, and Yuval Yarom. "Amplifying Side Channels Through Performance Degradation." In: *Cryptology ePrint Archive: Report 2015/1141* (2015).

[13] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying constant-time implementations." In: *USENIX Security*. 2016.

[14] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*. 2007.

[15] AMD. *AMD64 Technology: Speculative Store Bypass Disable*. 2018. URL: `https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf`.

[16] AMD. *Software Techniques for Managing Speculation on AMD Processors*. Revison 7.10.18. 2018.

[17] AMD. *Software techniques for managing speculation on AMD processors*. 2018.

[18] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "On subnormal floating point and abnormal timing." In: *S&P*. 2015.

[19] ARM. *Cache Speculation Side-channels*. 2018.

[20] ARM. *CoreLink Level 2 Cache Controller L2C-310*. 2010.

[21] ARM Limited. *ARM A64 Instruction Set Architecture ARMv8, for ARMv8-A architecture profile*. ARM Limited, 2018.

[22] ARM Limited. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. ARM Limited, 2012.

[23] ARM Limited. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.

[24] ARM Limited. *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*. 2018.

[25] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. "Determinating timing channels in compute clouds." In: *CCSW*. 2010.

[26] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "'Ooh Aah... Just a Little Bit': A small amount of side channel can go a long way." In: *CHES*. 2014.

[27] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. 2004. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[28] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. "Hardware prefetchers leak : A revisit of SVF for cache-timing attacks." In: *MICRO*. 2012.

[29] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention." In: *arXiv:1903.01843* (2019).

[30] Joseph Bonneau and Ilya Mironov. "Cache-collision timing attacks against AES." In: *CHES 2006*. 2006.

[31] Boris Zbarsky. *Reduce resolution of performance.now*. 2015. URL: https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab.

[32] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector." In: *S&P*. 2016.

[33] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *WOOT*. 2017.

[34] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. "Software mitigations to hedge AES against cache-based software side channel vulnerabilities." In: *Cryptology ePrint Archive, Report 2006/052* (2006).

[35] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M Moya. "Cache Misses and the Recovery of the Full AES 256 Key." In: *Applied Sciences* 9.5 (2019).

[36] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. "RELOAD+ REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks." In: *arXiv:1904.06278* (2019).

[37] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A systematic evaluation of transient execution attacks and defenses." In: *USENIX Security*. 2019.

[38] Carlos Cardenas and Rajendra V Boppana. "Detection and mitigation of performance attacks in multi-tenant cloud computing." In: *1st International IBM Cloud Academy Conference, Research Triangle Park, NC, US*. 2012, p. 48.

[39] Chandler Carruth. *RFC: Speculative Load Hardening (a Spectre variant #1 mitigation*. Mar. 2018. URL: https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.

[40] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. "SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution." In: *arXiv:1802.09085* (2018).

[41] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real time detection of cache-based side-channel attacks using hardware performance counters." In: *Applied Soft Computing* (2016).

[42] Munish Chouhan and Halabi Hasbullah. "Adaptive detection technique for Cache-based Side Channel Attack using Bloom Filter for secure cloud." In: *International Conference on Computer and Information Sciences (ICCOINS)*. 2016.

[43] Chromium. *window.performance.now does not support sub-millisecond precision on Windows*. 2015. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110.

[44] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. "The last mile: An empirical study of timing channels on seL4." In: *CCS*. 2014.

[45] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. "Practical mitigations for timing-based side-channel attacks on modern x86 processors." In: *S&P*. 2009.

[46] Jonathan Corbet. *Strengthening user-space Spectre v2 protection*. Sept. 2018. URL: https://lwn.net/Articles/764209/.

[47] Victor Costan and Srinivas Devadas. *Intel SGX explained*. Tech. rep. Cryptology ePrint Archive, Report 2016/086, 2016.

[48] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the feasibility of online malware detection with performance counters." In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 559–570.

[49] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. "Cacheaudit: A tool for the static analysis of cache side channels." In: *TISSEC* (2015).

[50] ECLYPSIUM. *System Management Mode Speculative Execution Attacks*. May 2018. URL: https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/.

[51] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. "Branchscope: A new side-channel attack on directional branch predictor." In: *ACM SIGPLAN Notices*. 2018.

[52] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Covert channels through branch predictors: a feasibility study." In: *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM. 2015.

[53] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR." In: *MICRO*. 2016.

[54] Bryan Ford, Minlan Yu, Abhishek Sharma, Ramesh Govindan, Chandra Krintz, and Haitao Wu. "Plugging side-channel leaks with timing information flow control." In: *HotCloud*. 2012.

[55] Cesar Pereida García and Billy Bob Brumley. "Constant-time callees with variable-time callers." In: *USENIX Security*. 2017.

[56] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware." In: *Journal of Cryptographic Engineering* 8.1 (2018).

[57] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. "Drive-by key-extraction cache attacks from portable code." In: *International Conference on Applied Cryptography and Network Security*. 2018.

[58] Michael Godfrey and Mohammad Zulkernine. "A server-side solution to cache-based side-channel attacks in the cloud." In: *IEEE CLOUD*. 2013.

[59] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX." In: *EuroSec*. 2017.

[60] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. 2017.

[61] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with {TLB} Attacks." In: *USENIX Security*. 2018.

[62] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses." In: *S&P*. 2018.

[63] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security*. 2015.

[64] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[65] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR." In: *ESSoS*. 2017.

[66] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. "Page Cache Attacks." In: *CCS*. 2019.

[67] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed JavaScript." In: *ESORICS*. 2015.

[68] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *CCS*. 2016.

[69] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA*. 2016.

[70] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." In: *USENIX Security*. 2017.

[71] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. "Use-After-FreeMail: Generalizing the Use-After-Free Problem and Applying it to Email Services." In: *AsiaCCS*. 2018.

[72] Shay Gueron. *Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01*. 2012.

[73] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *S&P*. 2011.

[74] Nishad Herath and Anders Fogh. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security." In: *Black Hat Briefings*. Aug. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf.

[75] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[76] Wei-Ming Hu. "Reducing timing channels with fuzzy time." In: *Journal of Computer Security* (1992).

[77] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. "EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs." In: *USENIX ATC*. 2018.

[78] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *S&P*. 2013.

[79] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. "Understanding contention-based channels and using them for defense." In: *HPCA*. 2015.

[80] Mehmet S Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud." In: *CHES*. 2016.

[81] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud*. Tech. rep. Cryptology ePrint Archive, Report 2015/898, 2015., 2015.

[82] Intel. *Intel Analysis of Speculative Execution Side Channels*. July 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.

[83] Intel. "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide." In: 253665 (2016).

[84] Intel Corp. *Deep Dive: Intel Analysis of L1 Terminal Fault*. Aug. 2018. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault.

[85] Intel Corp. *Deep Dive: Intel Analysis of Microarchitectural Data Sampling*. May 2019. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling.

[86] Intel Corp. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. May 2019. URL: https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations.

[87] Intel Corp. *Retpoline: A Branch Target Injection Mitigation*. June 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.

[88] Intel Corp. *Speculative Execution Side Channel Mitigations*. May 2018. URL: https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[89] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Lucky 13 Strikes Back." In: *AsiaCCS*. 2015.

[90] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *MASCAT: Stopping Microarchitectural Attacks Before Execution*. Cryptology ePrint Archive, Report 2016/1196. 2017.

[91] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES." In: *S&P*. 2015.

[92] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Systematic reverse engineering of cache slice selection in Intel processors." In: *DSD*. 2015.

[93] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES." In: *RAID'14*. 2014.

[94] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks." In: *USENIX Security*. 2019.

[95] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. *Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge*. Retrieved on July 16, 2015. URL: http://hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf.

[96] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX." In: *CCS*. 2016.

[97] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. "A high-resolution side-channel attack on last-level cache." In: *DAC*. 2016.

[98] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation." In: *arXiv:1806.05179* (2018).

[99] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "StealthMem: system-level protection against cache-based side channel attacks in the cloud." In: *USENIX Security*. 2012.

[100] Russel King. *ARM: spectre-v2: harden branch predictor on context switches*. 2018. URL: https://patchwork.kernel.org/patch/10427513/.

[101] Vladimir Kiriansky and Carl Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses." In: *arXiv:1807.03757* (2018).

[102] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors." In: *IACR Cryptology ePrint Archive* (May 2018).

[103] Kirill A. Shutemov. *Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace*. Retrieved on November 10, 2015. Mar. 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[104] Paul Kocher. *Spectre Mitigations in Microsoft's C/C++ Compiler*. 2018.

[105] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *S&P*. 2019.

[106] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems." In: *CRYPTO*. 1996.

[107] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times." In: *USENIX Security*. 2016.

[108] Boris Köpf and Markus Dürmuth. "A provably secure and efficient countermeasure against timing attacks." In: *IEEE Computer Security Foundations Symposium*. 2009.

[109] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. "Automatic quantification of cache side-channels." In: *International Conference on Computer Aided Verification*. 2012.

[110] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer." In: *WOOT*. 2018.

[111] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing." In: *USENIX Security*. 2017.

[112] Peng Li, Debin Gao, and Michael K Reiter. "Mitigating access-driven timing channels in clouds using StopWatch." In: *DSN*. 2013.

[113] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security*. 2016.

[114] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium*. 2018.

[115] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. "Nethammer: Inducing Rowhammer Faults through Network Requests." In: *arXiv:1711.08002* (2017).

[116] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript." In: *ESORICS*. 2017.

[117] Fangfei Liu and Ruby B. Lee. "Random Fill Cache Architecture." In: *MICRO*. 2014.

[118] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. "Catalyst: Defeating last-level cache side channel attacks in cloud computing." In: *HPCA*. 2016.

[119] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *S&P*. 2015.

[120] LWN. *Spectre V1 defense in GCC*. July 2018. URL: `https://lwn.net/Articles/759423/`.

[121] G. Maisuradze and C. Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers." In: *CCS*. 2018.

[122] Robert Martin, John Demme, and Simha Sethumadhavan. "TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks." In: *ACM SIGARCH Computer Architecture News* (2012).

[123] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel." In: *DIMVA*. 2015.

[124] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters." In: *RAID*. 2015.

[125] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. 2017.

[126] Marina Minkin et al. "Fallout: Reading Kernel Writes From User Space." In: *arXiv:1905.12701* (2019).

[127] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies The Power of Cache Attacks." In: *arXiv:1703.06986* (2017).

[128] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters." In: *HASP*. 2018.

[129] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. *Spectre attack against SGX enclave*. Jan. 2018. URL: https://github.com/lsds/spectre-attack-sgx.

[130] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks." In: *USENIX ATC*. 2018.

[131] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS*. 2015.

[132] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA*. 2006.

[133] John K Ousterhout et al. "Scheduling Techniques for Concurrent Systems." In: *ICDCS*. 1982.

[134] Ady Wahyudi Paundu, Doudou Fall, Daisuke Miyamoto, and Youki Kadobayashi. "Leveraging KVM Events to Detect Cache-Based Side Channel Attacks in a Virtualization Environment." In: *Security and Communication Networks* (2018).

[135] Matthias Payer. "HexPADS: a platform to detect "stealth" attacks." In: *ESSoS*. 2016.

[136] Colin Percival. "Cache missing for fun and profit." In: *Proceedings of BSDCan*. 2005.

[137] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time." In: *CCS*. 2016.

[138] Mike Perry. *Bug 1517: Reduce precision of time for Javascript.* 2015. URL: https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517.

[139] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security*. 2016.

[140] Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "Just a little bit more." In: *Cryptographers' Track at the RSA Conference*. 2015.

[141] Moinuddin K Qureshi. "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping." In: *MICRO*. 2018.

[142] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. "Adaptive insertion policies for high performance caching." In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 381.

[143] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "Dude, is my code constant time?" In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017.

[144] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds." In: *CCS*. 2009.

[145] Gururaj Saileshwar and Moinuddin K Qureshi. "Lookout for Zombies: Mitigating Flush+ Reload Attack on Shared Caches by Monitoring Invalidated Lines." In: *arXiv:1906.02362* (2019).

[146] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-Flight Data Load." In: *S&P* (2019).

[147] Jan H Schönherr, Ben Juurlink, and Jan Richling. "Topology-aware equipartitioning with coscheduling on multicore systems." In: *Workshop on Multi-/Many-core Computing Systems*. 2013.

[148] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In: *AsiaCCS*. 2018.

[149] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. "ConTExT: Leakage-Free Transient Execution." In: *arXiv:1905.09100* (2019).

[150] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC*. 2017.

[151] Michael Schwarz, Florian Lackner, and Daniel Gruss. "JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits." In: *NDSS*. 2019.

[152] Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks." In: *NDSS*. 2018.

[153] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In: *NDSS*. 2018.

[154] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *DIMVA*. 2017.

[155] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *ESORICS*. 2019.

[156] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." In: *DIMVA*. 2019.

[157] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs." In: *arXiv:1905.05725* (2019).

[158] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *CCS*. 2019.

[159] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring." In: *Conference on Dependable Systems and Networks Workshops*. 2011.

[160] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating controlled-channel attacks against enclave programs." In: *NDSS*. 2017.

[161] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage." In: *CCS*. 2018.

[162] Prakash Shrestha, Manar Mohamed, and Nitesh Saxena. "Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise." In: *WiSec*. 2016.

[163] Raphael Spreitzer and Thomas Plos. "Cache-Access Pattern Attack on Disaligned AES T-Tables." In: *COSADE*. 2013.

[164] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels." In: *arXiv:1806.07480* (2018).

[165] Raoul Strackx and Frank Piessens. "The Heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks." In: *arXiv:1712.08519* (2017).

[166] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. "Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in IaaS clouds." In: *NDSS*. 2018.

[167] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. "Memory Deduplication as a Threat to the Guest OS." In: *EuroSec*. 2011.

[168] Ya Tan, Jizeng Wei, and Wei Guo. "The micro-architectural support countermeasures against the branch prediction analysis attack." In: *Conference on Trust, Security and Privacy in Computing and Communications*. 2014.

[169] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. "Unsupervised anomaly-based malware detection using hardware features." In: *International Workshop on Recent Advances in Intrusion Detection*. 2014.

[170] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer attacks over the network and defenses." In: *USENIX ATC*. 2018.

[171] The Chromium Projects. *Site Isolation*. URL: `http://www.chromium.org/Home/chromium-security/site-isolation`.

[172] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. "Cache side-channel attacks and time-predictability in high-performance critical real-time systems." In: *DAC*. 2018.

[173] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. "Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols." In: *arXiv:1802.03802* (2018).

[174] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures." In: *Journal of Cryptology* 23.1 (July 2010), pp. 37–71.

[175] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018.

[176] Jo Van Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.” In: *USENIX Security*. 2018.

[177] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. “Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution.” In: *USENIX Security*. 2017.

[178] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Malicious management unit: why stopping cache attacks in software is harder than you think.” In: *USENIX Security*. 2018.

[179] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. “Scheduler-based defenses against cross-VM side-channels.” In: *USENIX Security*. 2014.

[180] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. “Eliminating fine grained timers in Xen.” In: *CCSW*. 2011.

[181] Pepe Vila, Boris Köpf, and José Francisco Morales. “Theory and practice of finding eviction sets.” In: *S&P*. 2019.

[182] W3C. *High Resolution Time Level 2 - W3C Working Draft 21 July 2015*. July 2015. URL: `http://www.w3.org/TR/2015/WD-hr-time-2-20150721/#privacy-security`.

[183] W3Techs. *Usage of JavaScript for websites*. Aug. 2017. URL: `https://w3techs.com/technologies/details/cp-javascript/all/all`.

[184] Daimeng Wang, Zhiyun Qian, Nael B Abu-Ghazaleh, and Srikanth V Krishnamurthy. “PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack.” In: *DAC*. 2019.

[185] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. “Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries.” In: *NDSS*. 2019.

[186] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. “Timing channel protection for a shared memory controller.” In: *High Performance Computer Architecture*. 2014.

[187] Zhenghong Wang and Ruby B. Lee. “A Novel Cache Architecture with Enhanced Performance and Security.” In: *MICRO*. 2008.

[188] Zhenghong Wang and Ruby B. Lee. “New cache designs for thwarting software cache-based side channel attacks.” In: *ACM SIGARCH Computer Architecture News* 35.2 (June 2007), p. 494.

[189] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA–Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries." In: *USENIX Security*. 2018.

[190] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGXJail: Defeating Enclave Malware via Confinement." In: *RAID*. 2019.

[191] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. "Single trace attack against RSA key generation in Intel SGX SSL." In: *AsiaCCS*. 2018.

[192] Weisse, Ofir and Van Bulck, Jo and Minkin, Marina and Genkin, Daniel and Kasikci, Baris and Piessens, Frank and Silberstein, Mark and Strackx, Raoul and Wenisch, Thomas F. and Yarom, Yuval. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. 2018.

[193] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization." In: *USENIX Security*. 2019.

[194] John C Wray. "An analysis of covert timing channels." In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

[195] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. "Eliminating timing side-channel leaks using program repair." In: *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018.

[196] Weiyi Wu, Ennan Zhai, David Isaac Wolinsky, Bryan Ford, Liang Gu, and Daniel Jackowitz. "Warding off timing attacks in Deterland." In: *Conference on Timely Results in Operating Systems*. 2015.

[197] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud." In: *IEEE/ACM Transactions on Networking* (2014).

[198] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. "An exploration of L2 cache covert channels in virtualized environments." In: *CCSW*. 2011.

[199] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy." In: *MICRO*. 2018.

[200] Yuval Yarom and Naomi Benger. "Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack." In: *IACR Cryptology ePrint Archive* (2014).

[201] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security*. 2014.

[202] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. "Mapping the Intel Last-Level Cache." In: *Cryptology ePrint Archive, Report 2015/905* (2015), pp. 1–12.

[203] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. "Speculative Taint Tracking (STT): A Comprehensive Protection for Transiently Accessed Secrets." In: *MICRO*. 2019.

[204] Andreas Zankl, Johann Heyszl, and Georg Sigl. "Automated detection of instruction cache leaks in modular exponentiation software." In: *International Conference on Smart Card Research and Advanced Applications*. 2016.

[205] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. "Predictive mitigation of timing channels in interactive systems." In: *CCS*. 2011.

[206] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds." In: *RAID*. 2016.

[207] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. "Memory DoS attacks in multi-tenant clouds: Severity and mitigation." In: *arXiv:1603.03404* (2016).

[208] Yinqian Zhang and MK Reiter. "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud." In: *CCS*. 2013.

[209] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys." In: *CCS*. 2012.

[210] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis." In: *S&P*. 2011.

[211] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. "A software approach to defeating side channels in last-level caches." In: *CCS*. 2016.

Part II

Publications

List of Publications

During the time of my thesis, I contributed to 21 papers which are published in conference proceedings, and 6 papers which are currently in submission to conferences. Out of these papers, 12 papers are published at Tier 1 conferences.

Publications Used in the Thesis

▶ *Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In:* FC. *2017*

▶ *Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In:* DIMVA. *2017*

▶ *Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In:* AsiaCCS. *2018*

▶ *Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In:* NDSS. *2018*

▶ *Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks." In:* NDSS. *2018*

▶ *Michael Schwarz, Florian Lackner, and Daniel Gruss. "JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits." In:* NDSS. *2019*

► *Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In:* CCS. *2019*

Other Contributions

► *Martin Schwarzl, Michael Schwarz, Thomas Schuster, and Daniel Gruss. "Ghost in the Syscall: Speculative Register Prefetcher." In:* under submission *(2020)*

► *Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take A Way: AMD's Cache Way Predictors Leak Secrets." In:* under submission *(2020)*

► *Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs." In:* arXiv:1905.05725 *(2019) (under submission)*

► *Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. "ConTExT: Leakage-Free Transient Execution." In:* arXiv:1905.09100 *(2019) (under submission)*

► *Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. "Fallout: Reading Kernel Writes From User Space." In:* arXiv:1905.12701 *(2019) (under submission)*

► *Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. "Nethammer: Inducing Rowhammer Faults through Network Requests." In:* arXiv:1711.08002 *(2017) (under submission)*

► *Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In:* ESORICS. *2019*

► *Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGX-*

Jail: Defeating Enclave Malware via Confinement." In: RAID. *2019*

▶ *Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A systematic evaluation of transient execution attacks and defenses." In:* USENIX Security. *2019*

▶ *Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization." In:* USENIX Security. *2019*

▶ *Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. "Page Cache Attacks." In:* CCS. *2019*

▶ *Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX." in:* DIMVA. *2019*

▶ *Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." In:* S&P. *2019*

▶ *Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space." In:* USENIX Security Symposium. *2018*

▶ *Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. "Use-After-FreeMail: Generalizing the Use-After-Free Problem and Applying it to Email Services." In:* AsiaCCS. *2018*

▶ *Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses." In:* S&P. *2018*

▶ *Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in*

Sandboxed JavaScript." In: ESORICS. *2017*

► *Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR." in:* ESSoS. *2017*

► *Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In:* NDSS. *2017*

► *Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In:* USENIX Security. *2016*

5

Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript

Publication Data

Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC*. 2017

Contributions

Main author.

Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript

Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard

Graz University of Technology, Austria

Abstract

Research showed that microarchitectural attacks like cache attacks can be performed through websites using JavaScript. These timing attacks allow an adversary to spy on users secrets such as their keystrokes, leveraging fine-grained timers. However, the W3C and browser vendors responded to this significant threat by eliminating fine-grained timers from JavaScript. This renders previous high-resolution microarchitectural attacks non-applicable.

We demonstrate the inefficacy of this mitigation by finding and evaluating a wide range of new sources of timing information. We develop measurement methods that exceed the resolution of official timing sources by 3 to 4 orders of magnitude on all major browsers, and even more on Tor browser. Our timing measurements do not only re-enable previous attacks to their full extent but also allow implementing new attacks. We demonstrate a new DRAM-based covert channel between a website and an unprivileged app in a virtual machine without network hardware. Our results emphasize that quick-fix mitigations can establish a dangerous false sense of security.

1 Introduction

Microarchitectural attacks comprise side-channel attacks and covert channels, entirely implemented in software. Side-channel attacks exploit timing differences to derive secret values used in computations. They have been studied extensively in the past 20 years with a focus on cryptographic algorithms [2, 10, 16, 29–31, 48]. Covert channels are special side channels where a sender and a receiver use the side channel actively to transmit

The original publication is available at `https://link.springer.com/chapter/10.1007/978-3-319-70972-7_13`.

data covertly. These attacks require highly accurate timing and thus are typically implemented in native binaries written in C or assembly language to use the best available timing source.

Side channels exist on virtually all systems and software not hardened against side channels. Thus, browsers are an especially easy target for an attacker, because browsers process highly sensitive data and attackers can easily trick a victim to open a malicious website in the browser. Consequently, timing side-channel attacks have been demonstrated and observed in the wild, to recover a user's browser history [8, 13, 41], but also a user's geolocation [14], whether a user is logged in to another website [4] and even CSRF tokens [11]. Van Goethem et al. [37] exploited more accurate in-browser timing to obtain information even from within other websites, such as contact lists or previous inputs.

Oren et al. [28] recently demonstrated that cache side-channel attacks can also be performed in browsers. Their attack uses the `performance.now` method to obtain a timestamp whose resolution is in the range of nanoseconds. It allows spying on user activities but also building a covert channel with a process running on the system. Gruss et al. [9] and Bosman et al. [5] demonstrated Rowhammer attacks in JavaScript, leveraging the same timing interface. In response, the W3C [40] and browser vendors [1, 3, 6] have changed the `performance.now` method to a resolution of 5 µs. The timestamps in the Tor browser are even more coarse-grained, at 100 ms [25]. In both cases, this successfully stops side-channel attacks by withholding necessary information from an adversary.

In this paper, we demonstrate that reducing the resolution of timing information or even removing these interfaces is completely insufficient as an attack mitigation. We propose several new mechanisms to obtain absolute and relative timestamps. We evaluated 10 different mechanisms on the most recent versions of 4 different browsers: Chrome, Firefox, Edge, as well as the Tor browser, which took even more drastic measures. We show that all browsers leak highly accurate timing information that exceeds the resolution of official timing sources by 3 to 4 orders of magnitude on all browsers, and by 8 on the Tor browser. In all cases, the resolution is sufficient to revive the attacks that were thought mitigated [28].

Based on our novel timing mechanisms, we are the first to exploit DRAM-based timing leaks from JavaScript. There were doubts whether DRAM-based timing leaks can be exploited from JavaScript, as it is not possible to directly reach DRAM [32]. We demonstrate that a DRAM-based covert channel can be used to exfiltrate data from highly restricted, isolated execution environments that are not connected to the network.

More specifically, we transmit data from an unprivileged process in a Virtual Machine (VM) without any network hardware to a website, by tunneling the data through the DRAM-based covert channel to the JavaScript running in a web browser on the same host machine.

Our key contributions are:

- We performed a comprehensive evaluation of known and new mechanisms to obtain timestamps. We compared 10 methods on the 3 major browsers on Windows, Linux and Mac OS X, as well as on Tor browser.
- Our new timing methods increase the resolution of official methods by 3 to 4 orders of magnitude on all browsers, and by 8 orders of magnitude on Tor browser. Our evaluation therefore shows that reducing the resolution of timer interfaces does not mitigate any attack.
- We demonstrate the first DRAM-based side channel in JavaScript to exfiltrate data from a highly restricted execution environment inside a VM with no network interfaces.
- Our results underline that quick-fix mitigations are dangerous, as they can establish a false sense of security.

The remainder of this paper is organized as follows. In Section 2, we provide background information. In Section 3, we comprehensively evaluate new timing measurement methods on all major browsers. In Section 4, we demonstrate the revival of cache attacks with our new timing primitives as well as a new DRAM-based covert channel between JavaScript in a website and a process that is strictly isolated inside a VM with no network hardware. Finally, we discuss effective mitigation techniques in Section 5 and conclude in Section 6.

2 Background

2.1 Microarchitectural attacks

A large body of recent work has focused on cross-VM covert channels. A first class of work uses the CPU cache for covert communications. Ristenpart et al. [33] are the first to demonstrate a cache-based covert channel between two Amazon EC2 instances, yielding 0.2 bps. Xu et al. [47] optimized this covert channel and assessed the difference in performance between theoretical and practical results. They obtain 215.11 bps with an error rate of 5.12%. Maurice et al. [23] built a cross-VM covert channel, using the last-level cache and a Prime+Probe approach, that achieves a

bit rate of 751 bps with an error rate of 5.7%. Liu et al. [21] demonstrated a high-speed cache-based covert channel between two VMs that achieves transmission speeds of up to 600 Kbps and an error rate of less than 1%. In addition to the cache, covert channels have also been demonstrated using memory. Xiao et al. [46] demonstrated a memory-based covert channel using page deduplication. Wu et al. [45] built a covert channel of 746 bps with error correction, using the memory bus. Pessl et al. [32] reverse engineered the DRAM addressing functions that map physical addresses to their physical location inside the DRAM. The mapping allowed them to build a covert channel that relies solely on the DRAM as shared resource. Their cross-core cross-VM covert channel achieves a bandwidth of 309 Kbps. Maurice et al. [24] demonstrated an error-free covert channel between two Amazon EC2 instances of more than 360 Kbps, which allows building an SSH connection through the cache.

2.2 JavaScript and timing measurements

JavaScript is a scripting language supported by all modern browsers, which implement just-in-time compilation for performance. Contrary to low-level languages like C, JavaScript is strictly sandboxed and hides the notion of addresses and pointers. The concurrency model of JavaScript is based on a single-threaded *event loop* [26], which consists of a message queue and a call stack. Events are handled in the message queue, moved to the call stack when the stack is empty and processed to completion. As a drawback, if a message takes too long to process, it blocks other messages to be processed, and the browser becomes unresponsive. Browsers received the support for multithreading with the introduction of *web workers*. Each web worker runs in parallel and has its own event loop [26].

For timing measurement, the timestamp counter of Intel CPUs provides the number of CPU cycles since startup and thus a high-resolution timestamp. In native code, the timestamp counter is accessible through the unprivileged `rdtsc` instruction. In JavaScript, we cannot execute arbitrary instructions such as the `rdtsc` instruction. One of the timing primitives provided by JavaScript is the High Resolution Time API [40]. This API provides the `performance.now` method that gives a sub-millisecond timestamp. The W3C standard recommends that the timestamp should be monotonically increasing and accurate to 5 µs. The resolution may be lower if the hardware has no support for such a high resolution.

Remarkably, until Firefox 36 the High Resolution Time API returned timestamps accurate to one nanosecond. This is comparable to the native

`rdtsc` instruction which has a resolution of 0.5 ns on a 2 GHz CPU. As a response to the results of Oren et al. [28], the timer resolution was decreased for security reasons [3]. In recent versions of Chrome and WebKit, the timing resolution was also decreased to the suggested 5 µs [1, 6]. The Tor project even reduced the resolution to 100 ms [25]. The decreased resolution of the high-resolution timer is supposed to prevent time-based side-channel attacks. In a concurrent work, Kohlbrenner et al. [18] showed that it is possible to recover a high resolution by observing clock edges, as well as to create new implicit clocks using browser features. Additionally, they implemented fuzzy time that aims to degrade the native clock as well as all implicit clocks.

2.3 Timing attacks in JavaScript

Van Goethem et al. [37] showed different timing attacks in browsers based on the processing time of resources. They aimed to extract private data from users by estimating the size of cross-origin resources. Stone [35] showed that the optimization in SVG filters introduced timing side channels. He showed that this side channel can be used to extract pixel information from iframes.

Microarchitectural side channels have only recently been exploited in JavaScript. Oren et al. [28] showed that it is possible to mount cache attacks in JavaScript. They demonstrated how to generate an eviction set for the last-level cache that can be used to mount a Prime+Probe attack. Based on this attack, they built a covert channel using the last-level cache that is able to transmit data between two browser instances. Furthermore, they showed that the timer resolution is high enough to create a spy application that tracks the user's mouse movements and network activity. As described in Section 2.2, this attack caused all major browsers to decrease the resolution of the `performance.now` method.

Gruss et al. [9] demonstrated hardware faults triggered from JavaScript, exploiting the so-called Rowhammer bug. The Rowhammer bug occurs when repeatedly accessing the same DRAM cells with a high frequency [15]. This "hammering" leads to bit flips in neighboring DRAM rows. As memory accesses are usually cached, they also implemented cache eviction in JavaScript.

All these attacks require a different timestamp resolution. The attacks from Goethem et al. [37] and Stone [35] require a timestamp resolution that is on the order of a microsecond, while the attack of Oren et al. [28] relies on the fine-grained timestamps on the order of nanoseconds. More

generally, as microarchitectural side channel attacks aim at exploiting timing differences of a few CPU cycles, they depend on the availability of fine-grained timestamps. We note that decreasing the resolution therefore only mitigates microarchitectural attacks on the major browsers that have a resolution of 5 µs, but mitigates more side-channel attacks on the Tor browser which has a resolution of 100 ms.

3 Timing Measurements in the JavaScript Sandbox

This section describes techniques to get accurate measurements with a high-resolution timestamp in the browser. In the first part, we describe methods to recover a high resolution for the provided High Resolution Time API. The second part describes different techniques that allow deriving highly accurate timestamps, with *implicit* timers. These methods are summarized in Table 5.1.

3.1 Recovering a high resolution

In both Chrome and Webkit, the timer resolution is decreased by rounding the timestamp down to the nearest multiple of 5 µs. As our measurements fall below this resolution, they are all rounded down to 0. We refer to the underlying clock's resolution as *internal resolution* and to the decreased resolution of the provided timer as *provided resolution*. It has already been observed that it is possible to recover a high resolution by observing the clock edges [18, 22, 34, 38]. The clock edge aligns the timestamp perfectly to its resolution, *i.e.*, we know that the timestamp is an exact multiple of its provided resolution at this time.

Clock interpolation

As the underlying clock source has a high resolution, the difference between two clock edges varies only as much as the underlying clock. This property gives us a very accurate time base to build upon. As the time between two edges is always constant, we interpolate the time between them. This method has also been used in JavaScript in a concurrent work [18].

Clock interpolation requires a calibration before being able to return accurate timestamps. For this purpose, we repeatedly use a busy-wait loop to increment a counter between two clock edges. This gives us the number of steps we can use for the interpolation. We refer to the average number

of increments as *interpolation steps*. The time it takes to increment the counter once equals the resolution we are able to recover. It can be approximated by dividing the time difference of two clock edges by the number of interpolation steps. This makes the timer independent from both the internal and the provided resolution.

The measurement with the improved resolution works as follows. We busy wait until we observe a clock edge. At this point, we start with the operation we want to time. After the timed operation has finished, we again busy wait for the next clock edge while incrementing a counter. We assume that the increment operation is a constant time operation, thus allowing us to linearly interpolate the passed time. From the calibration, we know the time of one interpolation step which will be a fraction of the provided resolution. Multiplying this time by the number of increments results in the interpolated time. Adding the interpolated time to the measured time increases the timer's resolution again.

Using this method, we recover a highly accurate timestamp.

Listing 7.1 shows the JavaScript implementation. Table 5.1 shows the recovered resolution for various values of provided resolution. Even for a timer rounded down to a multiple of 100 ms, we recover a resolution of 15 µs.

Edge thresholding

We do not require an exact timestamp in all cases. For many side-channel attacks it is sufficient to distinguish two operations f_{fast} and f_{slow} based on their execution time. We refer to the execution times of the short-running function and long-running function as t_{fast} and t_{slow} respectively.

We devise a new method that we call edge thresholding. This method again relies on the property that we can execute multiple constant-time operations between two edges of the clock. Edge thresholding works as long as the difference in the execution time is larger than the time it takes to execute one such constant-time operation. Figure 5.1 illustrates the main idea of edge thresholding. Using multiple constant-time operations, we generate a padding after the function we want to measure. The execution time of the padding $t_{padding}$ is included into the measurement, increasing the total execution time by a constant value. The size of the padding depends on the provided resolution and on the execution time of the functions. We choose the padding in such a way that $t_{slow} + t_{padding}$ crosses one more clock edge than $t_{fast} + t_{padding}$, *i.e.*, both functions take a different amount of clock edges.

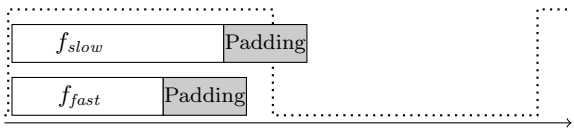


Figure 5.1: Edge thresholding: apply padding such that the slow function crosses one more clock edge than the fast function.

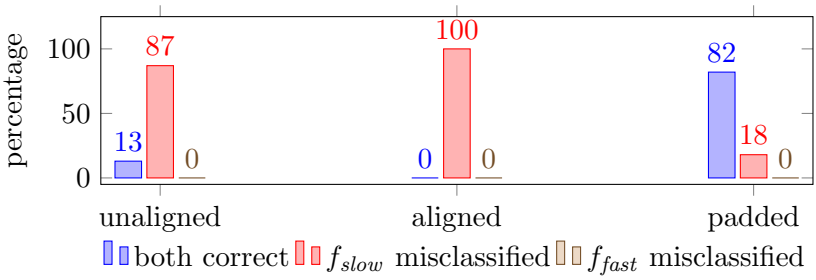


Figure 5.2: Results of edge thresholding where the difference between the function's execution time is less then the provided resolution.

To choose the correct padding, we start without padding and increase the padding gradually. We align the function start at a clock edge and measure the number of clock edges it takes to execute the short-running and the long-running function. As soon as the long-running function crosses one more clock edge than the short-running function, we have found a working padding. Subsequently, this padding is used for all execution time measurements. Figure 5.2 shows the results of classifying two functions with an execution time difference of 0.9 µs and a provided resolution of 10 µs. A normal, unaligned measurement is able to classify the two functions only in the case when one of the measurements crosses a clock edge, whereas the edge thresholding method categorizes over 80% of the function calls correctly by their relative execution time. Moreover, there are no false classifications.

3.2 Alternative timing primitives

In cases where the High Resolution Time API [40] is not available, e.g., on Tor browser, we have to resort to different timing primitives, as highlighted by Kohlbrenner et al. [18]. As there is no different high-resolution timer available in JavaScript and we cannot access any native timers, we have to create our own timing sources. In most cases, it is sufficient to have a fast-paced monotonically increasing counter as a timing primitive that

is not a real representation of time but an approximation of a highly accurate monotonic timer. While this concept was already presented by Wray in 1992 [44], Lipp et al. [20] recently demonstrated a practical high-resolution timing primitive on ARM using a counting thread. As JavaScript is inherently based on a single threaded event loop with no true concurrency, the timing primitive has to be based either on recurring events or non-JavaScript browser features.

We present several novel methods to construct timing primitives in JavaScript. We refer to them as *free-running timers* and *blocking timers*. Free-running timers do not depend on the JavaScript's event loop and run independently from the remaining code. Blocking timers are based on JavaScript events and are either only usable to recover a high resolution or in combination with web workers. If used in combination with web workers, the timers become free-running timers.

At first, it seems that timing primitives blocking the JavaScript event loop might not be useful at all. The higher the resolution of the timing primitive, the more events are added to the event queue and the less time remains for actual code. However, there are still two constructions that are able to use such primitives. First, these primitives can be used for very accurate interpolation steps when applying either clock interpolation or edge thresholding. Second, it is possible to take advantage of the multithreading support with web workers to run the timing primitive in parallel to the method to time.

Timeouts

The first asynchronous feature dating back to the introduction of JavaScript is the WindowTimers API. Specifically the `setTimeout` and `setInterval` functions allow scheduling a timer-based callback. The time is specified in a millisecond resolution. After specifying the timeout, the browser keeps track of the timer and calls the callback as soon as the timer has expired.

A concurrent timer-based callback allows us to simulate a counting thread. We create a callback function that increments a global counter and schedules itself again using the `setTimeout` function. This method has also been used in a concurrent work [18]. Although the minimal supported timeout is 0, the real timeout is usually larger. The HTML5 specification defines a timeout of at least 4 ms for nested timers, *i.e.*, specifying the timeout from within the callback function has a delay of at least 4 ms [42].

This limitation also applies to timeouts specified by the `setInterval` function.

Most browsers comply to the HTML5 specification and treat all timeouts below 4 ms as 4 ms. In Firefox, the minimum timeout is determined by the value of the flag `dom.min_timeout_value` which defaults to 4 ms as well. Note that the timeout only has such a high frequency if it is run in an active tab. Background tasks do not allow such high frequencies.

Microsoft implemented another timeout function in their browsers which is not standardized. The `setImmediate` function behaves similarly to the `setTimeout` function with a timeout of 0. The function is not limited to 4 ms and allows to build a high-resolution counting thread. A counting thread using this function results in a resolution of up to 50 µs which is three orders of magnitude higher than the `setTimeout` method.

Message passing

By default, the browser enforces a same-origin policy, *i.e.*, scripts are not allowed to access web page data from a page that is served from a different domain. JavaScript provides a secure mechanism to circumvent the same-origin policy and to allow cross-origin communication. Scripts can install message listeners to receive message events from cross-origin scripts. A script from a different origin is allowed to post messages to a listener.

Despite the intended use for cross-origin communication, we can use this mechanism within one script as well. The message listener is not limited to messages sent from cross-origin scripts. Neither is there any limitation for the target of a posted message. Adding checks whether a message should be handled is left to the JavaScript developer. According to the HTML standard, posted messages are added to the event queue, *i.e.*, the message will be handled after any pending event is handled. This behavior leads to a nearly immediate execution of the installed message handler. A counting thread using the `postMessage` functions achieves a resolution of up to 35 µs. An implementation is shown in Listing 7.2.

To obtain a free-running timing primitive, we have to move the message posting into separate web workers. This appears to be a straightforward task. However, there are certain limitations for web workers. Web workers cannot post messages to other web workers (including themselves). They can only post messages to the main thread and web workers they spawn, so called sub workers. Posting messages to the main thread again blocks the main thread's event loop, leaving sub web workers as the only viable

option. Listing 7.3 shows a sample implementation using one worker and one sub worker. The worker can communicate with the main thread and the sub worker. If the worker receives a message from the main thread, it sends back its current counter value. Otherwise, the worker continuously "requests" the current counter value from the sub worker. The sub worker increments the counter on each request and sends the current value back to the worker. The resulting resolution is even higher than with the blocking version of the method. On Tor browser, the achieved resolution is up to 15 µs, which is 4 orders of magnitude higher than the resolution of the native timer.

An alternative to sub workers are broadcast channels. Broadcast channels allow the communication between different sources from the same origin. A broadcast channel is identified by its name. In order to subscribe to a channel, a worker can create a `BroadcastChannel` object with the same name as an existing channel. A message that is posted to the broadcast channel is received by all other clients subscribed to this broadcast channel. We can build a construct that is similar to the sub worker scenario using two web workers. The web workers broadcast a message in their broadcast receiver to send the counter value back and forth. One of the web workers also responds to messages from the main thread to return the current counter value. With a resolution of up to 55 µs, this method is still almost as fast as the worker thread variant.

Message Channel

The Channel Messaging API provides bi-directional pipes to connect two clients. The endpoints of the pipe are called ports, and every port can both send and receive data. A message channel can be used in a similar way as cross-origin message passing. Listing 7.4 shows a simple blocking counting thread using a message channel.

As with the cross-origin message passing method, we can also adapt this code to work inside a web worker yielding a free-running timing primitive. Listing 7.5 shows the implementation for web workers. The resolution for the free-running message channel method is up to 30 µs, which is lower compared to the cross-origin communication method. However, it is currently the only method that works across browsers and has a resolution in the order of microseconds.

CSS animations

With CSS version 3, the support for animations [39] was added. These animations are independent of JavaScript and are rendered by the browser. Users can specify keyframes and attributes that will then be animated without any further user interaction.

We demonstrate a new method that uses CSS animations to build a timing primitive. A different method using CSS animations has been used in a concurrent work [18]. We define an animation that changes the width of an element from 0 px to 1 000 000 px within 1 s. Theoretically, if all animation steps are calculated, the current width is incremented every microsecond. However, browsers limit the CSS animations to 60 fps, *i.e.*, the resolution of our timing primitive is 16 ms in the best case. Indeed, most monitors have a maximum refresh rate of 60 Hz, *i.e.*, they cannot display more than 60 fps. Thus, a higher frame rate would only waste resources without any benefit. To get the current timestamp, we retrieve the current width of the element. In JavaScript, we can get the current width of the element using `window.getComputedStyle(elem, null).getPropertyValue("width")`.

SharedArrayBuffer

Web workers do not have access to any shared resource. The communication is only possible via messages. If data is passed using a message, either the data is copied, or the ownership of the data is transferred. This design prevents race conditions and locking problems without having to depend on a correct use of locks. Due to the overhead of message passing for high-bandwidth applications, approaches for sharing data between workers are discussed by the ECMAScript committee [27]. An experimental extension for web workers is the `SharedArrayBuffer`. The ownership of such a buffer can be shared among multiple workers, which can access the buffer simultaneously.

A shared resource provides a way to build a real counting thread with a negligible overhead compared to a message passing approach. This already raised concerns with respect to the creation of a high-resolution clock [19]. In this method, one worker continuously increments the value of the buffer without checking for any events on the event queue. The main thread simply reads the current value from the shared buffer and uses it as a high-resolution timestamp.

We implemented a clock with a parallel counting thread using the `SharedArrayBuffer`. An implementation is shown in Listing 7.6. The

resulting resolution is close to the resolution of the native timestamp counter. On our Intel Core i5 test machine, we achieve a resolution of up to 2 ns using the shared array buffer. This is equivalent to a resolution of only 4 CPU cycles, which is 3 orders of magnitude better than the timestamp provided by `performance.now`.

3.3 Evaluation

We evaluated all methods on an Intel Core i5-6200U machine using the most popular browsers, up to date at the time of writing: Firefox 51, Chrome 53, Edge 38.14393.0.0, and Tor 6.0.4. All tests were run on Ubuntu 16.10, Windows 10, and Mac OS X 10.11.4. Table 5.1 shows the timing resolution of every method for every browser and operating system combination. We also evaluated our methods using Fuzzyfox [17], the fork of Firefox hardened against timing attacks [18].

The introduction of multithreading in JavaScript opened several possibilities to build a timing primitive that does not rely on any provided timer. By building a counting thread, we are able to get a timer resolution of several microseconds. This is especially alarming for the Tor browser, where the provided timer only has a resolution of 100 ms. Using the demonstrated methods, we can build a reliable timer with a resolution of up to 15 µs. The lower resolution was implemented as a side channel mitigation and is rendered useless when considering the results of the alternative timing primitives.

The best direct timing source we tested is the experimental `SharedArrayBuffer`. The best measurement method we tested is edge thresholding. Both increase the resolution by at least 3 orders of magnitude compared to `performance.now` in all browsers. Countermeasures against timing side-channels using fuzzy time have been proposed by Hu et al. [12] and Vattikonda et al. [38]. They suggested to reduce the provided resolution and to randomize the clock edges. However, we can fall back to the constructed timing primitives if this countermeasure is not applied on all implicit clocks.

In a concurrent work, Kohlbrenner et al. [18] proposed Fuzzyfox, a fork of Firefox that uses fuzzy time on both explicit and implicit clocks, and aims to cap all clocks to a resolution of 100 ms. Our evaluation shows that the explicit timer `performance.now` is reduced to 100 ms, and is fuzzed enough that the interpolation and edge thresholding methods do not work to recover a high resolution. Similarly, some of the implicit timers, such as `setTimeout`, `postMessage`, and Message Channel, are also mitigated,

Table 5.1: Timing primitive resolutions on various browsers and operating systems.

	Free-running	Firefox 51	Chrome 53	Edge 38	Tor 6.0.4	Fuzzyfox
`performance.now`	✓	5 µs	5 µs	1 µs	100 ms	100 ms
CSS animations	✓	16 ms	16 ms	16 ms	16 ms	125 ms
setTimeout		4 ms	4 ms	2 ms	4 ms	100 ms
setImmediate		–	–	50 µs	–	–
postMessage		45 µs	35 µs	40 µs	40 µs	47 ms
Sub worker	✓	20 µs	–[2]	50 µs	15 µs	–
Broadcast Channel	✓	145 µs	–	–	55 µs	760 µs
MessageChannel		12 µs	55 µs	20 µs	20 µs	45 ms
MessageChannel (W)	✓	75 µs	100 µs	20 µs	30 µs	1120 µs
SharedArrayBuffer	✓	2 ns[3]	15 ns[4]	–	–	2 ns
Interpolation[1]		500 ns	500 ns	350 ns	15 µs	–
Edge thresholding[1]		2 ns	15 ns	10 ns	2 ns	–

with a resolution between 45 ms and 100 ms. However, the Broadcast Channel, Message Channel with web workers, and SharedArrayBuffer still have a fine grained resolution, between 2 ns and 1 ms. It is to be noted that, while these methods stay accurate, the resulting clock is too fuzzy to derive a finer clock with either interpolation or edge thresholding.

4 Reviving and Extending Microarchitectural Attacks

In this section, we demonstrate that with our timing primitives, we are able to revive attacks that were thought mitigated, and build new DRAM-based attacks.

4.1 Reviving Cache Attacks

Oren et al. [28] presented the first microarchitectural side-channel attack running in JavaScript. Their attack was mitigated by decreasing the timer resolution. We verified that the attack indeed does not work anymore on

[1] Uses `performance.now` for coarse-grained timing information.

[2] Sub workers do not work in Chrome, this is a known issue since 2010 [7].

[3] Currently only available in the nightly version.

[4] It has to be enabled by starting Chrome with `--js-flags=--harmony-sharedarraybuffer --enable-blink-feature=SharedArrayBuffer`.

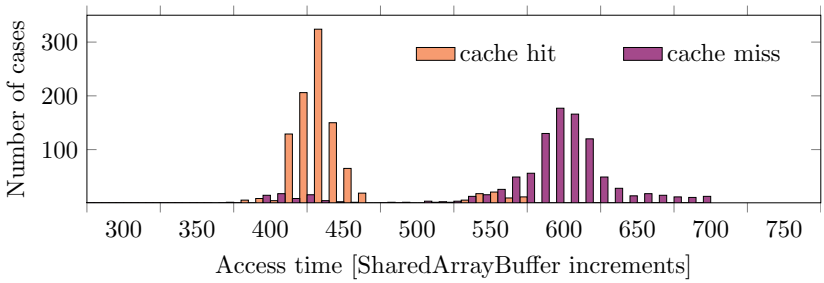


Figure 5.3: Histogram for cache hits and cache misses.

current browser versions. However, we are able to revive cache attacks by using our newly discovered timing sources. Figure 5.3 shows the timing difference between cache hits and cache misses, measured with the `SharedArrayBuffer` method. The ability to measure this timing difference is the building block of all cache attacks.

4.2 A New DRAM-based Covert Channel

Pessl et al. [32] established that timing differences in memory accesses can be exploited to build a cross-CPU covert channel. We demonstrate that this attack is also possible using JavaScript. In our scenario, the sender is an unprivileged binary inside a VM without a network connection. The receiver is implemented in sandboxed JavaScript running in a browser outside the VM, on the same host.

Overview

To communicate, the sender and the receiver agree on a certain bank and row of physical memory. This agreement can be done in advance and is not part of the transmission. The receiver continuously measures the access time to a value located inside the agreed row. For continuous accesses, the value will be cached in the row buffer and the access will be fast, resulting in a low access time. The receiver considers this as a 0. If the sender wants to transmit a 1, it accesses a different row inside the same bank. This access triggers a row conflict, resulting in a replacement of the row buffer content. On the receiver's next access, the request cannot be served from the row buffer but has to be fetched from the DRAM, resulting in a high access time.

Challenges

For the sender, we assume that we can run arbitrary unprivileged binary programs inside the VM. We implement the sender in C, which allows us to use the computer's high-resolution timestamp counter. Furthermore, we can flush addresses from the cache using the unprivileged *clflush* instruction. The only limitation on the sender is the absence of physical addresses.

On the receiver side, as the covert channel relies on timing differences that are in the order of tens of nanoseconds, we require a high-resolution timing primitive. We presented in Section 3 different methods to build timing primitives if the provided High Resolution Time API is not accurate enough. However, implementing this side channel in JavaScript poses some problems besides high-resolution timers. First, the DRAM mapping function requires the physical address to compute the physical location, *i.e.*, the row and the bank, inside the DRAM. However, JavaScript does not know the concept of pointers. Therefore, we neither have access to virtual nor physical addresses. Second, we have to ensure that memory accesses will always be served from memory and not the cache, *i.e.*, we have to circumvent the cache. Finally, the noise present on the system might lead to corrupt transfers. We have to be able to detect such bit inversions for reliable communication.

Address selection

The DRAM mapping function reverse engineered by Pessl et al. [32] takes a physical address and calculates the corresponding physical memory location. Due to the absence of addresses in JavaScript, we cannot simply use these functions. We have to rely on another side channel to be able to infer address bits in JavaScript.

We exploit the fact that heap memory in JavaScript is allocated on demand, *i.e.*, the browser acquires additional heap memory from the operating system if this is required. These heap pages are internally backed by 2 MB pages, called Transparent Huge Pages (THP). Due to the way virtual memory works, for THPs, the 21 least-significant bits of a virtual and physical address are the same. On many systems, this is already sufficient as input to the DRAM mapping function. This applies to the sender as well, with the advantage that we know the virtual address which we can use immediately without any further actions.

To get the beginning of a THP in JavaScript, we iterate through an array of multiple megabytes while measuring the time it takes to access

the array element, similarly to Gruss et al. [9]. As the physical pages for these THPs are also mapped on-demand, a page fault occurs as soon as an allocated THP is accessed for the first time. Such an access takes significantly longer than an access to an already mapped page. Thus, higher timings for memory accesses with a distance of 2 MB indicate the beginning of a THP. At this array index, the 21 least-significant bits of both the virtual and the physical address are 0.

Cache circumvention

To measure DRAM access times we have to ensure that all our accesses go to the DRAM and not to the cache. In native code, we can rely on the `clflush` instruction. This unprivileged instruction flushes a virtual address from all cache levels, *i.e.*, the next access to the address is ensured to go to the DRAM.

However, in JavaScript we neither have access to the `clflush` instruction nor does JavaScript provide a function to flush the cache. Thus, we have to resort to cache eviction. Cache eviction is the process of filling the cache with new data until the data we want to flush is evicted from the cache. The straightforward way is to fill a buffer with the size of the last-level cache with data. However, this is not feasible in JavaScript as writing multiple megabytes of data is too slow. Moreover, on modern CPUs, it might not suffice to iteratively write to the buffer as the cache replacement policy is not pseudo-LRU since Ivy Bridge [43].

Gruss et al. [9] demonstrated fast cache eviction strategies for numerous CPUs. They showed that their functions have a success rate of more than 99.75% when implemented in JavaScript. We also rely on these functions to evict the address which we use for measuring the access time.

Transmission

To transmit data from inside the VM to the JavaScript, they have to agree on a common bank. It is not necessary to agree on a bank dynamically, it is sufficient to have the bank hardcoded in both programs. The sender and the receiver both choose a different row from this bank. Again, this can be hardcoded, and there is no requirement for an agreement protocol.

On the sender side, the application inside the VM continuously accesses a memory address in its row if it wants to transmit a binary 1. These accesses cause row conflicts with the receiver's row. To send a binary 0, the sender does nothing to not cause any row conflict. On the receiver side, the JavaScript constantly measures the access time to a memory

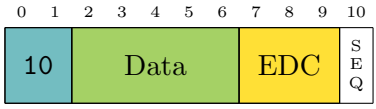


Figure 5.4: One packet of the covert channel. It has a 2-bit preamble ‘‘10’’, 5 data bits, 3 bits of error detection code and a 1 bit sequence number.

address from its row and evicts the address afterwards. If the sender has accessed its row, the access to the receiver's row results in a row conflict. As a row conflict takes significantly longer than a row hit, the receiver can determine if the sender has accessed its row.

To synchronize sender and receiver, the receiver measures the access time in a higher frequency than the sender is sending. The receiver maintains a constant-size sliding window that moves over all taken measurements. As soon as the majority of the measurements inside the sliding window is the same, one bit is received. The higher the receiver's sampling frequency is, compared to the sender's sending frequency, the lower the probability of wrongly measured bits. However, a higher sampling frequency also leads to a slower transmission speed due to the increased amount of redundant data.

Due to different noise sources on the system, we encounter transmission errors. Such noise sources are failed evictions, high DRAM activity of other programs or not being scheduled at all. To have a reliable transmission despite those interferences, we encapsulate the data into packets with sequence numbers and protect each packet with an error detection code as shown in Figure 5.4. The receiver is then able to detect any transmission error and to discard the packet. The sequence number ensures to keep the data stream synchronized. Thus, transmission errors only result in missing data, but the data stream is still synchronized after transmission errors. To deal with missing data, we can apply high-level error correction as shown by Maurice et al. [24].

Using the `SharedArrayBuffer`, we achieve a transmission rate of 11 bps for a 3 kB file with an error rate of 0% on our Intel Core i5 test machine. The system workload did not influence the transmission, as long as there is at least one core fully available to the covert channel. We optimized the covert channel for reliability and not speed. We expect that it is possible to further increase the transmission rate by using multiple banks to transmit data in parallel. However, the current speed is two orders of magnitude higher than the US government's minimum standard for covert channels[36].

5 Countermeasures

Lowering the timer resolution

As a reaction to the JavaScript cache attacks published by Oren et al. [28], browsers reduced the resolution of the high-resolution timer. Nevertheless, we are able to recover a higher resolution from the provided timer, as well as to build our own high-resolution timers.

Fuzzy time

Vattikonda et al. [38] suggested the concept of fuzzy time to get rid of high-resolution timers in hypervisors. Instead of rounding the timestamp to achieve a lower resolution, they move the clock edge randomly within one clock cycle. This method prevents the detection of the underlying clock edge and thus makes it impossible to recover the internal resolution. In a concurrent work, Kohlbrenner et al. [18] implemented the fuzzy time concept in Firefox to show that this method is also applicable in JavaScript. The implementation targets explicit clocks as well as implicit clocks. Nonetheless, we found different implicit clocks exceeding the intended resolution of 100 ms.

Shared memory and message passing

A proposed mitigation is to introduce thread affinity to the same CPU core for threads with shared memory [19]. This prevents true parallelism and should therefore prevent a real asynchronous timing primitive. However, we showed that even without shared memory we can achieve a resolution of up to 15 µs by using message passing. Enforcing the affinity to one core for all communicating threads would lead to a massive performance degradation and would effectively render the use of web workers useless. A compromise is to increase the latency of message passing which should not affect low- to moderate-bandwidth applications. Compared to Fuzzyfox's delay on the main event queue, this has two advantages. First, the overall usability impact is not as severe as only messages are delayed and not every event. Second, it also prevents the high accuracy of the Message Channel and Broadcast Channel method as the delay is not limited to the main event queue.

6 Conclusion and Outlook

High-resolution timers are a key requirement for side-channel attacks in browsers. As more side-channel attacks in JavaScript have been demonstrated against users' privacy, browser vendors decided to reduce the timer resolution.

In this article, we showed that this attempt to close these vulnerabilities was merely a quick-fix and did not address the underlying issue. We investigated different timing sources in JavaScript and found a number of timing sources with a resolution comparable to `performance.now`. This shows that even removing the interface entirely, would not have any effect. Even worse, an adversary can recover a resolution of the former `performance.now` implementation through measurement methods we proposed. We evaluated our new measurement methods on all major browsers as well as the Tor browser that has applied the highest penalty to the timer resolution. Our results are alarming for all browsers, including the privacy-conscious Tor browser, as we are able to recover a resolution in the order of nanoseconds in all cases. In addition to reviving attacks that were now deemed infeasible, we demonstrated the first DRAM-based side channel in JavaScript. In this side-channel attack, we implemented a covert channel between an unprivileged binary in a VM with no network interface and a JavaScript program in a browser outside the VM, on the same host.

While fuzzy timers can lower the resolution of the provided timer interfaces, we show that applying the same mitigation on all implicit clocks, including the one that are not discovered yet, is a complex task. Thus, we conclude that it is likely that an adversary can obtain sufficiently accurate timestamps when running arbitrary JavaScript code. As microarchitectural attacks are not restricted to JavaScript, we recommend to mitigate them at the system- or hardware-level.

Acknowledgments

We would like to thank our shepherd Jean Paul Degabriele, Georg Koppen from the Tor Browser project as well as all our anonymous reviewers. We would also like to thank the major browser vendors for their quick responses when reporting our findings. This project has received funding

from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

7 Appendix: JavaScript Code

```
1  function calibrate()
2  {
3    var counter = 0, next;
4    for(var i = 0; i < 10; i
       ++)
5    {
6        next = wait_edge();
7        counter += count_edge
       ();
8    }
9    next = wait_edge();
10   return (wait_edge() - next
       ) /
11     (counter / 10.0);
12 }
13
14 function measure(fnc)
15 {
16   var start = wait_edge();
17   fnc();
18   var count = count_edge();
19   return (performance.now()-
       start) - count *
       calibrate();
20 }
```

a: Clock interpolation.

```
1  function wait_edge()
2  {
3    var next, last =
       performance.now()
       ;
4    while((next =
       performance.now()
       ) == last) {}
5    return next;
6  }
7
8  function count_edge()
9  {
10   var last =
       performance.now()
       , count = 0;
11   while(performance.
       now() == last)
       count++;
12   return count;
13 }
```

b: Helper functions.

Listing 7.1: Clock interpolation: `calibrate` returns the time one increment takes, `measure` uses interpolation to measure the execution time of `fnc`

```
var count = 0;

function counter()
{
    count++;
    window.postMessage(null, window.location);
}
window.addEventListener("message", counter);
window.postMessage(null, window.location);
```

Listing 7.2: Abusing cross-origin communication to build a counting thread.

```
var ts = new
  Worker('subworker.js');
ts.postMessage(0);

function counter(event)
{
  timestamp = event.data;
}
ts.addEventListener("
    message", counter);
[...]

// get timestamp
ts.postMessage(0);
```

a: Timing measurement example.

```
var count = 0;

onmessage = function(event)
{
    count++;
    postMessage(count);
}
```

b: subworker2.js

```
var sub = new
  Worker("subworker2.
    js");
sub.postMessage(0);

var count = 0;

sub.onmessage = msg;
onmessage = msg;

function msg(event)
{
    if(event.data !=
    0)
    {
        count = event.
    data;
        sub.
    postMessage(0);
    }
    else
      self.postMessage
    (count);
}
```

c: subworker.js

Listing 7.3: Message passing with web workers to get a free-running timer.

```
var count = 0, channel = null;
function handleMessage(e)
{
    count++;
    channel.port2.postMessage(0);
}

channel = new MessageChannel();
channel.port1.onmessage = handleMessage;
channel.port2.postMessage(0);
```

Listing 7.4: A blocking timing primitive using a message channel.

```
var worker = new
  Worker("mcworker.js");
var main_channel = new
    MessageChannel();
var side_channel = new
    MessageChannel();

function handleMessage(e)
{
    timestamp = e.data;
}

main_channel.port2.
    onmessage =
    handleMessage;
worker.postMessage(0,
  [ main_channel.port1,
    side_channel.port1,
    side_channel.port2 ])
    ;
[...]

// get timestamp
main_channel.port2.
    postMessage(0);
```

a: Timing measurement example.

```
var main_port, port1,
    port2, count = 0;

self.onmessage =
    function(event)
{
    main_port = event.
    ports[0];
    port1 = event.ports
    [1];
    port2 = event.ports
    [2];
    main_port.onmessage
    =
      function()
      {
        main_port.
    postMessage(count);
      };
    port1.onmessage =
      function()
      {
        count++;
        port2.
    postMessage(0);
      };
    port2.postMessage(
    count);
};
```

b: mcworker.js

Listing 7.5: Message passing with web workers to get a free-running timer.

```
var buffer = new
    SharedArrayBuffer(16)
    ;
var counter = new
  Worker("counter.js");
counter.postMessage([
    buffer],
  [buffer]);
var arr = new
  Uint32Array(buffer);
[...]

timestamp = arr[0];
```

a: Timing measurement example.

```
self.onmessage =
    function(event)
{
  var [buffer] = event.
    data;
  var arr = new
    Uint32Array(buffer);
  while(1)
  {
    arr[0]++;
  }
}
```

b: counter.js

Listing 7.6: Parallel counting thread without additional overhead.

References

[1] Alex Christensen. *Reduce resolution of performance.now.* https://bugs.webkit.org/show_bug.cgi?id=146531. 2015.

[2] Daniel J. Bernstein. *Cache-Timing Attacks on AES.* http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2004.

[3] Boris Zbarsky. *Reduce resolution of performance.now.* https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab.

[4] Andrew Bortz and Dan Boneh. "Exposing private information by timing web applications." In: *WWW'07*. 2007.

[5] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector." In: *S&P'16*. 2016.

[6] Chromium. *window.performance.now does not support sub-millisecond precision on Windows.* https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110. 2015.

[7] Chromium Bug Tracker. *HTML5 nested workers are not supported in chromium.* https://bugs.chromium.org/p/chromium/issues/detail?id=31666. Retrieved on October 18, 2016. 2010.

[8] Edward W Felten and Michael A Schneider. "Timing attacks on web privacy." In: *CCS'00*. 2000.

[9] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA'16*. 2016.

[10] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *S&P'11*. 2011.

[11] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. "Scriptless attacks: stealing the pie without touching the sill." In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 760–771.

[12] Wei-Ming Hu. "Lattice Scheduling and Covert Channels." In: *S&P'92*. 1992, pp. 52–61.

[13] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "An empirical study of privacy-violating information flows in JavaScript web applications." In: *CCS'10*. 2010.

[14] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. "I know where you've been: Geo-inference attacks via the browser cache." In: *IEEE Internet Computing* 19.1 (2015), pp. 44–53.

[15] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors." In: *ISCA'14*. 2014.

[16] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems." In: *Crypto'96*. 1996, pp. 104–113.

[17] David Kohlbrenner and Hovav Shacham. *FTFuzzyfox*. `https://github.com/dkohlbre/gecko-dev/tree/fuzzyfox`. retrieved on January 23, 2017. 2016.

[18] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times." In: *USENIX Security Symposium*. 2016.

[19] Lars T Hansen. *Shared memory: Side-channel information leaks*. `https://github.com/tc39/ecmascript_sharedmem/blob/master/issues/TimingAttack.md`. 2016.

[20] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. 2016.

[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *S&P'15*. 2015.

[22] Robert Martin, John Demme, and Simha Sethumadhavan. "TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks." In: *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*. 2012.

[23] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel." In: *DIMVA'15*. 2015.

[24] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS'17*. to appear. 2017.

[25] Mike Perry. *Bug 1517: Reduce precision of time for Javascript.* `https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517`. 2015.

[26] Mozilla Developer Network. *Concurrency model and Event Loop.* `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`. 2016.

[27] Mozilla Inc. *ECMAScript Shared Memory and Atomics.* `http://tc39.github.io/ecmascript_sharedmem/shmem.html`. 2016.

[28] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS'15*. 2015.

[29] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA 2006*. 2006.

[30] Dan Page. "Theoretical use of cache memory as a cryptanalytic side-channel." In: *Cryptology ePrint Archive, Report 2002/169* (2002).

[31] Colin Percival. "Cache missing for fun and profit." In: *Proceedings of BSDCan*. 2005.

[32] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium*. 2016.

[33] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds." In: *CCS'09*. 2009.

[34] Mark Seaborn. *Comment on ECMAScript Shared Memory and Atomics.* `https://github.com/tc39/ecmascript_sharedmem/issues/1#issuecomment-144171031`. 2015.

[35] Paul Stone. "Pixel perfect timing attacks with HTML5." In: *Context Information Security (White Paper)* (2013).

[36] U.S. Department of Defense. *Trusted computing system evaluation "The Orange Book"*. Technical Report 5200.28-STD. 1985.

[37] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. "The clock is still ticking: Timing attacks in the modern web." In: *CCS'15*. 2015.

[38] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. "Eliminating fine grained timers in Xen." In: *CCSW'11*. 2011.

[39] W3C. *CSS Animations*. https://www.w3.org/TR/css3-animations/. 2016.

[40] W3C. *High Resolution Time Level 2*. https://www.w3.org/TR/hr-time/. 2016.

[41] Zachary Weinberg, Eric Y Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks." In: *S&P'11*. 2011.

[42] WHATWG. *HTML Living Standard – Timers*. https://html.spec.whatwg.org/multipage/webappapis.html#timers. Retrieved on October 18, 2016. 2016.

[43] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/. Retrieved on October 18, 2016.

[44] John C Wray. "An analysis of covert timing channels." In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

[45] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud." In: *IEEE/ACM Transactions on Networking* (2014).

[46] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. "A covert channel construction in a virtualized environment." In: *CCS'12*. 2012.

[47] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. "An exploration of L2 cache covert channels in virtualized environments." In: *CCSW'11*. 2011.

[48] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

6

Malware Guard Extension: Using SGX to Conceal Cache Attacks

Publication Data

Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *DIMVA*. 2017

Contributions

Main author.

Malware Guard Extension: Using SGX to Conceal Cache Attacks

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard

Graz University of Technology, Austria

Abstract

In modern computer systems, user processes are isolated from each other by the operating system and the hardware. Additionally, in a cloud scenario it is crucial that the hypervisor isolates tenants from other tenants that are co-located on the same physical machine. However, the hypervisor does not protect tenants against the cloud provider and thus the supplied operating system and hardware. Intel SGX provides a mechanism that addresses this scenario. It aims at protecting user-level software from attacks from other processes, the operating system, and even physical attackers.

In this paper, we demonstrate fine-grained software-based side-channel attacks from a malicious SGX enclave targeting co-located enclaves. Our attack is the first malware running on real SGX hardware, abusing SGX protection features to conceal itself. Furthermore, we demonstrate our attack both in a native environment and across multiple Docker containers. We perform a Prime+Probe cache side-channel attack on a co-located SGX enclave running an up-to-date RSA implementation that uses a constant-time multiplication primitive. The attack works although in SGX enclaves there are no timers, no large pages, no physical addresses, and no shared memory. In a semi-synchronous attack, we extract 96 % of an RSA private key from a single trace. We extract the full RSA private key in an automated attack from 11 traces.

1 Introduction

Modern operating systems isolate user processes from each other to protect secrets in different processes. Such secrets include passwords stored in password managers or private keys to access company networks. Leakage of

The original publication is available at `https://link.springer.com/chapter/10.1007/978-3-319-60876-1_1`.

these secrets can compromise both private and corporate systems. Similar problems arise in the cloud. Therefore, cloud providers use virtualization as an additional protection using a hypervisor. The hypervisor isolates different tenants that are co-located on the same physical machine. However, the hypervisor does not protect tenants against a possibly malicious cloud provider.

Although hypervisors provide functional isolation, side-channel attacks are often not considered. Consequently, researchers have demonstrated various side-channel attacks, especially those exploiting the cache [15]. Cache side-channel attacks can recover cryptographic secrets, such as AES [29] and RSA [33] keys, across virtual machine boundaries.

Intel introduced a new hardware extension SGX (Software Guard Extensions) [27] in their CPUs, starting with the Skylake microarchitecture. SGX is an isolation mechanism, aiming at protecting code and data from modification or disclosure even if all privileged software is malicious [10]. This protection uses special execution environments, so-called enclaves, which work on memory areas that are isolated from the operating system by the hardware. The memory area used by the enclaves is encrypted to protect application secrets from hardware attackers. Typical use cases include password input, password managers, and cryptographic operations. Intel recommends storing cryptographic keys inside enclaves and claims that side-channel attacks "are thwarted since the memory is protected by hardware encryption" [25].

Hardware-supported isolation also led to fear of super malware inside enclaves. Rutkowska [44] outlined a scenario where an enclave fetches encrypted malware from an external server and executes it within the enlave. In this scenario, it is impossible to debug, reverse engineer, or analyze the executed malware in any way. Costan et al. [10] eliminated this fear by arguing that enclaves always run with user space privileges and can neither issue syscalls nor perform any I/O operations. Moreover, SGX is a highly restrictive environment for implementing cache side-channel attacks. Both state-of-the-art malware and side-channel attacks rely on several primitives that are not available in SGX enclaves.

In this paper, we show that it is very well possible for enclave malware to attack its hosting system. We demonstrate a cross-enclave cache attack from within a malicious enclave that is extracting secret keys from co-located enclaves. Our proof-of-concept malware is able to recover RSA keys by monitoring cache access patterns of an RSA signature process in a semi-synchronous attack. The malware code is completely invisible to the operating system and cannot be analyzed due to the isolation provided

by SGX. We present novel approaches to recover physical address bits, as well as to recover high-resolution timing in absence of the timestamp counter, which has an even higher resolution than the native one. In an even stronger attack scenario, we show that an additional isolation using Docker containers does not protect against this kind of attack.

We make the following contributions:

1. We demonstrate that, despite the restrictions of SGX, cache attacks can be performed from within an enclave to attack a co-located enclave.
2. By combining DRAM and cache side channels, we present a novel approach to recover physical address bits even if 2 MB pages are unavailable.
3. We obtain high-resolution timestamps in enclaves without access to the native timestamp counter, with an even higher resolution than the native one.
4. In an automated end-to-end attack on the wide-spread *mbedTLS* RSA implementation, we extract 96 % of an RSA private key from a single trace.

Section 2 presents the required background. Section 3 outlines the threat model and attack scenario. Section 4 describes the measurement methods and the online phase of the malware. Section 5 explains the offline-phase key recovery. Section 6 evaluates the attack against an up-to-date RSA implementation. Section 7 discusses several countermeasures. Section 8 concludes our work.

2 Background

2.1 Intel SGX in Native and Virtualized Environments

Intel Software Guard Extensions (SGX) are a new set of x86 instructions introduced with the Skylake microarchitecture. SGX allows protecting the execution of user programs in so-called enclaves. Only the enclave can access its own memory region, any other access to it is blocked by the CPU. As SGX enforces this policy in hardware, enclaves do not need to rely on the security of the operating system. In fact, with SGX the operating system is generally not trusted. By doing sensitive computation inside an enclave, one can effectively protect against traditional malware, even if such malware has obtained kernel privileges. Furthermore, it allows running secret code in a cloud environment without trusting hardware and operating system of the cloud provider.

An enclave resides in the virtual memory area of an ordinary application process. This virtual memory region of the enclave can only be backed by physically protected pages from the so-called Enclave Page Cache (EPC). The EPC itself is a contiguous physical block of memory in DRAM that is encrypted transparently to protect against hardware attacks.

Loading of enclaves is done by the operating system. To protect the integrity of enclave code, the loading procedure is measured by the CPU. If the resulting measurement does not match the value specified by the enclave developer, the CPU will refuse to run the enclave.

Since enclave code is known to the (untrusted) operating system, it cannot carry hard-coded secrets. Before giving secrets to an enclave, a provisioning party has to ensure that the enclave has not been tampered with. SGX therefore provides remote attestation, which proves correct enclave loading via the aforementioned enclave measurement.

At the time of writing, no hypervisor with SGX support was available. However, Arnautov et al. [4] proposed to combine Docker containers with SGX to create secure containers. Docker is an operating-system-level virtualization software that allows applications to run in separate containers. It is a standard runtime for containers on Linux which is supported by multiple public cloud providers. Unlike virtual machines, Docker containers share the kernel and other resources with the host system, requiring fewer resources than a virtual machine.

2.2 Microarchitectural Attacks

Microarchitectural attacks exploit hardware properties that allow inferring information on other processes running on the same system. In particular, cache attacks exploit the timing difference between the CPU cache and the main memory. They have been the most studied microarchitectural attacks for the past 20 years, and were found to be powerful to derive cryptographic secrets [15]. Modern attacks target the last-level cache, which is shared among all CPU cores. Last-level caches (LLC) are usually built as n-way set-associative caches. They consist of S cache sets and each cache set consists of n cache ways with a size of 64 B. The lowest 6 physical address bits determine the byte offset within a cache way, the following $\log_2 S$ bits starting with bit 6 determine the cache set.

Prime+Probe is a cache attack technique that has first been used by Osvik et al. [39]. In a Prime+Probe attack, the attacker constantly primes (*i.e.*, evicts) a cache set and measures how long this step took. The runtime of the prime step is correlated to the number of cache ways

that have been replaced by other programs. This allows deriving whether or not a victim application performed a specific secret-dependent memory access. Recent work has shown that this technique can even be used across virtual machine boundaries [33, 34].

To prime (*i.e.*, evict) a cache set, the attacker uses n addresses in same cache set (*i.e.*, an *eviction set*), where n depends on the cache replacement policy and the number of ways. To minimize the amount of time the prime step takes, it is necessary to find a minimal n combined with a fast access pattern (*i.e.*, an *eviction strategy*). Gruss et al. [20] experimentally found efficient eviction strategies with high eviction rates and a small number of addresses. We use their eviction strategy on our Skylake test machine throughout the paper.

Pessl et al. [42] found a similar attack through DRAM modules. Each DRAM module has a row buffer that holds the most recently accessed DRAM row. While accesses to this buffer are fast, accesses to other memory locations in DRAM are much slower. This timing difference can be exploited to obtain fine-grained information across virtual machine boundaries.

2.3 Side-Channel Attacks on SGX

Intel claims that SGX features impair side-channel attacks and recommends using SGX enclaves to protect password managers and cryptographic keys against side channels [25]. However, there have been speculations that SGX could be vulnerable to side-channel attacks [10]. Xu et al. [50] showed that SGX is vulnerable to page fault side-channel attacks from a malicious operating system [1].

SGX enclaves generally do not share memory with other enclaves, the operating system or other processes. Thus, any attack requiring shared memory is not possible, e.g., Flush+Reload [51]. Also, DRAM-based attacks cannot be performed from a malicious operating system, as the hardware prevents any operating system accesses to DRAM rows in the EPC. However, enclaves can mount DRAM-based attacks on other enclaves because all enclaves are located in the same physical EPC.

In concurrent work, Brasser et al. [8], Moghimi et al. [37] and Götzfried et al. [17] demonstrated cache attacks on SGX relying on a malicious operating system.

2.4 Side-Channel Attacks on RSA

RSA is widely used to create asymmetric signatures, and is implemented by virtually every TLS library, such as OpenSSL or *mbedTLS*, which is used for instance in cURL and OpenVPN. RSA essentially involves modular exponentiation with a private key, typically using a square-and-multiply algorithm. An unprotected implementation of square-and-multiply is vulnerable to a variety of side-channel attacks, in which an attacker learns the exponent by distinguishing the square step from the multiplication step [15, 51]. *mbedTLS* uses a windowed square-and-multiply routine for the exponentiation. Liu et al. [33] showed that if an attack on a window size of 1 is possible, the attack can be extended to arbitrary window sizes.

Earlier versions of *mbedTLS* were vulnerable to a timing side-channel attack on RSA-CRT [3]. Due to this attack, current versions of *mbedTLS* implement a constant-time Montgomery multiplication for RSA. Additionally, instead of using a dedicated square routine, the square operation is carried out using the multiplication routine. Thus, there is no leakage from a different square and multiplication routine as exploited in previous attacks on square-and-multiply algorithms [33, 51]. However, Liu et al. [33] showed that the secret-dependent accesses to the buffer b still leak the exponent. Boneh et al. [7] and Blömer et al. [6] recovered the full RSA private key if only parts of the key bits are known.

3 Threat Model and Attack Setup

In this section, we present our threat model. We demonstrate a malware that circumvents SGX and Docker isolation guarantees. We successfully mount a Prime+Probe attack on an RSA signature computation running inside a different enclave, on the outside world, and across container boundaries.

3.1 High-Level View of the Attack

In our threat model, both the attacker and the victim are running on the same physical machine. The machine can either be a user's local computer or a host in the cloud. In the cloud scenario, the victim has its enclave running in a Docker container to provide services to other applications running on the host. Docker containers are well supported on many cloud providers, e.g., Amazon [13] or Microsoft Azure [36]. As these containers are more lightweight than virtual machines, a host can run up to several

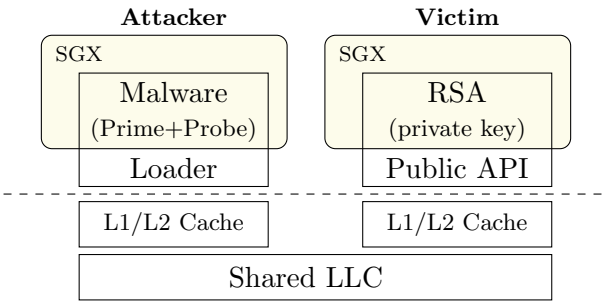


Figure 6.1: The threat model: both attacker and victim run on the same physical machine in different SGX enclaves.

hundred containers simultaneously. Thus, the attacker has good chances to get a co-located container on a cloud provider.

Figure 6.1 gives an overview of our native setup. The victim runs a cryptographic computation inside the enclave to protect it against any attacks. The attacker tries to stealthily extract secrets from this victim enclave. Both the attacker and the victim use Intel SGX features and thus are subdivided into two parts, the enclave and loader, *i.e.*, the main program instantiating the enclave.

The attack is a multi-step process that can be divided into an online and an offline phase. Section 4 describes the online phase, in which the attacker first locates the victim's cache sets that contain the secret-dependent data of the RSA private key. The attacker then monitors the identified cache sets while triggering a signature computation. Section 5 gives a detailed explanation of the offline phase in which the attacker recovers a private key from collected traces.

3.2 Victim

The victim is an unprivileged program that uses SGX to protect an RSA signing application from both software and hardware attackers. Both the RSA implementation and the private key reside inside the enclave, as suggested by Intel [25]. Thus, they can never be accessed by system software or malware on the same host. Moreover, memory encryption prevents physical information leakage in DRAM. The victim uses the RSA implementation of the widely deployed *mbedTLS* library. The *mbedTLS* library implements a windowed square-and-multiply algorithm, that relies on constant-time Montgomery multiplications. The window size is fixed to

1, as suggested by the official knowledge base [2]. The victim application provides an API to compute a signature for provided data.

3.3 Attacker

The attacker runs an unprivileged program on the same host machine as the victim. The goal of the attacker is to stealthily extract the private key from the victim enclave. Therefore, the attacker uses the API provided by the victim to trigger signature computations.

The attacker targets the exponentiation step of the RSA implementation. The attack works on arbitrary window sizes [33], including window size 1. To prevent information leakage from function calls, *mbedTLS* uses the same function (`mpi_montmul`) for both the square and the multiply operation. The `mpi_montmul` takes two parameters that are multiplied together. For the square operation, the function is called with the current buffer as both arguments. For the multiply operation, the current buffer is multiplied with a buffer holding the multiplier. This buffer is allocated in the calling function `mbedtls_mpi_exp_mod` using `calloc`. Due to the deterministic behavior of the tlibc `calloc` implementation, the used buffers always have the same virtual and physical addresses and thus the same cache sets. The attacker can therefore mount a Prime+Probe attack on the cache sets containing the buffer.

In order to remain stealthy, all parts of the malware that contain attack code reside inside an SGX enclave. The enclave can protect the encrypted real attack code by only decrypting it after a successful remote attestation after which the enclave receives the decryption key. As pages in SGX can be mapped as writable and executable, self-modifying code is possible and therefore code can be encrypted. Consequently, the attack is completely stealthy and invisible from anti-virus software and even from monitoring software running in ring 0. Note that our proof-of-concept implementation does not encrypt the attack code as this has no impact on the attack.

The loader does not contain any suspicious code or data, it is only required to start the enclave and send the exfiltrated data to the attacker.

3.4 Operating System and Hardware

Previous work was mostly focused on attacks on enclaves from untrusted cloud operating systems [10, 46]. However, in our attack we do not make any assumptions on the underlying operating system, *i.e.*, we do not rely on a malicious operating system. Both the attacker and the victim are

unprivileged user space applications. Our attack works on a fully-patched recent operating system with no known software vulnerabilities, *i.e.*, the attacker cannot elevate privileges.

We expect the cloud provider to run state-of-the-art malware detection software. We assume that the malware detection software is able to monitor the behavior of containers and inspect the content of containers. Moreover, the user can run anti-virus software and monitor programs inside the container. We assume that the protection mechanisms are either signature-based, behavioral-based, heuristics-based or use performance counters [12, 21].

Our only assumption on the hardware is that attacker and victim run on the same host system. This is the case on both personal computers and on co-located Docker instances in the cloud. As SGX is currently only available on Intel Skylake CPUs, it is valid to assume that the host is a Skylake system. Consequently, we know that the last-level cache is shared between all CPU cores.

4 Extracting Private Key Information

In this section, we describe the online phase of our attack. We first build primitives necessary to mount this attack. Then we show in two steps how to locate and monitor cache sets to extract private key information.

4.1 Attack Primitives in SGX

Successful Prime+Probe attacks require two primitives: a high-resolution timer to distinguish cache hits and misses and a method to generate an eviction set for arbitrary cache sets. Due to the restrictions of SGX enclaves, implementing Prime+Probe in enclaves is not straight-forward. Therefore, we require new techniques to build a malware from within an enclave.

High-resolution Timer

The unprivileged `rdtsc` and `rdtscp` instructions, which read the timestamp counter, are usually used for fine-grained timing outside enclaves. In SGX, these instructions are not permitted inside an enclave, as they might cause a VM exit [24]. Thus, we have to rely on a different timing source with a resolution in the order of 10 cycles to reliably distinguish cache hits from misses as well as DRAM row hits from row conflicts.

To achieve the highest number of increments, we handcraft a counter thread [31, 49] in inline assembly. The counter variable has to be accessible across threads, thus it is necessary to store the counter variable in memory. Memory addresses as operands incur an additional cost of approximately 4 cycles due to L1 cache access times [23]. On our test machine, a simple counting thread executing `1: incl (%rcx); jmp 1b` achieves one increment every 4.7 cycles, which is an improvement of approximately 2 % over the best code generated by `gcc`.

We can improve the performance—and thus the resolution—further, by exploiting the fact that only the counting thread modifies the counter variable. We can omit reading the counter variable from memory. Therefore, we introduce a "shadow counter variable" which is always held in a CPU register. The arithmetic operation (either `add` or `inc`) is performed on this register, unleashing the low latency and throughput of these instructions. As registers cannot be shared across threads, the shadow counter has to be moved to memory using the `mov` instruction after each increment. Similar to the `inc` and `add` instruction, the `mov` instruction has a latency of 1 cycle and a throughput of 0.5 cycles/instruction when copying a register to memory. The improved counting thread, `1: inc %rax; mov %rax, (%rcx), jmp 1b`, is significantly faster and increments the variable by one every 0.87 cycles, which is an improvement of 440 % over the simple counting thread. In fact, this version is even 15 % faster than the native timestamp counter, thus giving us a reliable timing source with even higher resolution. This new method might open new possibilities of side-channel attacks that leak information through timing on a sub-`rdtsc` level.

Eviction Set Generation

Prime+Probe relies on eviction sets, *i.e.*, we need to find virtual addresses that map to the same physical cache set. An unprivileged process cannot translate virtual to physical addresses and therefore cannot simply search for virtual addresses that fall into the same cache set. Liu et al. [33] and Maurice et al. [34] demonstrated algorithms to build eviction sets using large pages by exploiting the fact that the virtual address and the physical address have the same lowest 21 bits. As SGX does not support large pages, this approach is inapplicable. Oren et al. [38] and Gruss et al. [20] demonstrated automated methods to generate eviction sets for a given virtual address. Due to microarchitectural changes their approaches are

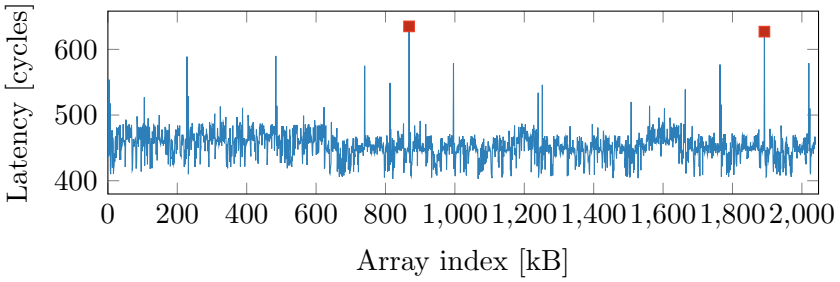


Figure 6.2: Access times when alternately accessing two addresses which are 64 B apart. The (marked) high access times indicate row conflicts.

either not applicable at all to the Skylake architecture or consume several hours on average before even starting the actual Prime+Probe attack.

We propose a new method to recover the cache set from a virtual address without relying on large pages. The idea is to exploit contiguous page allocation [28] and DRAM timing differences to recover DRAM row boundaries. The DRAM mapping functions [42] allow to recover physical address bits.

The DRAM organization into banks and rows causes timing differences. Alternately accessing pairs of two virtual addresses that map to the same DRAM bank but a different row is significantly slower than any other combination of virtual addresses. Figure 6.2 shows the average access time for address pairs when iterating over a 2 MB array. The highest two peaks show row conflicts, *i.e.*, the row index changes while the bank, rank, and channel stay the same.

To recover physical address bits we use the reverse-engineered DRAM mapping function as shown in Table 6.1. Our test machine is an Intel Core i5-6200U with 12 GB main memory. The row index is determined by physical address bits 18 and upwards. Hence, the first address of a DRAM row has the least-significant 18 bits of the physical address set to '0'. To

Table 6.1: Reverse-engineered DRAM mapping functions from Pessl et al. [42].

		Address Bit																	
		22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	
2 DIMMs	Channel				⊕	⊕					⊕	⊕			⊕	⊕			
	BG0									⊕							⊕		
	BG1	⊕				⊕													
	BA0				⊕				⊕										
	BA1		⊕				⊕												
	Rank			⊕				⊕											

detect row borders, we scan memory sequentially for an address pair in physical proximity that causes a *row conflict*. As SGX enclave memory is allocated contiguously we can perform this scan on virtual addresses.

A virtual address pair that causes row conflicts at the beginning of a row satisfies the following constraints:

1. The least-significant 18 physical address bits of one virtual address are zero. This constitutes a DRAM row border.
2. The bank address (BA), bank group (BG), rank, and channel determine the DRAM bank and must be the same for both virtual addresses.
3. The row index must be different for both addresses to cause a row conflict.
4. The difference of the two virtual addresses has to be at least 64 B (the size of one cache line) but should not exceed 4 kB (the size of one page).

Physical address bits 6 to 17 determine the cache set which we want to recover. Hence, we search for address pairs where physical address bits 6 to 17 have the same known but arbitrary value.

To find address pairs fulfilling the aforementioned constraints, we modeled the mapping function and the constraints as an SMT problem and used the Z3 theorem prover [11] to provide models satisfying the constraints. The model we found yields pairs of physical addresses where the upper address is 64 B apart from the lower one. There are four such address pairs within every 4 MB block of physical memory such that each pair maps to the same bank but a different row. The least-significant bits of the physical address pairs are either (`0x3fffc0`, `0x400000`), (`0x7fffc0`, `0x800000`), (`0xbfffc0`, `0xc00000`) or (`0xffffc0`, `0x1000000`) for the lower and higher address respectively. Thus, at least 22 bits of the higher addresses least-significant bits are 0. As the cache set is determined by the bits 6 to 17, the higher address has the cache set index 0. We observe that satisfying address pairs are always 256 KB apart. Since we have contiguous memory [28], we can generate addresses mapping to the same cache set by adding multiples of 256 KB to the higher address.

In modern CPUs, the last-level cache is split into cache slices. Addresses with the same cache set index map to different cache slices based on the remaining address bits. To generate an eviction set, it is necessary to only use addresses that map to the same cache set in the same cache slice. However, to calculate the cache slice, all bits of the physical address are required [35].

As we are not able to directly calculate the cache slice, we use another approach. We add our calculated addresses from the correct cache set to our eviction set until the eviction rate is sufficiently high. Then, we try to remove single addresses from the eviction set as long as the eviction rate does not drop. Thus, we remove all addresses that do not contribute to the eviction, and the result is a minimal eviction set. Our approach takes on average 2 seconds per cache set, as we already know that our addresses map to the correct cache set. This is nearly three orders of magnitude faster than the approach of Gruss et al. [20]. Older techniques that have been comparably fast do not work on current hardware anymore due to microarchitectural changes [33, 38].

4.2 Identifying and Monitoring Vulnerable Sets

With the reliable high-resolution timer and a method to generate eviction sets, we can mount the first stage of the attack and identify the vulnerable cache sets. As we do not have any information about the physical addresses of the victim, we have to scan the last-level cache for characteristic patterns corresponding to the signature process. We consecutively mount a Prime+Probe attack on every cache set while the victim is executing the exponentiation step.

We can then identify multiple cache sets showing the distinctive pattern of the signature operation. The number of cache sets depends on the RSA key size. Cache sets at the buffer boundaries might be used by neighboring buffers and are more likely to be prefetched [18, 51] and thus, prone to measurement errors. Consequently, we use cache sets neither at the start nor the end of the buffer.

The measurement method is the same as for detecting the vulnerable cache sets, *i.e.*, we again use Prime+Probe. Due to the deterministic behavior of the heap allocation, the address of the attacked buffer does not change on consecutive exponentiations. Thus, we can collect multiple traces of the signature process.

To maintain a high sampling rate, we keep the post-processing during the measurements to a minimum. Moreover, it is important to keep the memory activity at a minimum to not introduce additional noise on the cache. Thus, we only save the timestamps of the cache misses for further post-processing. As a cache miss takes longer than a cache hit, the effective sampling rate varies depending on the number of cache misses. We have to consider this effect in the post-processing as it induces a non-constant sampling interval.

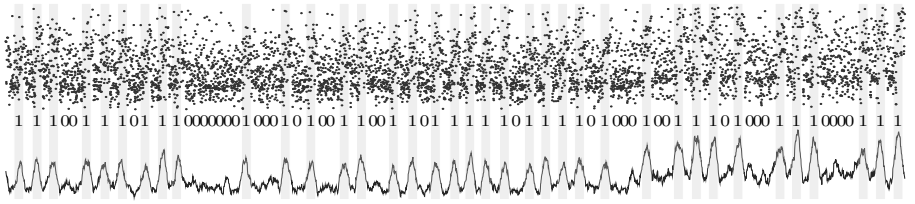


Figure 6.3: A raw measurement trace over 4 000 000 cycles. The peaks in the pre-processed trace on the bottom clearly indicate '1's.

5 Recovering the Private Key

In this section, we describe the offline phase of our attack: recovering the private key from the recorded traces of the victim enclave. This can either be done inside the malware enclave or on the attacker's server.

Ideally, an attacker would combine multiple traces by aligning them and averaging out noise. From the averaged trace, the private key can be extracted more easily. However, most noise sources, such as context switches, system activity and varying CPU clock, alter the timing, thus making trace alignment difficult. We pre-process all traces individually and extract a partial key out of each trace. These partial keys likely suffer from random insertion and deletion errors as well as from bit flips. To eliminate the errors, we combine multiple partial keys in the key recovery phase. This approach has much lower computational overhead than trace alignment since key recovery is performed on partial 4096-bit keys instead of full traces containing several thousand measurements.

Key recovery comes in three steps. First, traces are pre-processed. Second, a partial key is extracted from each trace. Third, the partial keys are merged to recover the private key. In the pre-processing step we filter and resample raw measurement data. Figure 6.3 shows a trace segment before (top) and after pre-processing (bottom). The pre-processed trace shows high peaks at locations of cache misses, indicating a '1' in the RSA exponent.

To automatically extract a partial key from a pre-processed trace, we first run a peak detection algorithm. We delete duplicate peaks, e.g., peaks where the corresponding RSA multiplications would overlap in time. We also delete peaks that are below a certain adaptive threshold, as they do not correspond to actual multiplications. Using an adaptive threshold is necessary since neither the CPU frequency nor our timing source (the counting thread) is perfectly stable. The varying peak height is shown in the right third of Figure 6.3. The adaptive threshold is the median

over the 10 previously detected peaks. If a peak drops below 90 % of this threshold, it is discarded. The remaining peaks correspond to the '1's in the RSA exponent and are highlighted in Figure 6.3. '0's can only be observed indirectly in our trace as square operations do not trigger cache activity on the monitored sets. '0's appear as time gaps in the sequence of '1' peaks, thus revealing all partial key bits. Note that since '0's correspond to just one multiplication, they are roughly twice as fast as '1's.

When a correct peak is falsely discarded, the corresponding '1' is interpreted as two '0's. Likewise, if noise is falsely interpreted as a '1', this cancels out two '0's. If either the attacker or the victim is not scheduled, we have a gap in the collected trace. However, if both the attacker and the victim are descheduled, this gap does not show up prominently in the trace since the counting thread is also suspended by the interrupt. This is an advantage of a counting thread over the use of the native timestamp counter.

In the final key recovery, we merge multiple partial keys to obtain the full key. We quantify partial key errors using the edit distance. The edit distance between a partial key and the correct key gives the number of bit insertions, deletions and flips necessary to transform the partial key into the correct key.

The full key is recovered bitwise, starting from the most-significant bit. The correct key bit is the result of the majority vote over the corresponding bit in all partial keys. To correct the current bit of a wrong partial key, we compute the edit distance to all partial keys that won the majority vote. To reduce the performance overhead, we do not calculate the edit distance over the whole partial keys but only over a lookahead window of a few bits. The output of the edit distance algorithm is a list of actions necessary to transform one key into the other. We apply these actions via majority vote until the key bit of the wrong partial key matches the recovered key bit again.

6 Evaluation

In this section, we evaluate the presented methods by building a malware enclave attacking a co-located enclave that acts as the victim. As discussed in Section 3.2, we use *mbedTLS*, in version 2.3.0.

For the evaluation, we attack a 4096-bit RSA key. The runtime of the multiplication function increases exponentially with the size of the key. Hence, larger keys improve the measurement resolution of the attacker. In

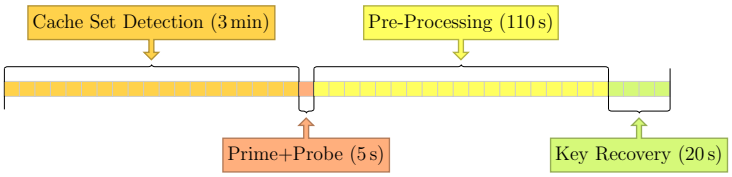


Figure 6.4: A high-level overview of the average times for each step of the attack.

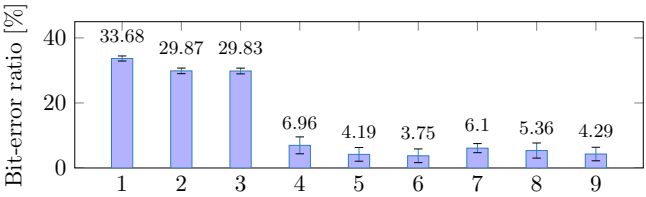


Figure 6.5: The 9 cache sets that are used by a 4096-bit key and their error ratio when recovering the key from a single trace.

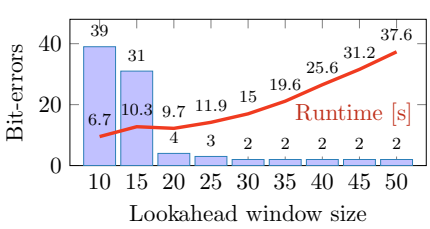
terms of cache side-channel attacks, large RSA keys do not provide higher security but degrade side-channel resistance [41, 48, 51].

6.1 Native Environment

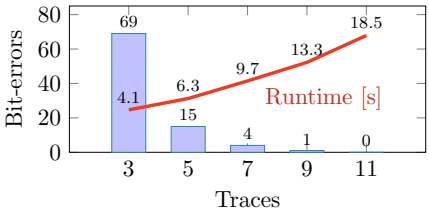
We use a Lenovo ThinkPad T460s with an Intel Core i5-6200U (2 cores, 12 cache ways) running Ubuntu 16.10 and the Intel SGX driver. Both the attacker enclave and the victim enclave are running on the same machine. We trigger the signature process using the public API of the victim.

Figure 6.4 gives an overview of how long the individual steps of an average attack take. The runtime of automatic cache set detection varies depending on which cache sets are used by the victim. The attacked buffer spans 9 cache sets, out of which 6 show a low bit-error ratio, as shown in Figure 6.5. For the attack we select one of the 6 sets, as the other 3 suffer from too much noise. The noise is mainly due to the buffer not being aligned to the cache set. Furthermore, as already known from previous attacks, the hardware prefetcher can induce a significant amount of noise [18, 51].

Detecting one vulnerable cache set within all 2048 cache sets requires about 340 trials on average. With a monitoring time of 0.21 s per cache set, we require a maximum of 72 s to eventually capture a trace from a



(a) Increasing the lookahead reduces bit errors and increases runtime.



(b) Increasing the number of traces reduces bit errors and increases runtime.

Figure 6.6: Relation between number of traces, lookahead window size, number of bit errors, and runtime.

vulnerable cache set. Thus, based on our experiments, we estimate that cache set detection—if successful—always takes less than 3 min.

One trace spans 220.47 million CPU cycles on average. Typically, '0' and '1' bits are uniformly distributed in the key. The estimated number of multiplications is therefore half the bit size of the key. Thus, the average multiplication takes 107 662 cycles. As the Prime+Probe measurement takes on average 734 cycles, we do not have to slow down the victim additionally.

When looking at a single trace, we can already recover about 96 % of the RSA private key, as shown in Figure 6.5. For a full key recovery we combine multiple traces using our key recovery algorithm, as explained in Section 5. We first determine a reasonable lookahead window size. Figure 6.6a shows the performance of our key recovery algorithm for varying lookahead window sizes on 7 traces. For lookahead windows smaller than 20, bit errors are pretty high. In that case, the lookahead window is too small to account for all insertion and deletion errors, causing relative shifts between the partial keys. The key recovery algorithm is unable to align partial keys correctly and incurs many wrong "correction" steps, increasing the overall runtime as compared to a window size of 20. While a lookahead window size of 20 already shows a good performance, a window size of 30 or more does not significantly reduce the bit errors. Therefore, we fixed the lookahead window size to 20.

To remove the remaining bit errors and get full key recovery, we have to combine more traces. Figure 6.6b shows how the number of traces affects the key recovery performance. We can recover the full RSA private key without any bit errors by combining only 11 traces within just 18.5 s.

Table 6.2: Our results show that cache attacks can be mounted successfully in the shown scenarios.

Attack on / Attack from	Benign Userspace	Benign Kernel	Benign SGX Enclave
Malicious Userspace	✓ [33, 39]	✓ [22]	✓ **new**
Malicious Kernel	—	—	✓ **new** [8, 17, 37]
Malicious SGX Enclave	✓ **new**	✓ **new**	✓ **new**

This results in a total runtime of less than 130 s for the offline key recovery process.

Generalization

Based on our experiments we deduced that attacks are also possible in a weaker scenario, where only the attacker is inside the enclave. On most computers, applications handling cryptographic keys are not protected by SGX enclaves. From the attacker's perspective, attacking such an unprotected application does not differ from attacking an enclave. We only rely on the last-level cache, which is shared among all applications, whether they run inside an enclave or not. We empirically verified that such attacks on the outside world are possible and were again able to recover RSA private keys.

Table 6.2 summarizes our results. In contrast to concurrent work on cache attacks on SGX [8, 17, 37], our attack is the only one that can be mounted from unprivileged user space, and cannot be detected as it runs within an enclave.

6.2 Virtualized Environment

We now show that the attack also works in a virtualized environment. As described in Section 2.1, no hypervisor with SGX support was available at the time of our experiments. Instead of full virtualization using a virtual machine, we used lightweight Docker containers, as used by large cloud providers, e.g., Amazon [13] or Microsoft Azure [36]. To enable SGX within a container, the host operating system has to provide SGX support. The SGX driver is then simply shared among all containers. Figure 6.7 shows our setup where the SGX enclaves communicate directly with the SGX driver of the host operating system. Applications running inside the container do not experience any difference to running on a native system.

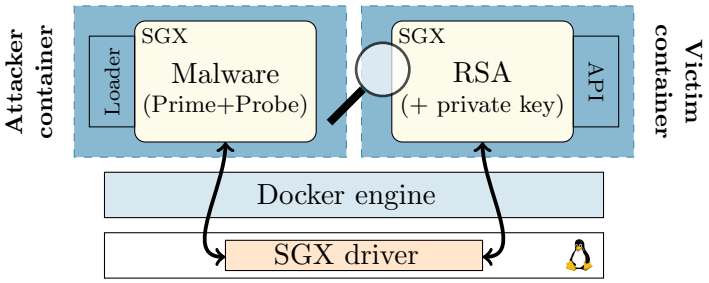


Figure 6.7: Running the SGX enclaves inside Docker containers to provide further isolation. The host provides both containers access to the same SGX driver.

Considering the performance within Docker, only I/O operations and network access have a measurable overhead [14]. Operations that only depend on memory and CPU do not see any performance penalty, as these operations are not virtualized. Thus, caches are also not affected by the container.

We were successfully able to attack a victim from within a Docker container without any changes in the malware. We can even perform a cross-container attack, *i.e.*, both the malware and the victim are running inside different containers, without any changes. As expected, we require the same number of traces for a full key recovery. Hence, containers do not provide additional protection against our malware at all.

7 Countermeasures

Most existing countermeasures cannot be applied to a scenario where a malicious enclave performs a cache attack and no assumptions about the operating system are made. In this section, we discuss 3 categories of countermeasures, based on where they ought to be implemented.

7.1 Source Level

A generic side-channel protection for cryptographic operations (e.g., RSA) is exponent blinding [30]. It will prevent the proposed attack, but other parts of the signature process might still be vulnerable to an attack [45]. More generally bit slicing can be applied to a wider range of algorithms to protect against timing side channels [5, 47]

7.2 Operating System Level

Implementing countermeasures against malicious enclave attacks on the operating system level requires trusting the operating system. This would weaken the trust model of SGX enclaves significantly, but in some threat models this can be a viable solution. However, we want to discuss the different possibilities, in order to provide valuable information for the design process of future enclave systems.

Detecting Malware

One of the core ideas of SGX is to remove the cloud provider from the root of trust. If the enclave is encrypted and only decrypted after successful remote attestation, the cloud provider has no way to access the secret code inside the enclave. Also, heuristic methods, such as behavior-based detection, are not applicable, as the malicious enclave does not rely on malicious API calls or user interaction which could be monitored. However, eliminating this core feature of SGX could mitigate malicious enclaves in practice, as the enclave binary or source code could be read by the cloud provider and scanned for malicious activities.

Herath and Fogh [21] proposed to use hardware performance counters to detect cache attacks. Subsequently, several other approaches instrumenting performance counters to detect cache attacks have been proposed [9, 19, 40]. However, according to Intel, SGX enclave activity is not visible in the thread-specific performance counters [26]. We verified that even performance counters for last-level cache accesses are disabled for enclaves. The performance counter values are three orders of magnitude below the values as compared to native code. There are no cache hits and misses visible to the operating system or any application (including the host application). This makes it impossible for current anti-virus software and other detection mechanisms to detect malware inside the enclave.

Enclave Coloring

We propose enclave coloring as an effective countermeasure against cross-enclave attacks. Enclave coloring is a software approach to partition the cache into multiple smaller domains. Each domain spans over multiple cache sets, and no cache set is included in more than one domain. An enclave gets one or more cache domains assigned exclusively. The assignment of domains is either done by the hardware or by the operating system. Trusting the operating system contradicts one of the core ideas of

SGX [10]. However, if the operating system is trusted, this is an effective countermeasure against cross-enclave cache attacks.

If implemented in software, the operating system can split the last-level cache through memory allocation. The cache set index is determined by physical address bits below bit 12 (the page offset) and bits > 12 which are not visible to the enclave application and can thus be controlled by the operating system. We call these upper bits a color. Whenever an enclave requests pages from the operating system (we consider the SGX driver as part of the operating system), it will only get pages with a color that is not present in any other enclave. This coloring ensures that two enclaves cannot have data in the same cache set, and therefore a Prime+Probe attack is not possible across enclaves. However, attacks on the operating system or other processes on the same host would still be possible.

To prevent attacks on the operating system or other processes, it would be necessary to partition the rest of the memory as well, *i.e.*, system-wide cache coloring [43]. Godfrey et al. [16] evaluated a coloring method for hypervisors by assigning every virtual machine a partition of the cache. They concluded that this method is only feasible for a small number of partitions. As the number of simultaneous enclaves is relatively limited by the available amount of SGX memory, enclave coloring can be applied to prevent cross-enclave attacks. Protecting enclaves from malicious applications or preventing malware inside enclaves is however not feasible using this method.

Heap Randomization

Our attack relies on the fact, that the used buffers for the multiplication are always at the same memory location. This is the case, as the used memory allocator (`dlmalloc`) has a deterministic best-fit strategy for moderate buffer sizes as used in RSA. Freeing a buffer and allocating it again will result in the same memory location for the re-allocated buffer.

We suggest randomizing the heap allocations for security relevant data such as the used buffers. A randomization of the addresses and thus cache sets bears two advantages. First, automatic cache set detection is not possible anymore, as the identified set will change for every run of the algorithm. Second, if more than one trace is required to reconstruct the key, heap randomization increases the number of required traces by multiple orders of magnitude, as the probability to measure the correct cache set by chance decreases.

Although not obvious at first glance, this method requires a certain amount of trust in the operating system. A malicious operating system could assign only pages mapping to certain cache sets to the enclave, similar to enclave coloring. Thus, the randomization is limited to only a subset of cache sets, increasing the probability for an attacker to measure the correct cache set.

Intel CAT

Recently, Intel introduced an instruction set extension called CAT (cache allocation technology) [24]. With Intel CAT it is possible to restrict CPU cores to one of the slices of the last-level cache and even to pin cache lines. Liu et al. [32] proposed a system that uses CAT to protect general purpose software and cryptographic algorithms. Their approach can be directly applied to protect against a malicious enclave. However, this approach does not allow to protect enclaves from an outside attacker.

7.3 Hardware Level

Combining Intel CAT with SGX

Instead of using Intel CAT on the operating system level it could also be used to protect enclaves on the hardware level. By changing the `eenter` instruction in a way that it implicitly activates CAT for this core, any cache sharing between SGX enclaves and the outside as well as co-located enclaves could be eliminated. Thus, SGX enclaves would be protected from outside attackers. Furthermore, it would protect co-located enclaves as well as the operating system and user programs against malicious enclaves.

Secure RAM

To fully mitigate cache- or DRAM-based side-channel attacks memory must not be shared among processes. We propose an additional fast, non-cachable secure memory element that resides inside the CPU.

The SGX driver can then provide an API to acquire the element for temporarily storing sensitive data. A cryptographic library could use this memory to execute code which depends on secret keys such as the square-and-multiply algorithm. Providing such a secure memory element per CPU core would even allow parallel execution of multiple enclaves.

Data from this element is only accessible by one program, thus cache attacks and DRAM-based attacks are not possible anymore. Moreover,

if this secure memory is inside the CPU, it is infeasible for an attacker to mount physical attacks. It is unclear whether the Intel eDRAM implementation can already be instrumented as a secure memory to protect applications against cache attacks.

8 Conclusion

Intel claimed that SGX features impair side-channel attacks and recommends using SGX enclaves to protect cryptographic computations. Intel also claimed that enclaves cannot perform harmful operations.

In this paper, we demonstrated the first malware running in real SGX hardware enclaves. We demonstrated cross-enclave private key theft in an automated semi-synchronous end-to-end attack, despite all restrictions of SGX, e.g., no timers, no large pages, no physical addresses, and no shared memory. We developed a timing measurement technique with the highest resolution currently known for Intel CPUs, perfectly tailored to the hardware. We combined DRAM and cache side channels, to build a novel approach that recovers physical address bits without assumptions on the page size. We attack the RSA implementation of *mbedTLS*, which uses constant-time multiplication primitives. We extract 96 % of a 4096-bit RSA key from a single Prime+Probe trace and achieve full key recovery from only 11 traces.

Besides not fully preventing malicious enclaves, SGX provides protection features to conceal attack code. Even the most advanced detection mechanisms using performance counters cannot detect our malware. This unavoidably provides attackers with the ability to hide attacks as it eliminates the only known technique to detect cache side-channel attacks. We discussed multiple design issues in SGX and proposed countermeasures for future SGX versions.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

[1] Ittai Anati, Frank McKeen, Shay Gueron, Haitao Huang, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos V. Rozas, and Hisham Shafi. *Intel Software Guard Extensions (Intel SGX)*. Tutorial Slides. Tutorial Slides presented at ICSA 2015. 2015.

[2] ARMmbed. *Reduce mbed TLS memory and storage footprint*. `https://tls.mbed.org/kb/how-to/reduce-mbedtls-memory-and-storage-footprint`. Retrieved on October 24, 2016. 2016.

[3] Cyril Arnaud and Pierre-Alain Fouque. "Timing attack against protected RSA-CRT implementation used in PolarSSL." In: *CT-RSA 2013*. 2013.

[4] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.

[5] Eli Biham. "A fast new DES implementation in software." In: *International Workshop on Fast Software Encryption*. 1997, pp. 260–272.

[6] Johannes Blömer and Alexander May. "New partial key exposure attacks on RSA." In: *Crypto'03*. 2003.

[7] Dan Boneh, Glenn Durfee, and Yair Frankel. "An attack on RSA given a small fraction of the private key bits." In: *International Conference on the Theory and Application of Cryptology and Information Security*. 1998.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: (2017). URL: `http://arxiv.org/abs/1702.07521`.

[9] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. *Real time detection of cache-based side-channel attacks using Hardware Performance Counters*. Cryptology ePrint Archive, Report 2015/1034. 2015.

[10] Victor Costan and Srinivas Devadas. *Intel SGX explained*. Tech. rep. Cryptology ePrint Archive, Report 2016/086, 2016.

[11] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[12] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the feasibility of online malware detection with performance counters." In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 559–570.

[13] Docker. *Amazon Web Services - Docker*. `https://docs.docker.com/machine/drivers/aws/`. 2016.

[14] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. "An updated performance comparison of virtual machines and linux containers." In: *2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*. 2015.

[15] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. *A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware*. Tech. rep. Cryptology ePrint Archive, Report 2016/613, 2016., 2016.

[16] Michael Misiu Godfrey and Mohammad Zulkernine. "Preventing Cache-Based Side-Channel Attacks in a Cloud Environment." In: *IEEE Transactions on Cloud Computing* (Oct. 2014).

[17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX." In: *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. 2017.

[18] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA'16*. 2016.

[20] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA'16*. 2016.

[21] Nishad Herath and Anders Fogh. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security." In: *Black Hat USA*. 2015.

[22] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *S&P'13*. 2013.

[23] Intel. "Intel® 64 and IA-32 Architectures Optimization Reference Manual." In: (2014).

[24] Intel. "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide." In: 253665 (2014).

[25] Intel Corporation. *Hardening Password Managers with Intel Software Guard Extensions: White Paper*. 2016.

[26] Intel Corporation. *Intel SGX: Debug, Production, Pre-release what's the difference?* `https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference`. Retrieved on October 24, 2016. 2016.

[27] Intel Corporation. *Intel Software Guard Extensions (Intel SGX)*. `https://software.intel.com/en-us/sgx`. Retrieved on November 7, 2016. 2016.

[28] Intel Corporation. *Intel(R) Software Guard Extensions for Linux* OS*. `https://github.com/01org/linux-sgx-driver`. Retrieved on November 11, 2016. 2016.

[29] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES." In: *RAID'14*. 2014.

[30] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems." In: *Crypto'96*. 1996.

[31] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. 2016.

[32] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. "Catalyst: Defeating last-level cache side channel attacks in cloud computing." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 2016.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *S&P'15*. 2015.

[34] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS'17*. 2017.

[35] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters." In: *RAID'15*. 2015.

[36] Microsoft. *Create a Docker environment in Azure using the Docker VM extension*. `https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-dockerextension/`. Oct. 2016.

[37] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies The Power of Cache Attacks." In: *arXiv preprint arXiv:1703.06986* (2017).

[38] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS'15*. 2015.

[39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA 2006*. 2006.

[40] Matthias Payer. "HexPADS: a platform to detect "stealth" attacks." In: *ESSoS'16*. 2016.

[41] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[42] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium*. 2016.

[43] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. "Resource Management for Isolation Enhanced Cloud Services." In: *Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW'09)*. 2009, pp. 77–84.

[44] Joanna Rutkowska. *Thoughts on Intel's upcoming Software Guard Extensions (Part 2)*. `http://theinvisiblethings.blogspot.co.at/2013/09/thoughts-on-intels-upcoming-software.html`. Retrieved on October 20, 2016. 2013.

[45] Werner Schindler. "Exclusive exponent blinding may not suffice to prevent timing attacks on RSA." In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 2015.

[46] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. "VC3: trustworthy data analytics in the cloud using SGX." In: 2015.

[47] M Sudhakar, Ramachandruni Venkata Kamala, and MB Srinivas. "A bit-sliced, scalable and unified Montgomery multiplier architecture for RSA and ECC." In: *2007 IFIP International Conference on Very Large Scale Integration*. 2007, pp. 252–257.

[48] Colin D Walter. "Longer keys may facilitate side channel attacks." In: *International Workshop on Selected Areas in Cryptography*. 2003.

[49] John C Wray. "An analysis of covert timing channels." In: *Journal of Computer Security* (1992).

[50] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *S&P'15*. May 2015.

[51] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

7

KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks

Publication Data

Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In: *NDSS*. 2018

Contributions

Main author.

KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks

Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard

Graz University of Technology, Austria

Abstract

Besides cryptographic secrets, software-based side-channel attacks also leak sensitive user input. The most accurate attacks exploit cache timings or interrupt information to monitor keystroke timings and subsequently infer typed words and sentences. These attacks have also been demonstrated in JavaScript embedded in websites by a remote attacker. We extend the state-of-the-art with a new interrupt-based attack and the first Prime+Probe attack on kernel interrupt handlers. Previously proposed countermeasures fail to prevent software-based keystroke timing attacks as they do not protect keystroke processing through the entire software stack.

We close this gap with *KeyDrown*, a new defense mechanism against software-based keystroke timing attacks. *KeyDrown* injects a large number of fake keystrokes in the kernel, making the keystroke interrupt density uniform over time, *i.e.*, independent of the real keystrokes. All keystrokes, including fake keystrokes, are carefully propagated through the shared library to make them indistinguishable by exploiting the specific properties of software-based side channels. We show that attackers cannot distinguish fake keystrokes from real keystrokes anymore and we evaluate *KeyDrown* on a commodity notebook as well as on Android smartphones. We show that *KeyDrown* eliminates any advantage an attacker can gain from using software-based side-channel attacks.

1 Introduction

Modern computer systems leak sensitive user information through side channels. Among software-based side channels, information can leak, for

The original publication is available at `http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_04B-1_Schwarz_paper.pdf`.

example, from the system or microarchitectural components such as the CPU cache [11] or the DRAM [43]. Historically, side-channel attacks have exploited these information leaks to infer cryptographic secrets [30, 40, 58], whereas more recent attacks even target keystroke timings and sensitive user input directly [14, 39, 43].

In general, keystroke attacks aim to monitor when a keyboard input occurs, which either allows inferring user input directly or launching follow-up attacks [50, 60]. In particular, mobile devices may expose this information through sensor data, but practical mitigations [48] have already been proposed. Furthermore, restrictions (on the procfs) have already been implemented in Android O [13, 24] and are likely to be upstreamed to the main Linux kernel. Consequently, attackers are left with side channels to obtain keystroke timings. Especially microarchitectural attacks allow monitoring memory accesses with a granularity of single cache lines, and thus also allow recovering keystroke timings with a high accuracy.

Keystroke timing attacks are hard to mitigate, compared to side-channel attacks on cryptographic implementations. Indeed, attacks on cryptographic implementations can be mitigated with changes in the algorithms, such as making execution paths independent of secret data. On the contrary, user input travels a long way, from the hardware interrupt through the operating system and shared libraries up to the user space application. In order to detect a keystroke, an attacker just needs to probe a single spot in the keystroke path for activity.

In the general case, keystrokes are non-repeatable low-frequency events, *i.e.*, if the attacker misses a keystroke, there is no way to repeat the measurement. However, an attacker that explicitly targets a password field can record more timing traces when the user enters the password again. While these traces have variations in timing, due to the variance of the typing behavior, it allows an attacker to combine multiple traces and to perform a more sophisticated attack. This makes attacks on password fields even harder to mitigate.

State-of-the art defense mechanisms [13, 24, 48] only restrict access to the system interfaces providing interrupt statistics [9, 60], and do not address all the layers involved in keystroke processing. Therefore, these defenses do not prevent all software-based keystroke timing attacks. We first demonstrate two novel side-channel attacks to infer keystroke timings, that work on systems where previous keystroke timing attacks are mitigated [13, 24]. The first attack uses the `rdtsc` instruction to determine the execution time of an interrupt service routine (ISR), which

is then used to determine whether or not the interrupt was caused by the keyboard. The second attack uses Multi-Prime+Probe on the kernel to determine when a keystroke is being processed in the kernel.

Based on these investigations and state-of-the-art attacks, we identify three essential requirements for successful elimination of keystroke timing attacks on the entire software stack. In the presence of the countermeasure:

1. Any classifier based on a single-trace side-channel attack may not provide any advantage over a random classifier.
2. The number of side-channel traces a classifier requires to detect all keystrokes correctly must be impractically high.
3. The implementation of the countermeasure may not leak information about its activity or computations.

Based on the identified requirements, we present *KeyDrown*, a new defense mechanism against keystroke timing attacks exploiting software-based side channels. *KeyDrown* covers the entire software stack, from the interrupt source to the user space buffer storing the keystroke, both on x86 systems and on ARM devices. We cover both the general case where an attacker can only obtain a single trace, and the case of password input where an attacker can obtain multiple traces. *KeyDrown* works in three layers:

1. To mitigate interrupt-based attacks, *KeyDrown* injects a large number of fake keyboard interrupts, making the keystroke interrupt density uniform over time, *i.e.*, independent of the real keystrokes. Prime+Probe attacks on the kernel module are mitigated by unifying the control flow and data accesses of real and fake keystrokes such that there is no difference visible in the cache or in the execution time.
2. To mitigate Flush+Reload and Prime+Probe attacks on shared libraries, *KeyDrown* runs through the same code path in the shared library for all, fake and real, keystrokes.
3. To mitigate Prime+Probe attacks on password entry fields, *KeyDrown* updates the widget buffer for every fake and real keystroke.

We evaluate *KeyDrown* on several state-of-the-art attacks as well as our two novel attacks. In all cases, *KeyDrown* eliminates any advantage an attacker can gain from the side channels, *i.e.*, the attacker cannot deduce sensitive information from the side channel.

We provide a proof-of-concept implementation, which can be installed as a Debian package compatible with the latest long-term support release of Ubuntu (16.04). It runs on commodity operating systems with unmodified applications and unmodified compilers. *KeyDrown* is started automatically

and is entirely transparent to the user, *i.e.*, requires no user interaction. Although our countermeasure inherently executes more code than an unprotected system, it has no noticeable effect on keystroke latency. Finally, we also define what *KeyDrown* cannot protect against, such as word completion lookups or immediate forwarding of single keystrokes over the network.

Contributions. The contributions of this work are:

1. We present two novel attacks to recover keystroke timings, that work in environments where previous attacks fail [13, 24].
2. We identify three essential requirements for an effective countermeasure against keystroke attacks.
3. We propose *KeyDrown*, a multi-layered solution to mitigate keystroke timing attacks.[1]
4. We evaluate *KeyDrown* and show that it eliminates all known attacks.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background information. In Section 3, we introduce our novel attacks and define requirements a defense mechanism has to provide to successfully mitigate attacks. In Section 4, we describe the three layers of *KeyDrown*. In Section 5, we demonstrate that *KeyDrown* successfully mitigates keystroke timing attacks. In Section 6, we discuss limitations and future work. We conclude in Section 7.

2 Background

In this section, we provide background information on interrupt handling as well as on software-based side channels that leak keystroke timing information.

2.1 Linux Interrupt Handling

Interrupt handling is one of the low-level tasks of an operating system and thus highly architecture and machine dependent. This section covers the general design of how interrupts and their handling within the Linux kernel work on both x86 PCs and ARMv7 smartphones.

[1]The code and a demo video are available in a GitHub repository: `https://github.com/IAIK/keydrown`.

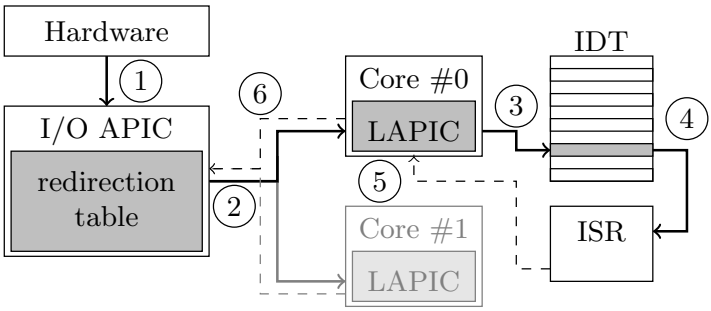


Figure 7.1: Linux interrupt handling on x86.

Interrupts on x86 and x86_64

Figure 7.1 shows a high-level overview of interrupt handling on a dual-core x86 CPU. Interrupts are handled by the Advanced Programmable Interrupt Controller (APIC) [21]. The APIC receives interrupts from different sources: locally and externally connected I/O devices, inter-processor interrupts, APIC internal interrupts, performance monitoring interrupts, and thermal sensor interrupts. On multi-core systems, every CPU core has a local APIC (LAPIC) to handle interrupts. All LAPICs are connected to one or more I/O APICs which handle the actual hardware interrupts. The I/O APICs are part of the chipset and provide multi-core interrupt management by distributing the interrupts to the LAPICs as described in the ACPI system description tables [36].

Interrupt-generating hardware, such as the keyboard, is connected to an I/O APIC pin (①). The I/O APIC uses a redirection table to redirect hardware interrupts and the raised interrupt vector to the destination LAPIC (②) [20]. In the case of multiple configured LAPICs for one interrupt, the I/O APIC chooses a CPU based on task priorities in a round-robin fashion [5].

The LAPIC receiving the interrupt vector fetches the corresponding entry from the Interrupt Descriptor Table (IDT) (③) which is set up by the operating system. The IDT contains an offset to the Interrupt Service Routine (ISR) for every interrupt vector. The CPU saves the current CPU flags and jumps to the interrupt service routine (④) which then handles the interrupt.

After processing, the interrupt service routine acknowledges the interrupt by sending an end-of-interrupt (EOI) to the LAPIC (⑤). It then returns using the `iret` instruction to restore the CPU flags and to enable interrupts again. The LAPIC forwards the EOI to the I/O APIC (⑥)

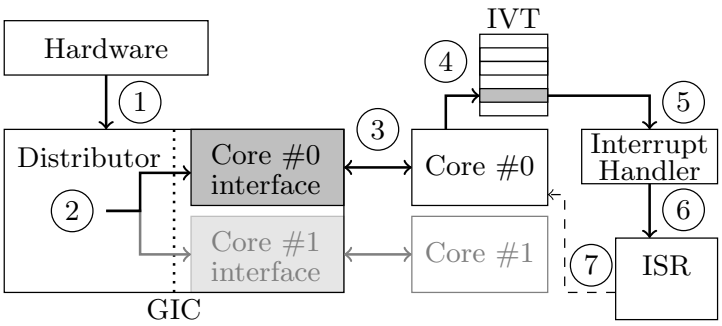


Figure 7.2: Linux interrupt handling on ARM.

which then resets the interrupt line to enable the corresponding interrupt again.

Interrupts on ARM

Figure 7.2 shows a high-level overview of interrupt handling on a dual-core ARMv7 CPU. On ARM, interrupts are handled by the General Interrupt Controller (GIC). The GIC is divided into two parts, the distributor, and a CPU interface for every CPU core [2]. Every interrupt-generating device is connected to the distributor of the GIC (①). The distributor (②) schedules between CPU interfaces according to the interrupt's affinity mask.

When a CPU interface receives an interrupt, it signals it to the corresponding CPU core (③). The core reads the interrupt number from the interrupt acknowledge register to acknowledge it. If the interrupt was sent to multiple CPU interfaces, all other CPU cores receive a spurious interrupt, as there is no more pending interrupt.

When receiving an interrupt, the CPU finishes executing the current instruction, switches to IRQ mode, and jumps to the IRQ entry of the Interrupt Vector Table (IVT) (④). The IVT contains exactly one instruction to jump to a handler function (⑤). In this handler function, the OS branches to the Interrupt Service Routine (ISR) corresponding to the interrupt number (⑥).

When the CPU is done servicing the interrupt, it writes the interrupt number to the End Of Interrupt register (⑦) to signal that it is ready to receive this interrupt again [1].

2.2 Microarchitectural Attacks

CPU caches are a small and fast type of memory, buffering frequently used data to speed-up subsequent accesses. There are typically three levels of caches in modern x86 CPUs, and two levels in modern ARM CPUs. The last-level cache is typically shared across cores of the same CPU, which makes it a target for cross-core side-channel attacks. On Intel x86 CPUs, the last-level cache is divided into one slice per core. The smallest unit managed by a cache is a cache line (typically 64 B). Modern caches are set-associative, *i.e.*, multiple cache lines are considered a set of equivalent storage locations. A memory location maps to a cache set and slice based on the physical address [19, 33, 59].

Flush+Reload. Flush+Reload [17, 58] is a technique that allows an attacker to monitor a victim's cache accesses at a granularity of a single cache line. The attacker flushes a cache line, lets the victim perform an operation, and then reloads and times the access to the cache line. A low timing indicates that the victim accessed the cache line. While very accurate, it can only be performed on shared memory, *i.e.*, shared libraries or binary code. Flush+Reload can neither be performed on dynamic buffers in a user program nor on code or data in the kernel. Gruss et al. [14] presented cache template attacks as a technique based on Flush+Reload to automatically find and exploit cache-based leakage in programs.

Prime+Probe. Prime+Probe [30, 40, 41] is a technique that allows an attacker to monitor a victim's cache accesses at a granularity of a cache set. The attacker primes a cache set, *i.e.*, fills the cache set with its own cache lines. It then lets the victim perform an operation. Finally, it probes its own cache lines *i.e.*, measures the access time to them. This technique does not require any shared memory between the attacker and the victim, but it is difficult due to the mapping between physical addresses and cache sets and slices. As Prime+Probe only relies on measuring the latency of memory accesses, it can be performed on any part of the software stack. It is possible to perform Prime+Probe on dynamically generated data [28] as well as kernel memory [40]. Preventing Prime+Probe attacks is difficult due to the huge attack surface and the fact that Prime+Probe uses only innocuous operations such as memory accesses on legitimately allocated memory, as well as timing measurements.

DRAMA. Besides the cache, the DRAM design also introduces side channels [43], *i.e.*, timing differences caused by the DRAM row buffer. A DRAM bank contains a row buffer caching an entire DRAM row (8 KB). Requests to the currently active row are served from this buffer, resulting in a fast access, whereas other requests are significantly slower. DRAM

side-channel attacks do not require shared memory and work across CPUs of the same machine sharing a DRAM module.

2.3 Keystroke Timing Attacks

Keystrokes from Keystroke Timing. Keystroke timing attacks attempt to recover what was typed by the user by analyzing keystroke timing measurements. These timings show characteristic patterns of the user, which depend on several factors such as keystroke sequences on the level of single letters, bigrams, syllables or words as well as keyboard layout and typing experience [44]. Existing attacks train probabilistic classifiers like hidden Markov models or neural networks to infer known words or to reduce the password-guessing complexity [49, 50, 60].

Most keystroke timing attacks exploit the inter-keystroke timing, *i.e.*, the timing difference between two keystrokes, but according to Idrus et al. [18] combinations of key press and key release events could also be exploited. Pinet et al. [44] report inter-keystroke interval values between 160 ms and 200 ms for skilled typists. Lee et al. [27] define the values depending on whether a text sequence was trained or entered for the first time, resulting in inter-keystroke intervals between 125 ms and 215 ms with a variance between 43 ms and 106 ms, again for trained and untrained text sequences.

Keystroke Timing from Software. A direct software side channel for keystroke timings is provided through OS interfaces, such as instruction pointer and stack pointer information leaked through `/proc/stat`, and interrupt statistics leaked through `/proc/interrupts` [60]. As the instruction pointer and stack pointer information became too unpredictable, Jana and Shmatikov [22] showed that CPU usage yields much more reliable information on keystroke timings. Diao et al. [9] demonstrated high-precision keystroke timing attacks based on `/proc/interrupts`. However, these attacks are not possible anymore in Android O [13, 24], as access to these resources has been restricted.

Vila et al. [53] recovered keystroke timings from timing differences caused by the event queue in the Chrome browser. Based on the native attack we present in Section 3.2, Lipp et al. [29] implemented the same attack in JavaScript. They recovered keystroke timings and identified user-typed URLs. They also showed that users can be distinguished based on this attack.

Gruss et al. [14] demonstrated that Flush+Reload allows distinguishing specific keys or groups of keys based on key-dependent data accesses in shared libraries. Ristenpart et al. [46] demonstrated a keystroke timing

attack using Prime+Probe with a false-negative rate of 5 % while measuring 0.3 false positive keystrokes per second. Pessl et al. [43] showed that it is possible to use DRAM attacks to monitor keystrokes, e.g., in the address bar of Firefox. However, this attack only works if the target application performs a massive amount of memory accesses to thrash the cache reliably on its own.

3 Keystroke Timing Attacks & Defenses

Due to the amount of code executed for every keystroke, there are many different side channels for keystroke timings. In this section, we introduce our two novel attacks and compare them to state-of-the-art keystroke timing attack vectors, in order to understand the requirements for effective countermeasures. Finally, we derive three requirements for countermeasures to be effective against keystroke timing attacks.

The requirements are defined based on precision and recall of side-channel attacks. The *precision* is the fraction of true positive detected keystrokes in all detected keystrokes. If the precision is low, the side channel yields too many false positives to derive the correct keystroke timings. The *recall* is the fraction of true positive detected keystrokes in all real keystrokes. If the recall is low, *i.e.*, the side channel misses too many true positives, inter-keystroke timings are corrupted too. A standard measure of accuracy is the *F-score*, *i.e.*, the geometric mean of precision and recall. An F-score of 1 describes a perfect side channel. An F-score of 0 describes that a side channel provides no information at all.

Note that there is only a limited number of keystroke time frames that can be reliably distinguished by an attacker, due to the typing speed and the variance of inter-keystroke timing (cf. Section 2.3). A keystroke timing attack providing nanosecond-accurate timestamps is actually only providing the binary information in which time frames a keystroke occurred. Hence, we can compare side-channel-based classifiers to binary decision classifiers for these time frames.

An *always-zero oracle* which never detects any event has an F-score of 0. An *always-one oracle* which "detects" an event in every possible time frame, *i.e.*, a large number of false positives, no false negatives, and no true negatives, is a channel which provides zero information. Similarly, a *random-guessing oracle*, which decides for every possible time frame whether it "detects" an event based on an apriori probability, also provides zero information. For 8 keystrokes and 100 possible time frames per second, the F-score for the always-one oracle is 0.15 which is strictly better than

the F-score of the random-guessing oracle (0.14). An attacker relying on any side-channel-based classifier with a lower F-score could achieve better results by simply using an always-one oracle, *i.e.*, in such a case it would not make sense to use the side-channel-based classifier in the first place. In the remainder of the paper, we assume that an attacker wants to find the real 8 keystrokes in 100 possible time frames per second.

This attack model does not have the concept of processes or windows. Indeed, this is an accurate representation, as side-channel attacks on keystroke timings are system-wide attacks on shared code, cache sets, or other shared parts of the microarchitecture. This makes them very powerful but also provides us a means to defeat them, *i.e.*, an attacker cannot distinguish real keyboard input in one process or window from fake keyboard input in another process or window.

3.1 Keystroke Timing Attack Surface

Keystroke processing involves computations on all levels of the software stack. Hence, targeted solutions like Cloak cannot provide complete protection in this case [16]. The keyboard interrupt is handled by one of the CPU cores, which interrupts the currently executed thread. A significant amount of code is executed in the *operating system kernel* and the keyboard driver until the preprocessed keystroke event is handed over to a user space *shared library* that is part of the user interface. The shared library distributes the keystroke event to all user interface elements listening for the event. Finally, the shared library hands over the keyboard input to the active *user space process* which further processes the input, e.g., store a password character in a buffer. This abundance of code and data that is executed and accessed upon a keystroke provides a multitude of possibilities to measure keystroke timings.

3.2 New Attack Vectors

Software side channels through `procfs` interfaces can be mitigated by restricting access to them [9, 60]. However, such restrictions do not prevent keystroke timing attacks. We demonstrate two new attacks to infer keystroke timings: the first one exploits interrupt timings to detect keystrokes, and the second one relies on Prime+Probe to attack a kernel module. Table 7.1 compares the novel attacks we describe in the following with the state-of-the-art attack vectors (cf. Section 2.3) in terms of attack techniques and the exploited attack surface.

Table 7.1: State-of-the-art Software-based Keystroke Timing Attacks and their Targets.

	Kernel	Shared library	User process
Interface-based	✓ [9, 22, 60]	✘	✘
Timing-based	✓ **ours**	✘	✘
Flush+Reload	✘	✓ [14]	✘
Prime+Probe on L1	✓ [46]	✓ [46]	✓ [46]
Prime+Probe on LLC	✓ **ours**	✓ **ours**	✓ **ours**
DRAMA	✘	✘	✓ [43]

Low-Requirement Interrupt Timing Attack. We propose a new *timing-based* attack that only requires unprivileged sand-boxed code execution on the targeted platform and an accurate timing source, e.g., the `rdtsc` instruction or a counter thread. The basic idea is to monitor differences in the execution time of acquiring high-precision time stamps, e.g., the `rdtsc` instruction, as outlined in Algorithm 1. While small differences between successive time stamps allow us to infer the CPU utilization, larger differences indicate that the measurement process was interrupted. In particular, I/O events like keyboard interrupts lead to clearly visible peaks in the execution time, due to the interaction of the keyboard ISR with hardware and the subsequent processing of keystrokes. Modern operating systems have core affinities for interrupts, which generally do not change until the system is rebooted, and core affinities for threads. Hence, once a thread runs on the core for the keyboard interrupt, it will continuously be interrupted by every keyboard interrupt, making this attack surprisingly reliable. By starting multiple threads an attacker can first run on all cores and after detecting which thread receives keyboard interrupts, terminate all threads but the one that is running on the right core.

Note that this attack does not benefit at all from attacker process and victim process running on the same core. The keyboard interrupt is scheduled based on its core affinity and not based on the core affinity of any victim thread. Hence, the attack works best if the attacker has a lot of computation time on the interrupt-handling core, but not the victim core.

Figure 7.3 illustrates these observations in a timing trace recorded while the user was typing a password. The bars indicate actual keystroke

```
for i ∈ {1, ..., N} do
    tsc[i] ← rdtsc();
    if tsc[i] − tsc[i − 1] > threshold then
        events[i] ← tsc[i];
        diff[i] ← tsc[i] − tsc[i − 1];
    end
end
```

Algorithm 1: Recording interrupt timing

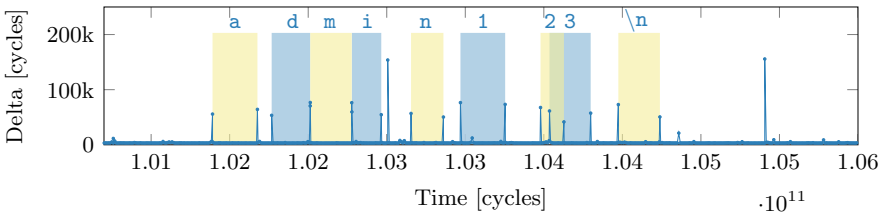


Figure 7.3: Measured delta between continuous `rdtsc` calls while entering a password. Keystroke events interrupt the attacker and thus cause higher deltas. Background color illustrates the keystroke ground truth. Periodic interrupts at 1.025 and 1.049 have a different interruption time.

events, which almost perfectly match certain measurement points. Based on this plot, we can clearly distinguish keyboard interrupts (around 60 000 cycles) from other interrupts. For example, rescheduling interrupts can be observed with a difference of about 155 000 cycles. In this attack, we achieve a precision of 0.89 and a recall of 1, resulting in an F-score of 0.94, which means a significant advantage over an always-one oracle of +537.4 %.

In our attack, we targeted the laptop keyboard of a Lenovo ThinkPad T460s (*i.e.*, a PS/2 keyboard), and touchscreens of multiple smartphones (cf. Appendix). The attack might not work on USB keyboards, as they are typically configured for polling instead of interrupts. However, the defense mechanism we present in Section 4 protects USB keyboards as well.

A preliminary version of our attack was the basis for an implementation without `rdtsc` in JavaScript embedded in websites [29]. The authors used this attack to detect the URL typed by the user with a high accuracy and even distinguish different users typing on the same machine. Showing that the attack even works in this much more constrained environment

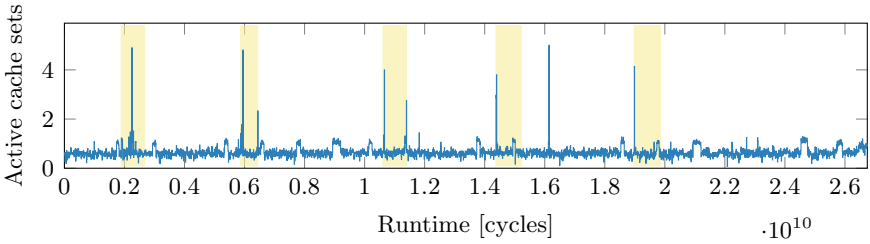


Figure 7.4: Multi-Prime+Probe attack on password input. Keystrokes cause higher activity in more cache sets. Background color illustrates the keystroke ground truth.

underlines the practicality of our attack. It is not influenced by foreground, background, or sandboxed operation.

Multi-Prime+Probe Attack on the Kernel. Our second attack relies on Prime+Probe to attack the keyboard interrupt handler within the kernel. More specifically, we target the code in the keyboard interrupt handler that is executed each time a key is pressed. Thereby, keystroke events can be inferred by observing cache activity in the cache set used by the keyboard interrupt handler.

To find the cache sets that are accessed by the keyboard interrupt handler, we first need to find the physical addresses where the code is located. We can use the TSX-based side channel by Jang et al. [23] to locate the code within the kernel. Kernel Address-Space-Layout Randomization was not enabled by default until Ubuntu 16.10. Thus, an attacker can also just use known physical addresses from an attacker-controlled system.

To reduce the influence of system noise, we developed a new form of Prime+Probe attack called Multi-Prime+Probe. Multi-Prime+Probe combines the information from multiple simultaneous Prime+Probe attacks on different addresses. Figure 7.4 shows the result of such a Multi-Prime+Probe attack on the keyboard interrupt handler. In a post-processing step, we smoothed the Multi-Prime+Probe trace with a 500 µs sliding window. The keystroke events cause higher activity in the targeted cache sets and thus produce clearly recognizable peaks for every key event. Despite doubts that such an attack can be mounted [15], our attack is the first highly accurate keystroke timing attack based on Prime+Probe on the last-level cache. More specifically, we achieve a precision of 0.71 and a recall of 0.92, resulting in an F-score of 0.81, which is significantly better than state-of-the-art Prime+Probe attacks.

3.3 Requirements for Elimination of Keystroke Timing Attacks

As demonstrated in the previous section, we are able to craft new attacks with fewer requirements than state-of-the-art attacks. Hence, countermeasures against keystroke timing attacks must be designed in a generic way, in all affected layers of the software stack, covering known and unknown attacks.

Attack Model. We assume that an attacker can run an unprivileged program on the target machine, with a recently updated system. As sensor-based attacks [6] are already addressed in [48], and Android O [13, 24] also mitigates various procfs attacks, we consider them out of scope for this paper.

The attacker is able to continuously monitor a side channel to obtain traces for all user input. We assume the (hypothetical) countermeasure against keystroke timing attacks was already installed when the attacker gained unprivileged access to the machine. Consequently, the attacker cannot obtain keystroke timing templates and thus cannot perform a template attack.

We assume that an attacker can generally obtain only a single trace for any user input sequence, but multiple traces for password input. In contrast to side-channel attacks on algorithms, which can be repeated multiple times, user input sequences are generally not (automatically) repeatable, and thus an attacker cannot obtain multiple traces. An exception are phrases that are repeatedly entered in the same way, such as login credentials and especially passwords. A countermeasure must address both cases.

To effectively eliminate keystroke timing attacks, we identify the 3 following requirements a countermeasure must fulfill.

***R1*: Minimize Side Channel Accuracy.** As user input sequences are in general not (automatically) repeatable, keystroke timing attacks require a high precision and high recall to succeed. To be effective, a countermeasure must reduce the F-score enough so that the attacker does not gain any advantage from using the side channel over an always-one oracle. More specifically, the F-score of the side-channel based classifier may not be above the F-score of the always-one oracle (0.15). Ristenpart et al. [46] reported a false-negative rate of 5 % with 0.3 false positives per second. At an average typing speed for a skilled typist of 8 keystrokes per second [44], the F-score is thus 0.96, which is an advantage over an always-one oracle of +545.3 %. Gruss et al. [14, 15] reported false-negative rates $\leq 8\,\%$ with no false positives, resulting in an F-score of > 0.96, which is an advantage

over an always-one oracle of +546.9 %. Thus, we assume a countermeasure is effective if it reduces the F-score of side channels significantly, such that using the side channel gives an advantage over an always-one oracle of ≤0.0 %.

***R2*: Reduction of Statistical Characteristics in Password Input.** In the case of a password input, we assume that an attacker can combine information from multiple traces, *i.e.*, exploit statistical characteristics. A countermeasure is effective if the attacker requires an impractical number of traces to reach the F-score of state-of-the-art attacks, *i.e.*, higher than 0.95.

Specifically, if the side-channel attack requires more traces than can be practically obtained, we consider the side-channel attack not practical. Studies [7, 8, 10, 47, 54] estimate that most users have 1–5 different passwords and enter 5 passwords per day on average. It is also estimated that 56% of users change their password at least once every 6 months. Thus, even if we assume that we attack a user with a single password that is entered 5 times per day, the expected number of measurement traces that an attacker is able to gather after 6 months is 913. Assuming that attackers might come up with new side-channel attacks, a generous security margin must be applied. We consider a countermeasure effective if it requires more than 1825 traces, *i.e.*, traces for a whole year, to reach an F-score of 0.95.

***R3*: Implementation Security.** *R1* and *R2* define how the countermeasure must be designed to be effective. However, the implementation itself can indirectly violate *R1* or *R2* by leaking side-channel information on computations of the countermeasure itself. Consequently, an attacker may be able to filter the true positive keystrokes. We thus require that the countermeasure may not have distinguishable code paths or data access patterns to guarantee that it is free from leakage.

If the implementation does not leak by itself, an attacker is only left with the low F-scores from *R1* and *R2*. If all requirements are met, classical password recovery attacks like brute force and more sophisticated attacks using Markov n-grams [32, 37], probabilistic context-free grammars (PCFG) [52, 55], or neural networks [35], are more practical than a side-channel attack in the presence of the countermeasure.

In the following section, we describe the design of a countermeasure that fulfills all three requirements.

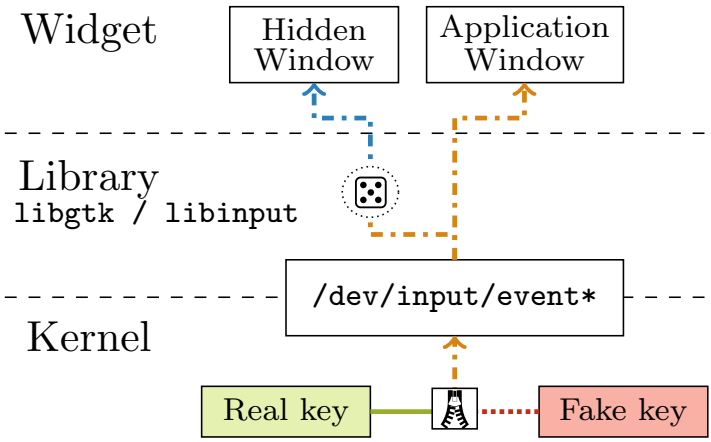


Figure 7.5: Multi-layered design of *KeyDrown*.

4 *KeyDrown* Multi-layer Design

We designed *KeyDrown* as a multi-layered countermeasure.[1] Each layer builds up on the layer beneath and adds additional protection. Figure 7.5 shows how the layers are connected to each other. The first layer implements a protection mechanism against interrupt-based attacks and timing-based attacks by artificially injecting interrupts. Any real keyboard interrupt only replaces one fake keyboard interrupt within a multitude of fake interrupts, *i.e.*, it perfectly blends in the stream of random fake keyboard interrupts. The implementation ensures that it makes the keystroke interrupt density uniform over time and thus, independent of the real interrupts. Figuratively speaking, plotting the number of keystroke interrupts over time will yield a line which has no deviations at the points in time where real keystrokes occur.

KeyDrown exploits that keystroke timing side channels do not provide the information which process or window is receiving the keystroke. These side channels are system-wide attacks on shared code, shared cache sets, or other shared parts of the microarchitecture. While this makes them very powerful (cross-core, cross-user attacks), it is also the basis for our defense mechanism. An attacker cannot distinguish real keyboard input in one process or window from fake keyboard input in another process or window. *KeyDrown* exploits this technicality and sends the fake keyboard input through the entire software stack into a special process and window.

[1]The code and a demo video are available in a GitHub repository: `https://github.com/keydrown/keydrown`.

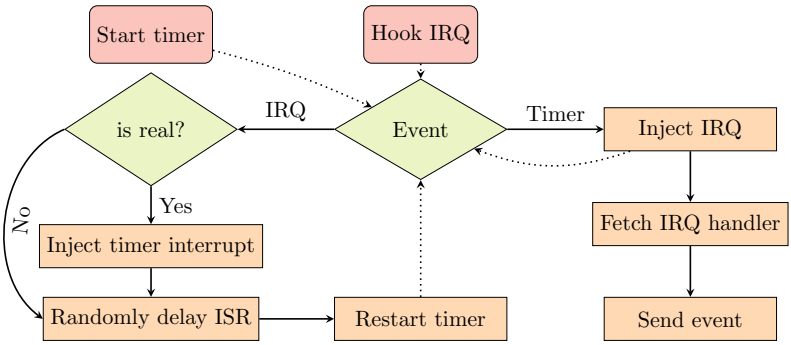


Figure 7.6: General flowchart of the kernel module.

All keystrokes, *i.e.*, real keystrokes and fake keystrokes, are passed to the library in a way which is indistinguishable for an attacker. The only difference is the key code value as well as the target process and window, which both cannot be obtained in keystroke timing side channels.

The second layer protects the library handling the user input against Flush+Reload attacks, including cache template attacks, and Prime+Probe attacks. For every keystroke event received from the kernel, a random keystroke is sent to a hidden window. The library cannot distinguish between real and fake keystrokes and thus both have the same execution path. Note that this also triggers screen redraw events, hence, the screen-redraw interrupt side channel is also covered by *KeyDrown*.

In the third layer, the actual password entry field is protected against Prime+Probe attacks by accessing the underlying buffer whenever a real or a fake keystroke is received.

Combining the three layers, the system-wide set of cache lines that are touched by the code paths through the entire software stack for real and fake keystrokes, are identical. As there is no difference, this voids any advantage an attacker could have gained from a cache side channel.

4.1 First Layer

Basic Concept. Figure 7.6 shows the program flow for the kernel part of *KeyDrown* for both x86 and ARM. We use a non-periodic one-shot timer

interrupt with a random delay to inject a fake keystroke.[1] This leads to a uniform random distribution of keystrokes over time.

The kernel module handles two types of events: Hardware interrupts from the input device, and the timer interrupts. If the kernel module receives one of our timer interrupts, it injects a keyboard interrupt. If it receives a keyboard interrupt, it injects a non-periodic one-shot timer interrupt. Thus, for real and fake keystrokes both interrupts occur. To minimize the effect of the real keyboard interrupt on the interrupt density, the upcoming non-periodic one-shot timer interrupt is canceled. Note that the time between the fake keyboard interrupt and the user pressing a key was also a random delay. *KeyDrown* acts as if this random delay was planned for the fake keyboard interrupt all along. That is, the real keyboard interrupt takes the place of our fake keyboard interrupt. Hence, the real keyboard interrupt has no additional influence on the keystroke interrupt density function. This guarantees that overall, the keystroke interrupt density remains uniform and real keystrokes cannot be distinguished from fake keystrokes.

For the fake keystrokes, the kernel uses a typically unused key value. The kernel does not have varying code paths and data accesses based on the key value, hence, the same code is executed for both real and fake keystrokes. In both cases, the keystroke handler is delayed by a small random delay to hide timing differences from interrupt runtimes. Finally, all keystrokes are passed to the library through the same data structures (cf. Figure 7.5). Consequently, the attacker cannot use a Prime+Probe or Multi-Prime+Probe attack on the kernel to distinguish real and fake keystrokes.

Implementation Details. The first layer of *KeyDrown* is implemented as a Linux kernel module that aims to prevent interrupt-based attacks on keystrokes. We do not require a custom kernel or any patches to the Linux kernel itself, but only the Linux kernel header files for the running kernel. All functionality is implemented in one generic kernel module that can be loaded into any Linux kernel from version 3.4 to 4.10, the newest release at the time of writing. The interrupt hardware and handling mechanism is compatible with all personal computers; thus, there is no further limitation on PC hardware or Linux distributions.

[1] Timer interrupts are often known as periodic interrupts triggering regular operations, e.g., scheduling. However, on modern systems there are significantly more features to timer interrupts, such as non-periodic one-shot timers [21]. One-shot timers are architectural features that can be used through legitimate kernel interfaces and have no side effects on any system timers.

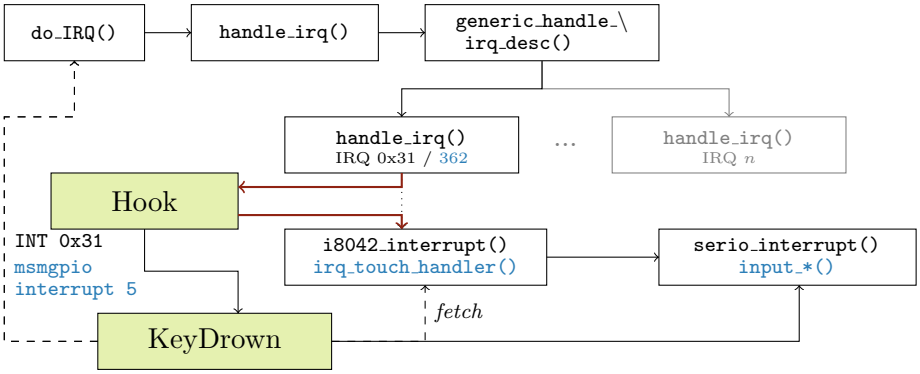


Figure 7.7: Linux kernel module design for x86 and the Snapdragon SoC. Snapdragon specific functions are marked in blue.

Figure 7.7 shows the implementation details of the *KeyDrown* kernel module. The non-periodic one-shot timer interrupts are implemented using the Linux platform-independent high-resolution timer API [31]. On Linux, a driver can register an interrupt handler for a specific interrupt which is called whenever the CPU receives the interrupt. The interrupt service routine `do_IRQ` calls the general `handle_irq` function which subsequently calls `generic_handle_irq_desc` to execute the correct handler for every interrupt. To receive all hardware interrupts, we change the input device's interrupt handler to a function within our kernel module. Afterwards, we forward the interrupt to the actual input device driver (*i.e.*, `i8042_interrupt` on x86, and `irq_touch_handler` on the Nexus 5 (ARM)). Every time the kernel receives one of the non-periodic timer interrupts or a real hardware interrupt, we restart the non-periodic timer with a new random delay to maintain the uniform random distribution over time.

The kernel module triggers a hardware interrupt for every non-periodic timer interrupt. On x86, we can simply execute the `int` assembly instruction with the corresponding interrupt number. This spurious keyboard interrupt travels up until the point where the keyboard driver tries to read the scancode from the hardware. As the driver does not execute the entire `i8042_interrupt` function for spurious interrupts, we access the remaining function to fetch it into the cache as if it was executed. In contrast, for real keys, we access the code that injects the keys to fetch it into the cache as if it was executed. From an attacker's point of view, there is no difference in cache activity between a data fetch and a code fetch, *i.e.*, a Prime+Probe attack cannot determine the difference.

We inject a scancode of a typically unused key, such as F16 or a Windows multimedia key using the standard `serio_interrupt` interface. Thus, from this point on the only difference between real and fake keystrokes is the scancode. Finally, all scancodes are sent to the upper software layers and run through the same execution path.

On the ARM platform, hardware interrupts and device drivers are hardware dependent. We decided to implement our proof-of-concept on the widespread Qualcomm Snapdragon Mobile Station Modem (MSM) SoC [45].

ARM processors generally do not provide an assembly instruction to generate arbitrary interrupts from supervisor mode. Instead, we have to communicate with the interrupt controller directly. The Snapdragon MSM SoC implements its own intermediate I/O interrupt controller. All interrupt generating hardware elements are connected to this interrupt controller and not directly to the GIC. Therefore, if we want to inject an interrupt, we write the interrupt state of the touchscreen interrupt via memory mapped I/O registers to the MSM I/O interrupt controller. The remaining execution path is analogous to the x86 module. When the driver aborts due to a spurious interrupt, we fetch the `irq_touch_handler` to produce the same cache footprint as if it was executed. We inject an out-of-bounds touch event using the `input_event`, `input_report_abs`, and `input_sync` functions, which is then handed to the upper layers.

4.2 Second Layer

Basic Concept. The second layer countermeasure ensures that the control flow within the key-handling library is exactly the same for both real and injected keystrokes. The fundamental idea of the second layer is that real and injected keystrokes should have the same code paths and data accesses in the library. We rely on the events injected in the first layer to propagate them further through the key-handling library. The injected keys sent by our first layer are valid, but typically unused keys, thus they travel all the way up to the user space to the receiving user space application. However, these unused keys might not have the exact same path within the library.

Gruss et al. showed that an attacker can build cache template attacks based on Flush+Reload [14] to detect keystrokes and even distinguish groups of keys. This cache leakage can also be measured with Multi-Prime+Probe. Both attacks exploit the cache activity of certain functions that are only called if a keystroke is handled, *i.e.*, varying execution paths

and access patterns [14]. We mitigate these attacks by duplicating every key event (cf. Figure 7.5) running through multiple execution paths and access sequences simultaneously. The key value of the duplicated key event is replaced by a random key value, and the key event is sent to a hidden window. Hence, the two key events, the real and the duplicated one, are processed simultaneously by the remainder of the library and the two applications. This introduces a significant amount of noise on cache template attacks on the library layer.

The real key event at this point may still be a fake keystroke from the kernel. However, we duplicate the key event in order to trigger key value processing and key drawing in the library and the hidden window for both fake and real keystrokes. Consequently, we cannot distinguish real and fake keystrokes on the library layer using a side channel anymore.

Implementation Details. One of the most popular user interface libraries for Linux is *GTK+* [57]. The *GTK+* library handles the user input for many desktop environments and is included in most Linux distributions [51]. As we cannot hide cache activity, we generate artificial cache activity for the same cache lines that are active when handling real user inputs.

The kernel provides all events, such as keyboard inputs, through the `/dev/input/event*` pseudo-files to the user space. The X Window System uses these files to provide all events to the *GTK+* event queue.

On x86, the second layer is a standalone *GTK+* application. On system startup, we create a hidden window containing a text field. The application uses `poll` to listen to the `/dev/input/event*` interface to get notified whenever a keyboard event occurs. This allows *KeyDrown* to have a very low performance overhead, as the application is not using CPU time as long as it is waiting inside the `poll` function. Whenever we receive a keystroke event from the kernel, we create an additional *GTK+* keystroke event with a random key that is associated with the text field of the hidden window. For every keystroke — regardless of whether it is a printable character or not — that comes from the kernel, the same path is active within the library. Thus, an attacker cannot distinguish an injected keystroke from a real keystroke anymore.

The second layer has no knowledge of an event's source. Thus, it cannot violate *R3*, as the information whether a keystroke is real or injected is not present in the second layer.

On Android, the handling of input events is considerably simpler. The injected events travel directly to the foreground application without going to any non-Android library. Thus, all events have exactly the

same execution path, and it is only necessary to drop our fake event immediately before the registered touch event handler is called. To not leak any information through the non-executed touch handler, we access the cache lines in the same way as if the touch handler was executed.

4.3 Third Layer

Basic Concept. While the first layer protects against interrupt-based attacks and the second layer prevents attacks on the library handling the user inputs, the buffer that stores the actual secret is not protected and can still be monitored using a Prime+Probe attack. The fake keystrokes sent by the kernel are unused key codes, which do not have any effect on the user interface element or the corresponding buffer. We mitigate cache attacks on this layer by generating cache activity on the cache lines that are used when the buffer is processed for any key code received from the kernel. More specifically, we access the buffer every time the library receives a keystroke event from the kernel. This ensures that the buffer is cached for both real and fake keystrokes.

An attacker who mounts a Flush+Reload attack against the library, or a Prime+Probe attack directly on the buffer, sees cache activity for both real and injected events. This is also the case for cache template attacks, as the injected events induce a significant amount of noise in both the profiling and the exploitation phase. Therefore, the third layer protects against attacks that are mounted against the Android keyboard as shown by Lipp et al. [28], or Multi-Prime+Probe attacks directly on the input field buffer (cf. Section 3.2).

Implementation Details. In *GTK+*, the `GtkEntry` widget implements the `GtkEditable` interface, which describes a text-editing widget, used as a single-line text and password entry field. By setting its *visibility* flag, entered characters are shown as a symbol and, thus, hidden from the viewer.

Implementing the countermeasure directly in the *GTK+* library would require rebuilding the library and all of its dependencies. As this is highly impractical, we chose a different approach: `LD_PRELOAD` allows listing shared objects that are loaded before other shared objects on the execution of the program [26]. By using this environment variable, we can overwrite the `gtk_entry_new` function that is called when a new object of `GtkEntry` should be created. In our own implementation, we register a key press event handler for the new entry field. This event handler is called on both real and injected keys and accesses the underlying buffer.

On Android, the basic concept is the same. It is, however, implemented as part of the keyboard and not the library. The keyboard relies on the `inotifyd` command to detect touch events provided by the kernel. If a password entry field is focused, the keyboard accesses the password entry buffer on every touch event by calling the key handling function with a dummy key. This ensures that both the buffer as well as the keyboard's key handling functions are active for every event.

5 Evaluation

We evaluate *KeyDrown* with respect to the requirements *R1*, *R2*, *R3* as well as discuss the performance of our implementation. We evaluate the x86 version of *KeyDrown* on a Lenovo ThinkPad T460s (Intel Core i5-6200U) and the ARM version on both an LG Nexus 5 (ARMv7) and a OnePlus 3T (ARMv8). A large comparison table can be found in the appendix. As the results are very similar for all architectures, we provide the results for the LG Nexus 5 (ARMv7) and the OnePlus 3T (ARMv8) in the appendix. We evaluate four different side channels with and without *KeyDrown*: `procfs`, `rdtsc`, Flush+Reload (including cache template attacks), and Prime+Probe on the last-level cache. We discuss Prime+Probe attacks on the L1 cache and DRAMA side-channel attacks. Table 7.2 gives an overview of all known and new attacks and whether *KeyDrown* prevents them.

To evaluate *KeyDrown*, we chose a uniform key-injection interval $[0\,\text{ms}, 20\,\text{ms}]$. Note that this is not a constant interrupt rate but quite the opposite. Any real keystroke replaces the currently scheduled key injection. Real keystrokes are much rarer and when splitting time into 20 ms intervals, the distribution of real keystrokes in these 20 ms intervals is uniform, identical to the uniform distribution of our key-injection delay. Hence, based on the time a keystroke arrives there is no side channel leaking whether it was a fake one, or a real one. This leads to a uniform interrupt density function with 100 events per second, independent of the real keystrokes.

As described in Section 3, we compare our results to an always-one oracle and a random-guessing oracle. A random-guessing oracle, which chooses randomly — without any information — for every 10 ms interval whether there was a keystroke based on an apriori probability, would achieve an F-score of 0.14. The always-one oracle performs slightly better, as it has a higher true positive rate of 100 %, but it also has a false positive rate of 100 %, *i.e.*, the oracle neither uses nor provides any information.

Table 7.2: Overview which attacks work (●), partly work (◐) and do not work (○) with enabled (✓) and disabled (✘) *KeyDrown*.

	Android < 8		Android ≥ 8		Linux	
KeyDrown	✘	✓	✘	✓	✘	✓
Interface-based [9, 22, 60]	●	◐	○	○	●	○
Interrupt-based (`rdtsc`, [53])	●	○	●	○	●	○
Prime+Probe on L1 [46]	●	○	●	○	●	○
Prime+Probe on LLC	●	○	●	○	●	○
Multi-Prime+Probe	●	○	●	○	●	○
Flush+Reload [14]	●	○	●	○	●	○
DRAMA [43]	●	◐	●	◐	●	◐

The F-score of the always-one oracle is 0.15 and thus, higher than the F-score of a random-guessing oracle. If a side channel yields an F-score of this value or below, the attacker gains no advantage over the always-one oracle from this side channel.

For all evaluated attacks, we provide the precision of the attack with and without *KeyDrown*, based on the best threshold distinguisher we can find. *KeyDrown* does not influence the recall, as it does not reduce the number of true positives and it also does not increase the number of real keystrokes. However, we provide the recall for all attacks with a recall below 1. The harmonic mean of precision and recall — the F-score — gives an indication how well the countermeasure works. We provide the advantage over the always-one oracle as a direct indicator on whether it makes sense to use the side channel or not.

5.1 Requirement *R1*

We evaluate *KeyDrown* with respect to *R1*, the elimination of single-trace attacks. *R1* defines that a side channel may not provide any advantage over an always-one oracle, *i.e.*, the advantage measured in the F-score must be $\leq 0.0\,\%$. We show that *KeyDrown* fulfills this requirement by mounting state-of-the-art attacks with and without *KeyDrown*. Table 7.3 summarizes the F-scores for all attacks with and without *KeyDrown*. In all cases, *KeyDrown* eliminates any advantage that can be gained from the side channel, when considering single-trace attacks only. In some cases, the numerous false positives and false negatives lead to an even worse F-score.

Table 7.3: F-score without and with *KeyDrown* and advantage over always-one oracle for state-of-the-art attacks. *KeyDrown* eliminates any side-channel advantage.

Side Channel	no *KeyDrown*	(Δ always-one)	*KeyDrown*	(Δ always-one)
`procfs`	1.00	(+575.0 %)	0.15	(+0.0 %)
`rdtsc`	0.94	(+537.4 %)	0.14	(−3.8 %)
Flush+Reload	0.99	(+569.3 %)	0.09	(−40.2 %)
LLC Prime+Probe	0.81	(+440.0 %)	0.11	(−27.7 %)

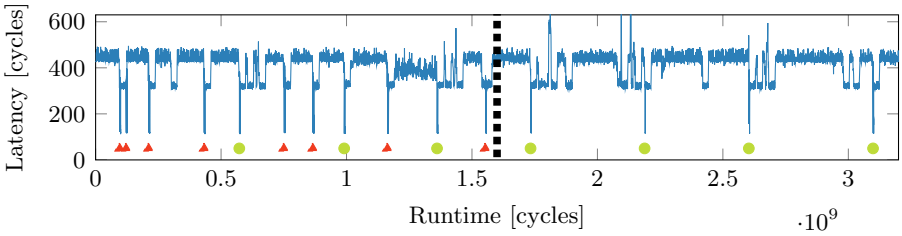


Figure 7.8: Flush+Reload attack on address `0x381c0` of `libgdk-3.so`. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

Flush+Reload. Flush+Reload allows an attacker to monitor accesses to memory addresses of a shared library with a very high accuracy. Figure 7.8 shows the result of such an attack against the `gdk_keymap_get_modifier_mask` function at address `0x381c0` of *libgdk-3.so* (v3.20.4 on Ubuntu Linux), the shared library isolating *GTK+* from the windowing system. This function is executed on every keystroke to retrieve the hardware modifier mask of the windowing system. The attacker measures cache hits on the monitored address whenever a key is pressed and, thus, can spy on the keystroke timings very accurately. While *KeyDrown* is active, the attacker measures additional cache hits on every injected keystroke and cannot distinguish between real and fake keystrokes. When *KeyDrown* is not active, the attack is successful.

For other addresses found using cache template attacks, we made the same observation. Without *KeyDrown*, both profiling and exploiting vulnerable addresses is possible. With *KeyDrown*, we still find all addresses that are loaded into the cache upon keystrokes, however, as we cannot distinguish between real and fake keystrokes we cannot exploit this anymore. Without *KeyDrown*, the precision is 1.00 and the F-score is 0.99, which is a +569.3 % advantage over an always-one oracle. If *KeyDrown*

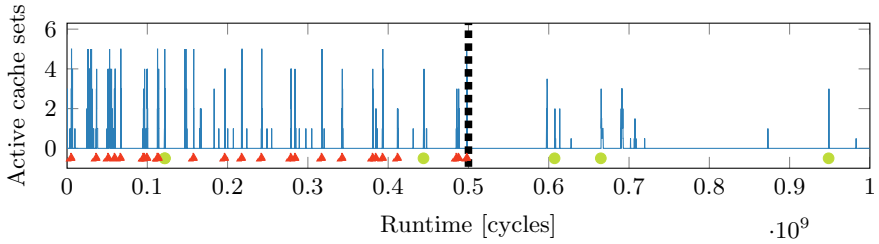


Figure 7.9: Multi-Prime+Probe attack on the 5 cache sets from `0x2514250` to `0x2514390` of `i8042_interrupt`. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

is active, the precision is lowered to 0.05 and, thus, the resulting F-score is 0.09, which is a (negative) advantage of −40.2 % over the always-one oracle.

Prime+Probe. If an attacker cannot use Flush+Reload, a fallback to Prime+Probe is possible. The disadvantage of a Prime+Probe attack on the last-level cache is the amount of noise that increases the false-positive rate. Prior to this work, there was no successful keystroke attack using Prime+Probe on the last-level cache. We perform the Multi-Prime+Probe attack presented in Section 3.2 to attack keystroke timings.

Figure 7.9 shows the results of inferring keystrokes by detecting the keyboard interrupt handler's cache activity using Multi-Prime+Probe. We monitored 5 cache sets in parallel for a higher noise robustness. Without *KeyDrown*, the precision is already at a quite low value of 0.71 with a recall of only 0.92, yielding an F-score of 0.81, which is an advantage over an always-one oracle of +440.0 %. Memory accesses to one of the cache sets by any other application cannot be distinguished from a cache set access by the keyboard interrupt handler, causing a high number of false positives. If we enable *KeyDrown*, the precision drops to 0.06, as the attacker additionally measures the noise generated by the injected keystrokes. The F-score is then 0.11, which is a (negative) advantage over an always-one oracle of -27.7 %.

Figure 7.10 shows the results of mounting a Multi-Prime+Probe attack on the buffer of a password field within a *GTK+* application. Although there is more noise visible in the traces, we achieve the same precision and F-score as for the attack on the kernel module when *KeyDrown* is disabled. If we enable *KeyDrown*, the precision drops to 0.05, which is a

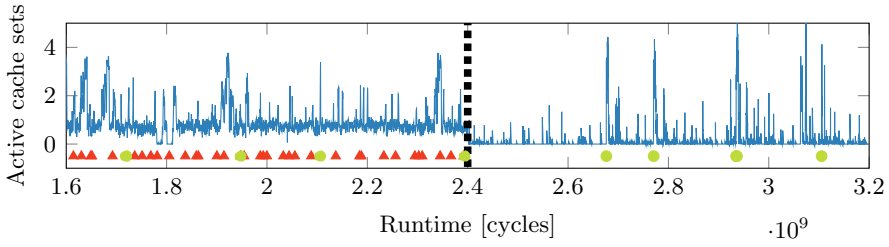


Figure 7.10: Multi-Prime+Probe attack on the 5 cache sets corresponding to a password field's buffer within a demo application. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

bit lower than the precision on the kernel, resulting in an F-score of 0.10, which is again no advantage over an always-one oracle.

Interrupts. *KeyDrown* also protects against interrupt-based attacks, including our new timing-based attack. For the attacks based on the `procfs` interface [9, 22], we measure an average reading interval of 980 cycles. With our new attack based on `rdtsc`, we can measure every 95 cycles on average, resulting in a probing frequency one order of magnitude higher.

Figure 7.11 and Figure 7.12 illustrate the effect of our countermeasure on the `procfs`-based interrupt attack and the `rdtsc`-based attack, respectively. Without *KeyDrown*, we achieve a precision of 1.00 for the `procfs`-based attack and a precision of 0.89 for the `rdtsc`-based attack, resulting in an F-score of 1.00 and 0.94 respectively. Enabling *KeyDrown* reduces the precision to 0.08 and 0.07 respectively. Thus, the resulting F-score is 0.15, which is exactly the same as the always-one oracle, for the `procfs`-based attack, and 0.14 for the `rdtsc`-based attack, which is a (negative) advantage over an always-one oracle of -3.8 %.

5.2 Requirement *R2*

KeyDrown reduced the F-score of all state-of-the-art attacks such that using the side channel gives an advantage over an always-one oracle of $\leq$0.0 %. An attacker might still be able to combine multiple traces from the same user and build a binary classifier, if the user predictably and repeatedly types the same character sequence. Such a classifier may achieve a higher precision and a higher F-score, as long as there is actually meaningful information in the corresponding traces. However, there is a

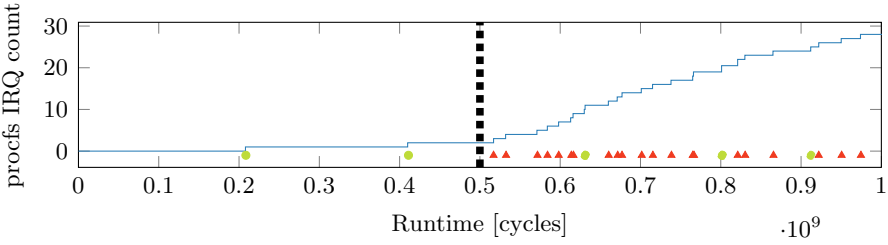


Figure 7.11: `procfs`-based attack. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (after dotted line).

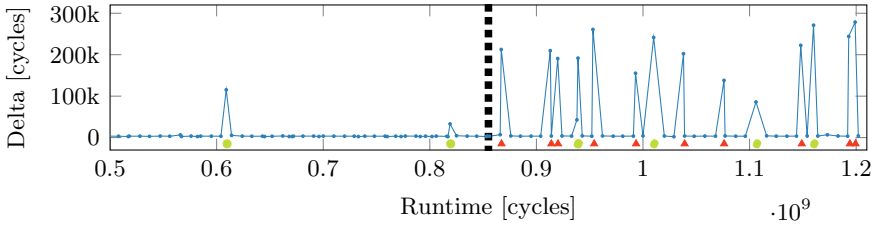


Figure 7.12: `rdtsc`-based attack. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (after dotted line).

practical limit on the number of traces an attacker can gather from the user, which *R2* estimates to be 1825 traces.

In our attack scenario, we model a powerful attacker who can take advantage of the following properties:

1. **Noise-free side channel:** The used side channel is noise-free, *i.e.*, only real and fake keystrokes are recorded, no other system noise.
2. **Perfect (re-)alignment:** The attacker can detect when a password input starts with a variance as low as the variance of a single inter-keystroke interval. Additionally, the attacker has an alignment-oracle providing perfect re-alignment for the traces after each guessed keystroke. This leads to the same variance for every key instead of an accumulated variance.
3. **Known length:** The attacker knows the exact length of the password and expects exactly as many keystrokes.

This attacker is far stronger than any practical attacker.

We generate simulated traces that fulfill the properties above and calculate the average of the perfectly (re-)aligned traces. As our attacker

knows the length n of the password, he finds the n most likely positions where a Gaussian distribution with the known inter-keystroke interval variance matches. If the expected value μ of each Gaussian curve is within the variance of the real keystroke, we assume that the number of traces was sufficient to extract the positions of the real keystrokes.

We set the simulated typing variance to ± 40 ms which is a bit less than the value reported by Lee et al. [27] for trained text sequences. In total, we generated 300 000 simulated traces, each containing 8 keystrokes within 2 s. From this set of simulated traces, we evaluated how many randomly chosen traces we have to combine to extract the correct positions of the keystrokes. We found that an attacker requires an average of 2458 traces to extract the correct positions. This is significantly more than the 1825 traces deemed to be secure in *R2*.

5.3 Requirement *R3*

As *KeyDrown* fulfills *R1* and *R2*, we can be assured that the underlying technique is a working countermeasure. However, as the implementation of a countermeasure itself can leak information, we need to ensure that *KeyDrown* does not create a new software-based side channel in order to satisfy *R3*.

First Layer. The first layer runs in the kernel and can thus only be attacked using Prime+Probe. Figure 7.7 shows that, in general, we have the same execution flow and data accesses. For the few deviations, we prevent any potential cache leakage from non-executed code paths by performing the same memory accesses as if they were executed. As an attacker cannot distinguish if a cache activity is caused by an execution or a memory read, the module's cache activity does not leak additional information to an attacker. We investigated the cache activity on the cache sets used by the *KeyDrown* kernel module in a Prime+Probe attack and found no leakage from our module.

Second Layer. To make use of the same noise as in the first layer, the second layer listens to the `/dev/input/event0` pseudo-file containing all keyboard events. This file is not world-readable but only readable by members of the `input` group. Thus, this layer runs as a separate *keydrown* user with default limited privileges and additional access to this file.

As the second layer is a user space binary, an attacker could theoretically mount a Flush+Reload attack against the second layer. However, attacking the second layer does not result in any additional information. The second layer does not know whether an event is generated from a real

or an injected keystroke. For every event, a random printable character is sent to the hidden window. Thus, the execution path for printable characters is always active, and the attacker cannot learn any additional information from attacking the second layer. The same is also true for Prime+Probe, even a successful attack does not provide additional information. We investigated the cache activity of the *KeyDrown* shared library parts and the *KeyDrown* user space binary using a template attack and did not find any leakage.

Third Layer. The third layer builds upon the second layer, and thus the same argumentation as for the second layer holds. An attacker cannot distinguish real and injected keystrokes in the second layer as all events are merged within the kernel. As the third layer relies on the same source as the second layer, there is also no leakage from the third layer. Thus, any attack on the third layer does not give an attacker any advantage over any other attack. We investigated the cache activity of the control flow and data accesses up to the point where the input is stored in the buffer in a Prime+Probe attack and found no leakage.

5.4 Performance

On the x86 architecture, we evaluate the performance impacts of running our *KeyDrown* implementation on standard Ubuntu 16.10. We use *lmbench* [34], a set of micro benchmarks for performance analysis of UNIX systems, and *PARSEC 3.0* [4], a benchmark suite intended to simulate a realistic workload on multicore systems.

The lmbench results for the latency benchmarks show a performance overhead of 6.9 %. However, as the execution time of the lmbench benchmarks is in the range of microseconds to nanoseconds, the overhead does not allow for definite conclusions about the overall system performance. Still, we can see that the injected interrupts have only a small impact on the kernel performance.

To measure the overall performance, we run the PARSEC 3.0 benchmark with different numbers of cores. The average performance overhead over all measurements for any number of cores is 2.5 %. For workloads that do not use all cores, the performance impact is only 2.0 % for one core and 2.5 % for two cores. Only if the CPU is under heavy load, we observe a higher performance impact of 3.1 % when running the benchmarks on all cores.

On ARM, we evaluate the battery consumption of *KeyDrown*. We measure the power consumption in three different scenarios, always over

the timespan of 5 min. First, if the screen is off, our fake interrupts are completely disabled, and thus, *KeyDrown* does not increase the power consumption if the mobile phone is not used. Second, if the screen is turned on, but the keyboard is not shown, *KeyDrown* increases the power consumption slightly by 3.9 %. Third, if the keyboard is shown, the power consumption with *KeyDrown* increases by 15.6 %. However, as most of the time, the keyboard is not shown, *KeyDrown* does not have great impacts on the overall power consumption. In total, *KeyDrown* reduces the battery life time of an average user by 4.6 %.[1]

Note that all the performance measurements were done using the proof-of-concept. We expect that the proof-of-concept can be considerably improved in terms of performance overhead and battery usage by not injecting the fake interrupts all the time but only while the user is actually entering text.

5.5 Other Attacks

While we already demonstrated that the most powerful side-channel attacks are mitigated, we discuss three other attacks subsequently. The Prime+Probe side channel results from the victim program evicting a cache line of the attacker. As the last-level cache is inclusive, any eviction from the last-level cache also evicts this line from the L1 cache. However, if a cache line is evicted from the L1 cache it may still be in the last-level cache. In this case, the attacker would miss the eviction and thus the targeted event. In our evaluation, we find that the recall is very close to 1 in all cases. This means that we do not miss any events. Hence, there is no additional information that an attacker could gain from a Prime+Probe attack on the L1 cache. Consequently, evaluating Prime+Probe on the last-level cache is sufficient to conclude that Prime+Probe on the L1 cache does not leak additional information.

The DRAMA side-channel attack presented by Pessl et al. [43] results from a massive number of secret-dependent memory accesses that lead to heavy cache thrashing, *i.e.*, the victim program accesses lots of memory locations that are mapped to the same cache lines. It is therefore unclear whether or not *KeyDrown* protects against DRAMA. In particular, it does not protect against the specific attack against keystrokes in the Firefox

[1] For an average user, with a screen-on-time of 145 minutes, 2617 touch actions [56], and 1 charge per day (21.7 hours standby time) [25], an average typing speed of 20 words per minute [3] and hence, 100 characters per minute [38], we can assume a keyboard-shown time of 26 minutes per battery charge. For modern devices, screen-on consumes approximately 33 times more battery than standby [12].

address bar (cf. Section 6). However, we observe that *KeyDrown* adds significant amounts of noise to the attack.

To our surprise, we found that *KeyDrown* also mitigates the keystroke timing attack based on the event queue of the Chrome browser by Vila et al. [53] (USENIX Sec'17). They state that the leakage is due to the time it takes Chrome to enqueue and dispatch every keystroke event. However, we investigated their attack and were able to reproduce it on MacOS systems reliably, but not on other operating systems, indicating that this effect is not purely Chrome-specific, but also has other influences. We believe that their attack exploits multiple effects in combination: the Chrome event queue and the interruption by the hardware interrupts as in our `rdtsc`-based attack, which is additionally amplified by the significantly higher I/O latency caused by the atypical MacOS design for interrupt handling [42].[1]

A preliminary version of our `rdtsc`-based interrupt timing attack was the basis for the same attack in JavaScript [29]. They were able to identify the user typed URL, and distinguish different users based on this attack. As they report, *KeyDrown* successfully mitigates their attack in JavaScript as well.

KeyDrown has a significant effect on the attack by Jana et al. [22], exploiting CPU utilization spikes. The fake keystrokes introduce similar small CPU utilization spikes making their attack impractical. Similarly, *KeyDrown* triggers screen redraws through the hidden window (cf. Section 4). Hence, *KeyDrown* also makes the screen-redraw-based attack by Diao et al. [9] impractical.

6 Limitations and Future Work

KeyDrown mitigates software-based side-channel attacks on keystrokes and keystroke timings in general. This includes even the application layer without changing an existing application if either:

- the input is processed only after the user finished entering the text (e.g., pressing a button on a login form), and there is no immediate

[1] Interrupt handlers on MacOS only enqueue the task to handle an interrupt in a queue, taking almost zero time. This queue is processed by an interrupt service thread, doing the actual interrupt handling. This additional step increases the total computation time compared to traditional interrupt handling. As the attack is not influenced by which thread does the actual interrupt handling, this increased interruption time amplifies the side channel.

action when a key is pressed (e.g., as with password fields or simple text input fields),

- or the application is designed to remove side-channel information.

Otherwise, the application layer might still leak timing information when performing intense computations for every single keystroke, e.g., autocomplete or live search features [43].

Song et al. [50] demonstrated keystroke timing attacks performed by a malicious observer on the same network. Zhang et al. [60] speculated that this attack vector could also be exploited through `/proc/net`, which might still be available in Android O. However, this is not a local software-based attack but a side channel for a remote attacker. Hence, dedicated countermeasures beyond *KeyDrown* should be implemented to prevent this attack.

Some software-based side channel attacks may be unaffected by *KeyDrown*, e.g., the sensor-based attacks exploiting the accelerometer [6], but these attacks can be thwarted by introducing noise [48].

KeyDrown protects against software-based attacks on keystrokes as well as touch events. However, swipe movements are not protected as their interrupt rate is too high. While this is not a problem in the case of a password input — if a password can be swiped and thus pasted from a dictionary, there is little to protect — it is future work to investigate how to extend *KeyDrown* to protect swipe movements.

Furthermore, our novel side channels emphasize the necessity to deploy *KeyDrown* widely. Multi-Prime+Probe attacks provide a significantly higher accuracy than previous Prime+Probe attacks on dynamic memory and kernel memory. It is likely that Multi-Prime+Probe works similarly in cloud systems and thus allows highly accurate attacks like keystroke timing attacks across virtual machine boundaries.

Our current proof-of-concept is not optimized for usability. For most of the system, the real and fake keystrokes are indistinguishable, the keystrokes are just led to different windows. Known limitations are that fake keystrokes interrupt key repetition and may interfere with input methods in modern computer games. However, these limitations can be overcome by adapting how key repetition is implemented.

7 Conclusion

Keystrokes are processed on many different layers of the software stack and are thus not entirely covered by previously proposed defense mechanisms. In this article, we presented *KeyDrown*, a novel defense mechanism that

mitigates keystroke timing attacks. *KeyDrown* injects a large number of fake keystrokes on the kernel level and propagates them — through all layers of the software stack — up to the user space application. A careful design and implementation of this countermeasure ensures that all software routines involved in the processing of a keystroke are loaded, irrespective of whether a real or a fake keystroke is processed. Thereby, *KeyDrown* mitigates interrupt-based attacks, Prime+Probe attacks, and Flush+Reload attacks on the entire software stack. With *KeyDrown*, an attacker cannot distinguish fake from real keystrokes in practice anymore. Our evaluation shows that *KeyDrown* eliminates any advantage an attacker can gain from side channels, *i.e.*, $\leq$0.0 % advantage over an always-one oracle, thus, it successfully mitigates keystroke timing attacks.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable feedback and Johannes Winter for insights on ARM interrupt handling. This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

Appendix

We compare the accuracy of four different side channels with and without *KeyDrown* (`procfs`, `rdtsc`, Flush+Reload, and Prime+Probe on the last-level cache) on three different architectures: a Lenovo ThinkPad T460s (Intel Core i5-6200U), an LG Nexus 5 (ARMv7), and a OnePlus 3T (ARMv8). Table 7.4 summarizes the F-scores for all attacks with and without *KeyDrown*. *KeyDrown* prevents keystroke timing attacks in all cases when considering single-trace attacks only.

We performed our experiments on the touchscreen soft-keyboard of the Nexus 5. With *KeyDrown*, the precision is lowered to 0.01 and, thus, the

Table 7.4: F-score without and with *KeyDrown* for state-of-the-art attacks.

Device	Side Channel	unprotected	*KeyDrown*
ThinkPad T460s	`procfs`	1.00	0.15
LG Nexus 5	`procfs`	1.00	0.15
OnePlus 3T	`procfs`	1.00	0.15
ThinkPad T460s	Interrupt-timing (`rdtsc`)	0.94	0.14
LG Nexus 5	Interrupt-timing	0.94	0.14
OnePlus 3T	Interrupt-timing	0.99	0.15
ThinkPad T460s	Flush+Reload	0.99	0.09
LG Nexus 5	Flush+Reload	0.99	0.02
OnePlus 3T	Flush+Reload	0.93	0.10
ThinkPad T460s	Prime+Probe on LLC	0.81	0.11
LG Nexus 5	Prime+Probe on LLC	0.80	0.11
OnePlus 3T	Prime+Probe on LLC	0.89	0.07

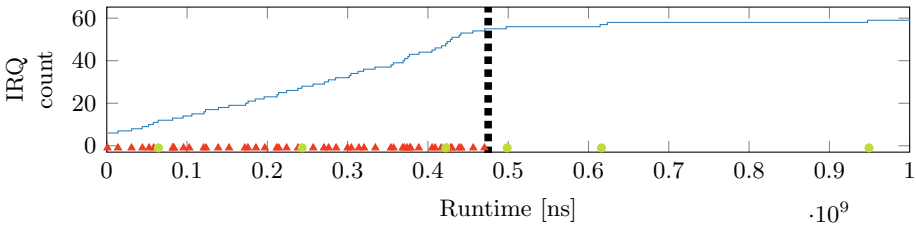


Figure 7.13: `procfs`-based attack on the Nexus 5. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

resulting F-score of 0.02 means a $\leq -86.5\,\%$ advantage over an always-one oracle.

Figure 7.13 and Figure 7.14 show a `procfs`-based interrupt attack and a timing-based attack, both on the Nexus 5. Without *KeyDrown*, we achieve a precision of 1.00 for the `procfs`-based attack and 0.89 for the timing-based attack, resulting in an F-score of 1.00 and 0.94 respectively. Enabling *KeyDrown* reduces the precision to only 0.08 and 0.07 respectively. Thus, the resulting F-score is 0.15 for the `procfs`-based attack, and 0.14 for the timing-based attack, which is an advantage of $\leq 0.0\,\%$ over an always-one oracle.

Figure 7.15 shows the results of inferring keystrokes by detecting the touchscreen interrupt handler's cache activity using Multi-Prime+Probe on the Nexus 5. We monitored 5 cache sets in parallel for noise robustness. Without *KeyDrown*, the precision is 0.71 with a recall of only 0.92, as

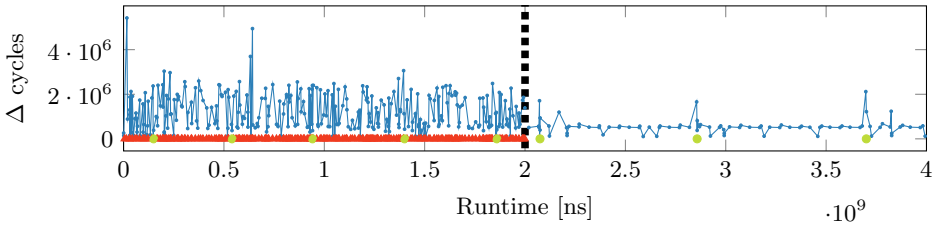


Figure 7.14: Timing-based attack on the Nexus 5. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

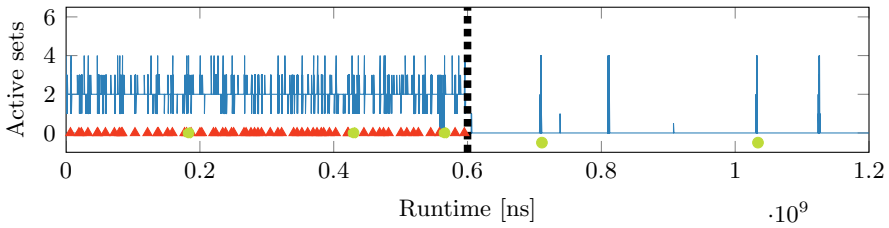


Figure 7.15: Multi-Prime+Probe attack on the 5 cache sets from `0x382659be` to `0x38265abe` of `touch_irq_handler` on the Nexus 5. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

an access to one of the cache sets by any other application cannot be distinguished from a cache set access by the touchscreen interrupt handler, resulting in a high number of false positives. If we enable *KeyDrown*, the precision drops to 0.06, as the attacker additionally measures the noise generated by the injected keystrokes. Thus, the F-score is 0.11.

We performed our experiments on the OnePlus 3T touchscreen soft-keyboard. Figure 7.16 shows a Flush+Reload attack on `libflinger.so`. Without *KeyDrown*, the precision is 0.88 and the F-score is thus 0.93. If *KeyDrown* is active, the precision is lowered to 0.05 and, thus, the resulting F-score of 0.10 means a $\leq -32.5\,\%$ advantage over an always-one oracle.

Figure 7.17 and Figure 7.18 show a `procfs`-based interrupt attack as well as a timing-based attack, both on the OnePlus 3T. The attack has a precision of 1.00 (F-score of 1.00) and 0.99 (F-score of 0.99) respectively. Enabling *KeyDrown* reduces the precision to only 0.08 (F-score is 0.15) and 0.07 (F-score is 0.15) respectively, which is a $0.0\,\%$ advantage over an always-one oracle.

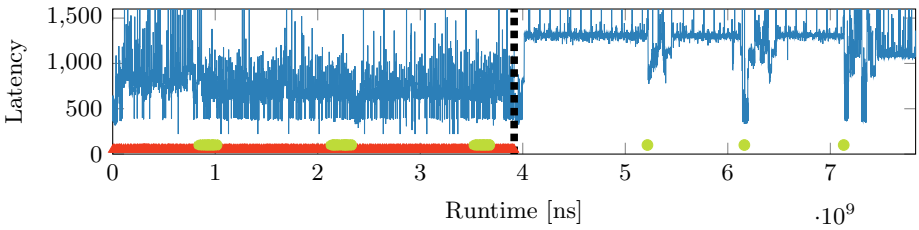


Figure 7.16: Flush+Reload attack on address `0x28ec0` of `libflinger.so` on the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

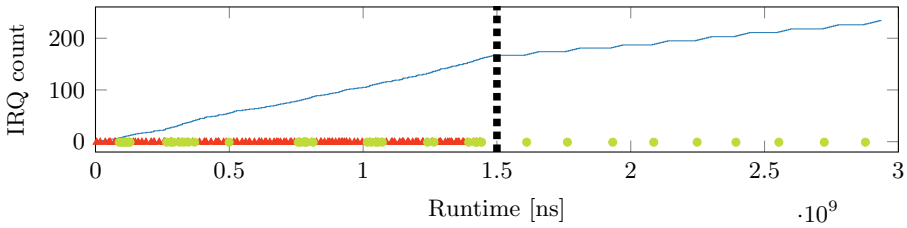


Figure 7.17: `procfs`-based attack on the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

Figure 7.19 shows the results of inferring keystroke timings by detecting the touchscreen interrupt handler's cache activity using Multi-Prime+Probe on the OnePlus 3T. We monitored 5 cache sets in parallel for a higher noise robustness. Without *KeyDrown*, the precision is already at a quite low value of 0.80 with a recall of only 1.00, as access to one of the cache sets by any other application cannot be distinguished from a cache set access by the touchscreen interrupt handler. Thus, this attack has a high number of false positives. If we enable *KeyDrown*, the precision drops to 0.10, as the attacker additionally measures the noise generated by the injected keystrokes. Thus, the F-score is 0.07, which is a $\leq -52.7\,\%$ advantage over an always-one oracle.

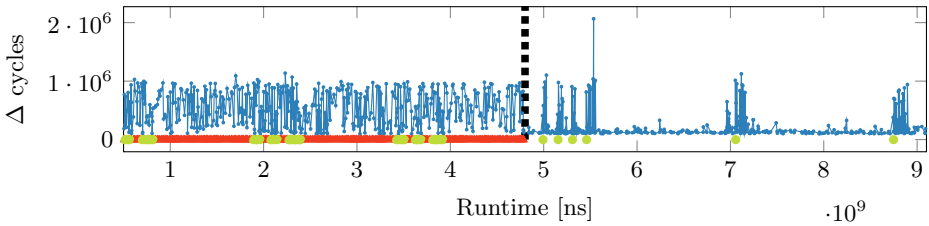


Figure 7.18: Timing-based attack on the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable with *KeyDrown* (before dotted line).

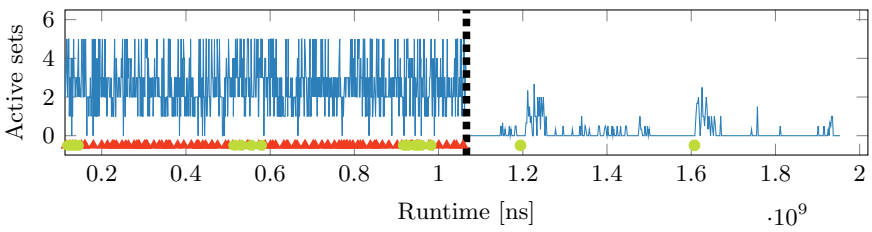


Figure 7.19: Multi-Prime+Probe attack on the 5 cache sets from `0x3fc0355c28` to `0x3fc0355d68` of `msm_gpio_irq_handler` of the OnePlus 3T. Injected keystrokes (▲) and real events (●) are not distinguishable when *KeyDrown* is active (before dotted line).

References

[1] ARM. *Application Note 176 – How a GIC works*. 2007. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0176c/ar01s03s02.html.

[2] ARM. *ARM Generic Interrupt Controller Architecture version 2.0*. 2013.

[3] Patti Bao, Jeffrey Pierce, Stephen Whittaker, and Shumin Zhai. "Smart phone use by non-mobile business users." In: *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*. 2011.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC benchmark suite: Characterization and architectural implications." In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 2008.

[5] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005.

[6] Liang Cai and Hao Chen. "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion." In: *USENIX Workshop on Hot Topics in Security – HotSec*. 2011.

[7] CSID. *Consumer Survey: Password Habits*. 2012. URL: http://www.csid.com/wp-content/uploads/2012/09/CS_PasswordSurvey_FullReport_FINAL.pdf.

[8] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. "The Tangled Web of Password Reuse." In: *NDSS'14*. 2014.

[9] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis." In: *S&P'16*. 2016.

[10] Shirley Gaw and Edward W. Felten. "Password Management Strategies for Online Accounts." In: *SOUPS'06*. 2006.

[11] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2016), pp. 1–27. DOI: 10.1007/s13389-016-0141-6.

[12] Nirave Gondhia. *Samsung Galaxy S7 battery life review*. 2016. URL: http://www.androidauthority.com/samsung-galaxy-s7-battery-life-review-683968/.

[13] Google. *Android O prevents access to /proc/stat*. June 2017. URL: https://issuetracker.google.com/issues/37140047.

[14] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[15] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA'16*. 2016.

[16] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." In: *USENIX Security Symposium*. 2017.

[17] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *S&P'11*. 2011.

[18] Syed Idrus, Estelle Cherrier, Christophe Rosenberger, and Patrick Bours. "Soft Biometrics for Keystroke Dynamics: Profiling Individuals While Typing Passwords." In: *Computers & Security* 45 (2014), pp. 147–155. ISSN: 0167-4048.

[19] Mehmet S Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud." In: *CHES'16*. 2016.

[20] Intel. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)*. 1996.

[21] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2014.

[22] Suman Jana and Vitaly Shmatikov. "Memento: Learning Secrets from Process Footprints." In: *S&P'12*. 2012.

[23] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX." In: *CCS'16*. 2016.

[24] Nick Kralevich. "Honey, I Shrunk the Attack Surface." In: *Black Hat 2017 Briefings*. 2017.

[25] DU Global Battery Lab. *Global App Power Consumption Report 2016, H1*. 2016. URL: https://medium.com/@DU_Global_Battery_Lab/e7f9b845bed.

[26] *ld.so(8) Linux Programmer's Manual*. Linux man-pages project, 2016. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html.

[27] Po-Ming Lee, Wei-Hsuan Tsui, and Tzu-Chien Hsiao. "The Influence of Emotion on Keyboard Typing: An Experimental Study Using Auditory Stimuli." In: *PLOS ONE* 10 (2015), pp. 1–16.

[28] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. 2016.

[29] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript." In: *ESORICS'17*. (to appear). 2017.

[30] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *S&P'15*. 2015.

[31] LWN. *The high-resolution timer API*. Jan. 2006. URL: https://lwn.net/Articles/167897/.

[32] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. "A Study of Probabilistic Password Models." In: *S&P'14*. 2014.

[33] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters." In: *RAID'15*. 2015.

[34] Larry W McVoy, Carl Staelin, et al. "lmbench: Portable Tools for Performance Analysis." In: *USENIX ATC'96*. 1996.

[35] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks." In: *USENIX Security Symposium*. 2016.

[36] Microsoft. *ACPI system description tables*. July 2016. URL: https://msdn.microsoft.com/en-us/windows/hardware/drivers/bringup/acpi-system-description-tables#madt.

[37] Arvind Narayanan and Vitaly Shmatikov. "Fast Dictionary Attacks on Passwords Using Time-space Tradeoff." In: *CCS'05*. 2005.

[38] Peter Norvig. *English letter frequency counts: Mayzner revisited.* 2013. URL: http://norvig.com/mayzner.html.

[39] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS'15*. 2015.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA*. 2006.

[41] Colin Percival. "Cache missing for fun and profit." In: *Proceedings of BSDCan*. 2005.

[42] *Performance Considerations*. Apple Inc., 2013. URL: https://developer.apple.com/library/content/documentation/Darwin/Conceptual/KernelProgramming/performance/performance.html.

[43] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium*. 2016.

[44] Svetlana Pinet, Johannes C. Ziegler, and F.-Xavier Alario. "Typing Is Writing: Linguistic Properties Modulate Typing Execution." In: *Psychon Bull Rev* 23.6 (Apr. 2016), pp. 1898–1906.

[45] Qualcomm. *Snapdragon Mobile Processors and Chipsets*. Jan. 2017. URL: https://www.qualcomm.com/products/snapdragon.

[46] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds." In: *CCS'09*. 2009.

[47] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "Encountering Stronger Password Requirements: User Attitudes and Behaviors." In: *SOUPS'10*. 2010.

[48] Prakash Shrestha, Manar Mohamed, and Nitesh Saxena. "Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise." In: *WiSec'16*. 2016.

[49] Laurent Simon, Wenduan Xu, and Ross Anderson. "Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards." In: *Proceedings on Privacy Enhancing Technologies* (2016).

[50] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH." In: *USENIX Security Symposium*. 2001.

[51] The GTK+ Team. *GTK+ Features*. 2016. URL: https://www.gtk.org/features.php.

[52] Raphael Veras, Christopher Collins, and Julie Thorpe. "On Semantic Patterns of Passwords and their Security Impact." In: *NDSS'14*. 2014.

[53] Pepe Vila and Boris Köpf. "Loophole: Timing Attacks on Shared Event Loops in Chrome." In: *USENIX Security Symposium*. 2017.

[54] Rick Wash, Rader Rader, Ruthie Berman, and Zac Wellmer. "Understanding Password Choices: How Frequently Entered Passwords Are Re-used across Websites." In: *SOUPS'16*. 2016.

[55] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. "Password Cracking Using Probabilistic Context-Free Grammars." In: *S&P'09*. 2009.

[56] Michael Winnick and Jess Mons. *Mobile touches: a study on humans and their tech*. 2016. URL: https://blog.dscout.com/mobile-touches.

[57] X.org Foundation. *xorg Documentation*. Oct. 2014. URL: https://www.x.org/wiki/Documentation/.

[58] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

[59] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. "Mapping the Intel Last-Level Cache." In: *Cryptology ePrint Archive, Report 2015/905* (2015), pp. 1–12.

[60] Kehuan Zhang and XiaoFeng Wang. "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems." In: *USENIX Security Symposium*. 2009.

8

JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks

Publication Data

Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks." In: *NDSS*. 2018

Contributions

Main author.

JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks

Michael Schwarz, Moritz Lipp, and Daniel Gruss

Graz University of Technology, Austria

Abstract

Modern web browsers are ubiquitously used by billions of users, connecting them to the world wide web. From the other side, web browsers do not only provide a unified interface for businesses to reach customers, but they also provide a unified interface for malicious actors to reach users. The highly optimized scripting language JavaScript plays an important role in the modern web, as well as for browser-based attacks. These attacks include microarchitectural attacks, which exploit the design of the underlying hardware. In contrast to software bugs, there is often no easy fix for microarchitectural attacks.

We propose *JavaScript Zero*, a highly practical and generic fine-grained permission model in JavaScript to reduce the attack surface in modern browsers. *JavaScript Zero* facilitates advanced features of the JavaScript language to dynamically deflect usage of dangerous JavaScript features. To implement *JavaScript Zero* in practice, we overcame a series of challenges to protect potentially dangerous features, guarantee the completeness of our solution, and provide full compatibility with all websites. We demonstrate that our proof-of-concept browser extension *Chrome Zero* protects against 11 unfixed state-of-the-art microarchitectural and side-channel attacks. As a side effect, *Chrome Zero* also protects against 50 % of the published JavaScript 0-day exploits since Chrome 49. *Chrome Zero* has a performance overhead of 1.82% on average. In a user study, we found that for 24 websites in the Alexa Top 25, users could not distinguish browsers with and without *Chrome Zero* correctly, showing that *Chrome Zero* has no perceivable effect on most websites. Hence, *JavaScript Zero* is a practical solution to mitigate JavaScript-based state-of-the-art microarchitectural and side-channel attacks.

The original publication is available at `http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_07A-3_Schwarz_paper.pdf`.

1 Introduction

Over the past 20 years, JavaScript has evolved to the predominant language on the web. Of the 10 million most popular websites, 94.7 % use JavaScript [54]. Dynamic content relies heavily on JavaScript, and thus, most pages use JavaScript to improve the user experience, using, e.g., AJAX and dynamic page manipulation. Especially for platform-independent HTML5 applications, JavaScript is a vital component.

With the availability of modern browsers on mobile devices, web applications target smartphones and tablets as well. Furthermore, mobile platforms typically provide a number of sensors and features not present on commodity laptops and desktop computers. To make use of these additional features, the World Wide Web Consortium (W3C) provides drafts and recommendations for additional APIs [53]. Examples include the Geolocation API [52] and the Battery Status API [49]. These APIs are supported by most browsers and allow developers to build cross-platform web applications with similar functionality as native applications.

Undoubtedly, allowing every website to use such APIs has security and privacy implications. Websites can exploit sensors to fingerprint the user [51] by determining the number of sensors, their update frequency, and also their value (e.g., for battery level or geolocation). Furthermore, sensor data can be exploited to mount side-channel attacks on user input [7, 23].

Microarchitectural attacks can also be implemented in JavaScript, exploiting properties inherent to the design of the microarchitecture, such as timing differences in memory accesses. Although JavaScript code runs in a sandbox, Oren et al. [33] demonstrated that it is possible to mount cache attacks in JavaScript. Since their work, a series of microarchitectural attacks have been mounted from websites, such as page deduplication attacks [14], Rowhammer attacks [15], ASLR bypasses [13], and DRAM addressing attacks [40].

As a response to these attacks, some—but not all—of the APIs have been restricted by reducing the resolution (e.g., High Precision Time API) [2, 6, 9] or completely removing them (e.g., DeviceOrientation Event Specification) [50]. However, these countermeasures are incomplete as they do not cover all sensors and are circumventable [13, 40].

A common trait of all attacks is that they rely on the behavior of legitimate JavaScript features, which are rarely required by benign web applications. However, removing these JavaScript features entirely breaks the compatibility with the few websites that use them in a non-malicious

way. This is, for example, the case with NoScript, a browser extension that completely blocks JavaScript on a page [10].

We propose *JavaScript Zero*, a fine-grained JavaScript permission system, which combines ideas from existing permission systems in browsers and smartphone operating systems. *JavaScript Zero* facilitates advanced features of the JavaScript language to overcome the following three challenges:

- C1 Restrictions must not be circumventable using self-modifying code, such as higher-order scripts.
- C2 Restricting access to potentially dangerous features must be irreversible for a website.
- C3 Restrictions must not have a significant impact on compatibility and user experience.

To overcome challenge C1, we utilize advanced language features such as virtual machine layering [19] for security. In contrast to previous approaches [56], virtual machine layering allows redefining any function of the JavaScript virtual machine at runtime. Hence, we can cope with obfuscated code and higher-order scripts, *i.e.*, scripts that generate scripts. *JavaScript Zero* replaces all potentially dangerous functions with secure wrappers. When calling such a function, *JavaScript Zero* decides whether to perform a pre-defined action or ask the user for permission to execute the function.

To overcome challenge C2, we utilize closures, another advanced feature of the JavaScript language, for security. Variables in closures cannot be accessed from any outside scope, providing us with language-level protection for our countermeasure. With closures, we make all references to original unprotected functions inaccessible to the website.

We provide a proof-of-concept implementation as a Chrome browser extension, *Chrome Zero*. In contrast to previous protection techniques [18, 24], *Chrome Zero* requires no changes to the browser source code. Hence, *Chrome Zero* requires lower maintenance efforts while at the same time users also benefit from the security of the most up-to-date browser.

To overcome challenge C3, we keep user interactions to a minimum and provide multiple *protection levels* with pre-defined restrictions. We do not only provide a binary permission system to block or allow certain functionality, but we also allow to modify the semantics of functions and objects. For example, the user can allow the usage of the high-precision timing API but decides to reduce the available resolution to 100 ms instead of 5 µs. These settings can be configured by either the user or the community, through a *protection list*.

We evaluate the efficacy of *Chrome Zero* on 23 recent side-channel attacks, microarchitectural attacks, and 0-day exploits. We show that it successfully prevents all microarchitectural and side-channel attacks in JavaScript. Although not a main goal of *Chrome Zero*, it also prevents 50 % of the published JavaScript 0-day exploits since Chrome 49. This shows that we were able to solve challenges C1 and C2.

To evaluate whether *Chrome Zero* solves challenge C3, we measure the performance overhead and the impact on the user experience for the Alexa Top 25 websites, at the second highest security level. On average, we observe a performance overhead of 1.82 %. In a double-blind user study, we found that for 24 websites out of the Alexa Top 25, users could not distinguish browsers with and without *Chrome Zero* showing that *Chrome Zero* has no significant effect on the user experience.

Contributions. The contributions of this work are:

1. We propose *JavaScript Zero*, a fine-grained JavaScript permission system to mitigate state-of-the-art microarchitectural and side-channel attacks.
2. We show that combining advanced and novel features of the JavaScript language, e.g., virtual machine layering, closures, proxy objects, and object freezing, can be retrofitted to form a basis for strong security boundaries.
3. We show that *JavaScript Zero* successfully prevents all published microarchitectural and side-channel attacks and as a side effect also mitigates 50 % of the published JavaScript 0-day exploits since Chrome 49.
4. We evaluate our proof-of-concept implementation *Chrome Zero* in terms of performance and usability. *Chrome Zero* has 1.82 % performance overhead on average on the Alexa Top 10 websites. In a double-blind user study, we show that users cannot distinguish a browser with and without *Chrome Zero* for 24 of the Alexa Top 25 websites.

The remainder of the paper is organized as follows. Section 2 provides preliminary information necessary to understand the defenses we propose. Section 3 defines the threat model. Section 4 describes the design of *JavaScript Zero*. Section 5 details our proof-of-concept implementation *Chrome Zero*. Section 7 provides a security analysis of *Chrome Zero* as an instance of *JavaScript Zero*. Section 7 provides a usability analysis of *Chrome Zero*. Section 8 discusses related work. We conclude our work in Section 8.

Table 8.1: Requirements of state-of-the-art side-channel attacks in JavaScript.

	Memory addresses	Accurate timing	Multithreading	Shared data	Sensor API
Rowhammer.js [15]	●	●	○	○	○
Practical Memory Deduplication Attacks in Sandboxed Javascript [14]	◐	●	○	○	○
Fantastic Timers and Where to Find Them [40]	●	●†	◐	◐	○
ASLR on the Line [13]	●	●†	◐	◐	○
The spy in the sandbox [33]	◐	●	○	○	○
Loophole [47]	○	◐	●	○	○
Pixel perfect timing attacks with HTML5 [44]	○	●†	◐	◐	○
The clock is still ticking [45]	○	●	◐	○	○
Practical Keystroke Timing Attacks in Sandboxed JavaScript [20]	○	◐†	●	◐	○
TouchSignatures [23]	○	○	○	○	●
Stealing sensitive browser data with the W3C Ambient Light Sensor API [31]	○	○	○	○	●

† If accurate timing is not available, it can be approximated using a combination of multithreading and shared data.

2 Preliminaries

In this section, we provide preliminary information on microarchitectural attacks in native code and JavaScript, and on JavaScript exploits.

2.1 Microarchitectural Attacks

Modern processors are highly optimized for computational power and efficiency. However, optimizations often introduce side effects that can be exploited in so-called microarchitectural attacks. Microarchitectural attacks comprise side-channel and fault attacks on microarchitectural elements or utilizing microarchitectural elements, e.g., pipelines, caches, buses, DRAM. Attacks on caches have been investigated extensively in the past 20 years, with a focus on cryptographic implementations [4, 17]. The timing difference between a cache hit and a cache miss can be exploited to learn secret information from co-located processes and virtual machines. Modern attacks use either Flush+Reload [55], if read-only shared memory is available, or Prime+Probe [34] otherwise. In both attacks, the attacker manipulates the state of the cache and later on checks whether the state has changed. Besides attacks on cryptographic implementations [34, 55], these attack primitives can also be used to defeat ASLR [13] or to build covert-channels [22].

2.2 Microarchitectural and Side-Channel Attacks in JavaScript

Microarchitectural attacks were only recently exploited from JavaScript. As JavaScript code is sandboxed and inherently single-threaded, attackers face certain challenges in contrast to attacks in native code. We identified several requirements that are the basis for microarchitectural attacks, *i.e.*, every attack relies on at least one of these primitives. Moreover, sensors found on many mobile devices, as well as modern browsers introduce side-channels which can also be exploited from JavaScript. Table 8.1 gives an overview of, to the best of our knowledge, all 11 known microarchitectural and side-channel attacks in JavaScript and their requirements.

Memory Addresses JavaScript is a sandboxed scripting language which does not expose the concept of pointers to the programmer. Even though pointers are used internally, the language never discloses virtual addresses to the programmer. Thus, an attacker cannot use language features to gain knowledge of virtual addresses. The closest to virtual addresses are `ArrayBuffer`s, contiguous blocks of virtual memory. `ArrayBuffer`s are used in the same way as ordinary arrays but are faster and more memory efficient, as the underlying data is actually an array which cannot be resized [26]. If one virtual address within an `ArrayBuffer` is identified, the remaining addresses are also known, as both the addresses of the memory and the array indices are linear [13, 14].

Gras et al. [13] showed that `ArrayBuffer`s can be exploited to reconstruct virtual addresses. An attacker with knowledge of virtual addresses has effectively defeated address space layout randomization, thus circumventing an important countermeasure against memory corruption attacks.

Microarchitectural attacks typically do not rely on virtual addresses but physical addresses. Cache-based attacks [15, 33] rely on parts of the physical address to determine cache sets. DRAM-based attacks [14, 15, 40] also rely on parts of the physical address to determine the beginning of a page or a DRAM row. However, for security reasons, an unprivileged user does not have access to the virtual-to-physical mapping. This is not only true for JavaScript, but also for any native application running on a modern operating system.

Consequently, microarchitectural attacks have to resort to side-channel information to recover this information. Gruss et al. [14] and Gras et al. [13] exploit the fact that browser engines allocate `ArrayBuffer`s always

page aligned. The first byte of the `ArrayBuffer` is therefore at the beginning of a new physical page and has the least significant 12 bits set to '0'.

For DRAM-based attacks, this is not sufficient, as they require more bits of the physical address. These attacks exploit another feature of browser engines and operating systems. If a large chunk of memory is allocated, browser engines typically use `mmap` to allocate this memory, which is optimized to allocate 2 MB transparent huge pages (THP) instead of 4 KB pages [15, 40]. As these physical pages are mapped on demand, *i.e.*, as soon as the first access to the page occurs, iterating over the array indices results in page faults at the beginning of a new page. The time to resolve a page fault is significantly higher than a normal memory access. Thus, an attacker knows the index at which a new 2 MB page starts. At this array index, the underlying physical page has the 21 least significant bits set to '0'.

Accurate Timing Accurate timing is one of the most important primitives, inherent to nearly all microarchitectural and side-channel attacks. As most of the microarchitectural and side-channel attacks exploit some form of timing side channel, they require a way to measure timing differences. The required resolution depends greatly on the underlying side channel. For example, DRAM-row conflicts [15, 40], cache-timing differences [13, 33], and interrupt timings [20] require a timing primitive with a resolution in the range of nanoseconds, whereas for detecting page faults [14, 15, 40], exploiting SVG filters [44], or mounting cross-origin timing attacks [45], a resolution in the range of milliseconds is sufficient.

JavaScript provides two interfaces for measuring time. The `Date` object represents an instance in time, used to get an absolute timestamp. The object provides a method to get a timestamp with a resolution of 1 ms. The second interface is the `Performance` object which is used to provide information about page performance. This interface provides several timing relevant properties and functions, such as the `performance.now()` function, which provides a highly accurate timestamp in the order of microseconds [40]. Another part of the `Performance` object is the User Timing API, a benchmarking feature for developers, which also provides timestamps in the order of microseconds.

However, the resolution of these built-in timers is not high enough to measure microarchitectural side channels, where the timing differences are mostly in the order of nanoseconds. Thus, such attacks require a custom timing primitive. Usually, it is sufficient to measure timing differences, and

an absolute timestamp is not necessary. Thus, access to a full-blown clock is not required, and attackers usually settle for some form of a monotonically incremented counter as a clock replacement. Kohlbrenner et al. [18] and Schwarz et al. [40] investigated new methods to get highly accurate timing. Most of their timing primitives rely on either building counting loops using message passing [40] or on interfaces for multimedia content [18]. Using such self-built timers, it is possible to measure timing differences with a nanosecond resolution.

Multithreading JavaScript is inherently single-threaded and based on an event loop. All events, such as function calls or user inputs, are pushed to this queue and then serially, and thus synchronously, handled by the engine. HTML5 introduced multithreading to JavaScript, in the form of worker threads (web workers), allowing real parallelism for JavaScript code. With web workers, every worker has its own (synchronous) event queue. The synchronization is handled via messages, which are again events. Thus, JavaScript does not require explicit synchronization primitives.

The support for true parallelism allows to mount new side-channel attacks. Vila et al. [47] exploited web workers to spy on different browser windows by measuring the dispatch time of the event queue. This timing side-channel attack allows to detect user inputs and identify pages which are loaded in a different browser window. A similar attack using web workers was shown by Lipp et al. [20]. However, this attack does not exploit timing differences in the browser engine, but on the underlying microarchitecture. An endless loop running within a web worker detects CPU interrupts, which can then be used to deduce keystroke information.

Shared Data To synchronize and exchange data, web workers have to rely on message passing. Message passing has the advantage over unrestricted memory sharing as there is no requirement for synchronization primitives. Sending an object to a different worker transfers the ownership of the object as well. Thus, objects can never be changed by multiple workers in parallel.

As transferring the ownership of objects can be slow, JavaScript introduced `SharedArrayBuffer`s. A `SharedArrayBuffer` is a special object which behaves the same as a normal `ArrayBuffer`, but it can be simultaneously accessed by multiple workers. Inherently, this can reintroduce synchronization problems.

Schwarz et al. [40] and Gras et al. [13] showed that this shared data can be exploited to build timing primitives with a nanosecond resolution.

Their timing primitive requires only one worker running in an endless loop and incrementing the value in a `SharedArrayBuffer`. The main thread can simply use the value inside this shared buffer as a timestamp. Using this method, it is possible to get a timestamp resolution of 2 ns, which is almost as high as Intel's native timestamp counter, and thus sufficient to mount DRAM- and cache-based side-channel attacks.

Sensor API As JavaScript is also used on mobile devices, HTML5 introduced interfaces to interact with device sensors. Some sensors are already restricted by the existing permission system in the browser, such as the geolocation API. This permission system uses callback functions to deliver results. Hence, it is inherently incompatible with existing synchronous APIs and cannot be instrumented to protect arbitrary JavaScript functions. As these sensors can affect the user's privacy, the user has to explicitly permit usage of these interfaces on a per-page basis. However, several other sensors are not considered invasive in terms of security or privacy.

Mehrnezhad et al. [23] showed that access to the motion and orientation sensor can compromise security. By recording the data from these sensors, they were able to infer PINs and touch gestures (e.g., zoom) of the user. Although not implemented in JavaScript, Spreitzer [42] showed that access to the ambient light sensor (as specified by the W3C [48]) can also be exploited to infer user PINs. Similarly, Olejnik [31] utilized the Ambient Light Sensor API to recover information on the user's browsing history, to violate the same-origin policy, and to steal cross-origin data.

2.3 JavaScript Exploits

In addition to microarchitectural and side-channel attacks, there are also JavaScript-based attacks exploiting vulnerabilities in the JavaScript engine. An exploit triggers an implementation error in the engine to divert the control flow of native browser code. These implementation errors can—and should—be fixed by browser vendors. Side-channel attacks, however, often arise from the hardware design. In contrast to software, the hardware and hardware design cannot be easily changed.

As exploits are based on implementation errors and not design issues, we cannot identify general requirements for such attacks. Every JavaScript function and each interface can be potentially abused if there is a vulnerability in the engine. Thus, we cannot provide a general protection against exploits, and exploits are therefore not in the scope of this paper.

However, we can still reduce the attack surface of the browser, and we provide practical protection against 50 % of the published JavaScript 0-day exploits since Chrome 49.

Exploits often rely on arrays to craft their payload. Moreover, bugs are often triggered by errors in functions responsible for parsing complex data (e.g., JSON). As some of the functions used in exploits are also requirements for microarchitectural and side-channel attacks, we also evaluate exploits in this paper to confirm that our permission system is also applicable to reduce the general attack surface of the browser, *i.e.*, hardening browsers against 0-day exploits until they are fixed by the browser vendors.

3 Threat Model

In our threat model, we assume that the attacker is capable of performing state-of-the-art microarchitectural and software-based side-channel attacks in JavaScript. This is a reasonable assumption, as we found most published attacks to be accompanied with proof-of-concept source code allowing us to reproduce the attacks.

We assume that the victim actively uses a browser, either natively, or in a virtual machine. The attacker resides either in a different, co-located virtual machine [14, 40] or—for most attacks—somewhere else on the internet. In all state-of-the-art microarchitectural and software-based side-channel attacks, the attacker has some form of remote code execution. In line with these works, we assume that the attacker was able to maliciously place the attack code in a benign website. This can be achieved if the benign website either includes content from a (malicious) third party, such as advertisements or libraries, or if an attacker has compromised the benign site in some way. Another possibility is that the victim navigated to a malicious website controlled by the attacker. Hence, in all cases, the *attacker-controlled JavaScript code is executed in the victim's browser.*

The browser contains a JavaScript engine that executes code embedded in a website inside the browser sandbox. The sandbox ensures that JavaScript code cannot access any system resources not intended to be accessed. Furthermore, every page has its own execution context protected by the sandbox, *i.e.*, code on different pages cannot influence each other. We assume that an attacker is not aware of exploitable bugs in the JavaScript engine, and hence, can *only use legitimate JavaScript features.* Exploiting bugs in the interpreter, sandbox, or other execution environments, is out of scope for this paper.

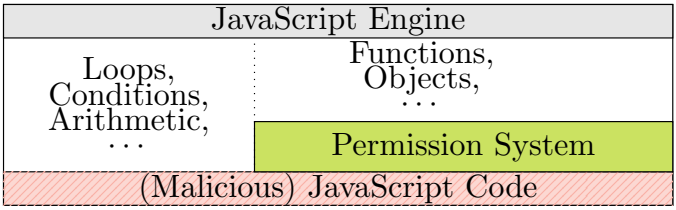


Figure 8.1: The permission system acts as an abstraction layer between the JavaScript engine and the interfaces provided to a JavaScript developer.

4 Design of *JavaScript Zero*

In this section, we present the design of our JavaScript permission system, *JavaScript Zero*. We propose a fine-grained policy-based system which allows to change the behavior of standard JavaScript interfaces and functions. Policies enforce certain restrictions to a website to protect users from malicious JavaScript. They allow to quickly adapt the permission system to protect against newly discovered attacks. Furthermore, different policies can be combined by the user, depending on the desired level of protection.

The idea of *JavaScript Zero* is to introduce an abstraction layer between the JavaScript engine and the interface provided to a (malicious) JavaScript developer. The basic idea of this layer is to protect functions, interfaces, and object properties, as shown in Fig. 8.1. The abstraction layer can block, modify, or simply forward every interaction of the code with the JavaScript engine. The layer is completely transparent to the web application and, thus, no modification of any existing source code is required to deploy *JavaScript Zero*. *JavaScript Zero* can intercept all calls to functions provided by the language, which also includes constructors of objects and getters of object properties. However, it does not interfere with the constructs of the language itself, *i.e.*, loops and primitive data types bypass the abstraction layer. In the remainder of this paper, we use the term "functions" to refer to general functions, object constructors, and getters of object properties for the sake of brevity.

The exact behavior of *JavaScript Zero* is defined by a *protection policy*. A protection policy is a machine-readable description which contains a policy for every function, property, or object that should be handled by the permission system. Listing 8.1 shows an excerpt of such a policy. In this sample policy, the function to go back to the last website is

```
function : {
  "window.performance.now":
    { action: "modify",
      return: "Math.floor(window.performance.now()
        / 1000.0) * 1000.0" },
  "history.back":
    { action: "block" },
  "navigator.getBattery":
    { action: "ask", default: "null" }
```

Listing 8.1: Excerpt of a protection policy. The function `performance.now` is modified to return timestamps with a lower resolution, the function `history.back` is blocked.

completely blocked, *i.e.*, if a script calls this function, it does nothing. Furthermore, the resolution of the high-resolution timer is reduced from several microseconds to one second. Finally, the battery API requires permission from the user, and if the user denies access to the function, it simply returns no information.

The policies can be designed by any user and shared with other users. Thus, as soon as a new exploit, side-channel attack, or microarchitectural attack emerges, a new policy preventing it can be created and shared with all users. We propose a community-maintained policy repository where users can subscribe to certain kinds of policies, e.g., more or less strict policies for their specific hardware and software. The functionality of *JavaScript Zero* does not fundamentally rely on the community, and every user can also write their own policies, or thoroughly inspect third-party policies before applying them. Hence, a careful user can avoid the inherent limitations of a community-maintained policy, e.g., adversarial modifications, which can happen in any open-source project.

For every policy, there are four different possibilities how it affects a function used on a website:

1. **Allow.** The function is explicitly allowed or not specified in the policy. In this case, no action is performed and the function can be used normally.
2. **Block.** The function is blocked. In this case, *JavaScript Zero* replaces the function by a stub that only returns a given default value.
3. **Modify.** The function is modified. In this case, *JavaScript Zero* replaces the original function with a policy-defined function. This function can still call the original function if necessary.

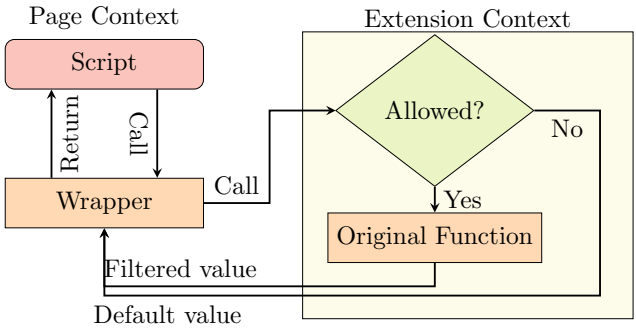


Figure 8.2: A policy replaces a function by a wrapper. The extension implements the logic to ask the user whether the function shall be blocked or the original function is executed.

4. **User permission.** The function requires the permission of the user. In this case, *JavaScript Zero* has to pause execution of the current function, display a notification to the user, and wait for the response of the user.

In the fourth case, the user has to explicitly grant permission. The user can opt to save the decision, to not be bothered again, and, thus, user interruptions are kept to a minimum.

We opted for a browser extension, as it can be easily installed in a user's browser and neither relies on modification of the source code of the website or the browser, nor any external service, e.g., a web proxy. Thus, there is no constant maintenance of a forked browser source base necessary. Moreover, by designing *JavaScript Zero* as a browser extension, it can easily be implemented for any browser supporting extensions (e.g., Chrome, Firefox, Edge) as the design of *JavaScript Zero* is independent of the browser.

Figure 8.2 shows the general design of this approach. Functions are replaced by wrapper functions which can either immediately return a result or divert the control flow to the browser extension. The browser extension can then ask the user whether to allow the function call or block it.

To allow regular users to use such a browser extension on a day-to-day basis, we propose a simple interface for handling protection policies. This interface defines so-called *protection levels*, each grouping one or more protection policies. Thus, a user only chooses a protection level out of a predefined set of levels, e.g., one of *none*, *low*, *medium*, *high*, *paranoid*. Although this simplification reduces the flexibility of the extension, we

chose this approach as for a regular user it is clearly not feasible to choose from protection policies or even define custom protection policies.

5 Implementation of *Chrome Zero*

In this section, we describe *Chrome Zero*, our open-source[1] proof-of-concept implementation of *JavaScript Zero* for Google Chrome 49 and newer. Implementing *Chrome Zero* faces certain challenges:

- C1 Restrictions must not be circumventable using self-modifying code, such as higher-order scripts.
- C2 Restricting access to potentially dangerous features must be irreversible for a website.
- C3 Restrictions must not have a significant impact on compatibility and user experience.

In addition to the aforementioned challenges, implementing *JavaScript Zero* as a browser extension results in a trade-off between compatibility with up-to-date browsers and functionality we can use, *i.e.*, we cannot change the browser and thus have to rely on functions provided by the extension API.

First, we describe in Section 5.1 how to retrofit virtual machine layering for security and extend it for objects using *proxy objects* [28]. Virtual machine layering was originally developed for low-overhead run-time monitoring of functions [19]. We use it to guarantee that a policy is always applied to a function (Challenge C1). In Section 5.2, we show that JavaScript closures in combination with object freezing can be utilized to secure the virtual machine layering approach to guarantee irreversibility of policies (Challenge C2). This combination of virtual machine layering and closures provides strong security boundaries for *Chrome Zero*. Finally, we discuss in Section 5.3 how to maintain practical usability of *Chrome Zero* (Challenge C3), despite the restrictions it introduces.

5.1 Virtual machine layering

To ensure that our own function is called instead of the original function, without modifying the browser, we facilitate a technique known as virtual machine layering [19]. Although this technique was originally developed for low-overhead run-time monitoring, we show that in combination with JavaScript closures, it can also be applied as a security mechanism. For

[1] *Chrome Zero*: `https://github.com/IAIK/ChromeZero`

```
var original_reference = window.performance.now;
window.performance.now = function() { return 0; };
// call the new function (via function name)
alert(window.performance.now()); // == alert(0)
// call the original function (only via reference)
alert(original_reference.call(window.performance))
    ;
```

Listing 8.2: Virtual machine layering applied to the function `performance.now`. The function name points to the new function, the original function can only be called using the reference.

security, virtual machine layering has a huge advantage over state-of-the-art source rewriting techniques [36, 38, 56], where functions are replaced directly in the source code. Ensuring that source rewriting cannot be circumvented is a hard problem, as function calls can be generated dynamically and are thus not always visible in the source code [1]. Support for such higher-order scripts is strictly necessary for full protection, as failing to apply a policy to only one function breaks the security guarantees of the security policies. However, higher-order scripts are often out-of-scope or not fully supported [24]. In contrast, virtual machine layering ensures that functions are replaced at the lowest level, right before they are executed by the JavaScript engine.

Listing 8.2 shows an example of virtual machine layering. As JavaScript allows to dynamically extend and modify prototypes, existing functions can be changed, and new functions can be added to every object. Virtual machine layering takes advantage of this language property. *Chrome Zero* saves a reference to the original function of an object and replaces the original function with a wrapper function. The original function can only be called using the saved reference. Calling the function by using the function name will automatically call the wrapper function. As *Chrome Zero* has full access to the website, it can use virtual machine layering to replace any original function. We can ensure that the code of *Chrome Zero* is executed before the page is rendered, and thus, that no other code can save a reference to the original function.

Additionally, virtual machine layering covers higher-order scripts without any additional costs. Higher-order scripts are scripts which are dynamically created (or loaded) and executed by existing scripts. There are multiple ways of creating higher-order scripts, including `eval`, script injection, function constructors, event handlers, and `setTimeout`. As any

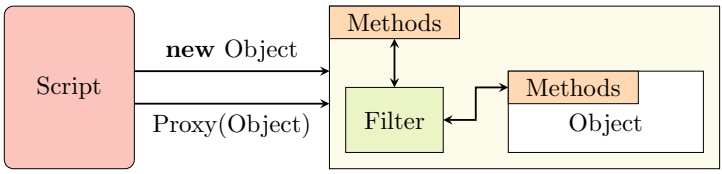


Figure 8.3: Dangerous objects are wrapped in proxy objects. The proxy object decides whether methods of the original object are called or substituted by a different functions.

higher-order script automatically uses the re-defined function without further changes, policies are automatically applied to higher-order scripts as well, and there is no possibility for obfuscated code to circumvent the function replacement.

As *JavaScript Zero* supports policies not only for functions but also for properties and objects, we have to extend virtual machine layering, which was originally only intended for functions.

Properties

Typically, a property of a prototype is not a function, but a simple data value. JavaScript allows to replace all properties by special functions called *accessor properties*. Accessor properties are a special kind of property, where every access to the property triggers a user-defined function. In case the property was already an accessor property, *Chrome Zero* simply replaces the original function. Thus, regardless of the type of property, we can convert every property to an accessor property using `Object.defineProperty`.

Objects

To be able to apply policies to objects, we wrap the original object within a *proxy object* as shown in Fig. 8.3. The proxy object contains the original object and forwards all functions (which are not overwritten) to the original object. Thus, all states are still handled by the original object, and only functions for which a policy is defined have to be re-implemented in the proxy object.

Although JavaScript prototypes have a constructor, simply applying virtual machine layering to the constructor function is not sufficient. The constructor function is only a pointer to the real constructor and not used to actually instantiate the object. The alternative to the proxy

```
(function() {
// original is only accessible in this scope
var original = window.performance.now;
window.performance.now = function() {
  return Math.floor((original.call(window.
    performance))
    / 1000.0) * 1000.0;
}; })();
```

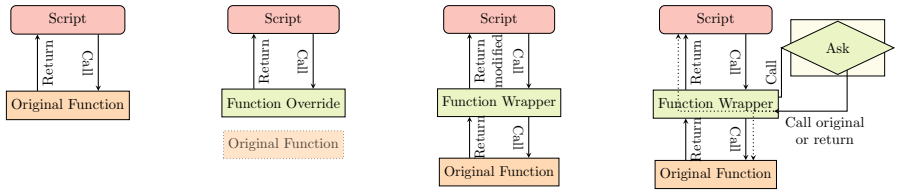
Listing 8.3: Virtual machine layering applied to the function `performance.now` within a closure. The function name points to the new function, the original function can only be called using the reference. However, the reference is not visible outside the scope, *i.e.*, only the wrapper function can access the reference to the original function.

object for replacing the constructor is to re-implement the entire object with all methods and properties. However, as this requires considerable engineering effort and cannot easily be automated, it is not feasible, and we thus rely on the proxy object.

5.2 Self-protection

An important part of the implementation is that it is not possible for an attacker to circumvent the applied policies (Challenge C2). Thus, an implementation has to ensure that an attacker cannot get hold of any reference to the original functions. We utilize JavaScript closures for security, by creating anonymous scopes not connected to any object and thus inaccessible to code outside of the closure. Listing 8.3 shows virtual machine layering wrapped in a closure.

With the original version of virtual machine layering as shown in Listing 8.2, an attacker could simply guess the name of the variable holding the original reference. Furthermore, all global variables are members of the JavaScript root object `window`, and an attacker could use reflection to iterate over all variables until the function reference is discovered. Closures provide a way to store data in a scope not connected to the `window` object. Thus, by applying the virtual machine layering process within a closure, the reference to the original function is still available to the wrapper function but inaccessible to any code outside of the closure. This guarantees that the virtual machine layering is irreversible.



(a) Normal function

(b) Blocked function

(c) Modified function

(d) Function with user permission

Figure 8.4: (a) A normal, unmodified function call as reference. (b) If a function is blocked, it can be immediately replaced with a function returning the default value. (c) If the return value has to be modified, the function can be replaced by an anonymous JavaScript closure which applies the modification directly on the page. (d) Only if the user has to be asked for permission, a switch into the extension context is necessary.

At the time of writing, there is no mechanism to modify a function without redefining it. Thus, an attacker cannot inject new code into the wrapper function (or modify the existing code) without destroying the closure and therefore losing the reference to the original function.

Additionally, objects have to be protected using `Object.freeze` after the virtual machine layering process. This ensures that deleting the function does not revert to the original function pointer, as it is otherwise the case in Google Chrome.

If a policy requires user interaction, *i.e.*, the user has to decide whether a function shall be executed or not, this logic must also be protected against an attacker. By relying on a browser extension, we already have the advantage of a different execution context and thus a different security context. A website cannot access data inside an extension or influence code running inside an extension. This also protects the policies, which are stored within the extension. Therefore, there is no possibility for a malicious website to modify or inject new policies.

5.3 User Interface

Challenge C3 is to have no significant impact on compatibility and user experience. This implies that *Chrome Zero* must not have a perceivable performance impact (cf. Section 7).

As diverting the control flow into the extension (cf. Figure 8.2) is relatively costly, we only do that if absolutely necessary. Figure 8.4 shows

how *Chrome Zero* only diverts the control flow to the extension if a policy requires that the user is asked for permission. In all other cases, we can directly replace the function with a stub or wrapper function (Figure 8.4b and Figure 8.4c) before loading a page.

As JavaScript does not provide a mechanism to block scripts, except for the built-in pop-up boxes (e.g., `alert`), pausing a function to ask the user for permission requires interaction with the browser extension. *Chrome Zero* relies on the Google Chrome Debugger API [12] which extends the functionality of JavaScript to influence and inspect the internal state of the JavaScript engine. Using Chrome's remote debugging protocol [11], *Chrome Zero* registers a function which is called whenever a script uses the `debugger` keyword. The effect of the `debugger` keyword is that the JavaScript engine pauses all currently executing scripts before calling the registered function [27].

While the script—and thus the entire page—is paused, *Chrome Zero* asks the user for permission to execute the current function. The result is then returned to the calling function by writing it to a local variable within the closure, and function execution is resumed using the Debugger API. Note that only *Chrome Zero* can access the local variable that stores the result, as all variables within the closure are inaccessible to the remaining page (cf. Section 5.2). The function then either resumes execution of the function, or returns a default value in case the user does not give permission to execute the function. Spurious usage of the `debugger` keyword on a (malicious) website has no effect, as *Chrome Zero* just continues if no policy is found for the current function.

Chrome Zero does not instrument the existing browser permission system, as it cannot be retrofitted to protect arbitrary functions and objects. The existing browser permission system only works for APIs designed to be used with the permission system, *i.e.*, the API has to be asynchronous by relying on callback functions or promises. The browser asks the user for permission, and if the user accepts, the browser calls a callback function with the result, e.g., the current geolocation. Hence, synchronous APIs, e.g., the result of the function call is provided as a return value, cannot be protected with the browser's asynchronous permission system. For the protection to be complete, we have to handle both synchronous as well as asynchronous function calls, and can therefore not rely on the browser's internal permission system.

Table 8.2: All discussed policies (except for sensors) and their effect on attacks.

Policy \ Prevents	Rowhammer.js [15]	Page Deduplication [14]	DRAM Covert Channel [40]	Anti-ASLR [13]	Cache Eviction [13, 15, 33, 40]	Keystroke Timing [20, 47]	Browser [44, 45, 47]	Exploits (cf. Section 6.2)
Buffer ASLR	○	◐	○	●	●	○	○	◐
Array preloading	●	○	●	○	○	○	○	○
Non-deterministic array	●	◐	◐	●	●	○	○	○
Array index randomization	○	●	○	●	○	○	○	◐
Low-resolution timestamp	○	◐	○	○	○	◐	◐	○
Fuzzy time	○	◐*	○	○*	○	●*	●*	○
`WebWorker` polyfill	○	○	●	●	●	●	○	○
Message delay	○	○	○	○	○	◐	◐	○
Slow `SharedArrayBuffer`	○	○	●	◐	●	○	○	○
No `SharedArrayBuffer`	○	○*	●	●*	●	○*	○*	◐
Summary	●	●	●	●	●	●	●	◐

Symbols indicate whether a policy fully prevents an attack, (●), partly prevents and attack by making it more difficult (◐), or does not prevent an attack (○).

A star (*) indicates that all policies marked with a star must be combined to prevent an attack.

6 Security Evaluation

In this section, we evaluate *JavaScript Zero* by means of our proof-of-concept Chrome extension, *Chrome Zero*. In the first part of the evaluation, we show how *Chrome Zero* prevents all microarchitectural and side-channel attacks that can be mounted from JavaScript (cf. Table 8.1). Furthermore, we show how policies can prevent exploits. We evaluate how many exploits are automatically prevented by protecting users against microarchitectural and side-channel attacks.

6.1 Microarchitectural and Side-Channel Attacks

To successfully prevent microarchitectural and side-channel attacks, we have to eliminate the requirements identified in Section 2.4. Depending on the requirements we eliminate, microarchitectural and side-channel

Table 8.3: A table of how policies correspond to the protection levels of *Chrome Zero*.

Requirement \ Protection Level	Off	Low	Medium	High	Paranoid
Memory addresses	-	Buffer ASLR	Array preloading	Non-deterministic array	Array index randomization
Accurate Timing	-	Ask	Low-resolution timestamp	Fuzzy time	Disable
Multithreading	-	-	Message delay	`WebWorker` polyfill	Disable
Shared data	-	-	Slow `SharedArrayBuffer`	Disable	Disable
Sensor API	-	-	Ask	Fixed value	Disable

attacks are not possible anymore (cf. Table 8.1). Consequently, we discuss policies to eliminate each requirement. Table 8.2 shows a summary of all policies and how they affect state-of-the-art attacks. Table 8.3 shows which policy is active on which protection level.

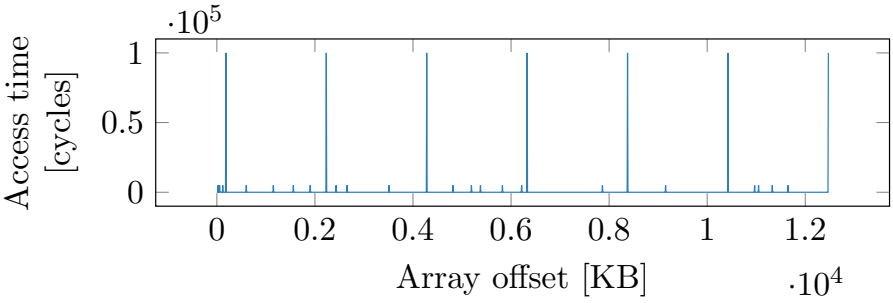
Memory Addresses

In all known attacks, array buffers are used to retrieve information on the underlying memory address. An attacker can exploit that array buffers are page-aligned to learn the least significant 12 bits of both the virtual and physical address [33]. Thus, we have to ensure that array buffers are not page-aligned and that an attacker does not know the offset of the array buffer within the page.

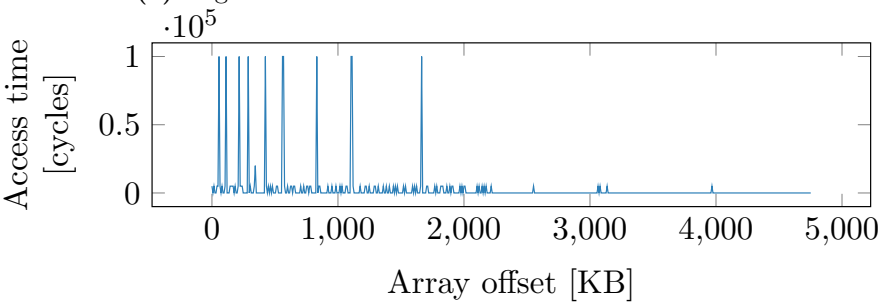
Array buffers are raw binary data buffers, storing values of arrays. However, their content cannot be accessed directly, but only using typed arrays or `DataView`s. Thus, we have to proxy both the `DataView` object as well as all typed arrays (e.g., `Uint8Array`, `Uint16Array`, etc.).

Buffer ASLR To prevent the arrays from being page-aligned, we introduce buffer ASLR, which randomizes the start of the array buffer. We overwrite the length argument of the constructor to allocate additional 4 KB. This allows us to move the start of the array anywhere on the page by generating a random offset in the range $[0; 4096)$. This offset is then added to the array index for every access. Hence, all data is shifted, *i.e.*, the value at index 0 is not page-aligned but starts at a random position within the page. Thus, an attacker cannot rely on the property anymore that the 12 least significant address bits of the first array buffer index are '0'.

Preloading However, the protection given by buffer ASLR is not complete, as an attacker can still iterate over a large array to detect page borders using page faults [15, 40]. With 21 bits of the virtual and physi-



(a) Page border detection without random accesses.

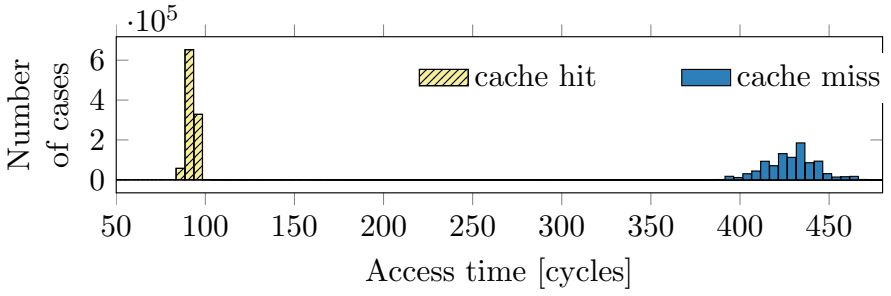


(b) Page border detection with random accesses.

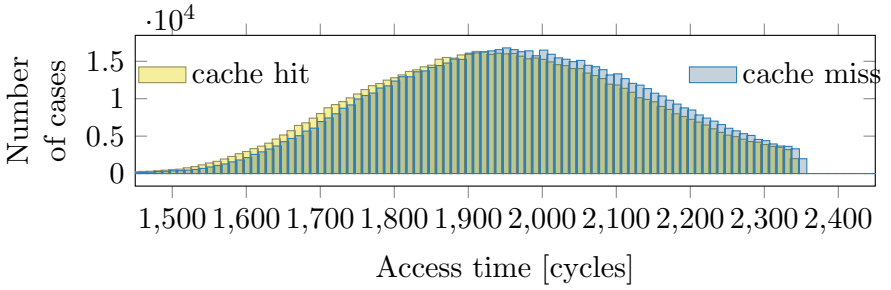
Figure 8.5: Page border detection without and with *Chrome Zero*. When iterating over an array, page faults cause a higher timing than normal accesses, visible as timing peaks.

cal address, a THP page border contains even more information for an attacker. One simple prevention for this attack is to iterate through the array after constructing it. Accessing all underlying pages triggers a page fault for every page, and an attacker subsequently cannot learn anything from iterating over the array, as the memory is already mapped. Thus, Rowhammer.js [15] and the DRAM covert channel [40] are prevented by *Chrome Zero*.

Non-determinism Instead of preloading, (*i.e.*, iterating over the array after construction), we can modify the setter of the array to add a memory access to a random array index for every access to the array. This has two advantages in terms of security. First, with only preloading, an attacker could wait for the pages to be swapped out, or deduplicated, enabling page border detection again. The random accesses prevent the page border



(a) Prime+Probe results without random accesses.



(b) Prime+Probe results with random accesses.

Figure 8.6: When adding random accesses, the timings for cache hits and misses blend together, making it impractical to decide whether an access was a cache hit or a miss. In contrast to a benign use case, the access time is significantly increased as the adversary is priming (*i.e.*, thrashing) the cache and any memory access is likely a cache miss.

detection, as an attacker cannot know whether the page fault was due to the regular access or due to a random access. As shown in Figure 8.5, with the random accesses, the probability to trigger a page fault for the first accesses is relatively high, as pages are not mapped in the beginning. This probability decreases until all pages are mapped. Thus, an attacker cannot reliably detect the actual border of a page but only the number of pages. Second, this prevents the generation of eviction sets [13, 15, 33, 40]. A successful eviction of a cache set requires an attacker to measure the access time of a special memory access pattern [15]. Adding random accesses in between prevents an attacker from obtaining accurate measurements, as the random accesses influence the overall timing (cf. Figure 8.6). Note that the access time is significantly increased as the adversary is priming (*i.e.*,

thrashing) the cache and thus, any additional memory access is likely a cache miss. Hence, this does not relate to any benign use case performance or access time.

Array Index Randomization One attack that cannot be thwarted with the discussed policies is the page-deduplication attack [14]. In this attack, an attacker only has to be in control over the content of one page to deduce if a page with the same content already exists in memory. Allocating a large array still gives an attacker control over at least one page. To prevent a page deduplication attack from being successful, we have to ensure that an attacker cannot deterministically choose the content of an entire page. One possible implementation is to make the mapping between the array index and the underlying memory unpredictable by using a random linear function.

We overwrite the array to access memory location $f(x)$ when accessing index x with $f(x) = ax+b \mod n$ where a and b are randomly chosen and n is the size of the `ArrayBuffer`. Furthermore, a and n must be co-prime to generate a unique mapping from indices to memory locations. To find a suitable a, we can simply choose a random a in the array constructor and test whether $\gcd(a, n) = 1$. As the probability of two randomly chosen numbers to be co-prime is $\frac{1}{\zeta(2)} = \frac{6}{\pi^2} \approx 61\%$ (for $n \to \infty$) [30], this approach is computationally efficient. That is, the expected value is 1.64 random choices to find two co-prime numbers (for $n \to \infty$).

As a and b are inaccessible to an attacker, the mapping ensures that an attacker cannot predict how data is stored in physical memory. An attacker can also not reverse a and b because there is no way to tell whether a guess was correct, as buffer ASLR and preloading prevent any reliable conclusions from page faults. Thus, an attacker cannot control the content of a page, thwarting page deduplication attacks [14].

Accurate Timing

Many attacks within the browser require highly accurate time measurements. Especially microarchitectural attacks require time measurements with a resolution in the range of nanoseconds [13, 15, 33, 40]. Such high-resolution timestamps were readily available in browsers [33] but have been replaced by lower resolution timestamps in all modern browsers in 2015 [2, 6, 9, 25]. However, Schwarz et al. [40] showed that it is still possible to get a nanosecond resolution from these timestamps by exploiting the underlying high-resolution clock.

```
(function() {
var wpn = window.performance.now, last = 0;
window.performance.now = function() {
var fuzz = Math.floor(Math.random() * 1000), //1ms
    now = Math.floor(wpn.call(window.performance)
    *1000);
var t = now - now % fuzz;
if(t > last) last = t;
return last / 1000.0;
};})();
```

Listing 8.4: Fuzzy time [46] applied to the high-resolution timing API with a 1 ms randomization.

Low-resolution timestamps As a policy, we can simply round the result of the high-resolution timestamp (`window.performance.now`) to a multiple of 100 ms. This is exactly the same behavior as implemented in the Tor browser. Thus, we achieve the same protection as the Tor browser, where the recoverable resolution is only 15 µs, rendering it useless for many attacks [40].

Fuzzy time A different approach is to apply fuzzy time to timers [46]. In addition to rounding the timestamp, fuzzy time adds random noise to the timestamp to prevent exact timing measurements but still guarantees that the timestamps are monotonically increasing. Kohlbrenner et al. [18] implemented this concept in Fuzzyfox, a Firefox fork. We can achieve the same results using a simple policy that implements a variant of the algorithm proposed by Vattikonda et al. [46] shown in Listing 8.4, without requiring constant maintenance of a browser fork.

We evaluated our policies for low-resolution timestamps and fuzzy time by creating two functions with a runtime below the resolution of the protected high-resolution timer. Using edge thresholding [40], we tried to distinguish the two functions based on their runtime. For the evaluation, we rounded timestamps to a multiple of 1 ms and used a 1 ms randomization interval for the fuzzy time. The two functions f_{slow} and f_{fast}, which we distinguish, have an execution time difference of 300 µs. Figure 8.7 shows the results of this evaluation. If no policy is applied to the high-resolution timer, the functions can always be distinguished based on their runtime. With the low-resolution timestamp and edge thresholding, the functions are correctly distinguished in 97 % of the cases, as the underlying clock still has a resolution in the range of nanoseconds.

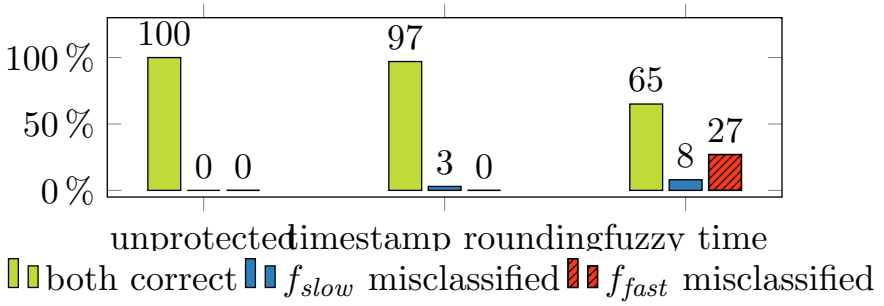


Figure 8.7: Edge thresholding to distinguish whether the function f_{slow} takes longer than f_{fast}. The difference between the execution times is less then the provided resolution.

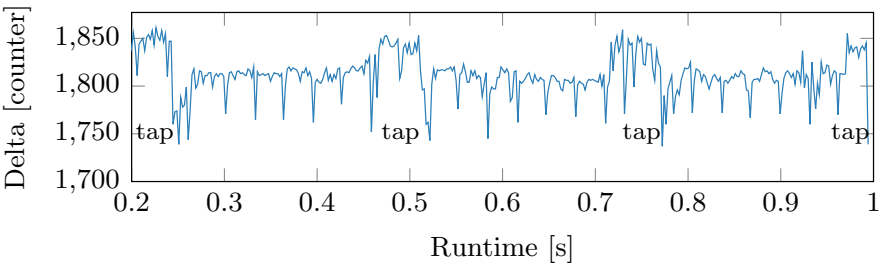
When fuzzy time is enabled, the functions are correctly distinguished in only 65 % of the cases, and worse, in 27 % of the cases the functions are wrongly classified, *i.e.*, the faster-executing function is classified as the slower function.

Figure 8.8 shows the result of fuzzy time on the JavaScript keystroke detection attack by Lipp et al. [20]. Without fuzzy time, it can be clearly seen whenever the user taps on the touch screen of a mobile device (Figure 8.8a). By enabling the fuzzy time policy, the attack is fully prevented, and no taps can be seen in the trace anymore (Figure 8.8b).

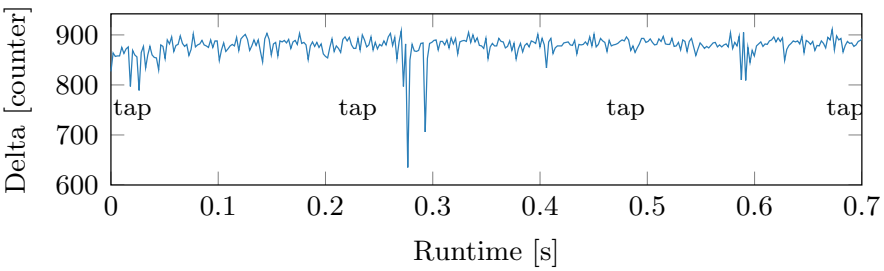
Multithreading

As the resolution of the built-in high-resolution timer has been reduced by all major browsers [2, 6, 9, 25], alternative timing primitives have been found [13, 18, 40]. Although several new timing primitives work without multithreading [18, 40], only the timers using multithreading achieve a resolution that is high enough to mount microarchitectural attacks [13, 20, 40, 47].

`WebWorker` polyfill A drastic—but effective—policy is to prevent real parallelism. To achieve this, we can completely replace `WebWorkers` by a polyfill intended for unsupported browsers. The polyfill [29] simulates `WebWorkers` on the main thread, trying to achieve similar functionality without support for real parallelism. Thus, all attacks relying on real parallelism [13, 20, 40, 47] do not work anymore.



(a) Without *Chrome Zero*.



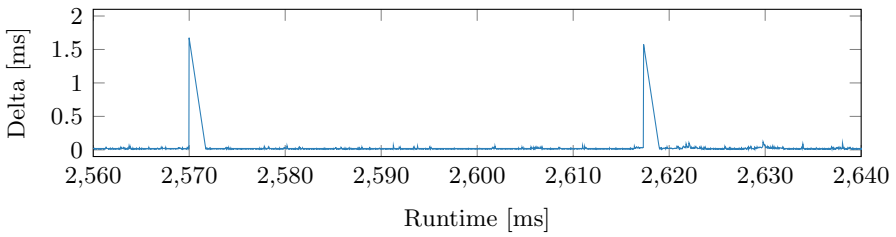
(b) With *Chrome Zero*.

Figure 8.8: Without *Chrome Zero*, taps can be clearly seen in the attack by Lipp et al. [20] (Figure 8.8a). With *Chrome Zero*, the attack is prevented and no taps are visible (Figure 8.8b).

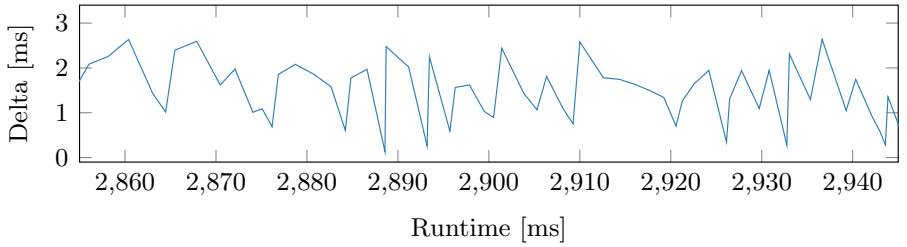
Message delay A different policy to specifically prevent certain timing primitives [40] and attacks on the browser's rendering queue [47] is to delay the `postMessage` function. If the `postMessage` function randomly delays messages (similar to Fuzzyfox [18]), the attack presented by Vila et al. [47] does not work anymore, as shown in Figure 8.9.

Shared Data

`SharedArrayBuffer` is the only data type which can be shared across multiple workers in JavaScript. This shared array can then be abused to build a high-resolution timer. One worker periodically increments the value stored in the shared array while the main thread uses the value as a timestamp. This technique is the most accurate timing primitive at the time of writing, creating a timer with a resolution in the range of nanoseconds [13, 40].



(a) Without *Chrome Zero*.



(b) With *Chrome Zero*.

Figure 8.9: Running the attack by Vila et al. [47] shows keystrokes among other system and browser activity (Figure 8.9a). With *Chrome Zero* in place, the `postMessage` timings are delayed and thus keystrokes cannot be detected anymore (Figure 8.9b).

No `SharedArrayBuffer` At the time of writing, the `SharedArrayBuffer` is by default deactivated in modern browsers. Thus, websites should not rely on this functionality anyway, and a policy can simply keep the `SharedArrayBuffer` disabled if vendors enable it.

Slow `SharedArrayBuffer` On our Intel i5-6200U test machine, we achieve a resolution of 0.77 ± 0.01 ns with the `SharedArrayBuffer` timing primitive. This allows use to clearly distinguish cache hits from cache misses (Figure 8.10a), and thus mount the attacks proposed by Schwarz et al. [40] as well as Gras et al. [13]. To protect users from these attacks, our policy randomly delays the access to the `SharedArrayBuffer`. Using this policy, we reduce the resolution to 4215.25 ± 69.39 ns, which is in the same range as the resolution of the native `performance.now()` timer. Thus, these microarchitectural attacks, which require a high-resolution timer, do not work anymore as shown in Figure 8.10b.

Sensor API

As mobile devices are equipped with many sensors, JavaScript provides several APIs allowing websites to access sensor data, e.g., accelerometer, ambient light, or battery status. While some of those interfaces allow developers to build more functional and user-friendly applications, they also facilitate leakage of sensitive information. While modern browsers explicitly ask the user for permission if the running websites want to access the user's geolocation, access to other APIs is silently permitted.

Battery Status API After Olejnik et al. [32] showed the potential privacy risk of the HTML5 Battery Status API as a tracking identifier, Firefox disabled the interface with version 52 [8]. However, Chrome still offers unrestricted access to this API without asking for permission. Thus, we introduce a policy allowing to either randomize the properties of the battery interface, to set them to fixed values, or to disable the interface entirely. With this policy, the Battery Status API can not be used as a tracking identifier anymore.

Ambient Light Sensor The ambient light sensor can be used to infer user PINs [42] or to recover browsing history information [31]. While the API needs to be enabled manually in the Chrome browser, it is enabled by default in the Firefox browser. By introducing a policy that either returns constant values for the ambient light sensor, or disables the interface, an attacker is unable to perform these attacks.

Device Motion, Orientation, and Acceleration The motion sensor data and orientation sensor data of mobile web browsers can be exploited to infer user PIN input [23]. Both are available to websites without any permissions. In order to mitigate such attacks, we introduce a policy that allows to spoof the sensor data or to prohibit access entirely.

6.2 Exploits

Although exploits are out-of-scope for the permission system (cf. Section 2.3), we investigate the (side-)effect of our policies on JavaScript exploits. For this, we investigate CVEs that are exploitable via JavaScript and were discovered since Chrome 49, as *Chrome Zero* requires Chrome 49 and later.

To evaluate whether *Chrome Zero* protects a user from a specific exploit, we first reproduce the exploit without *Chrome Zero* and then activate

Chrome Zero to check whether the exploit still works. We reproduced all 12 CVEs[1] for the Chrome JavaScript engine, which were discovered since 2016 for Chrome 49 or later and for which we could find proof-of-concept implementations online. All of the 12 CVEs lead to either a crash of the current tab or to information leakage.

With *Chrome Zero* in place, half of them are prevented, leaving only 6 CVEs that are still exploitable. The prevented CVEs all rely on at least one object which we modify (e.g., `ArrayBuffer`) and thus do not work with the modified object. Furthermore, we expect that actual remote code execution using the working exploits gets more complicated if policies such as array index randomization or buffer ASLR are in place. Thus, *Chrome Zero* provides additional protection against 0-days without requiring explicit policies. Creating policies to specifically target CVEs is left to future research.

7 Usability Evaluation

In this section, we analyze the usability impact of *Chrome Zero* by performing a performance analysis and a double-blind user study. We first analyze how many websites use functionality which is also used in microarchitectural and side-channel attacks. We then analyze the performance impact on the Alexa Top 10 websites. Finally, we show whether the protection mechanism has any impact on the browsing experience, *i.e.*, whether there are pages that do not work as expected anymore, for the Alexa Top 25 websites.

7.1 Performance

We evaluated the performance overhead of *Chrome Zero* in both micro and macro benchmarks.

First, we evaluated the impact of *Chrome Zero* on the loading time of a page. We measured a page loading latency between 10.64 ms if no policy is active, and 89.08 ms if policies protecting against all microarchitectural and side-channel attacks are active. As on every page load the current policies are loaded and injected into the current tab, the latency grows linearly with the number of policies, and delays the actual rendering of the page. On average, we measured a latency of approximately 3.4 ms per

[1] CVE-2016-1646, 1653, 1665, 1669, 1677, 5129, 5172, 5198, 5200, 9651, 2017-5030, 5053

Table 8.4: Results of the JSZJetStream benchmark.

	Without *Chrome Zero*	With *Chrome Zero*
Latency	71.46 ± 4.43	71.33 ± 2.43
Throughput	220.45 ± 6.80	214.71 ± 3.50
Total	134.90 ± 5.96	132.81 ± 2.92

The higher the score, the better the performance.

active policy, *i.e.*, every policy delays the loading of a newly opened page by 3.4 ms.

Second, we investigated the overhead for Chrome's JavaScript engine by using the internal profiler. Figure 8.11 shows the overhead for the Alexa Top 10 websites. The runtime increase of the JavaScript engine had a median of 1.82 %, which corresponds to only 16 ms.

Finally, we used the JSZJetStream [3] browser benchmark, which is developed as part of the WebKit engine. We measure a performance overhead of 1.54 % when using *Chrome Zero*. Table 8.4 shows the detailed scores of the benchmark.

A reason for the low overhead of *Chrome Zero* is the JavaScript Just-In-Time (JIT) compiler. Chrome's JIT consists of several compilers, producing code on different optimization levels [39]. As the code is continuously optimized by the JIT, our injected functions are compiled to highly efficient native code, with a negligible performance difference compared to the original native functions. The results of our benchmarks show that *Chrome Zero* does not have a visible impact on the user experience for an everyday usage.

7.2 Compatibility

For *Chrome Zero* to be usable on a day-to-day basis, it is important that the majority of websites is still usable if policies are active. We analyzed the Alexa Top 100 websites with a protection level of *high*, the second highest protection level (cf. Table 8.3). Out of the 100 pages, all of them used JavaScript, and 57 relied on functions for which the protection level *high* defines policies. For all these pages, we verified that *Chrome Zero* did not cause any error when testing some of the site's basic functionality. For a thorough evaluation, we conducted a double-blind user study to test whether *Chrome Zero* has an impact on the browsing experience. We designed the study to have 24 participants to have a

maximum standard error below 15 % at a confidence level of 85 %. The 24 participants, which we recruited by advertising it through word-of-mouth, had different backgrounds, ranging from students without any IT background to information-security post-doctoral researchers.

Method

We explained every participant that we developed a browser extension which provides additional protection against attacks. We showed two instances of Google Chrome to the participant, one without *Chrome Zero* (A) and one with *Chrome Zero* set to protection level *high* (B) for every website in the Alexa Top 25. For every website, a fully automated script randomly chose whether browser instance A or B had *Chrome Zero* activated, without any interaction of the study conductor or the study participant. Hence, neither the study participant nor the study conductor knew which of the two browsers had *Chrome Zero* activated, making the study double blind. After 1 minute, the script asked the user whether there was any noticeable difference between the two pages, and if so, whether browser A or browser B had *Chrome Zero* enabled. The results of these questions were saved in a file and automatically evaluated after the user tested all 25 pages. Every correct user answer was counted as a 100 % correct answer, if the user did not notice any difference, we counted it as a 50% probability to make the correct guess, and if the user answered incorrectly, we counted a 0 % correct answer.

Results

Figure 8.12 shows the results of our user study. The overall probability to correctly detect the presence of *Chrome Zero* was 50.2 %. The maximum average success rate of a participant was 60 %; the minimum was 40 %.

The maximum detection rate for a website was 62.5 % for yahoo.com (standard error ±0.05). For all other websites, the detection rate was not better than random guessing. The minimum detection rate for a website was 41.7 % for amazon.com and sina.com.cn (standard error ±0.05).

For the 6 websites where no *Chrome Zero* policy was active, users guessed correctly in 48.3 % on average (standard error ±0.049), *i.e.*, a deviation of 1.7*pp* to random guessing. For the 19 websites where at least one *Chrome Zero* policy was active, users guessed correctly in 50.8 % on average (standard error ±0.055), *i.e.*, a deviation of 0.8*pp* to random guessing. This highlights how negligible the differences in the user experience are.

While their classification of the instance was many times incorrect, participants stated loading time, cookie-policy dialogues and website redirections as the reason for selecting the instance as the one using *Chrome Zero*.

Although our implementation is only a proof-of-concept implementation, the results of the study confirm that *JavaScript Zero* is practical, and our implementation of *Chrome Zero* is usable on a day-to-day basis.

8 Related Work

In this section, we discuss related work on protecting users from the execution of potential harmful JavaScript code. While there are several proposed solutions, *JavaScript Zero* is the only technique fully implemented as a browser extension only (*Chrome Zero*) without negatively affecting the browsing experience. *Chrome Zero* does not rely on any changes to existing source code or the system's environment and thus does not require support by developers or browser vendors.

Browser extensions Browser extensions such as NoScript [10] or uBlock [37] allow users to define policies to permit or prohibit the execution of JavaScript depending on their origin, *i.e.*, a page can either completely block JavaScript or execute it without any restrictions. In contrast, *Chrome Zero* offers a more fine-grained permission model that operates on function level and does not interfere with dynamic website content. Furthermore, *JavaScript Zero* directly targets attack prevention, whereas existing browser extensions aim primarily at blocking advertisements and third-party tracking code.

In concurrent work, Snyder et al. [41] proposed a browser extension to protect against exploits in general, based on the same generic idea as *JavaScript Zero*. They first exhaustively evaluate the usage statistics of JavaScript APIs and their connection to CVEs and then, similar to our approach, selectively block the corresponding JavaScript APIs. Based on this approach they block 52% of all CVEs, while only impacting the usability of 4% to 7% of the tested websites. When a usability impact on 16% of the tested websites is still acceptable, they can even block 72% of all CVEs. This is a significantly lower usability impact than previous approaches like NoScript [10] or uBlock [37]. With our focus on mitigating microarchitectural and side-channel attacks, we complement the work by Snyder et al. [41] and show that the underlying generic idea is not only

applicable to CVEs or side-channel attacks, but to both types of attacks. This highlights the strength of the underlying idea of both papers.

Modified browsers Meyerovich et al. [24] modified the JavaScript engine of Internet Explorer 8 to enforce fine-grained application-specific runtime security policies by the website developer. In contrast, *JavaScript Zero* is implemented as a browser extension and does not rely on any developer to define security policies. Patil et al. [35] analyzed the access control requirements in modern web browsers and proposed JCShadow, a fine-grained access control mechanism in Firefox. JCShadow splits the running JavaScript into groups with an assigned isolated copy of the JavaScript context. A security policy then defines which code is allowed to access objects in other shadow contexts to separate untrusted third-party JavaScript code from the website. Stefan et al. [43] proposed COWL, a label-based mandatory access control system to sandbox third-party scripts. Bichhawat et al. [5] proposed WebPol, a fine-grained policy framework to define the aspects of an element accessible by third-party domains by exposing new native APIs. All these approaches assume a benign website developer protecting their website from untrusted—and possibly malicious—third-party libraries trying to manipulate their website. In contrast, *JavaScript Zero* does not make any assumptions in this direction. Any website or library developer may be malicious, trying to attack the user. *JavaScript Zero* neither relies on website developers nor requires any modifications of the browser or the JavaScript engine.

Kohlbrenner et al. [18] proposed Fuzzyfox, a modified version of Firefox that mediates all timing sources by degrading the resolution of explicit timers and implicit clocks to 100 ms. In contrast to Fuzzyfox, *JavaScript Zero* successfully prevents not only timing attacks but also attacks which do not require high-resolution timing measurements. Mao et al. [21] studied timing-based probing attacks that indirectly infer sensitive information from the website. Their tool only allows to identify malicious operations performing timing-based probing attacks based on generalized patterns, e.g., the frequency of timing API usage. *JavaScript Zero* directly prevents the attack by either disallowing timers or making them too coarse-grained.

Code rewriting Reis et al. [38] implemented BrowserShield, a service that automatically rewrites websites and embedded JavaScript to apply run-time checks to filter known vulnerabilities. Yu et al. [56] proposed to automatically rewrite untrusted JavaScript code through a web proxy, in order to ask the user how to proceed on possible dangerous behavior,

e.g., opening many pop-ups or cookie exfiltration attacks. Their model only covers policies with respect to opening URLs, windows, and cookie accesses, and does not protect against side-channel attacks. Moreover, *JavaScript Zero* does neither rewrite any existing code nor rely on any possibly platform-dependent service such as a web proxy.

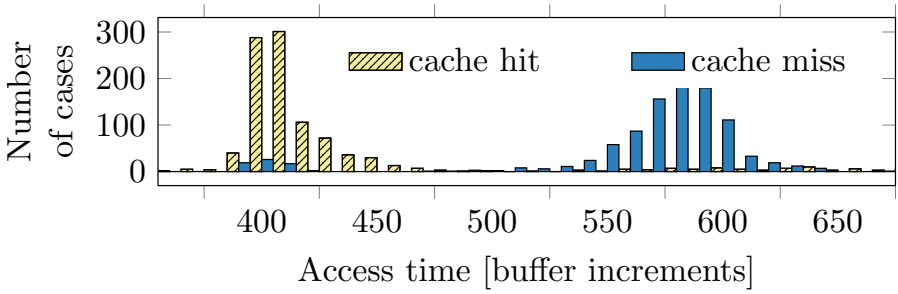
JavaScript frameworks Agten et al. [1] presented JSand, a client-side JavaScript sandboxing framework that enforces a server-specified policy to jail included third-party libraries. Phung et al. [36] proposed to modify code in order to protect it from inappropriate behavior of third-party libraries. Their implementation requires website developers to manually add protection code to their website. However, their protection does not apply to scripts loaded in a new context, *i.e.*, with `<frame>`, `<iframe>`, or refresh directives. Guan et al. [16] studied the privacy implications of the HTML5 `postMessage` function and developed a policy-based framework to restrict unintended cross-origin messages. As our countermeasure is implemented solely as a browser extension, it does not rely on any website developer to use a certain library or to apply any changes to the code.

9 Conclusion

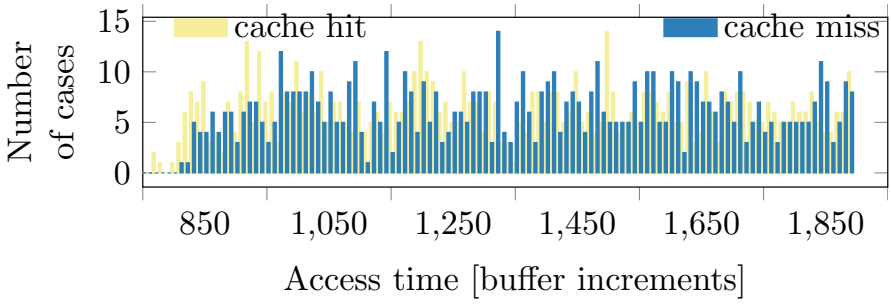
In this paper, we presented *JavaScript Zero*, a highly practical and generic fine-grained permission model in JavaScript to reduce the attack surface in modern browsers. *JavaScript Zero* leverages advanced JavaScript language features, such as virtual machine layering, closures, proxy objects, and object freezing, for security and privacy. Hence, *JavaScript Zero* is fully transparent to website developers and users and even works with obfuscated code and higher-order scripts. Our proof-of-concept Google Chrome extension, *Chrome Zero*, successfully protects against 11 unfixed state-of-the-art microarchitectural and side-channel attacks. As a side effect, *Chrome Zero* successfully protects against 50 % of the published JavaScript 0-day exploits since Chrome 49. *Chrome Zero* has a low-performance overhead of only 1.82% on average. In a double-blind user study, we found that for 24 websites in the Alexa Top 25, users could not distinguish browsers with and without *Chrome Zero* correctly, showing that *Chrome Zero* has no perceivable (negative) effect on most websites. Our work shows that transparent low-overhead defenses against JavaScript-based state-of-the-art microarchitectural attacks and side-channel attacks are practical.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable feedback and our study participants for their time. This work has been supported by the Austrian Research Promotion Agency (FFG), the Styrian Business Promotion Agency (SFG), the Carinthian Economic Promotion Fund (KWF) under grant number 862235 (DeSSnet) and has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).



(a) Without *Chrome Zero*.



(b) With *Chrome Zero*.

Figure 8.10: Using `SharedArrayBuffer` in combination with web worker to build a high-resolution timing primitive as proposed by Gras et al. [13] and Schwarz et al. [40]. Without *Chrome Zero*, cache hits and misses are clearly distinguishable (Figure 8.10a). Configuring *Chrome Zero* to delay accesses to the `SharedArrayBuffer` leads to a uniform distribution in timings, thwarting the attacks (Figure 8.10b).

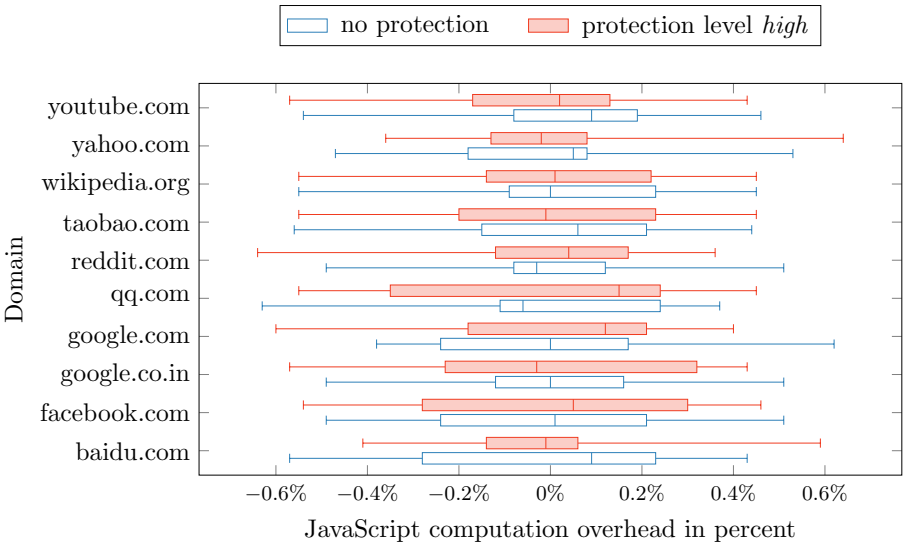


Figure 8.11: The computation overhead of the JavaScript engine while loading each of the Alexa Top 10 websites without protection and with protection level *high*. The performance overhead is negligible for most websites.

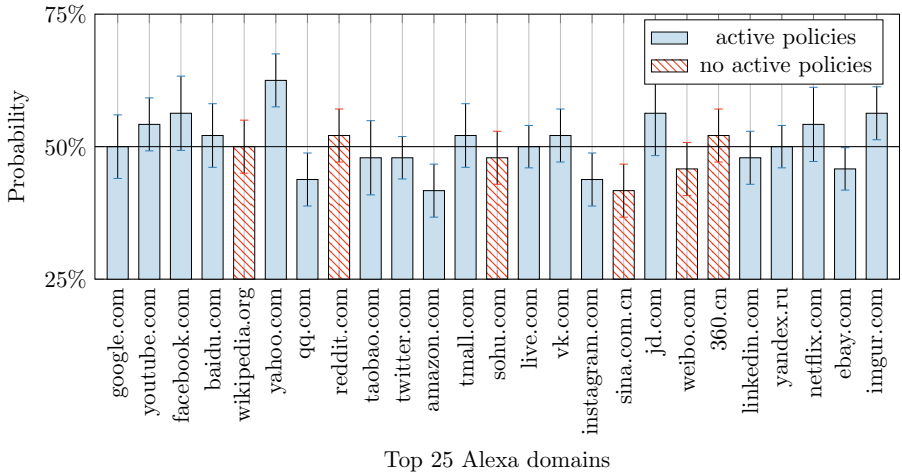


Figure 8.12: Probability that a user correctly identifies the browser with *Chrome Zero* and the corresponding standard error. Only for one website (yahoo.com) the users had a chance of identifying the browser with *Chrome Zero* that was clearly above the random guessing probability.

References

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications." In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012.

[2] Alex Christensen. *Reduce resolution of performance.now*. 2015. URL: https://bugs.webkit.org/show_bug.cgi?id=146531.

[3] Apple. *JSZJetStream 1.1*. Aug. 2017. URL: http://browserbench.org/JSZJetStream.

[4] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. 2004. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[5] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. "WebPol: Fine-grained Information Flow Policies for Web Browsers." In: *ESORICS'17*. (to appear). 2017.

[6] Boris Zbarsky. *Reduce resolution of performance.now*. 2015. URL: https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab.

[7] Liang Cai and Hao Chen. "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion." In: *USENIX Workshop on Hot Topics in Security – HotSec*. 2011.

[8] Chris Peterson. *Bug 1313580: Remove web content access to Battery API*. 2016. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1313580.

[9] Chromium. *window.performance.now does not support sub-millisecond precision on Windows*. 2015. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110.

[10] Giorgio Maone. *JSZNoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!* July 2017. URL: https://noscript.net.

[11] Google. *Chrome DevTools Protocol Viewer*. 2017. URL: https://developer.chrome.com/devtools/docs/debugger-protocol.

[12] Google. *chrome.debugger*. 2017. URL: https://developer.chrome.com/extensions/debugger.

[13] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS'17*. 2017.

[14] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed JavaScript." In: *ESORICS'15*. 2015.

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA'16*. 2016.

[16] Chong Guan, Kun Sun, Zhan Wang, and WenTao Zhu. "Privacy Breach by Exploiting postMessage in HTML5: Identification, Evaluation, and Countermeasure." In: *ASIACCS'16*. 2016.

[17] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems." In: *CRYPTO'96*. 1996.

[18] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times." In: *USENIX Security Symposium*. 2016.

[19] Erick Lavoie, Bruno Dufour, and Marc Feeley. "Portable and efficient run-time monitoring of javascript applications using virtual machine layering." In: *European Conference on Object-Oriented Programming*. 2014.

[20] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript." In: *ESORICS'17*. (to appear). 2017.

[21] Jian Mao, Yue Chen, Futian Shi, Yaoqi Jia, and Zhenkai Liang. "Toward Exposing Timing-Based Probing Attacks in Web Applications." In: *International Conference on Wireless Algorithms, Systems, and Applications*. 2016.

[22] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS'17*. 2017.

[23] Maryam Mehrnezhad, Ehsan Toreini, Siamak F Shahandashti, and Feng Hao. "Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript." In: *Journal of Information Security and Applications* (2016).

[24] Leo A Meyerovich and Benjamin Livshits. "ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser." In: *S&P'10*. 2010.

[25] Mike Perry. *Bug 1517: Reduce precision of time for Javascript.* 2015. URL: https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517.

[26] Mozilla Developer Network. *ArrayBuffer.* 2017. URL: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer.

[27] Mozilla Developer Network. *debugger.* 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/debugger.

[28] Mozilla Developer Network. *Proxy.* 2017. URL: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[29] Nolan Lawson. *A tiny and mostly spec-compliant WebWorker polyfill.* Nov. 2016. URL: https://github.com/nolanlawson/pseudo-worker.

[30] JE Nymann. "On the probability that k positive integers are relatively prime." In: *Journal of Number Theory* (1972).

[31] Lukasz Olejnik. *Stealing sensitive browser data with the W3C Ambient Light Sensor API.* 2017. URL: https://blog.lukaszolejnik.com/stealing-sensitive-browser-data-with-the-w3c-ambient-light-sensor-api/.

[32] Lukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. "The Leaking Battery." In: *Revised Selected Papers of the 10th International Workshop on Data Privacy Management, and Security Assurance - Volume 9481.* 2016.

[33] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS'15.* 2015.

[34] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA.* 2006.

[35] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. "Towards fine-grained access control in javascript contexts." In: *31st International Conference on Distributed Computing Systems (ICDCS).* 2011.

[36] Phu H Phung, David Sands, and Andrey Chudnov. "Lightweight self-protecting JavaScript." In: *ASIACCS'09.* 2009.

[37] Raymond Hill. *JSZuBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean.* July 2017. URL: https://github.com/gorhill/JSZuBlock.

[38] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. "BrowserShield: Vulnerability-driven Filtering of Dynamic HTML." In: *USENIX Symposium on Operating Systems Design and Implementation*. 2006.

[39] Ross McIlroy. *Firing up the Ignition Interpreter*. Aug. 2017. URL: https://v8project.blogspot.co.at/2016/08/firing-up-ignition-interpreter.html.

[40] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC'17*. 2017.

[41] Peter Snyder, Cynthia Taylor, and Chris Kanich. "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security." In: *CCS'17*. 2017.

[42] Raphael Spreitzer. "Pin skimming: Exploiting the ambient-light sensor in mobile devices." In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014.

[43] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. "Protecting Users by Confining JavaScript with COWL." In: *USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[44] Paul Stone. "Pixel perfect timing attacks with HTML5." In: *Context Information Security (White Paper)* (2013).

[45] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. "The clock is still ticking: Timing attacks in the modern web." In: *CCS'15*. 2015.

[46] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. "Eliminating fine grained timers in Xen." In: *CCSW'11*. 2011.

[47] Pepe Vila and Boris Köpf. "Loophole: Timing Attacks on Shared Event Loops in Chrome." In: *USENIX Security Symposium*. (to appear). 2017.

[48] W3C. *Ambient Light Sensor*. 2017. URL: https://www.w3.org/TR/ambient-light/.

[49] W3C. *Battery Status API*. 2016. URL: https://www.w3.org/TR/battery-status/.

[50] W3C. *DeviceOrientation Event Specification*. 2017. URL: https://www.w3.org/TR/orientation-event/.

[51] W3C. *Generic Sensor API*. 2017. URL: https://www.w3.org/TR/2017/WD-generic-sensor-20170530/.

[52] W3C. *Geolocation API Specification 2nd Edition*. 2016. URL: https://www.w3.org/TR/geolocation-API/.

[53] W3C. *Javascript APIs Current Status*. 2017. URL: https://www.w3.org/standards/techs/js.

[54] W3Techs. *Usage of JavaScript for websites*. Aug. 2017. URL: https://w3techs.com/technologies/details/cp-javascript/all/all.

[55] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

[56] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. "JavaScript instrumentation for browser security." In: *ACM SIGPLAN Notices*. 2007.

9

Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features

Publication Data

Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In: *AsiaCCS*. 2018

Contributions

Main author.

Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features

Michael Schwarz[1], Daniel Gruss[1], Moritz Lipp[1], Clémentine Maurice[2], Thomas Schuster[1], Anders Fogh[3], Stefan Mangard[1]

[1] Graz University of Technology, Austria [2] CNRS, IRISA, France
[3] G DATA Advanced Analytics, Germany

Abstract

Double-fetch bugs are a special type of race condition, where an unprivileged execution thread is able to change a memory location between the time-of-check and time-of-use of a privileged execution thread. If an unprivileged attacker changes the value at the right time, the privileged operation becomes inconsistent, leading to a change in control flow, and thus an escalation of privileges for the attacker. More severely, such double-fetch bugs can be introduced by the compiler, entirely invisible on the source-code level.

We propose novel techniques to efficiently detect, exploit, and eliminate double-fetch bugs. We demonstrate the first combination of state-of-the-art cache attacks with kernel-fuzzing techniques to allow fully automated identification of double fetches. We demonstrate the first fully automated reliable detection and exploitation of double-fetch bugs, making manual analysis as in previous work superfluous. We show that cache-based triggers outperform state-of-the-art exploitation techniques significantly, leading to an exploitation success rate of up to 97 %. Our modified fuzzer automatically detects double fetches and automatically narrows down this candidate set for double-fetch bugs to the exploitable ones. We present the first generic technique based on hardware transactional memory, to eliminate double-fetch bugs in a fully automated and transparent manner. We extend defensive programming techniques by retrofitting arbitrary code with automated double-fetch prevention, both in trusted execution environments as well as in syscalls, with a performance overhead below 1 %.

The original publication is available at `https://dl.acm.org/citation.cfm?id=3196508`.

1 Introduction

The security of modern computer systems relies fundamentally on the security of the operating system kernel, providing strong isolation between processes. While kernels are increasingly hardened against various types of memory corruption attacks, race conditions are still a non-trivial problem. Syscalls are a common scenario in which the trusted kernel space has to interact with the untrusted user space, requiring sharing of memory locations between the two environments. Among possible bugs in this scenario are time-of-check-to-time-of-use bugs, where the kernel accesses a memory location twice, first to check the validity of the data and second to use it (double fetch) [65]. If such double fetches are exploitable, they are considered double-fetch bugs. The untrusted user space application can change the value between the two accesses and thus corrupt kernel memory and consequently escalate privileges. Double-fetch bugs can not only be introduced at the source-code level but also by compilers, entirely invisible for the programmer and any source-code-level analysis technique [5]. Recent research has found a significant amount of double fetches in the kernel through static analysis [70], and memory access tracing through full CPU emulation [38]. Both works had to manually determine for every double fetch, whether it is a double-fetch bug.

Double fetches have the property that the data is fetched twice from memory. If the data is already in the cache (*cache hit*), the data is fetched from the cache, if the data is not in the cache (*cache miss*), it is fetched from main memory into the cache. Differences between fetches from cache and memory are the basis for so-called cache attacks, such as Flush+ Reload [56, 77], which obtain secret information by observing memory accesses [21]. Instead of exploiting the cache side channel for obtaining secret information, we utilize it to detect double fetches.

In this paper, we show how to efficiently and automatically detect, exploit, and eliminate double-fetch bugs, with two new approaches: DECAF and DropIt.

DECAF is a double-fetch-exposing cache-guided augmentation for fuzzers, which automatically *detects* and *exploits* real-world double-fetch bugs in a two-phase process. In the profiling phase, DECAF relies on cache side channel information to detect whenever the kernel accesses a syscall parameter. Using this novel technique, DECAF is able to detect whether a parameter is fetched multiple times, generating a candidate set containing double fetches, *i.e.*, some of which are potential double-fetch bugs. In the exploitation phase, DECAF uses a cache-based trigger signal to flip values

while fuzzing syscalls from the candidate set, to trigger actual double-fetch bugs. In contrast to previous purely probability-based approaches, cache-based trigger signals enable deterministic double-fetch-bug exploitation. Our automated exploitation exceeds state-of-the-art techniques, where checking the double-fetch candidate set for actual double-fetch bugs is tedious manual work. We show that DECAF can also be applied to trusted execution environments, e.g., ARM TrustZone and Intel SGX.

DropIt is a protection mechanism to *eliminate* double-fetch bugs. DropIt uses hardware transactional memory to efficiently drop the current execution state in case of a concurrent modification. Hence, double-fetch bugs are automatically reduced to ordinary non-exploitable double fetches. In case user-controlled memory locations are modified, DropIt continues the execution from the last consistent state. Applying DropIt to syscalls induces no performance overhead on arbitrary computations running in other threads and only a negligible performance overhead of 0.8 % on the process executing the protected syscall. We show that DropIt can also be applied to trusted execution environments, e.g., ARM TrustZone and Intel SGX.

Contributions. We make the following contributions:

1. We are the first to combine state-of-the-art cache attacks with kernel-fuzzing techniques to build DECAF, a generic double-fetch-exposing cache-guided augmentation for fuzzers.
2. Using DECAF, we are the first to show fully automated reliable detection and exploitation of double-fetch bugs, making manual analysis as in previous work superfluous.
3. We outperform state-of-the-art exploitation techniques significantly, with an exploitation success rate of up to 97 %.
4. We present DropIt, the first generic technique to eliminate double-fetch bugs in a fully automated manner, facilitating newfound effects of hardware transactional memory on double-fetch bugs. DropIt has a negligible performance overhead of 0.8 % on protected syscalls.
5. We show that DECAF can also fuzz trusted execution environments in a fully automated manner. We observe strong synergies between Intel SGX and DropIt, enabling efficient preventative protection from double-fetch bugs.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on cache attacks, race conditions, and kernel fuzzing. In Section 3, we discuss the building blocks for finding and eliminating double-fetch bugs. We present the profiling phase of DECAF

in Section 4 and the exploitation phase of DECAF in Section 5. In Section 6, we show how hardware transactional memory can be used to eliminate all double-fetch bugs generically. In Section 7 we discuss the results of we obtained by instantiating DECAF. We conclude in Section 8.

2 Background

2.1 Fuzzing

Fuzzing describes the process of testing applications with randomized input to find vulnerabilities.

The term "fuzzing" was coined 1988 by Miller [49], and later on extended to an automated approach for testing the reliability of several user-space programs on Linux [51], Windows [18] and Mac OS [50]. There is an immense number of works exploring user space fuzzing with different forms of feedback [12, 14, 19, 22–25, 32, 39, 61, 67]. However, these are not applicable to this work, as we focus on fuzzing the kernel and trusted execution environments.

Fuzzing is not limited to testing user-space applications, but it is also, to a much smaller extent, used to test the reliability of operating systems. Regular user space fuzzers cannot be used here, but a smaller number of tools have been developed to apply fuzzy testing to operating system interfaces. Carrette [9] developed the tool CrashMe that tests the robustness of operating systems by trying to execute random data streams as instructions. Mendoncca et al. [48] and Jodeit et al. [36] demonstrate that fuzzing drivers via the hardware level is another possibility to attack an operating system. Other operating system interfaces that can be fuzzed include the file system [7] and the virtual machine interface [20, 46].

The syscall interface is a trust boundary between the trusted kernel, running with the highest privileges, and the unprivileged user space. Bugs in this interface can be exploited to escalate privileges. Koopman et al. [40] were among the first to test random inputs to syscalls. Modern syscall fuzzers, such as Trinity [37] or syzkaller [69], test most syscalls with semi-intelligent arguments instead of totally random inputs. In contrast to these generic tools, Weaver et al. [71] developed perf_fuzzer, which uses domain knowledge to fuzz only the performance monitoring syscalls.

2.2 Flush+Reload

Flush+Reload is a side-channel attack exploiting the difference in access times between CPU caches and main memory. Yarom and Falkner [77]

presented Flush+Reload as an improvement over the cache attack by Gullasch et al. [31]. Flush+Reload relies on shared memory between the attacker and the victim and works as follows:

1. Establish a shared memory region with the victim (e.g., by mapping the victim binary into the address space).
2. Flush one line of the shared memory from the cache.
3. Schedule the victim process.
4. Measure the access time to the flushed cache line.

If the victim accesses the cache line while being scheduled, it is again cached. When measuring the access time, the attacker can distinguish whether the data is cached or not and thus infer whether the victim accessed it. As Flush+Reload works on cache line granularity (usually 64 B), fine-grained attacks are possible. The probability of false positives is very low with Flush+Reload, as cache hits cannot be caused by different programs and prefetching can be avoided. Gruss et al. [28] reported extraordinarily high accuracies, above 99 %, for the Flush+Reload side channel, making it a viable choice for a wide range of applications.

2.3 Double Fetches and Double-Fetch Bugs

In a scenario where shared memory is accessed multiple times, the CPU may fetch it multiple times into a register. This is commonly known as a **double fetch**. Double fetches occur when the kernel accesses data provided by the user multiple times, which is often unavoidable. If proper checks are done, ensuring that a change in the data during the fetches is correctly handled, double fetches are **non-exploitable** valid constructs.

A **double-fetch bug** is a time-of-check-to-time-of-use race condition, which is **exploitable** by changing the data in the shared memory between two accesses. Double-fetch bugs are a subset of double fetches. A double fetch is a double-fetch bug, if and only if it can be exploited by concurrent modification of the data. For example, if a syscall expects a string and first checks the length of the string before copying it to the kernel, an attacker could change the string to a longer string after the check, causing a buffer overflow in the kernel. This can lead to code execution within the kernel.

Wang et al. [70] used Coccinelle, a transformation and matching engine for C code, to find double fetches. With this static pattern-based approach, they identified 90 double fetches inside the Linux kernel. However, their work incurred several days of manual analysis of these 90 double fetches, identifying only 3 exploitable double-fetch bugs. A further limitation

of their work is that double-fetch bugs not matching the implemented patterns, cannot be detected. Xu et al. [75] used static code analysis in combination with symbolic checking to identify 23 new bugs in Linux. Again, double-fetch bugs not matching their formal definition are not identified.

Not all double fetches, and thus not all double-fetch bugs, can be found using static code analysis. Blanchou [5] demonstrated that especially in lock-free code, compilers can introduce double fetches that are not present in the code. Even worse, these compiler-introduced double fetches can become double-fetch bugs in certain scenarios (e.g., CVE-2015-8550). Jurczyk et al. [38] presented a dynamic approach for finding double fetches. They used a full CPU emulator to run Windows and log all memory accesses. Note that this requires significant computation and storage resources, as just booting Windows already consumes 15 hours of time, resulting in a log file of more than 100 GB [38]. In the memory access log, they searched for a distinctive double-fetch pattern, e.g., two reads of the same user-space address within a short time frame. They identified 89 double fetches in Windows 7 and Windows 8. However, their work also required manual analysis, in which they found that only 2 out of around 100 unique double fetches were exploitable double-fetch bugs. Again, if a double-fetch bug does not match the implemented double-fetch pattern, it is not detected. In summary, we find that all techniques for double-fetch bug detection are probabilistic and hence incomplete.

Race Condition Detection

Besides research on double fetches and double-fetch bugs, there has been a significant amount of research on race condition detection in general. Static analysis of source code and dynamic runtime analysis have been used to find data race conditions in multithreaded applications. Savage et al. [62] described the Lockset algorithm. Their tool, Eraser, dynamically detects race conditions in multithreaded programs. Pozniansky et al. [59, 60] extended their work to detect race conditions in multithreaded C++ programs on-the-fly. Yu et al. [79] described RaceTrack, an adaptive detection algorithm that reports suspicious activity patterns. These algorithms have been improved and made more efficient by using more lightweight data structures [17] or combining various approaches [74].

While these tools can be applied to user space programs, they are not designed to detect race conditions in the kernel space. Erickson et al. [15] utilized breakpoints and watchpoints on memory accesses to detect

data races in the Windows kernel. With RaceHound [54], the same idea has been implemented for the Linux kernel. The SLAM [3] project uses symbolic model checking, program analysis, and theorem proving, to verify whether a driver correctly interacts with the operating system. Schwarz et al. [63] utilized software model checking to detect security violations in a Linux distribution.

More closely related to double-fetch bugs, other time-of-check-to-time-of-use bugs exist. By changing the content of a memory location that is passed to the operating system, the content of a file could be altered after a validity check [4, 8, 72]. Especially time-of-check-to-time-of-use bugs in the file system are well-studied, and several solutions have been proposed [13, 42, 57, 58, 68].

2.4 Hardware Transactional Memory

Hardware transactional memory is designed for optimizing synchronization primitives [16, 78]. Any changes performed inside a transaction are not visible to the outside before the transaction succeeds. The processor speculatively lets a thread perform a sequence of operations inside a transaction. Unless there is a conflict due to a concurrent modification of a data value, the transaction succeeds. However, if a conflict occurs before the transaction is completed (e.g., a concurrent write access), the transaction aborts. In this case, all changes that have been performed in the transaction are discarded, and the previous state is recovered. These fundamental properties of hardware transactional memory imply that once a value is read in a transaction, the value cannot be changed from outside the transaction anymore for the time of the transaction.

Intel TSX is a hardware transactional memory implementation with cache line granularity. It is available on several CPUs starting with the Haswell microarchitecture. Intel TSX maintains a read set which is limited to the size of the L3 cache and a write set limited to the size of the L1 cache [26, 34, 45, 80]. A cache line is automatically added to the read set when it is read inside a transaction, and it is automatically added to the write set when it is modified inside a transaction. Modifications to any memory in the read set or write set from other threads cause the transaction to abort.

Previous work has investigated whether hardware transactional memory can be instrumented for security features. Guan et al. [30] proposed to protect cryptographic keys by only decrypting them within TSX transactions. As the keys are never written to DRAM in an unencrypted form,

they cannot be read from memory even by a physical attacker probing the DRAM bus. Kuvaiskii et al. [41] proposed to use TSX to detect hardware faults and roll-back the system state in case a fault occurred. Shih et al. [66] proposed to exploit the fact that TSX transactions abort if a page fault occurred for a memory access to prevent controlled-channel attacks [76] in cloud scenarios. Chen et al. [10] implemented a counting thread protected by TSX to detect controlled-channel attacks in SGX enclaves. Gruss et al. [29] demonstrated that TSX can be used to protect against cache side-channel attacks in the cloud.

Shih et al. [66] and Gruss et al. [29] observed that Intel TSX has several practical limitations. One observation is that executed code is not considered transactional memory, *i.e.*, virtually unlimited amount of code can be executed in a transaction. To evade the limitations caused by the L1 and L3 cache sizes, Shih et al. [66] and Gruss et al. [29] split transactions that might be memory-intense into multiple smaller transactions.

3 Building Blocks to Detect, Exploit, and Eliminate Double-Fetch Bugs

In this section, we present building blocks for detecting double fetches, exploiting double-fetch bugs, and eliminating double-fetch bugs. These building blocks are the base for DECAF and DropIt.

We identified three primitives, illustrated in Figure 9.1, for which we propose novel techniques in this paper:

- $\boldsymbol{\mathcal{P}1}$: Detecting double fetches via the Flush+Reload side channel.
- $\boldsymbol{\mathcal{P}2}$: Distinguishing (exploitable) double-fetch bugs from (non-exploitable) double fetches by validating their exploitability by automatically exploiting double-fetch bugs.
- $\boldsymbol{\mathcal{P}3}$: Eliminating (exploitable) double-fetch bugs by using hardware transactional memory.

In Section 4, we propose a novel, fully automated technique to detect double fetches ($\boldsymbol{\mathcal{P}1}$) using a multi-threaded Flush+Reload cache side-channel attack. Our technique complements other work on double-fetch bug detection [38, 70] as it covers scenarios which lead to false positives and false negatives in other detection methods. Although relying on a side channel may seem unusual, this approach has certain advantages over state-of-the-art techniques, such as memory access tracing [38] or static code analysis [70]. We do not need any model of what constitutes a double

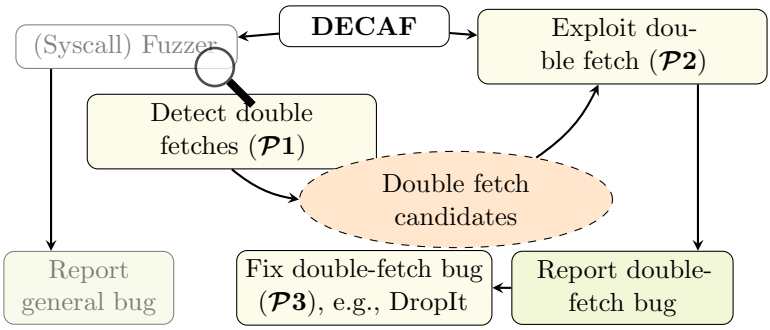


Figure 9.1: An overview of the framework. Detecting (Primitive $\mathcal{P}\mathbf{1}$) and exploiting (Primitive $\mathcal{P}\mathbf{2}$) double fetches runs in parallel to the syscall fuzzer. Reported double-fetch bugs can be eliminated (Primitive $\mathcal{P}\mathbf{3}$) after the fuzzing process.

fetch in terms of memory access traces or static code patterns. Hence, we can detect any double fetch regardless of any double fetch model.

Wang et al. [70] identified as limitations of their initial approach that false positives occur if a pointer is changed between two fetches and memory accesses, in fact, go to different locations or if user-space fetches occur in loops. Furthermore, false negatives occur if multiple pointers point to the same memory location (pointer aliasing) or if memory is addressed through different types (type conversion), or if an element is fetched separately from the corresponding pointer and memory. With a refined approach, they reduced the false positive rate from more than 98 % to only 94 %, *i.e.*, 6 % of the detected situations turned out to be actual double-fetch bugs in the manual analysis. Wang et al. [70] reported that it took an expert "only a few days" to analyze them. In contrast, our Flush+Reload-based approach is oblivious to language-level structures. The Flush+Reload-trigger only depends on actual accesses to the same memory location, irrespective of any programming constructs. Hence, we inherently bypass the problems of the approach of Wang et al. [70] by design.

Our technique does not replace existing tools, which are either slow [38] or limited by static code analysis [70] and require manual analysis. Instead, we complement previous approaches by utilizing a side channel, allowing fully automatic detection of double-fetch bugs, including those that previous approaches may miss.

In Section 5, we propose a novel technique to automatically determine whether a double fetch found using $\mathcal{P}\mathbf{1}$ is an (exploitable) double-fetch

bug (**$\mathcal{P}$2**). State-of-the-art techniques are only capable of automatically detecting double fetches using either dynamic [38] or static [70] code analysis, but cannot determine whether a found double fetch is an exploitable double-fetch bug. The double fetches found using these techniques still require manual analysis to check whether they are valid constructs or exploitable double-fetch bugs. We close this gap by automatically testing whether double fetches are exploitable double-fetch bugs (**$\mathcal{P}$2**), eliminating the need for manual analysis. Again, this technique relies on a cache side channel to trigger a change of the double-fetched value between the two fetches (**$\mathcal{P}$2**). This is not possible with previous techniques [38, 70].

As the first automated technique, we present DECAF, a double-fetch-exposing cache-guided augmentation for fuzzers, leveraging **$\mathcal{P}$1** and **$\mathcal{P}$2** in parallel to regular fuzzing. This allows us to automatically detect double fetches in the kernel and to automatically narrow them down to the exploitable double-fetch bugs (cf. Section 5), as opposed to previous techniques [38, 70] which incurred several days of manual analysis work by an expert to distinguish double-fetch bugs from double fetches. Similar to previous approaches [38, 70], which inherently could not detect all double-fetch bugs in the analyzed code base, our approach is also probabilistic and might not detect all double-fetch bugs. However, due to their different underlying techniques, the previous approaches and ours complement each other.

In Section 6, we present a novel method to simplify the elimination of detected double-fetch bugs (**$\mathcal{P}$3**). We observe previously unknown interactions between double-fetch bugs and hardware transactional memory. Utilizing these effects, **$\mathcal{P}$3** can protect code without requiring to identify the actual cause of a double-fetch bug. Moreover, **$\mathcal{P}$3** can even be applied as a preventative measure to protect critical code.

As a practical implementation of **$\mathcal{P}$3**, we built DropIt, an open-source [1] instantiation of **$\mathcal{P}$3** based on Intel TSX. We implemented DropIt as a library, which eliminates double-fetch bugs with as few as 3 additional lines of code. We show that DropIt has the same effect as rewriting the code to eliminate the double fetch. Furthermore, DropIt can automatically and transparently eliminate double-fetch bugs in trusted execution environments such as Intel SGX, in both desktop and cloud environments.

[1] The source can be found at `https://www.github.com/IAIK/libdropit`.

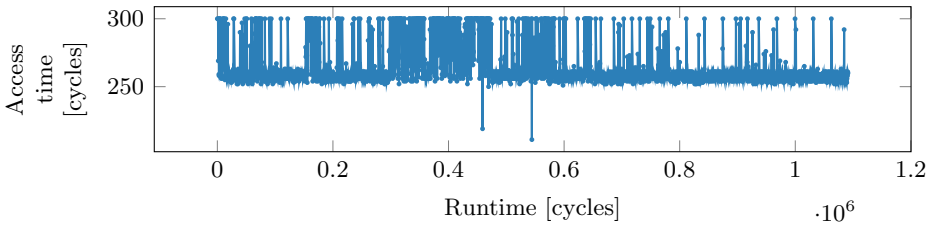


Figure 9.2: Flush+Reload timing trace for a syscall with a double fetch. The two downward peaks show when the kernel accessed the argument.

4 Detecting Double Fetches

We propose a novel dynamic approach to detect double fetches based on their cache access pattern (**$\mathcal{P}$1**, cf. Section 3). The main idea is to monitor the cache access pattern of syscall arguments of a certain type, *i.e.*, pointers or structures containing pointers. These pointers may be accessed multiple times by the kernel and, hence, a second thread can change the content. Other arguments that are statically copied or passed by value, and consequently are not accessed multiple times, cannot lead to double fetches.

To monitor access to potentially vulnerable function arguments, we mount a Flush+Reload attack on each argument in dedicated monitoring threads. A monitoring thread continuously flushes and reloads the memory location referenced by the function argument. As soon as the kernel accesses the function argument, the data is loaded into the cache. In this case, the Flush+Reload attack in the corresponding monitoring thread reports a *cache hit*.

Figure 9.2 shows a trace generated by a monitoring thread. The trace contains the access time in cycles for the memory location referenced by the function argument. If the memory is accessed twice, *i.e.*, a double fetch, we can see a second cache hit, as shown in Figure 9.2. This provides us with primitive **$\mathcal{P}$1**.

4.1 Classification of Multiple Cache Hits

Multiple cache hits within one trace usually correspond to multiple fetches. However, there are rare cases where this is not the case. To entirely eliminate spurious cache hits from prefetching, we simply disabled the prefetcher in software through MSR `0x1A4` and allocated memory on

different pages to avoid spatial prefetching. Note that this does not have any effect on the overall system stability and only a small performance impact. We want to discuss two other factors influencing the cache access pattern in more detail.

Size of data type Depending on the size of the data type, there are differences in the cache access pattern. If the data fills exactly one cache line, accesses to the cache line are clearly seen in the cache access pattern. There are no false positives due to unrelated data in the same cache set, and every access to the referenced memory is visible in the cache access pattern.

To avoid false positives if the data size is smaller than a cache line (*i.e.*, 64 B), we allocate memory chunks with a multiple of the page size, ensuring that dynamically allocated memory never shares one cache line. Hence, accesses to unrelated data (*i.e.*, separate allocations) do not introduce any false positives, as they are never stored in the same cache line. Thus, false positives are only detected if the cache line contains either multiple parameters, local variables or other members of the same structure.

Parameter reuse With call-by-reference, one parameter of a function can be used both as input and output, e.g., in functions working in-place on a given buffer. Using Flush+Reload, we cannot distinguish whether a cache hit is due to a read of or write to the memory. Thus, we can only observe multiple cache hits without knowing whether they are both caused by a memory read access or by other activity on the same cache line.

4.2 Probability of Detecting a Double Fetch

The actual detection rate of a double fetch depends on the time between two accesses. Each Flush+Reload cycle consists of flushing the memory from the cache and measuring the access time to this memory location afterwards. Such a cycle takes on average 298 cycles on an Intel i7-6700K. Thus, to detect a double fetch, the time between the two memory accesses has to be at least two Flush+Reload cycles, *i.e.*, 596 CPU cycles.

We obtain the exact same results when testing a double fetch in kernel space as in user space. Also, due to the high noise-resistance of Flush+Reload (cf. Section 2), interrupts, context switches, and other system activity have an entirely negligible effect on the result. With the minimum distance of 596 CPU cycles, we can already detect double fetches if the

scheduling is optimal for both applications. The further the memory fetches are apart, the higher the probability of detecting the double fetch. The probability of detecting double fetches increases monotonically with the time between the fetches, making it quite immune to interrupts such as scheduling. If the double fetches are at least 3000 CPU cycles apart, we almost always detect such a double fetch. In the real-world double-fetch bugs we examined, the double fetches were always significantly more than 3000 CPU cycles apart. Fig. 9.9 (Appendix A) shows the relation between the detection probability and the time between the memory accesses, empirically determined on an Intel i7-6700K.

On a Raspberry Pi 3 with an ARMv8 1.2 GHz Broadcom BCM2837 CPU, a Flush+Reload cycle takes 250 cycles on average. Hence, the double fetches must be at least 500 cycles apart to be detectable with a high probability.

4.3 Automatically Finding Affected Syscalls

Using our primitive $\boldsymbol{\mathcal{P}1}$, we can already automatically and reliably detect whether a double fetch occurs for a particular function parameter. This is the first building block of DECAF. DECAF is a two-phase process, consisting of a profiling phase which finds double fetches and an exploitation phase narrowing down the set of double fetches to only double-fetch bugs. We will now discuss how DECAF augments existing fuzzers to discover double fetches within operating system kernels fully automatically.

To test a wide range of syscalls and their parameters, we instantiate DECAF with existing syscall fuzzers. For Linux, we retrofitted the well-known syscall fuzzer *Trinity* with our primitive $\boldsymbol{\mathcal{P}1}$. For Windows, we extended the basic *NtCall64* fuzzer to support semi-intelligent parameter selection similar to Trinity. Subsequently, we retrofitted our extended NtCall64 fuzzer with our primitive $\boldsymbol{\mathcal{P}1}$ as well. Thereby, we demonstrate that DECAF is a generic technique and does not depend on a specific fuzzer or operating system.

Our augmented and extended NtCall64 fuzzer, *NtCall64DECAF* works for double fetches and double-fetch bugs in proof-of-concept drivers. However, due to the severely limited coverage of the NtCall64 fuzzer, we did not include it in our evaluations. Instead, we focus on Linux only and leave retrofitting a good Windows syscall fuzzer with DECAF for future work.

In the profiling phase of DECAF, the augmented syscall fuzzer chooses a random syscall to test. The semi-intelligent parameter selection of the

syscall fuzzer ensures that the syscall parameters are valid parameters in most cases. Hence, the syscall is executed and does not abort in the initial sanity checks.

Every syscall parameter that is either a pointer, a file or directory name, or an allocated buffer, can be monitored for double fetches. As Trinity already knows the data types of all syscall parameters, we can easily extend the main fuzzing routine. After Trinity selects a syscall to fuzz, it chooses the arguments to test with and starts a new process. Within this process, we spawn a Flush+Reload monitoring thread for every parameter that may potentially contain a double-fetch bug. The monitoring threads continuously flush the corresponding syscall parameter and measure the reload time. As soon as the parameter is accessed from kernel code, the monitoring thread measures a low access time. The threads report the number of detected accesses to the referenced memory after the syscall has been executed. These findings are logged, and simultaneously, all syscalls with double fetches are added to a candidate set for the interleaved exploitation phase. In Sect. 5, we additionally show how the second building block $\boldsymbol{\mathcal{P}2}$, allows to automatically test whether such a double fetch is exploitable. Figure 9.8 (Appendix A) shows the process structure of our augmented version of Trinity, called *TrinityDECAF*.

4.4 Double-Fetch Detection for Black Boxes

The Flush+Reload-based detection method ($\boldsymbol{\mathcal{P}1}$) is not limited to double fetches in operating system kernels. In general, we can apply the technique for all black boxes fulfilling the following criteria:

1. Memory references can be passed to the black box.
2. The referenced memory is (temporarily) shared between the black box and the host.
3. It is possible to run code in parallel to the execution of the black box.

This generalization does not only apply to syscalls, but it also applies to trusted execution environments.

Trusted execution environments are particularly interesting targets for double fetch detection and double-fetch-bug exploitation. Trusted execution environments isolate programs from other user programs and the operating system kernel. These programs are often neither open source nor is the unencrypted binary available to the user. Thus, if the vendor did not test for double-fetch bugs, researchers not affiliated with the vendor have no possibility to scan for these vulnerabilities. Moreover, even

the vendor might not be able to apply traditional double-fetch detection techniques, such as dynamic program analysis, if these tools are not available within the trusted execution environment.

Both Intel SGX [47] and ARM TrustZone [1] commonly share memory buffers between the host application and the trustlet running inside the trusted execution environment through their interfaces. Therefore, we can again utilize a Flush+Reload monitoring thread to detect double fetches by the trusted application ($\boldsymbol{\mathcal{P}1}$).

5 Exploiting Double-Fetch Bugs

In this section, we detail the second building block of DECAF, primitive $\boldsymbol{\mathcal{P}2}$, the base of the DECAF exploitation phase. It allows us to exploit any double fetch found via $\boldsymbol{\mathcal{P}1}$ (cf. Section 4) reliably and automatically. In contrast to state-of-the-art value flipping [38] (success probability 50 % or significantly lower), our exploitation phase has a success probability of 97 %. The success probability of value flipping is almost zero if multiple sanity checks are performed, whereas the success probability of $\boldsymbol{\mathcal{P}2}$ decreases only slightly.

5.1 Flush+Reload as a Trigger Signal

We propose to use Flush+Reload as a trigger signal to deterministically and reliably exploit double-fetch bugs. Indeed, Flush+Reload is a reliable approach to detect access to the memory, allowing us to flip the value immediately after an access. This combination of a trigger signal and targeted value flipping forms primitive $\boldsymbol{\mathcal{P}2}$.

The idea of the double-fetch-bug exploitation ($\boldsymbol{\mathcal{P}2}$) is therefore similar to the double-fetch detection ($\boldsymbol{\mathcal{P}1}$). As soon as one access to a parameter is detected, the value of the parameter is flipped to an invalid value. Just as the double-fetch detection (cf. Sect. 4), we can use a double-fetch trigger signal for every black box which uses memory references as parameters in the communication interface.

As shown in Figure 9.3, the exploitation phase can target double-fetch bugs with an even lower time delta between the accesses, than the double-fetch detection in the profiling phase (cf. Section 4). The reason is that only the first access has to be detected and changing the value is significantly faster than a full Flush+Reload cycle. Thus, it is even possible to exploit double fetches where the time between them is already too short to detect them. Consequently, every double fetch detected in

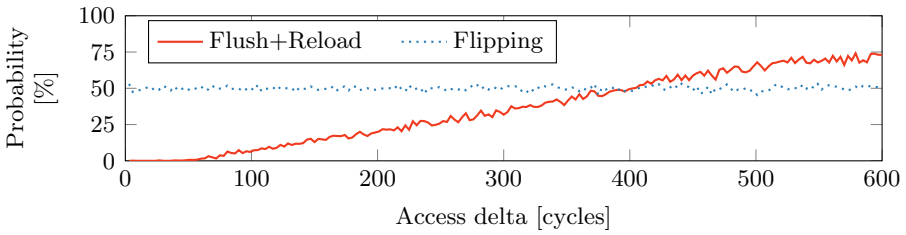


Figure 9.3: The probability of successfully exploiting a double-fetch bug depending on the time between the accesses.

the profiling phase can clearly be tested for exploitability using $\boldsymbol{\mathcal{P}2}$ in the exploitation phase.

As a fast alternative to Flush+Reload, Flush+Flush [28] could be used. Although Flush+Flush is significantly faster than Flush+Reload, Flush+Reload is usually the better choice as it has less noise.

5.2 Automated Syscall Exploitation

With the primitive $\boldsymbol{\mathcal{P}2}$ from Section 5.1, we add the second building block to DECAF, to not only detect double fetches but also to immediately exploit them. This has the advantage that exploitable double-fetch bugs can be found without human interaction, as the automated exploitation leads to evident errors and system crashes. As described in Section 4, DECAF does not only report the double fetches but also adds them to a candidate set for double-fetch bug testing. If a candidate is added to this set, the double-fetch bug test ($\boldsymbol{\mathcal{P}2}$) is immediately interleaved into the regular fuzzing process.

We randomly switch between four different methods to change the value: setting it to zero, flipping the least significant bit, incrementing the value, and replacing it by a random value. Setting a value to zero or a random value is useful to change pointers to invalid locations. Furthermore, it is also effective on string buffers as it can shorten the string, extend the string, or introduce invalid characters. Incrementing a value or flipping the least significant bit is especially useful if the referenced memory contains integers, as it might trigger off-by-one errors.

In summary, in the exploitation phase of DECAF, we reduce the double-fetch candidate set (obtained via $\boldsymbol{\mathcal{P}1}$) to exploitable double-fetch bugs without any human interaction ($\boldsymbol{\mathcal{P}2}$), complementing state-of-the-art techniques [38, 70]. The coverage of DECAF highly depends on the fuzzer used. Fuzzing is probabilistic and might not find every exploitable double

fetch, but with growing coverage of fuzzers, the coverage of DECAF will automatically grow as well.

6 Eliminating Double-Fetch Bugs

In this section, we propose the first transparent and automated technique to entirely eliminate double-fetch bugs (***P*3**). We utilize previously unknown interactions between double-fetch bugs and hardware transactional memory. ***P*3** protects code without requiring to identify the actual cause of a double-fetch bug and can even be applied as a preventative measure to protect critical code.

We present the DropIt library, an instantiation of ***P*3** with Intel TSX. DropIt eliminates double-fetch bugs, having the same effect as rewriting the code to eliminate the double fetch. We also show its application to Intel SGX, a trusted execution environment that is particularly interesting in cloud scenarios.

6.1 Problems of State-of-the-Art Double-Fetch Elimination

Introducing double-fetch bugs in software happens easily, and they often stay undetected for many years. As shown recently, modern operating systems still contain a vast number of double fetches, some of which are exploitable double-fetch bugs [38, 70]. As shown in Section 4 and Section 5, identifying double-fetch bugs requires full code coverage, and before our work, a manual inspection of the detected double fetches. Even when double-fetch bugs are identified, they are usually not trivial to fix.

A simple example of a double-fetch bug is a syscall with a string argument of arbitrary length. The kernel requires two accesses to copy the string, first to retrieve the length of the string and allocate enough memory, and second, to copy the string to the kernel.

Writing this in a naïve way can lead to severe problems, such as unterminated strings of kernel buffer overflows. One approach is to use a retry logic, as shown in Algorithm 2 (Appendix B), as it used in the Linux kernel whenever user data of unknown length has to be copied to the kernel. Such methods increase the complexity and runtime of code, and they are hard to wrap into generic functions.

Finally, compilers can also introduce double fetches that are neither visible in the source code nor easily detectable, as they are usually within just a few cycles [5].

6.2 Generic Double-Fetch Bug Elimination

Eliminating double-fetch bugs is not equivalent to eliminating double fetches. Double fetches are valid constructs, as long as a change of the value is successfully detected, or it is not possible to change the value between two memory accesses. Thus, making a series of multiple fetches atomic is sufficient to eliminate double-fetch bugs, as there is only one operation from an attacker's view (see Sect. 2.4). Curiously, the concept of hardware transactional memory provides exactly this atomicity.

As also described in Section 2.4, transactional memory provides atomicity, consistency, and isolation [33]. Hence, by wrapping code possibly containing a double fetch within a hardware transaction, we can benefit from these properties. From the view of a different thread, the code is one atomic memory operation. If an attacker changes the referenced memory while the transaction is active, the transaction aborts and can be retried. As the retry logic is implemented in hardware and not simulated by software, the induced overhead is minimal, and the amount of code is drastically reduced.

In a nutshell, hardware transactional memory can be instrumented as a hardware implementation of software-based retry solutions discussed in Section 6.1. Thus, wrapping a double-fetch bug in a hardware transaction does not hide, but actually eliminates the bug ($\boldsymbol{\mathcal{P}3}$). Similar to the software-based solution, our generic double-fetch bug elimination can be automatically applied in many scenarios, such as the interface between trust domains (e.g., `ECALL` in SGX). Naturally, solving a problem with hardware support is more efficient, and less error-prone, than a pure software solution.

In contrast to software-based retry solutions, our hardware-assisted solution ($\boldsymbol{\mathcal{P}3}$) does not require any identification of the resource to be protected. For this reason, we can even prevent undetectable or yet undetected double-fetch bugs, regardless of whether they are introduced on the source level or by the compiler. As these interfaces are clearly defined, the double-fetch bug elimination can be applied in a transparent and fully automated manner.

6.3 Implementation of DropIt

To build DropIt, our instantiation of $\boldsymbol{\mathcal{P}3}$, we had to rely on real-world hardware transactional memory, namely Intel TSX. Intel TSX comes with a series of imperfections, inevitably introducing practical limitations for security mechanisms, as observed in previous work [29] (cf. Section 2.4).

However, as hardware transactional memory is exactly purposed to make multiple fetches from memory consistent, Intel TSX is sufficient for most real-world scenarios.

To eliminate double-fetch bugs, DropIt relies on the `XBEGIN` and `XEND` instructions of Intel TSX. `XBEGIN` specifies the start of a transaction as well as a fall-back path that is executed if the transaction aborts. `XEND` marks the successful completion of a transaction.

We find that on a typical Ubuntu Linux the kernel usually occupies less than 32 MB including all code, data, and heap used by the kernel and kernel modules. With an 8 MB L3 cache we could thus read or execute more than 20 % of the kernel without causing high abort rates [29] (cf. Section 2.4). In Section 7.4, we show that for practical use cases the abort rates are almost 0 % and our approach even improves the system call performance in several cases.

DropIt abstracts the transactional memory as well as the retry logic from the programmer. Hence, in contrast to existing software-based retry logic (cf. Section 6.1), e.g., in the Linux kernel, DropIt is mostly transparent to the programmer. To protect code, DropIt takes the number of automatic retries as a parameter as well as a fall-back function for the case that the transaction is never successful, *i.e.*, for the case of an ongoing attack. Hence, a programmer only has to add 3 lines of code to protect arbitrary code from double fetch exploitation. Listing 9.2 (Appendix E) shows an example how to protect the insecure `strcpy` function using DropIt. The solution with DropIt is clearly simpler than current state-of-the-art software-based retry logic (cf. Algorithm 2). Finally, replacing software-based retry logic by our hardware-assisted DropIt library can also improve the execution time of protected syscalls.

DropIt is implemented in standard C and does not have any dependencies. It can be used in user space, kernel space, and in trusted environments such as Intel SGX enclaves. If TSX is not available, DropIt immediately executes the fall-back function. This ensures that syscalls still work on older systems, while modern systems additionally benefit from possibly increased performance and elimination of yet unknown double-fetch bugs.

DropIt can be used for any code containing multiple fetches, regardless of whether they have been introduced on a source-code level or by the compiler. In case there is a particularly critical section in which a double fetch can cause harm, we can automatically protect it using DropIt. For example, this is possible for parts of syscalls that interact with the user space. As these parts are known to a compiler, a compiler can simply add the DropIt functions there.

DropIt is able to eliminate double-fetch bugs in most real-world scenarios. As Intel TSX is not an ideal implementation of hardware transactional memory, use of certain instructions in transactions is restricted, such as port I/O instructions [35]. However, double fetches are typically caused by string handling functions and do not rely on any restricted instructions. Especially in a trusted environment, such as Intel SGX enclaves, where I/O operations are not supported, all functions interacting with the host application can be automatically protected using DropIt. This is a particularly useful protection against an attacker in a cloud scenario, where an enclave containing an unknown double-fetch bug may be exposed to an attacker over an extended period of time.

7 Evaluation

The evaluation consists of four parts. The first part evaluates DECAF ($\boldsymbol{\mathcal{P}1}$ and $\boldsymbol{\mathcal{P}2}$), the second part compares $\boldsymbol{\mathcal{P}2}$ to state-of-the-art exploitation techniques, the third part evaluates $\boldsymbol{\mathcal{P}1}$ on trusted execution environments, and the fourth part evaluates DropIt ($\boldsymbol{\mathcal{P}3}$).

First, we demonstrate the proposed detection method using Flush+Reload. We evaluate the double-fetch detection of TrinityDECAF on both a recent Linux kernel 4.10 and an older Linux kernel 4.6 on Ubuntu 16.10 and discuss the results. We also evaluate the reliability of using Flush+Reload as a trigger in TrinityDECAF to exploit double-fetch bugs ($\boldsymbol{\mathcal{P}2}$). On Linux 4.6, we show that TrinityDECAF successfully finds and exploits CVE-2016-6516.

Second, we compare our double-fetch bug exploitation technique ($\boldsymbol{\mathcal{P}2}$) to state-of-the-art exploitation techniques. We show that $\boldsymbol{\mathcal{P}2}$ outperforms value-flipping as well as a highly optimized exploit crafted explicitly for one double-fetch bug. This underlines that $\boldsymbol{\mathcal{P}2}$ is both generic and extends the state of the art significantly.

Third, we evaluate the double-fetch detection ($\boldsymbol{\mathcal{P}1}$) on trusted execution environments, *i.e.*, Intel SGX and ARM TrustZone. We show that despite the isolation of those environments, we can still use our techniques to detect double fetches.

Fourth, we demonstrate the effectiveness of DropIt, our double-fetch bug elimination method ($\boldsymbol{\mathcal{P}3}$). We show that DropIt eliminates source-code-level double-fetch bugs with a very low overhead. Furthermore, we reproduce CVE-2015-8550, a compiler-introduced double-fetch bug. Based on this example we demonstrate that DropIt also eliminates double-fetch bugs which are not even visible in the source code. Finally, we measure the

performance of DropIt protecting 26 syscalls in the Linux kernel, where TrinityDECAF reported double fetches.

7.1 Evaluation of DECAF

To evaluate DECAF, we analyze the double fetches and double-fetch bugs reported by TrinityDECAF. Our goal here is not to fuzz an excessive amount of time, but to demonstrate that DECAF constitutes a sensible and practical complement to existing techniques. Hence, we also used old and stable kernels where we did not expect to find new bugs, but validate our approach.

Reported Double-Fetch Bugs Besides many double fetches TrinityDECAF reports in Linux kernel 4.6, it identifies one double-fetch bug which is already documented as CVE-2016-6516. It is a double-fetch bug in one of the `ioctl` calls. The syscall is used to share physical sections of two files if the content is identical.

When calling the syscall, the user provides a file descriptor for the source file as well as a starting offset and length within the source file. Furthermore, the syscall takes an arbitrary number of destination file descriptors with corresponding offsets and lengths. The kernel checks whether the given destination sections are identical to the source section and if this is the case, frees the sections and maps the source section into the destination file.

As the function allows for an arbitrary number of destination files, the user has to supply the number of provided destination files. This number is used to determine the amount of memory required to allocate. Listing 9.1 (Appendix C) shows the corresponding code from the Linux kernel. Changing the number between the allocation and the actual access to the data structure leads to a kernel heap-buffer overflow. Such an overflow can lead to a crash of the kernel or even worse to a privilege escalation.

Trinity already has rudimentary support for the `ioctl` syscall, which we extended with semi-intelligent defaults for parameter selection. Consequently, while Trinity does not find CVE-2016-6516, TrinityDECAF indeed successfully detects this double fetch in the profiling phase. Fig. 9.4 shows a cache trace while calling the vulnerable function on Ubuntu 16.04 running an affected kernel 4.6. Although the time between the two accesses is only 10 000 cycles (approximately 2.5 µs on our i7-6700K test machine), we can clearly detect the two memory accesses.

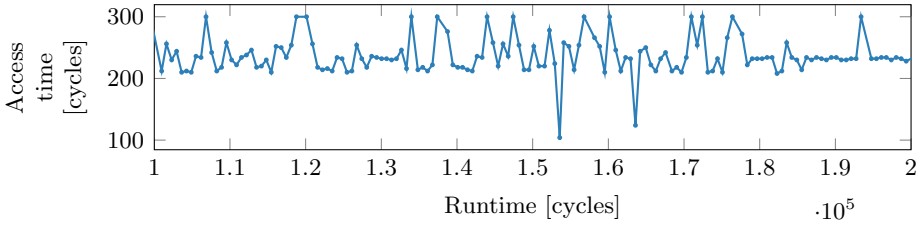


Figure 9.4: The two memory accesses of the `FIDEDUPERANGE` `ioctl` in Linux kernel 4.5 to 4.7 can be clearly seen at around $1.5 \cdot 10^5$ and $1.6 \cdot 10^5$ cycles.

When, in the exploitation phase, the monitoring thread changes the value to a higher value (cf. Section 5.2) exceeding the actual number of provided file descriptors, the kernel iterates out-of-bounds, as the number of file descriptors does not match the actual number of file descriptors anymore. This out-of-bounds access to the heap buffer results in a denial-of-service of the kernel and thus a hard reboot is required. Consequently, the denial-of-service shows that the double fetch is an exploitable double-fetch bug.

This demonstrates that DECAF is a useful complement to state-of-the-art fuzzing techniques, allowing to automatically detect bugs that cannot be found with traditional fuzzing approaches.

Reported Double Fetches Besides Linux kernel 4.6, we also tested TrinityDECAF on a recent Linux kernel 4.10. We let TrinityDECAF investigate all 64-bit syscalls (currently 295) without exceptions for one hour on an Intel i7-6700K. On average, every syscall was executed 8058 times. Due to the semi-intelligent parameter selection of TrinityDECAF, most syscalls are called with valid parameters. In our test run, 75.12 % of the syscalls executed successfully. Hence, on average, every syscall was successfully executed 6053 times, indicating a high code coverage for every syscall.

For every syscall parameter, TrinityDECAF displays a percentage of the calls where it detected a double fetch. Out of the 295 tested 64-bit syscalls, TrinityDECAF reported double fetches for 68 syscalls in Linux kernel 4.10. This is not surprising and in line with state-of-the-art work [70] which reported 90 double fetches in Linux, but only 33 in syscalls. For each of the reported syscalls, we investigated the respective implementation. Table 9.1 (Appendix A) shows a complete list of reported syscalls and the

reason why TrinityDECAF detected a double fetch. We can group the reported syscalls into 5 major categories, explaining the detected double fetch.

- **Filenames.** Most syscalls handling filenames (or paths) are reported by TrinityDECAF. Many of them use `getname_flags` internally to copy a filename to a kernel buffer. This function checks whether the filename is already cached in the kernel, and copies it to the kernel if this is not the case, resulting in multiple accesses to the file name. The exploitation phase automatically filtered out all non-exploitable double fetches in this category.
- **Shared input/output parameters.** We found 5 syscalls which are reported by TrinityDECAF although they do not contain a double fetch. In these syscalls, one of the syscall parameters was used as input and output. As reads and writes are not distinguishable through the cache access pattern (cf. Sect. 4.1), these syscalls are filtered out automatically in the exploitation phase.
- **Strings of arbitrary length.** As with filenames, some syscalls expect strings from the user that do not have a fixed length. To safely copy such arbitrary length strings, some syscalls (e.g., `mount`) use an algorithm similar to Algorithm 2. Thus, the detected double fetch is due to the length check and the subsequent string copy. The exploitation phase automatically filtered out all non-exploitable double fetches in this category.
- **Sanity check.** Many syscalls check—either directly, or in a subroutine—whether the supplied argument is sane. There are sanity checks that check whether it is safe to access a user-space pointer before actually copying data from or to it. Such a check can also trigger a cache hit if the value was actually accessed. All correct sanity checks were automatically filtered out in the exploitation phase. The exploitation phase correctly identified the `ioctl` syscall in the Linux kernel 4.6, but also correctly filtered it out in Linux kernel 4.10.
- **Structure elements.** If a syscall has a structure as parameter, double fetches can be falsely detected if structure members fall into the same cache line (cf. Sect. 4.1). If members are either copied element-wise or neighboring members are simply accessed, TrinityDECAF will detect a double fetch although two different variables are accessed. Again, these false positives are filtered out in the exploitation phase.

Our evaluation showed that TrinityDECAF provides a sensible complement to existing double-fetch bug detection techniques. The fact that we found only 1 exploitable double-fetch bug in 68 double fetches is not

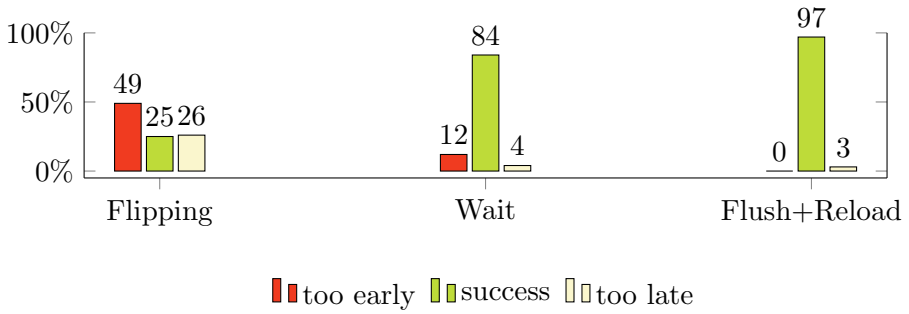


Figure 9.5: Comparing three exploits for CVE-2016-6516. Our Flush+Reload-based trigger in TrinityDECAF succeeds in 97 %, outperforming the provided proof-of-concept (84 %) and the state-of-the-art method of value flipping (25 %).

surprising, and in line with previous work, e.g., Wang et al. [70] found 3 exploitable double-fetch bugs by manually inspecting 90 double fetches they found. However, it also shows that the coverage of DECAF highly depends on the fuzzer used to instantiate it. Future work may retrofit other fuzzers with DECAF, to extend the spectrum of bugs that the fuzzer covers and thereby also extend the coverage of DECAF. Furthermore, as Trinity is continuously extended, the coverage of TrinityDECAF grows automatically with the coverage of Trinity.

7.2 Evaluation of $\mathcal{P}2$

To evaluate $\mathcal{P}2$ in detail, we compare three different variants to exploit the double-fetch bug reported in CVE-2016-6516.

First, the provided exploit, which calls `ioctl` multiple times, always changing the affected variable after a slightly increased delay. Second, we use state-of-the-art value flipping to switch the affected variable as fast as possible between the valid and an invalid value. Third, the automated approach $\mathcal{P}2$, integrated into TrinityDECAF.

Fig. 9.5 shows the success rate of 1000 executions of each of the three variants. Value flipping has by far the worst success rate, although in theory, it should have a success rate of approximately 50 %. In half of the cases, the value is flipped before the first access. Thus, the exploit fails, as the value is smaller at the next access. In the other cases, the probability to switch the value at the correct time is again 50 % resulting in an overall success rate of 25 %.

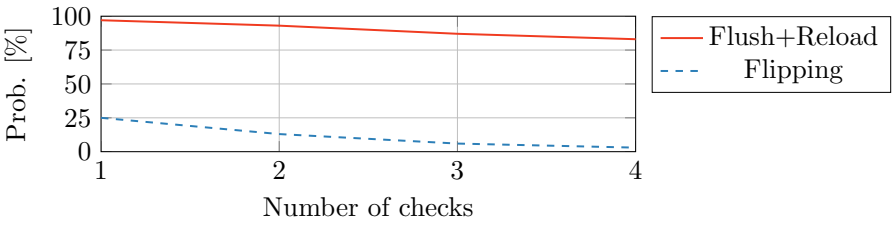


Figure 9.6: The probability of double-fetch bug exploitation decreases with the number of sanity checks as it only succeeds if the value changes between the last two accesses.

The original exploit is highly optimized for this specific vulnerability. It uses a trial-and-error busy wait with steadily increasing timeouts, which works surprisingly well, as there is sufficient time between the two accesses. Depending on the scheduling, the attacker sometimes sleeps too long (4 %) and sometimes too short (12 %). Still, the busy wait outperforms the value flipping in this scenario, increasing the success probability from 25 % to 84 %.

Even though our Flush+Reload-based trigger (***P*2**) is generic and does not require fine-tuning of the sleep intervals, it has the highest success rate. There is no case where the value was changed too early, as there are no false positives with Flush+Reload in this scenario. Furthermore, as the time between the two memory accesses is long enough, we achieve an almost perfect success rate of 97 %. The remaining 3 %, where we do not trigger a change of the value, are caused by unfortunate scheduling of the application.

The success rate of value flipping drops significantly if the two values have to fulfill specific constraints, e.g., the value has to be higher on the second access. For example, if an application does not only fetch the value twice, but multiple times for sanity checking, the probability of successfully exploiting it using value flipping decreases exponentially.

Fig. 9.6 shows the probability to exploit a double fetch similar to CVE-2016-6516, with additional fetches for sanity checks. To successfully exploit the vulnerability, the value has to be the same for all sanity checks and must be higher for the last access. Flipping the value between a valid and an invalid value decreases the chances by 50 % for every additional sanity check.

Our Flush+Reload-based method (***P*2**) does not suffer significantly from additional sanity checks. We can accurately trigger on the second to

last access to change the value. The slightly decreased probability is only due to missed accesses.

7.3 Evaluation of $\mathcal{P}1$ on Trusted Execution Environments

We evaluate $\boldsymbol{\mathcal{P}1}$ on trusted execution environments by successfully detecting double fetches in Intel SGX and ARM TrustZone.

Intel SGX Intel SGX allows running code in secure enclaves without trusting the user or the operating system. A program inside an enclave is not accessible to the operating system due to hardware isolation provided by SGX. Weichbrodt et al. [73] showed that synchronization bugs, such as double fetches, inside SGX enclaves, can be exploited to hijack the control flow or bypass access control.

As it has been shown recently, enclaves leak information through the last-level cache, even to unprivileged user space applications, as they share the last-level cache with regular user space applications [6, 27, 53, 64]. SGX enclaves provide a communication interface using so-called `ecall`s and `ocall`s, similar to the syscall interface. Enclaves fulfill the properties of Sect. 4.4, and we can thus detect double fetches within enclaves, even without access to the binary. Therefore, we can apply our method to identify double fetches within SGX enclaves.

To test our Flush+Reload detection mechanism ($\boldsymbol{\mathcal{P}1}$), we implemented a small enclave application. This application consists of only one `ecall`, which takes a memory reference as a parameter. As enclaves can access non-enclave memory, the user can simply allocate memory and provide the pointer to the enclave. The enclave accesses the memory once, idles a few thousand cycles and reaccesses the memory. Although the enclave should be isolated from other applications, the monitoring application can clearly detect the 2 cache hits. Fig. 9.11 (Appendix D) shows the measurement of the Flush+Reload thread running outside the enclave on an Intel i5-6200U. Similarly, Appendix D evaluates $\boldsymbol{\mathcal{P}1}$ on ARM TrustZone.

7.4 Evaluation of DropIt

To evaluate our open-source library DropIt, as an instantiation of $\boldsymbol{\mathcal{P}3}$, we investigate two real-world scenarios. In the first scenario, we demonstrate how DropIt eliminates a compiler-introduced real-world double-fetch bug in Xen (CVE-2015-8550). In the second scenario, we evaluate the effect of DropIt on Linux syscalls with double fetches. Our findings show that

DropIt successfully eliminates all double-fetch bugs and can be used as a preventative measure to protect double fetches in syscalls generically.

Eliminating Compiler-Introduced Double-Fetch Bugs As discussed in Section 2.3, compilers can also introduce double-fetch bugs. Especially switch statements are prone to double-fetch bugs if the variable is subject to a race condition [5, 52]. This is not an issue with the compiler, as the compiler is allowed to assume an atomic value for the switch condition [11]. We are aware of two scenarios where code generated by gcc contains a double-fetch bug.

If a switch is translated into a jump table with a default case, gcc generates two accesses to the switch variable. The first access checks whether the parameter is within the bounds of the jump table, the second access calculates the actual jump target. Thus, if the parameter changes between the accesses, a malicious user can divert the control flow of the program.

If the switch is implemented as multiple conditional jumps, the compiler is allowed to fetch the variable for every conditional jump. This leads to cases where the switch executes the default case as the variable changes while checking the conditions [52].

We evaluated DropIt on the real-world compiler-introduced double-fetch bug CVE-2015-8550. This vulnerability in Xen allowed arbitrary code execution due to a compiler-introduced double fetch within a switch statement. Note that such a switch statement is a common construct and can occur in any other kernel, e.g., Linux, or Windows, if a memory buffer is shared between user space and kernel space. Wrapping the switch statement using DropIt results in a clean and straightforward fix without relying on the compiler. With DropIt, any compiler-introduced switch-related double-fetch bug is successfully eliminated using only 3 lines of additional code.

To compare the overhead of traditional locking and DropIt, we implemented a minimal working example of a compiler-introduced double-fetch bug. Our example consists of a switch statement that has 5 different cases as well as a default case. The condition is a pointer which is subject to a race condition. The average execution time of the switch statement without any protection is 7.6 cycles. Using a spinlock to protect the variable increased the average execution time to 83.7 cycles. DropIt achieved a higher performance than the traditional spinlock with an average execution time of 68.0 cycles. Thus, DropIt is not only easy to deploy but also achieves a better performance than traditional locking mechanisms.

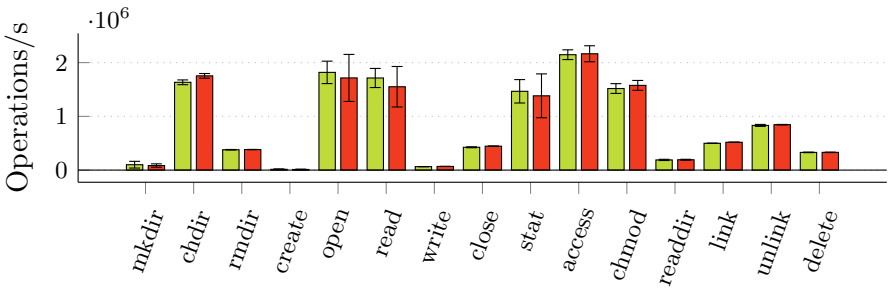


Figure 9.7: The number of executed file operations per second of our re-implemented `getname_flags` using DropIt (green) does not significantly differ from the version in the vanilla kernel (red) measured with the IOZone Fileops benchmark.

Preventative Protection of Linux Syscalls To show that DropIt provides an automated and transparent generic solution to eliminate double-fetch bugs, we also used DropIt in the Linux kernel. As discussed in Sect. 7.1, a majority of the double fetches we detected in the Linux kernel are due to the `getname_flags` function handling file names. We replaced this function with a straight-forward implementation protected by DropIt. With this small change, all double fetches previously reported in 26 syscalls were covered by DropIt, and thus all potential double-fetch bugs were eliminated.

To compare the performance of DropIt with the vanilla implementation, we executed 210 million file operations in both cases. All benchmarks were run on a bare metal kernel to reduce the impact of system noise. Fig. 9.7 shows the result of the IOzone filesystem benchmark [55]. On average, the benchmarks show a 0.8 % performance degradation on the tested file operations that are affected by our kernel change. In some cases, DropIt even has a better performance than the vanilla implementation. We therefore conclude that DropIt has no perceptible performance impact. The variances in the tests are probably due to the underlying hardware, *i.e.*, the SSDs on which we performed the file operations.

Thus, DropIt provides a reliable and straightforward way to cope with double-fetch bugs. It is easily integrable into existing C projects and does not negatively influence the performance compared to state-of-the-art solutions. Furthermore, it even increases the performance compared to traditional locking mechanisms. DropIt in SGX performs even better, since many operations interrupting TSX transactions are forbidden in SGX enclaves anyway.

8 Conclusion

In this paper, we proposed novel techniques to efficiently detect, exploit, and eliminate double-fetch bugs. We presented the first combination of state-of-the-art cache attacks with kernel-fuzzing techniques. This allowed us to find double fetches in a fully automated way. Furthermore, we presented the first fully automated reliable detection and exploitation of double-fetch bugs. By combining these two primitives, we built DECAF, a system to automatically find and exploit double-fetch bugs. DECAF is the first method that makes manual analysis of double fetches as in previous work superfluous. We show that cache-based triggers, as we use in DECAF, outperform state-of-the-art exploitation techniques significantly, leading to an exploitation success rate of up to 97 %.

DECAF constitutes a sensible complement to existing double-fetch detection techniques. Future work may retrofit more fuzzers with DECAF, extending the spectrum of bugs covered by fuzzers. Hence, double-fetch bugs do not require separate detection tools anymore, but testing for these bugs can now be a part of regular fuzzing. With continuously growing coverage of fuzzers, the covered search space for potential double-fetch bugs grows as well.

With DropIt, we leverage a newfound interaction between hardware transactional memory and double fetches, to completely eliminate double-fetch bugs. Furthermore, we showed that DropIt can be used in a fully automated manner to harden Intel SGX enclaves such that double-fetch bugs cannot be exploited. Finally, our evaluation of DropIt in the Linux kernel showed that it can be applied to large systems with a negligible performance overhead below 1 %.

Acknowledgments

We would like to thank our anonymous reviewers for their feedback. This work has been supported by the Austrian Research Promotion Agency (FFG), the Styrian Business Promotion Agency (SFG), the Carinthian Economic Promotion Fund (KWF) under grant number 862235 (DeSSnet) and has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

A TrinityDECAF and Detected Double Fetches

In this section, we show implementation details of TrinityDECAF (our augmented version of Trinity) as well as a complete list of syscalls reported by TrinityDECAF.

Figure 9.8 shows the process structure of our augmented version of Trinity, called *TrinityDECAF*. The syscall fuzzer Trinity is extended with one monitoring threads per syscall argument. Each of the monitoring threads mounts a Flush+Reload attack to detect double fetches (cf. Section 4.3).

Figure 9.9 shows the probability that TrinityDECAF detects a double fetch depending on the time between the two accesses to the memory (cf. Section 4.2)

Table 9.1 is a complete table of reported syscalls and the reason why TrinityDECAF detected a double fetch. The categories are discussed in detail in Section 7.1.

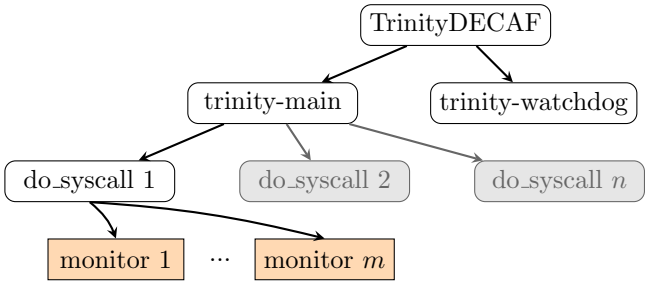


Figure 9.8: The structure of TrinityDECAF with the Flush+Reload monitoring threads for the syscall parameters.

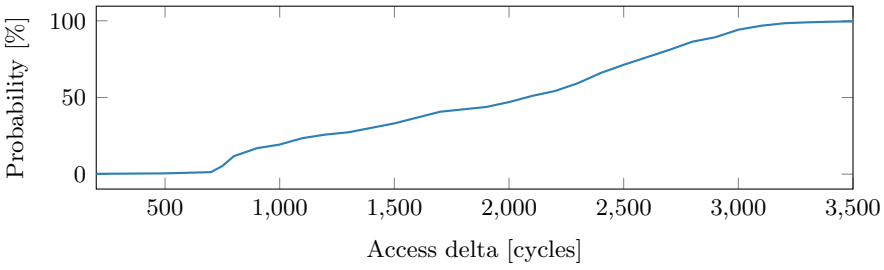


Figure 9.9: The probability of detecting a double fetch depending on the time between the accesses.

Table 9.1: Double fetches found by TrinityDECAF.

Category	Syscall
Filenames	open, newstat, truncate, chdir, rename(at), mkdir(at), rmdir, creat, unlink, link, symlink(at), readlink(at), chmod, (l)chown, utime, mknod, statfs, chroot, quotactl, *xattr, fchmodat
Shared input/output	sendfile, adjtimex, io_setup, recvmmsg, sendmmsg
Strings	mount, memfd_create
Sanity check	sched_setparam, ioctl, sched_setaffinity, io_cancel, sched_setscheduler, futimesat, sysctl, settimeofday, gettimeofday
Structure elements	recvmsg, msgsnd, sigaltstack, utime

B Safe String Copy

In this section, we show the pseudo-code of a standard algorithm used to safely copy an arbitrary-length string. Algorithm 2 first retrieves the length of the string, to allocate a buffer and copy the string up to this length. Then, it checks whether the string is terminated, and if not, retries again as the buffer was apparently changed before copying it.

A similar algorithm is used in the Linux kernel whenever user data of unknown length has to be copied to the kernel.

input : *string*
copy:
len ← strlen(string);
buffer ← allocate(len + 1);
strncpy(buffer, string, len);
if ***not*** *isNullTerminated(buffer)* **then**
  free(buffer);
  goto copy; // or abort with error if too many retries
end

Algorithm 2: Safe string copy for arbitrary string lengths with software-based retry logic.

C CVE-2016-6516

CVE-2016-6516 is a double-fetch bug in an `ioctl` call used to share physical sections of two files if the content is identical. This deduplicates the identical section to save physical storage. On a write access, the identical section has to be copied to ensure that the changes are only visible within the changed file.

The user provides a file descriptor for the source file as well as a starting offset and length within the source file. Additionally, the syscall takes an arbitrary number of destination file descriptors including offsets and lengths. The kernel maps the source section into the destination file if the given destination sections are identical to the source section.

The function supports an arbitrary number of destination files. Thus, the user has to supply the number of provided destination files, so that the kernel can determine the required amount of memory to allocate. Listing 9.1 shows the corresponding code from the Linux kernel. If the number changes between the allocation and the actual access to the data structure, the kernel accesses the buffer out-of-bounds, leading to a heap-buffer overflow.

```
1 // first access of dest_count
2 if (get_user(count, &argp->dest_count)) { [...] }
3 // allocation based on dest_count
4 size = offsetof(struct file_dedupe_range __user,
5   info[count]);
6 same = memdup_user(argp, size);
7 if (IS_ERR(same)) { [...] }
8 ret = vfs_dedupe_file_range(file, same);
9 // function accesses same->dest_count, not count
```

Listing 9.1: The vulnerable `ioctl_file_dedupe_range` function that was present in the Linux kernel from version 4.5 to 4.7. The `dest_count` member is accessed twice and can thus be changed between the accesses by a malicious user, leading to a kernel heap-buffer overflow.

D ARM TrustZone

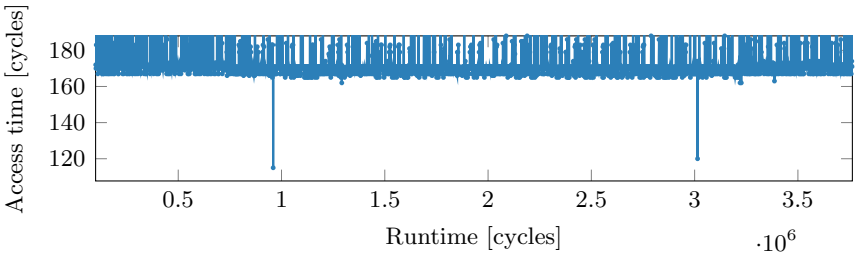


Figure 9.10: A double fetch of a trustlet running inside the ARM TrustZone of a Raspberry Pi 3. The cache hits can be clearly seen at around $0.96 \cdot 10^6$ and $3.01 \cdot 10^6$ cycles as the access time drops from >160 cycles to <120 cycles.

ARM TrustZone is a trusted execution environment for the ARM platform. The processor can either run in the normal world or the trusted world. As with Intel SGX, the worlds are isolated from each other using hardware mechanisms. Trustlets—applications running inside the secure world—provide a well-defined interface to normal world applications. This interface is accessed through a secure monitor call, similar to a syscall.

To use the ARM TrustZone, the normal-world operating system requires TrustZone support. Furthermore, a secure-world operating system has to run inside the TrustZone. For the evaluation, we used the TrustZone of a Raspberry Pi 3. We use the open-source trusted execution environ-

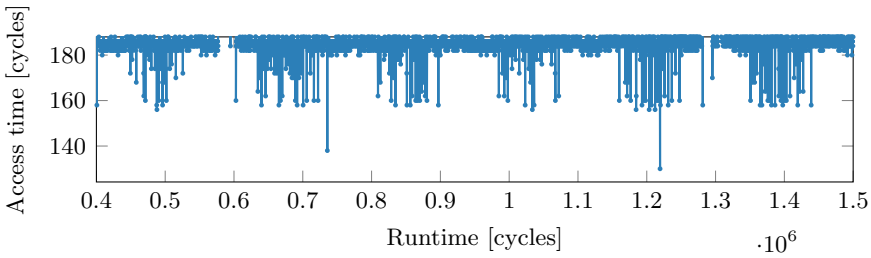


Figure 9.11: Monitoring a double fetch inside an SGX enclave. The cache hits can be clearly seen at around $0.75 \cdot 10^6$ and $1.21 \cdot 10^6$ cycles as the access time drops from >150 cycles to <140 cycles.

ment OP-TEE [43] as a secure-world operating system. The normal world runs a TrustZone-enabled Linux kernel.

As with Intel SGX (cf. Section 7.3), we again implement a trustlet providing a simple interface for receiving a pointer to a memory location. However, there are some subtle differences compared to the SGX enclave. First, trustlets are not allowed to simply access normal-world memory. To pass data or messages from normal world to secure world and vice versa, world shared memory is used, a region of non-secure memory, mapped both in the normal as well as in the secure world. With the world shared memory, we fulfill all criteria of Sect. 4.4.

On ARM, there are generally no unprivileged instructions to flush the cache or get a high-resolution timestamp [2]. However, they can be used from the operating system. Thus, in contrast to the double-fetch detection in syscalls or Intel SGX, we require root privileges to detect double fetches inside the TrustZone. This is not a real limitation, as we use the detection only for testing, and discovering bugs. An attacker using Flush+Reload as a trigger to exploit a double-fetch bug can rely on different time sources and eviction strategies as proposed by Lipp et al. [44].

Fig. 9.10 shows a recorded cache trace of the trustlet. Similarly to Figure 9.11, a trace from Intel SGX, the cache hits are clearly distinguishable from the cache misses. Thus, we can detect double-fetch bugs in trustlets, even without having access to the corresponding binaries.

E Example of DropIt

In this section, we show a small example of how to use DropIt. Listing 9.2 shows an example how to protect the insecure `strcpy` function using DropIt. A programmer only has to add 3 lines of code (highlighted in the

```
1  dropit_t config = dropit_init(1000);
2  dropit_start(config);
3  len = strlen(str); // First access
4  if (len < sizeof(buffer)) {
5    strcpy(buffer, str); // 2nd access,
6      // length of 'str' could have changed
7  } else {
8    printf("Too long!\n");
9  }
10 dropit_end(config, { printf("Fail!"); exit(-1);});
```

Listing 9.2: Using DropIt to protect a simple string copy containing a double-fetch bug from being exploited. Only the highlighted lines (1, 2, and 10) have to be added to the existing code to eliminate the double-fetch bug.

listing) to protect arbitrary code from double fetch exploitation. DropIt is clearly simpler than current state-of-the-art software-based retry logic (cf. Algorithm 2).

DropIt is implemented in standard C without any dependencies on other libraries, and can thus be used in user space, kernel space, as well as in trusted execution environments (e.g., Intel SGX). If Intel TSX is not available, DropIt has the possibility to execute a fall-back function instead.

References

[1] Tiago Alves. *Trustzone: Integrated hardware and software security.* 2004.

[2] ARM Limited. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition.* ARM Limited, 2012.

[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft." In: *International Conference on Integrated Formal Methods.* Springer. 2004.

[4] Matt Bishop, Michael Dilger, et al. "Checking for race conditions in file accesses." In: *Computing systems* 2.2 (1996).

[5] Marc Blanchou. "Shattering Illusions in Lock-Free Worlds – Compiler and Hardware behaviors in OSes and VMs." In: *Black Hat Briefings.* 2013.

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *WOOT.* 2017.

[7] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. "EXE: automatically generating inputs of death." In: *ACM Transactions on Information and System Security (TISSEC)* (2008).

[8] Xiang Cai, Yuwei Gui, and Rob Johnson. "Exploiting UNIX file-system races via algorithmic complexity attacks." In: *S&P.* 2009.

[9] George J Carrette. *CRASHME: Random input testing.* 1996.

[10] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu." In: *AsiaCCS.* 2017.

[11] ANSI Technical Committee, ISO/IEC JTC 1 Working Group, et al. *Rationale for international standard, Programming languages - C.* 1999.

[12] Bogdan Copos and Praveen Murthy. "Inputfinder: Reverse engineering closed binaries using hardware performance counters." In: *5th Program Protection and Reverse Engineering Workshop.* 2015.

[13] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. "RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities." In: *USENIX Security Symposium.* 2001.

[14] M Eddington. *Peach fuzzer*. 2008.

[15] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. "Effective Data-race Detection for the Kernel." In: *OSDI*. 2010.

[16] Cesare Ferri, Ruth Iris Bahar, Mirko Loghi, and Massimo Poncino. "Energy-optimal synchronization primitives for single-chip multiprocessors." In: *19th ACM Great Lakes symposium on VLSI*. 2009.

[17] Cormac Flanagan and Stephen N. Freund. "FastTrack: Efficient and Precise Dynamic Race Detection." In: *Commun. ACM* (2010).

[18] Justin E Forrester and Barton P Miller. "An empirical study of the robustness of Windows NT applications using random testing." In: *4th USENIX Windows System Symposium*. 2000.

[19] Vijay Ganesh, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing." In: *31st International Conference on Software Engineering*. 2009.

[20] Amaury Gauthier, Clément Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. "Enhancing fuzzing technique for OKL4 syscalls testing." In: *Sixth International Conference on Availability, Reliability and Security (ARES)*. 2011.

[21] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2016).

[22] Patrice Godefroid. "Random testing for security: blackbox vs. whitebox fuzzing." In: *2nd International Workshop on Random Testing*. 2007.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *ACM Sigplan Notices*. Vol. 40. 6. 2005, pp. 213–223.

[24] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing." In: *ACM Sigplan Notices*. Vol. 43. 6. 2008, pp. 206–215.

[25] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." In: *Queue* 10.1 (2012), p. 20.

[26] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A McKee, and Per Stenstrom. "Performance and energy analysis of the restricted transactional memory implementation on haswell." In: *IEEE IPDPS*. 2014.

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX." In: *EuroSec*. 2017.

[28] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[29] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory." In: *USENIX Security Symposium*. 2017.

[30] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. "Protecting private keys against memory disclosure attacks using hardware transactional memory." In: *S&P*. 2015.

[31] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *S&P*. 2011.

[32] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations." In: *USENIX Security Symposium*. 2013.

[33] Tim Harris, James Larus, and Ravi Rajwar. "Transactional memory." In: *Synthesis Lectures on Computer Architecture* 5.1 (2010), pp. 1–263.

[34] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2017.

[35] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. 2012.

[36] Moritz Jodeit and Martin Johns. "Usb device drivers: A stepping stone into your kernel." In: *IEEE European Conference on Computer Network Defense*. 2010.

[37] Dave Jones. "Trinity: A system call fuzzer." In: *13th Ottawa Linux Symposium*. 2011.

[38] Mateusz Jurczyk, Gynvael Coldwind, et al. *Identifying and exploiting windows kernel race conditions via memory access patterns*. 2013.

[39] Ulf Kargén and Nahid Shahmehri. "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing." In: *10th Joint Meeting on Foundations of Software Engineering*. 2015.

[40] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. "Comparing operating systems using robustness benchmarks." In: *16th Symposium on Reliable Distributed Systems*. 1997.

[41] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "HAFT: Hardware-assisted fault tolerance." In: *EuroSys*. 2016.

[42] Kyung-Suk Lhee and Steve J Chapin. "Detection of file-based race conditions." In: *International Journal of Information Security* 4.1 (2005).

[43] Linaro. *OP-TEE: Open Portable Trusted Execution Environment*. 2016. URL: `https://www.op-tee.org/`.

[44] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. 2016.

[45] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. "Concurrent and consistent virtual machine introspection with hardware transactional memory." In: *High Performance Computer Architecture (HPCA)*. 2014.

[46] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. "Testing system virtual machines." In: *19th International Symposium on Software Testing and Analysis*. 2010.

[47] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA*. 2013.

[48] Manuel Mendonça and Nuno Neves. "Fuzzing wi-fi drivers to locate security vulnerabilities." In: *IEEE EDCC*. 2008.

[49] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities." In: *Communications of the ACM* (1990).

[50] Barton P Miller, Gregory Cooksey, and Fredrick Moore. "An empirical study of the robustness of macos applications using random testing." In: *1st International Workshop on Random Testing*. 2006.

[51] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Tech. rep. University of Wisconsin, 1995.

[52] MITRE. *CWE-365: Race Condition in Switch*. 2017. URL: `https://cwe.mitre.org/data/definitions/365.html`.

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies The Power of Cache Attacks." In: *CHES*. 2017.

[54] Nikita Komarov. *Racehound: Data race detector for Linux kernel modules*. 2015. URL: `https://github.com/winnukem/racehound`.

[55] William D Norcott and Don Capps. *IOzone filesystem benchmark*. 2016. URL: `http://www.iozone.org/`.

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA*. 2006.

[57] Mathias Payer and Thomas R Gross. "Protecting applications against TOCTTOU races by user-space caching of file metadata." In: *ACM SIGPLAN Notices*. 2012.

[58] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. "Operating system transactions." In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009.

[59] Eli Pozniansky and Assaf Schuster. "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs." In: *PPoPP*. 2003.

[60] Eli Pozniansky and Assaf Schuster. "MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles." In: *Concurr. Comput. : Pract. Exper.* (2007).

[61] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "Vuzzer: Application-aware evolutionary fuzzing." In: *NDSS*. 2017.

[62] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." In: *ACM Transactions on Computer Systems* (1997).

[63] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. "Model checking an entire linux distribution for security violations." In: *Computer Security Applications Conference, 21st Annual*. 2005.

[64] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *DIMVA*. 2017.

[65] Fermin J Serna. *MS08-061 : The case of the kernel mode double-fetch*. 2008. URL: `https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/`.

[66] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating controlled-channel attacks against enclave programs." In: *NDSS*. 2017.

[67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. 2016.

[68] Prem Uppuluri, Uday Joshi, and Arnab Ray. "Preventing race condition attacks on file-systems." In: *ACM Symposium on Applied Computing*. 2005.

[69] Dmitry Vyukov. *syzkaller - linux syscall fuzzer*. 2016. URL: `https://github.com/google/syzkaller`.

[70] Pengfei Wang and Jens Krinke. "How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel." In: *USENIX Security Symposium*. 2017.

[71] Vincent M Weaver and Dave Jones. *perf fuzzer: Targeted fuzzing of the perf_event_open() system call*. Tech. rep. Technical Report, University of Maine, 2015.

[72] Jinpeng Wei and Calton Pu. "TOCTTOU Vulnerabilities in UNIX-style File Systems: An Anatomical Study." In: *FAST*. 2005.

[73] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves." In: *ESORICS*. 2016.

[74] Xinwei Xie and Jingling Xue. "Acculock: Accurate and Efficient Detection of Data Races." In: *CGO*. 2011.

[75] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels." In: *S&P*. 2018.

[76] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *S&P*. May 2015.

[77] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

[78] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing." In: *International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013.

[79] Yuan Yu, Tom Rodeheffer, and Wei Chen. "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking." In: *SOSP*. 2005.

[80] Georgios Zacharopoulos. *Employing Hardware Transactional Memory in Prefetching for Energy Efficiency*. Examensarbete, Uppsala Universitet. 2015.

10

JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits

Publication Data

Michael Schwarz, Florian Lackner, and Daniel Gruss. "JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits." In: *NDSS*. 2019

Contributions

Main author.

JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits

Michael Schwarz, Florian Lackner, Daniel Gruss

Graz University of Technology, Austria

Abstract

Today, more and more web browsers and extensions provide anonymity features to hide user details. Primarily used to evade tracking by websites and advertisements, these features are also used by criminals to prevent identification. Thus, not only tracking companies but also law-enforcement agencies have an interest in finding flaws which break these anonymity features. For instance, for targeted exploitation using zero days, it is essential to have as much information about the target as possible. A failed exploitation attempt, e.g., due to a wrongly guessed operating system, can burn the zero-day, effectively costing the attacker money. Also for side-channel attacks, it is of the utmost importance to know certain aspects of the victim's hardware configuration, e.g., the instruction-set architecture. Moreover, knowledge about specific environmental properties, such as the operating system, allows crafting more plausible dialogues for phishing attacks.

In this paper, we present a fully automated approach to find subtle differences in browser engines caused by the environment. Furthermore, we present two new side-channel attacks on browser engines to detect the instruction-set architecture and the used memory allocator. Using these differences, we can deduce information about the system, both about the software as well as the hardware. As a result, we cannot only ease the creation of fingerprints, but we gain the advantage of having a more precise picture for targeted exploitation. Our approach allows automating the cumbersome manual search for such differences. We collect all data available to the JavaScript engine and build templates from these properties. If a property of such a template stays the same on one system but differs on a different system, we found an environment-dependent property.

We found environment-dependent properties in Firefox, Chrome, Edge, and mobile Tor, allowing us to reveal the underlying operating system, CPU architecture, used privacy-enhancing plugins, as well as exact browser version. We stress that our method should be used in the development

of browsers and privacy extensions to automatically find flaws in the implementation.

1 Introduction

Today, more than half of the world's population is connected to the internet [35]. Regardless of whether people use websites from a computer or a smartphone, they require a web browser to do so. Most web browsers follow the standards defined by the World Wide Web Consortium (W3C), an international organization responsible for standards concerning the world wide web. Although the standards define many aspects of how websites are rendered and how they behave, they do not define everything on the implementation level.

As a consequence, implementation details differ significantly between different browsers. The differences can be found in supported standardized features, browser-specific features, as well as aspects which are undefined according to the standard [18]. With JavaScript, a scripting language supported by all modern browsers, websites can gather information about the concrete implementation of the browser. Furthermore, JavaScript allows to obtain details about the host system, e.g., the screen resolution, operating system, installed plugins. This can be used to adapt a website to the specific properties of a user's device and environment, providing an optimal user experience. However, the amount of information available to a website can also be abused to create a fingerprint consisting of a set of properties. Such a fingerprint can be used to uniquely identify a browser, and therefore a user, across multiple sessions and even across webpages [19, 38, 50].

Browsers aiming at the protection of the privacy of the user, such as the Tor browser, try to prevent fingerprinting. They do so by removing differences caused by the browser as well as the environment. They also block functionality such as Canvas elements [57]. There are also approaches to prevent fingerprinting by adding randomness instead of removing functionality [39]. The aim is always to prevent the creation of unique fingerprints of a browser and thus also user.

There are many legitimate reasons to prevent tracking and identification, and for certain groups, such as journalists or whistleblowers, it is in many cases even vital. However, for criminal actors, it is undoubtedly also beneficial to prevent tracking and unique identification. Thus, browsers

The original publication is available at `https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01B-4_Schwarz_paper.pdf`.

such as the Tor browser are also heavily used for criminal activities [17, 60]. The anti-fingerprinting methods ensure that users cannot be tracked across websites, preventing deanonymization through the user's usage pattern of websites [57]. Thus, attackers trying to reveal the identity of such users cannot rely on simply tracking a user with fingerprinting.

However, an attacker does not necessarily want to uniquely identify a user for the purpose of tracking. For an attacker, it might be even more desirable to gather as much information about the environment as possible to mount a targeted attack [50]. Especially for nation-state actors or law-enforcement agencies, it can already be advantageous if only some information is known about a user. Information fragments can then be used to, e.g., link a suspect to a browser session, or mount a targeted exploit on the user.

In this paper, we propose a method to automate the search for data leakage which reveals information about the user's environment. To automate the leakage detection, we build so-called templates over properties in different environments. A property can be anything which can be read by JavaScript. Multiple runs on one system reveal unstable properties, resulting in a deterministic set of static properties for a specific environment. We analyzed all unstable properties and show that in most cases they do not provide any reliable information about the environment. We also show how our method can be extended to the unstable properties that can be used for fingerprinting. The JavaScript property template we obtain allows us to match a specific target system to one of the environments in our template. Hence, an attacker can deduce what the environment of the target system is, and thus, which attacks can be mounted.

It is well known that law-enforcement agencies actively try to deanonymize Tor users [16, 24, 61, 78]. Various exploits have already been used to do this, some of which were discovered later on by researchers. The exploits are usually zero-day exploits mainly targetting users with Windows operating system [22, 78]. However, exploits are not limited to zero-day exploits.

Nowadays, there is a repertoire of powerful, software-based side-channel attacks. These side-channel attacks exploit various microarchitectural elements, most prominently caches [33, 55, 56, 81], DRAM [58], or branch prediction [2, 3, 21]. Side-channel attacks are not only able to break cryptographic algorithms [5, 37, 64], but are even able to read arbitrary memory contents [8, 36, 41, 74, 80]. With powerful side-channel attacks, it is plausible that nation-state actors also use side-channel attacks to de-anonymize Tor users.

Although some side-channel attacks can be mounted directly from the browser [23, 25, 26, 31, 32, 36, 43, 62] or even remotely [42, 65, 71], many powerful side-channel attacks require native code execution. Both zero-day exploits, as well as side-channel attacks, require knowledge of the attacked system. Trying to use an attack for a system which is not affected might draw attention to the exploit, and worse, might even leak a zero-day to the public, rendering it useless for future attacks.

Hence, there is an arms race between browser vendors emphasizing on the privacy of the user (e.g., Tor), and attackers and tracking companies trying to learn as much about the system as possible. Attackers try to find new ways to leak information which browser vendors prevent as soon as they become public. This requires considerable effort on both sides. Thus, both parties have an interest in automating this approach. Automated leakage detection has already been used to detect leakage from the cache [29], memory accesses [79], procfs pseudo-file system [68], and Android API [69].

Our fully-automated approach we propose can replace the tedious work of identifying such properties manually. As it is easily integrated into the development and testing chain, it will allow providing strong guarantees for this security and privacy aspect of modern browsers.

Furthermore, we present two new side-channel attacks which can be mounted from JavaScript. They allow an attacker to reveal the instruction-set architecture and the memory allocator. Both properties are essential aspects of both side-channel attacks as well as traditional zero-day exploits.

Contributions. The contributions of this work are:

1. We are the first to propose a fully-automated method to identify browser properties which can be used for fingerprinting.
2. We show that we can deduce information about the host system even in browsers employing anti-fingerprinting techniques.
3. We present two new side-channel attacks in JavaScript to deduce further information about the host system.
4. We show that privacy-enhancing browser extension can leak more information than they disguise and can even be semi-automatically circumvented, leading to a false sense of security.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background information on browser fingerprinting and side-channel attacks. In Section 3, we present our fully-automated method to find leakage from browser properties. In Section 4, we present two novel side-channel attacks to deduce information about the environment. In

Section 5, we apply the method to real-world scenarios and discuss the detected properties which are useful for targeted attacks and fingerprinting. In Section 6, we analyze the coverage we reach with our approach. In Section 7, we discuss the limitations of the approach. We conclude in Section 8.

2 Background and Related Work

In this section, we provide background about state-of-the-art browser fingerprinting and anti-fingerprinting mechanisms employed in current browsers. Furthermore, we also discuss related work which aims to automatically detect leakage in similar scenarios and give a short overview of side-channel attacks in JavaScript.

2.1 Browser Fingerprinting

Browser fingerprinting tries to uniquely identify a user across multiple webpages or visits to the same webpage without storing information in the browser. Thus, browser fingerprinting does not rely on classical tracking mechanisms such as cookies, making it hard for a user to prevent tracking.

Fingerprinting is usually done via a script which is executed when a user visits a website. This script collects several properties of the browser, such as the browser version, operating system, screen resolution, or installed plugins. While each of the properties itself does not allow tracking of a user, the combination of properties is unique enough to identify a user [19, 38].

There are many properties that can be used to fingerprint users. These properties include fonts [6, 50], plugins [50], rendering differences [9, 70], the battery status [53], and audio processing [20].

2.2 Anti-Fingerprinting Mechanisms

To prevent the tracking and identification of users, several software- and research projects try to minimize the fingerprinting surface. There are mainly two approaches to accomplish this goal.

First, some applications, such as, e.g., the Tor browser [57], hide the actual values of properties by either not exposing them or replacing them with the same value on all platforms. The Tor browser tries to prevent fingerprinting attempts which rely on browser properties that can be retrieved using JavaScript, plugins, or CSS [57]. Thus, the fingerprint of all Tor browsers in their default configuration is supposed to be the same.

There are also browser extensions for hiding values of properties as well as complete functionality that can be used for fingerprinting from a website. Such extensions include, e.g., Canvas Defender or the WebAPI Manager [66].

Second, some applications, such as, e.g., the FPRandom browser [39] or PriVaricator [51], try to break the stability of fingerprints by randomizing properties. FP-Block [72] spoofs properties in a way that they are the same for subsequent visits to one site, but differ between domains to prevent cross-domain tracking.

However, anti-fingerprinting mechanisms can be detected through additional, missing, or inconsistent values they create [1, 19, 46, 50].

2.3 Microarchitectural Attacks

Microarchitectural attacks have recently gained a lot of attention. Typically they are timing attacks that exploit the behavior of the microarchitecture, e.g., caches, branch predictors, or DRAM. The cache, in particular, was exploited in many attacks over the past years, leading to different attack techniques. Osvik et al. [55] described Evict+Time, where the attacker measures the influence of evicting a cache set on the runtime of an algorithm run by the victim, and Prime+Probe, where the attacker continuously measures whether the victim evicted a cache line in a specific cache set. Yuval and Falkner [81] described Flush+Reload, where the attacker continuously measures whether the victim reloaded a cache line. Several variations of these attacks were proposed, e.g., Flush+Flush [30], Evict+Reload [29, 40]. The recently discovered Meltdown [8, 41, 74, 80] and Spectre [36] attacks are significantly more powerful microarchitectural attacks. In some cases, they can infer values from arbitrary memory locations from other contexts, e.g., other processes or the operating system kernel.

2.4 Microarchitectural and Side-Channel Attacks in JavaScript

Although microarchitectural attacks exploit effects on a very low level of the CPU, they can even be exploited from JavaScript. In contrast to native code, JavaScript code is sandboxed and less powerful in terms of multithreading. Thus, there are several challenges an attacker has to overcome [63].

Still, many microarchitectural properties can be inferred from JavaScript [23, 25, 26, 31, 36, 43, 54, 62]. Moreover, sensors found on many

mobile devices as well as modern browsers, introduce side channels which can be exploited from JavaScript [44, 52, 67]. It has also been shown that microarchitectual properties can be used for fingerprinting [46].

2.5 Template-based Leakage Detection

Chari et al. [10] introduced template attacks as a strong form of side-channel attacks. They first collect side-channel traces from an attacker-controlled device, the so-called template. Then, they collect a single trace from an identical device processing an unknown secret. The unknown secret can then be recovered by comparing the trace to the recorded templates.

Brumley and Hakala [7] applied template attacks to cache-based timing attacks. They rely on Prime+Probe to automatically detect and exploit cache leakage. However, their method is limited to an attacker who runs on the same CPU core as the victim. Gruss et al. [29] demonstrated a Flush+Reload-based template attack to detect and exploit cache leakage automatically. As their attack leverages the shared last-level cache, it does not rely on the attacker's ability to run on the same core as the victim. Weiser et al. [79] dynamically instrumented binaries to generate templates consisting of all memory access. By comparing templates for different secret inputs, they can automatically detect whether the binary contains secret-dependent memory accesses.

On a higher level, Spreitzer et al. used template attacks on Android to infer application launches and visited websites via the procfs pseudo-file system [68] as well as the Android API [69]. For both approaches, they first create a template by gathering all available information from the proc file system [68] or Android API [69]. In the analysis phase, they compare templates gathered from different applications to classify application launches and fingerprint websites based on the templates.

3 JavaScript Template Attacks

JavaScript Template Attacks can automatically identify language features of JavaScript that leak information about the environment, e.g., the operating system or hardware. For this purpose, they leverage the well-known concept of template attacks (cf. Section 2.5) and apply it to JavaScript. As with all template attacks, JavaScript Template Attacks detect leakage through template differences caused by a secret. For JavaScript, the secret is the environment of the website, *i.e.*, the browser,

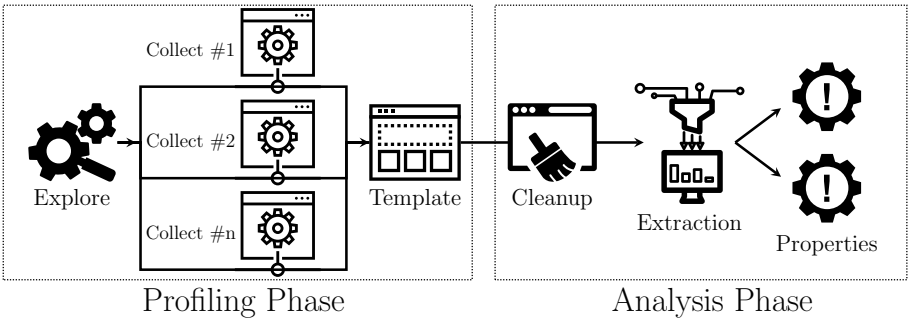


Figure 10.1: A JavaScript Template Attack consists of two phases. In the profiling phase, all available properties of a browser are collected multiple times. In the analysis phase, the template is pruned by removing duplicates and changing values. The resulting identifed properties leak properties about certain aspects of the environment.

operating system, and underlying hardware. A template is a matrix of properties (rows) for various environments (columns). All properties, e.g., browser properties, are retrieved through JavaScript.

Finding leakage is equivalent to detecting differences in these collected properties of the templates. The advantage of template attacks is that it is not necessary to understand the cause of the information leak. Hence, the template attack works fully automated. If the template contains different properties for different environments, our attack can deduce information about the (inaccessible) environment. This information can then be used by an attacker to mount a targeted exploit.

Our attack works in two phases which are outlined in Figure 10.1. The first phase is the *profiling phase*, which creates several profiles by collecting a set of properties, which are accessible via JavaScript, in different environments. These profiles are then combined to a template. In the *analysis phase*, we compare the properties of templates to automatically find differences caused by the environment. These discovered differences leak information about the environment which can be used on any webpage to mount a targeted attack.

3.1 Profiling Phase

The first phase is the profiling phase which builds the templates consisting of multiple profiles. The profiling phase runs entirely inside the browser and is implemented in JavaScript.

```
1 function getProperties(o) {
2   var result = [];
3
4   while(o !== null) {
5     result = result.concat(Reflect.ownKeys(o));
6     o = Object.getPrototypeOf(o);
7   }
8   return result;
9 }
```

Listing 3.1: Using reflections on all objects of the prototype chain results in a list of property names defined either directly in the object or inherited from an object on the prototype chain.

As a first step, the profiling code creates a list of properties which are accessible from JavaScript. In JavaScript, the accessible properties are either functions, numbers, strings, booleans, arrays, or objects. We refer to numbers, strings, and booleans as *primitive types*, as they have a single value which can directly be accessed and read. Objects and arrays (which are only a special type of object) are *complex types*, as they do not have a single generically comparable value. Instead, they are comprised of multiple primitive types and possibly further complex types.

Functions are more complex and require at least a certain understanding of the semantics to invoke them. This is an orthogonal problem [34], and thus, the properties that are returned by function calls are subject to future work. Solving this problem also allows applying JavaScript Template Attacks trivially to properties returned by functions. Even though we do not evaluate functions, we can still leverage functions for the templates. First, functions itself have a set of properties, e.g., `name` or `length`. Second, with *artificial properties*, we describe a way to add custom properties to the profiling phase. This allows us to convert simple functions, e.g., the `toString` function, into properties.

We distinguish between *native properties*, which are defined by the language or the browser, and *artificial properties* which can be added manually or automatically before the profiling phase.

- **Native Properties.** Native properties are primitive or complex types which are defined either by the language, *i.e.*, in the ECMAScript standard, or by the browser. Examples include the `length` property of almost every object or the `document` property of the `window` object. Moreover, browsers often introduce own properties to support features which are not yet standardized, or which aid

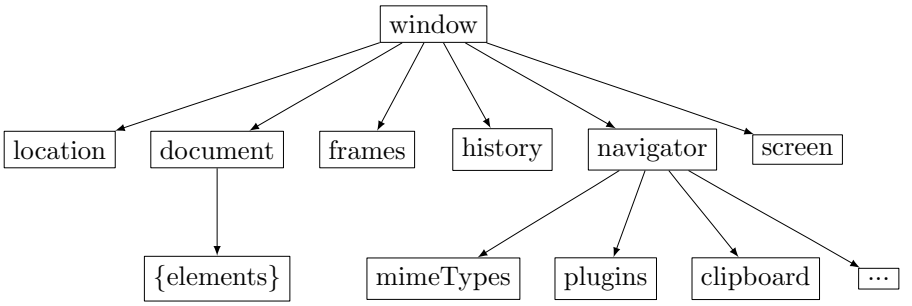


Figure 10.2: In the JavaScript object hierarchy, every object is derived from `Object`. The `window` object is the root of all accessible objects and thus, for JavaScript Template Attacks.

developers in the debugging process of web applications. Examples include the `window.chrome` property in Google Chrome or the `window.sidebar` property in Mozilla Firefox.

- **Artificial Properties.** We introduce the term *artificial properties* for properties which are typically not available in JavaScript. As JavaScript allows adding properties dynamically to any object, additional properties can be added to the profiled objects. These additional properties can, for example, be results of preceding function calls.

  Moreover, accessor properties can be added to the profiled objects. These properties are actually functions, as they do not return a static value but the result of a function. In contrast to functions, these properties do not support arguments. Thus, functions without arguments (e.g., `toString`) can be converted to artificial properties, allowing them to be used in the profiling phase.

Exploration Step

The first step of the profiling phase is to explore the list of all accessible properties. We leverage both reflections and the JavaScript functionality of iterating through properties of an object. Listing 3.1 shows our method to collect all properties from a given object. The properties include both inherited properties, which are not defined directly in the object but in the prototype chain, and non-inherited properties.

The goal is to identify as many properties as possible. There is no list of all available objects which can be used in the exploration step. However, in JavaScript, objects are linked with each other in so-called prototype

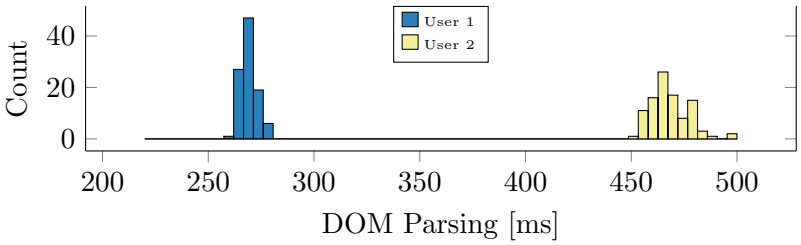


Figure 10.3: The histogram of non-static properties (e.g., the DOM parsing time) can be used to, e.g., create fingerprints.

chains. This is similar to class inheritance in other languages such as C++. Thus, from an arbitrary object, we can traverse all child elements and all parent elements. The root object of every object is `Object`.

Furthermore, JavaScript has an object hierarchy as illustrated in Figure 10.2. Accessible objects (e.g., global objects, functions, HTML DOM) are referenced in the `window` object (representing the browser window), or in one of its child elements. Hence, by starting the property exploration step at the `window` object, we reach all accessible properties. The result of the exploration step is a list of accessible properties.

The exploration step has to run only once per environment, as the set of properties is static and does not change.

Collection Step

During the collection step, the JavaScript code creates a profile consisting of the properties identified in the exploration step and their values. The collection step runs again inside the browser in JavaScript.

Our property collection algorithm takes a list of properties which were identified in the exploration step. For every property, the collection step acquires the actual value of the property. As we only considered properties which have a concrete value (e.g., no property which first requires a function to be called), we can directly read the value of every such property. Note that this step is not limited to properties with concrete values, as adding properties resulting from function calls works the same if there is a way to call functions in an automated way. We refer to the set of collected properties as a profile. Combining profiles by running the collection step in different environments results in a *template*.

The template still contains properties which are not useful in the further analysis (cf. Section 3.2), as they are not static. Examples include the page load time or the render time. These values change every time the

page is reloaded. Exploiting such properties requires an understanding of the semantics of the values which is an orthogonal problem. Although semantics could theoretically be inferred using machine learning, our manual investigations already showed that these non-static properties did not contain any information we deemed usable for deducing environment information. Thus, we focussed on the more interesting static properties. For fingerprinting, non-static properties might still be useful and can be exploited by collecting histograms of the values which can then be matched to single users (cf. Figure 10.3).

To later on detect which properties are not static (cf. Section 3.2), *i.e.*, which properties do not have the same value on every read, the collection step needs to run multiple times. Every run collects the same properties and the hash of the corresponding object. Thus, after multiple runs (typically 3 to 4), there is a list of values for every property from the exploration step, composing the profile.

The profile is finally transmitted to the back-end server (e.g., using AJAX) for incorporation into the template used for further analysis.

3.2 Analysis Phase

The analysis phase is an offline phase which finds the properties leaking information about the environment. In contrast to the profiling phase, this second phase of the templating process does not run inside the browser.

The input to the analysis is the template generated in the profiling phase. Depending on the profiles contained in the template, the analysis phase can detect properties leaking different aspects of the environment. For example, if all profiles are recorded with the same browser on different operating systems, the analysis phase detects properties leaking the operating system.

The analysis phase is also split into two steps, the cleanup step and the property extraction step.

Cleanup Step

In the first step, the template has to be cleaned. Profiles collected in the profiling phase often contain duplicate properties. There are multiple reasons for this.

First, JavaScript objects are often heavily linked to other objects. This creates entries in the profile which appear to have a different name but are the same properties. For example, `window.frames.window.name` is the same property as `window.name`. These properties are detected if the

objects have the same hash (which was stored in the collection phase), and are then unified.

Second, due to our method of collecting all properties (cf. Listing 3.1), the same property for one object might be collected multiple times. As we iterate through the entire prototype chain, we might get properties which are already overwritten by the child object. For example, the `name` property is collected for every object in the prototype chain. However, we can only access the `name` property of the last child, as it overwrites this property for all other objects in the prototype chain. These properties are trivial to remove as they have exactly the same name.

After the pruning of duplicates, the cleanup step has to identify properties which are not static, *i.e.*, properties which have changing values on different reads. For the collected values of every property, we test whether all the values are identical. If at least one of the values is different, we do not consider this property further. For example, the timestamp when the page was fully loaded (`window.performance.timing.responseEnd`) differs between multiple runs of the collection step. Although this property contains information about the environment, we cannot use it in an automated manner, as our automated method does not understand the semantics of properties (*i.e.*, that this is a timestamp).

In all observed cases, it was sufficient to run the collection step 3 to 4 times to filter out non-static properties in the cleanup step.

Property Extraction Step

Using the cleaned template, the property extraction step identifies properties which leak information. In this step, we first create the union of all properties from all profiles of the template. This is necessary, as in many cases not all properties are present in all profiles.

For every property in the unified property list, the collected values in the different profiles are compared. If a property has the same value in all profiles, it can be ignored as it does not contain any information. This is the case for the majority of the properties, as properties are in the most cases not influenced by the environment, but only the current page.

However, if the value of a property varies between different profiles in the template, this property contains information that can be used to distinguish the environments. The same holds true if a property cannot be found in a template at all. The absence of a property is treated as a value of `undefined` for this property. In Section 5, we show that the absence of

Browser	Profiling (once)	Profiling (twice)	Analysis	Total
Firefox	0.8 s	3.4 s	<0.1 s	3.5 s
Chrome	1.8 s	5.6 s	<0.1 s	5.7 s
Tor browser	0.7 s	3.2 s	<0.1 s	3.3 s
FPRandom	0.7 s	3.2 s	<0.1 s	3.3 s

Table 10.1: The time it takes to run a JavaScript Template Attack for various browsers. As the analysis phase does not run inside the browser, the time difference is due to the number of collected properties. For all browsers, the total time is well below 10 s.

properties can, for example, be used to detect whether a browser is used in private-browsing mode.

The final output of the analysis phase is a matrix of properties (rows) and their corresponding values for a set of different environments (columns). For all properties of the template matrix (*i.e.*, for each row), the value differs for at least one environment column. The more templates contain a different value for the property, the higher the entropy of the property, and thus the more it is able to deliver information about the environment. Section 5 shows the results of the JavaScript Template Attack on various browsers, including the properties which leak information.

3.3 Performance

In contrast to other template attacks [29, 68, 69, 79], JavaScript Template Attacks are extremely fast. Table 10.1 shows the runtime of the profiling and analysis phase for several different browsers. For all browsers, the runtime is well below 10 s, and could still be optimized.

The performance of the profiling phase depends on the performance of the JavaScript engine in the browser, and also on the number of properties provided by the browser. The higher the number of properties collected during the profiling phase, the longer this phase takes. If only native properties, *i.e.*, properties which are provided by the browser, are collected, the time of the profiling phase is below 2 s for all tested browsers. The artificial properties increase the runtime measurably.

The collection step of the profiling phase has to be run at least twice to remove properties which are not static, thus, the real time of the profiling phase increases by the number of runs. However, in all tests, the maximum number of required runs was 4. Moreover, to filter out properties which only change every second (e.g., a timestamp), we wait for 2 s between each

run of the collection phase. Still, the profiling phase for most browsers is below 5 s.

As the analysis phase is offline, *i.e.*, it does not run in the browser, and thus, there are only negligible performance differences for different browsers, due to the number of properties and environment provided. The resulting runtime in all our tests was less than 0.1 s.

The total runtime of a JavaScript Template Attack is the sum of the profiling phase(s) and the analysis phase. This time slightly depends on the browser, but for most our tests it is below 5 s.

4 Low-Level Properties

In this section, we show how the JavaScript Template Attack (cf. Section 3) can be augmented with properties reflecting low-level properties of the environment. For this, we add artificial properties (cf. Section 3.1) to the browser before running the profiling phase. The artificial properties are not properties per se but the result of functions deriving information about the underlying architecture or even microarchitecture.

Neither architectural nor microarchitectural properties are directly accessible in JavaScript. JavaScript code is platform independent. Thus, environmental properties have to be abstracted by the JavaScript engine. Moreover, for security reasons, JavaScript code runs in a sandbox and has no direct access to the underlying environment.

Still, recent research showed that such low-level properties can be obtained via side channels in JavaScript [26, 31, 43, 54, 62]. In this section, we present 2 new side channels to obtain architectural properties.

4.1 Instruction-Set Architecture

JavaScript is an interpreted language executed in a sandbox. Thus, the language itself is independent of the instruction-set architecture (ISA) of the machine it runs on. However, for performance reasons, JavaScript functions which are frequently executed are compiled to machine code using a just-in-time (JIT) compiler [15, 73].

Although JavaScript is oblivious to the ISA, the JIT compiler is limited by the ISA of the current platform. Thus, the JIT compiler behaves differently on CPUs with different ISAs. We can exploit this to distinguish one ISA from another ISA in JavaScript.

We craft a code snippet for which the JIT compiler can generate efficient code for one ISA and cannot generate equally efficient code for

a different ISA. Then, we compare the runtime of this code snippet to a very similar code snippet for which the JIT compiler can generate efficient code on both ISAs. Using the runtime differences between the two code snippets, we can infer the underlying ISA.

```
var a = 0.9, b = c = d
    = e = f = g = 0;
for(var i = 0; i <
    10000000; i++) {
  b = 1.0 / a;
  c = 2.2 / b;
  d = 3.4 / c;
  e = 4.1 / d;
  f = 5.8 / e;
  g = 6.6 / f;
  // no operation
  a = a + b + c + d + e
    + f + g + g;
}
```

```
var a = 0.9, b = c = d
    = e = f = g = h =
    0;
for(var i = 0; i <
    10000000; i++) {
  b = 1.0 / a;
  c = 2.2 / b;
  d = 3.4 / c;
  e = 4.1 / d;
  f = 5.8 / e;
  g = 6.6 / f;
  h = 7.1 / g;
  a = a + b + c + d + e
    + f + g + h;
}
```

Listing 4.1: Two nearly identical code snippets to detect whether the code runs in a 32-bit or 64-bit environment. In 64-bit environments, both functions have basically the same execution time, whereas in 32-bit environments, the Firefox/Tor browser just-in-time compiler generates slower code for the right function as fewer registers are available to store intermediate results.

Listing 4.1 contains two functions which are very similar. Both functions have data-dependent calculations with floating point numbers. However, the first function has one operation less. On x86, the JIT compiler uses the SSE XMM registers for floating point operations. There are 8 XMM registers available on x86-32 but 16 XMM registers on x86-64.

Thus, on x86-64, all intermediate values can be kept in the registers for both functions. However, on x86-32, all intermediate values can be kept in the registers for the first function but not for the second function. This increases the runtime of the 32-bit code significantly, as registers have to be reused and thus temporarily saved on the stack (cf. Listing 4.2). As the function is executed multiple thousand times, the runtime difference is accumulated and can easily be measured.

The same approach can also be used to distinguish 32-bit ARM vs. 64-bit ARM environments. There, the number of floating-point registers is the same, however, the number of general registers differ. On 32-bit ARM,

```
1 vaddss %xmm0,%xmm1,%
      xmm1
2 vdivsd %xmm7,%xmm6,%
      xmm6
3 vmovsd %xmm7,0x8(%esp)
4 vxorpd %xmm2,%xmm2,%
      xmm2
5 vxorpd %xmm7,%xmm7,%
      xmm7
```

```
1 vaddsd %xmm0,%xmm1,%
      xmm0
2 vdivsd %xmm2,%xmm11,%
      xmm3
3 vaddsd %xmm2,%xmm0,%
      xmm0
4 vdivsd %xmm3,%xmm10,%
      xmm4
```

Listing 4.2: The 32-bit x86 JIT compiler (left) cannot use as many registers as the 64-bit JIT compiler (right) and has to reuse registers and also save them onto the stack.

only 10 general registers (r0-r9) are used by the JIT compiler, whereas on 64-bit ARM, 32 general registers (r0-r31) are used by the JIT compiler.

We performed the measurement 10 000 times each on multiple 32-bit and 64-bit environments. In our tests, 32-bit environments can always be detected, 64-bit environments are in some cases classified as 32-bit due to scheduling or other noise which results in a slower execution of the fast function. However, we can still detect whether it is x86-32 or x86-64 with a probability of >98 % for all tested environments.

In fact, the measurement does not even require a high-precision timer. Noise does not play a role, as it can be averaged out by repeating the measurements, and the timer resolution does not matter, as the number of loop iterations (cf. Listing 4.1) can be increased until it is distinguishable. The `performance.now` function with a resolution of 100 ms in Tor is already sufficient to measure the difference if combined with edge thresholding [26, 62].

4.2 Memory Allocator

Many browser exploits rely on the underlying memory allocator [4, 28]. Buffer overflows as well as use-after-free vulnerabilities often require knowledge of the memory layout to craft reliable exploits. As browsers use different memory allocators, reliable exploits require information about the allocation strategy.

Memory allocators differ between browsers, e.g., PartitionAlloc in Chrome [14] and jemalloc in Firefox [4]. Due to platform-specific virtual memory APIs, the memory allocator behavior in one browser can even differ between operating systems [13]. However, all memory allocators

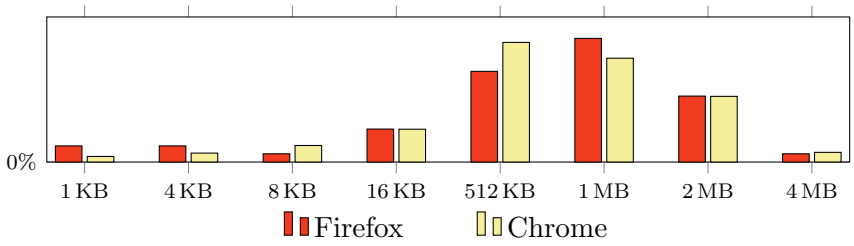


Figure 10.4: Iterating over a large array shows timing spikes at different array indices. The distances are caused by the internal memory allocator which has to allocate new memory blocks. The timings which are the easiest to detect (and thus have the highest frequency in the histogram) are slow timings caused by the allocator requesting more memory from the operating system.

have in common that they allocate memory in blocks. The size of such a block is usually a power of 2.

Thus, there are two scenarios if a resizable data structure in JavaScript has to grow. Either, there is still sufficient space in the allocated memory block, and the data structure just uses this space. Or, the memory has to be resized, which can lead to a reallocation of the memory and thus also the data structure. In the latter case, we can measure a timing difference, as this operation requires large amounts of memory to be copied which is a slow process.

Moreover, memory allocators distinguish between small and large allocations. While small allocations are handled directly by the memory allocator, large allocations are delegated to the operating system. The operating system can then directly map the required memory segments, e.g., with `mmap` on Unix or `VirtualAlloc` on Windows. Attacks which require knowledge of physical addresses [32, 62] exploited the fact that memory mapped by the operating system is not initialized. When iterating over the memory, the operating system has to handle a page fault for every page that is accessed for the first time, which takes significantly longer than an access to an already mapped page. Thus, an attacker learns where a new page starts, and thus the least significant bits of the physical address.

We only focus on the timing differences from the allocator itself, not on timing differences caused by the operating system. Note that page faults can of course also be used to learn information about the environment. However, as most systems use pages with a size of 4 KB, there is not much information to gain from exploiting this side channel.

To infer information about the memory allocator of the browser, we first allocate a small array of several kilobytes. We then choose a step size of 512 B and continuously resize the array by this step size. For every resize, we measure the time it takes using `performance.now()` in combination with edge thresholding [26, 62]. This results in a sufficiently high timer resolution to see the activity of the memory allocator. The activity manifests itself in slightly higher timings compared to accesses without memory allocator activity.

By comparing the distances between the high timings, we can infer the allocated size of the memory region. Figure 10.4 shows a histogram of the timing differences for Firefox and Chrome, grouped into typical sizes used by memory allocators. The default allocation size is detected correctly for both Chrome (512 KB) and Firefox (1 MB).

Measurement noise due to the coarse-grained `performance.now` timing function and interrupts leads to spurious high timings and missed high timings. The smaller buckets in the histogram are due to some smaller buckets used by the memory allocators, as well as spurious high timings. If the activity of a memory allocator (*i.e.*, a high timing) is missed, the bucket size is incorrectly identified as too large. However, as we see in the histogram, in the majority of the cases (*i.e.*, the highest peak in the histogram) the allocation size is determined correctly.

4.3 Graphics

WebGL allows the browser to access low-level properties and functions of the graphics card. The amount of information which can be gathered from the graphics card has already been used as a source for browser fingerprinting [9, 38]. Especially as WebGL does not require any browser permissions, it is an easy-to-use source for properties. In this section, we show that JavaScript Template Attacks can be trivially extended to also detect leaking properties in the WebGL extension.

The WebGL extension is not a static object which is always available through the object hierarchy (cf. Figure 10.2). Thus, on a blank site, there is no reference to a WebGL object or any of the WebGL extensions. However, by simply creating a WebGL element and attaching it to the `window` object, we can use a JavaScript Template Attack on the WebGL element as well.

Listing 4.3 shows the corresponding code to add WebGL as an artificial property to the object hierarchy. WebGL requires an HTML `canvas` element to instantiate the WebGL extension. We also add the `canvas`

```
<canvas id="glCanvas" width="640" height="480"/>
<script type="text/javascript">
  // add artificial property "canvas"
  window.canvas =
    document.querySelector("#glCanvas");
  // add artificial property "gl" for WebGL
  window.gl = window.canvas.getContext("webgl");
</script>
```

Listing 4.3: Adding the canvas element as well as the WebGL object as an artificial property to the `window` object.

element to the `window` object as an artificial property as it contains properties as well.

The WebGL object contains 435 properties. 296 out of the 435 properties are only constants which refer to specific WebGL parameters that can be actively queried from OpenGL. Thus, these properties itself do not contain any information. Hence, we have to automatically query the values of all parameters and again add them to `window` object as artificial properties. Querying the value of a parameter is as simple as `window.wgl[param] = gl.getParameter(gl[param])` for every property of the WebGL object.

Adding the base WebGL parameters as artificial properties adds already close to 300 properties accessible to a JavaScript Template Attack. Another large set of parameters corresponding to WebGL, and therefore the underlying hardware and environment, is not directly accessible through the WebGL object but through WebGL extensions. WebGL extensions provide additional functions and parameters of OpenGL to the browser. All specified and not-yet specified extensions are registered in the WebGL Extension Registry [27].

For every WebGL extension which is currently specified, `gl.getExtension (extensionName)` returns either an object of the extension if it is supported, or `null`. If the browser and environment support the extension, we can use it in the same way as the normal WebGL object. Again, every extension provides constant properties which can be used to query the parameter value from the extension. This is fully automated in the same manner as for the WebGL object.

Adding the parameters of all extensions adds around 100 additional properties to the `window` object. While the Tor browser does not provide any WebGL extension, there are 96 parameters from 23 extensions in Chrome and 115 parameters from 24 extensions in Firefox. In Section 5,

Device	ISA	Operating System	Browser
PC1	x86-64	Kubuntu 16.04.4 LTS	Chrome, Firefox, Tor
		Windows 10	Chrome, Firefox, Tor, Edge
PC2	x86-64	Kubuntu 18.04 LTS	Chrome, Firefox, Tor
		Windows 7	Chrome, Firefox, Tor, Edge
PC3	x86-64	Kubuntu 16.04.5 LTS	Chrome, Firefox, Tor
		Windows 10	Chrome, Firefox, Tor, Edge
VM1	x86-32	Windows XP	Chrome, Firefox, Tor
VM2	x86-64	Kubuntu 17.04	Chrome, Firefox, Tor
VM3	x86-64	Windows 10	Chrome, Firefox, Tor, Edge
Phone1	AArch64	Android 7.0	Chrome, Firefox, Tor
Phone2	ARMv7	Android 6.0.1	Chrome, Firefox, Tor
Phone3	AArch64	Ubuntu 16.04	Chrome, Firefox, Tor

Table 10.2: List of environments used for the case studies.

we show that the properties created from WebGL parameters can be used to infer information about the environment.

4.4 Microarchitectural Elements

There is a variety of other low-level properties which have already been used in side-channel attacks from JavaScript [26, 31, 36, 43, 44, 54, 62, 75, 77]. All these properties can theoretically also be added as artificial properties. However, these attacks are already powerful attacks itself. Furthermore, these attacks are often quite fragile and require information about the system itself, without providing information about the environment, but only about specific secrets. Thus, an attacker would rather use such microarchitectural side-channel attacks to complement a JavaScript Template Attack.

Moreover, as a consequence to the Spectre attacks, which have not only been shown in native code but also in JavaScript, browser manufacturers limited the access to high-precision timers rigorously. This does not only include the provided `performance.now` function but also self-built timers using `SharedArrayBuffers` [26, 62]. As a result, many of the well-known microarchitectural attacks are currently prevented until a new timing source is found, or browser vendors re-enable `SharedArrayBuffers` and precise timers as, e.g., Google plans to do with Chrome [59].

5 Case Studies

In this section, we provide multiple case studies of our JavaScript Template Attack in various environments. We scan all native properties which are in the hierarchy starting at `window` (cf. Figure 10.2). Additionally, we add the artificial properties described in Section 4, which includes all WebGL properties and WebGL extension properties, the memory allocator and the ISA. As browsers, we used Google Chrome 67.0.3396.99, Mozilla Firefox 61.0.1, Tor 7.5.6, and–if available–Microsoft Edge 42.17134.1.0. Table 10.2 shows a table of all the environments we used for testing.

For all case studies, we used our open-source JavaScript Template Attack framework.[1] In the case studies, we tried to automatically infer as much information about the environment as possible.

The collected information can be used directly or indirectly to mount targeted exploits. Directly usable information includes, for example, the operating system and architecture, which is required knowledge for many exploits. Indirectly usable information includes, for example, the use of privacy extensions or private mode which can be used to imitate plausible looking system messages or dialogues, e.g., for phishing [11, 12].

In all use cases, we assume that we cannot simply read the correct information directly from the browser, e.g., from the user agent. The user agent string contains among others operating system, browser name and version. Even if we get this information directly, an attacker cannot rely on this information, as it can easily be modified using browser extensions. Moreover, some browsers such as Tor do not even provide any information about the environment in the user agent.

5.1 Browser Detection

The major browsers all have their own JavaScript and rendering engine. Thus, exploitable bugs are usually limited to one browser. Especially exploits which heavily rely on the internal functionality of the browser are limited to a specific browser.

The differences between the browsers do not only prevent one browser exploit to work in a different browser, but it also makes it easy to distinguish browsers. Every browser supports a distinct set of functions [18] and also provides browser-specific properties through so-called vendor prefixes [49]. Already the number of documented properties for the major

[1] The source of the framework can be found in a GitHub repository at `https://github.com/IAIK/jstemplate`

browsers differs significantly, with 2698 for Chrome, 2247 for Firefox, and 1806 for Edge [47].

Moreover, as the JavaScript engine differs between browsers, the values of properties are also different. We added the `toString` representation of functions as simple artificial properties. As the representation is not strictly defined, it differs between browsers. This difference has also been exploited to detect the manipulation of the user-agent string [76].

However, not only the values of properties are different but also the available properties differ between browsers. We compared all accessible native and artificial properties of Firefox and Chrome running in exactly the same environment. Every property which was not implemented was assumed to have the value `undefined`, which is the case for every undefined variable.

In total, our JavaScript Template Attack discovered 14 544 properties which differed between Firefox and Chrome. With 60.1 %, the majority of differing properties is the string representation of functions. Without these artificial properties, there are still 5796 properties which differ between the two browsers. Similarly, there are 15 670 different properties between Edge and Firefox, and 8913 between Edge and Chrome.

Even between Firefox and the Tor browser (which is based on Firefox) we found 3055 properties with different values. Again, the majority of differing properties (63.6 %) is the string representation of functions. However, as both browsers share the same code base, the difference is not in the format of the string representing the function. The differences are caused by functions which are only available in one of the two browsers. Without considering functions, there are still 1111 properties with different values between the two browsers.

Summarizing, even browsers which share a common code base can be easily distinguished using our JavaScript Template Attack. For all tested browsers, there are more than 1000 properties with different values which can be used to uniquely identify a specific browser. We were successfully able to distinguish all of the 40 tested setups (cf. Table 10.2) without any false positives or false negatives. Even in the hypothetical case that native properties do not leak this information anymore, the artificial memory-allocator property (cf. Section 4.2) can be used to distinguish browsers.

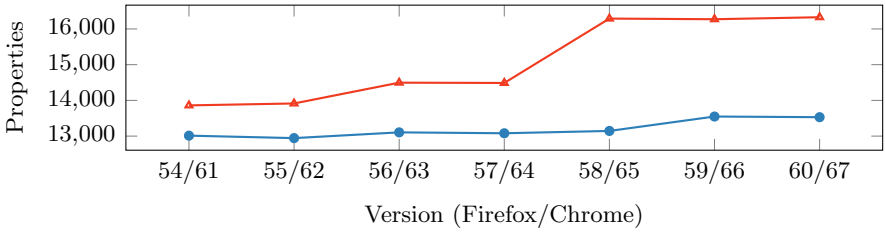


Figure 10.5: The number of identified properties from Chrome 60 to 67 (●) and Firefox 53 to 60 (▲). The trend shows that the number of properties increases over time.

Browser Version

For many exploits, it is not only necessary to know which browser the victim uses but also the exact browser version. As exploits are disclosed, they are usually fixed by the browser vendor in one of the next versions. Thus, to reliably run an exploit on a browser, knowing the browser version is important for selecting a working exploit.

Figure 10.5 shows the number of properties discovered using a JavaScript Template Attack for every Firefox and Chrome version since 53 and 60 respectively. For all versions of Firefox and Chrome, there are many unique properties. We further compared the number of properties between all versions of the browsers. There is always at least one property which has changed between any two versions. For all tested browsers in all setups, we were able to distinguish the versions of the browsers. We can see a clear trend to an increasing number of properties, although in some versions properties are removed due to changes in the standards or deprecation of functions.

Summarizing, for all major browsers, it is easy to detect the actual browser version by counting the number of implemented properties, even without inspecting the values of the properties. As the trend is to continuously add more features instead of removing features, we expect the browser version detection to work on newer versions of the browsers as well.

5.2 Privacy-Extension Detection

There are several privacy-enhancing extensions for browsers, e.g., ad blocker or anti-tracking extensions. Some of them modify the information sent to servers (e.g., FP-Block [72]) or overwrite JavaScript functionality

vs.	Medium	High	Tin Foil Hat	Sample Expression
Low	27	29	27	`!!((Worker&&Worker.toString().indexOf(‘‘postMessage’’)==-1)|0)`
Medium	-	28	28	`!!((addEventListener&&addEventListener.toString().indexOf(‘‘block’’)!=-1)|0)`
High	-	-	28	`!!((performance.now&&performance.now.toString().indexOf(‘‘fuzz’’)!=-1)|0)`
Tin Foil Hat	-	-	-	`!!((Array&&Array.toString().indexOf(‘‘Proxy’’)!=-1)|0)`

Table 10.3: Every row of the table represents a protection level of Chrome Zero [63]. On the left side of the table is the number of properties which have a different value compared to the protection level in the corresponding column. The right side of the table shows one sample expression which is only true if the corresponding protection level is active.

(e.g., Chrome Zero [63]). Often, such plugins change properties which are accessible from JavaScript. Thus, a JavaScript Template Attack can detect the presence of such plugins.

Note that the detection of such plugins can have various uses. First, it allows an attacker to create dialogues which look as if they are coming from such a browser extension, tricking the user into interacting with them. For example, a user might be tricked into clicking on a fake update dialogue from an extension, which actually triggers, e.g., a switch to fullscreen mode or a file download. Second, exploits can be automatically adapted to avoid functions which are modified by a browser extension such as Chrome Zero [63]. Finally, as already described by Mowery et al. [46], Eckersley [19], Acar et al. [1], or Nikiforakis et al. [50], such plugins are a source for fingerprinting, as they lead to inconsistencies.

We evaluated Chrome Zero [63], Chameleon [45], Canvas Defender[1], CyDec Platform AntiFingerprint[2], Ghostery[3], and WebAPI Manager [66]. For Chrome Zero, we are not only able to detect that it is active but also the current protection level (cf. Table 10.3).

Mounting a JavaScript Template Attack with the WebAPI Manager extension [66] active leads to similar results. Again, we can detect that

[1] `https://multiloginapp.com/canvasdefender-browser-extension/`

[2] `https://addons.mozilla.org/en-US/firefox/addon/cydec-platform-antifingerprint/`

[3] `https://www.ghostery.com/`

vs.	Lite	Conservative	Aggressive
None	1492	1539	2381
Lite	-	67	894
Conservative	-	-	843

Table 10.4: Every row of the table represents a protection level of the Web API Manager [66]. The table contains the number of properties with a different value compared to the protection level in the corresponding column.

the extension is active as it modifies properties. Similar to the Chrome Zero extension, we can also detect which protection level is used (lite, conservative, aggressive) as shown in Table 10.4. As with Chrome Zero, it is not possible to access the references to the original functions.

For Canvas Defender, we cannot only determine that it is used (105 distinguishing properties) but also semi-automatically circumvent it. Canvas Defender replaces functions which are used or can potentially be used for fingerprinting with its own functions. However, as it requires the original functionality as well, it stores references to the original functions as properties of the `window` object. Thus, a JavaScript Template Attack does not only discover the use of the extension, but it also reveals the original functions. From an attacker's perspective, the function references are conveniently named the same as the original functions and just prefixed with a random string. Thus, JavaScript Template Attacks cannot only detect the tested extension. It can even be used to circumvent it, leaving more than 30 000 users who have this extension installed with a false sense of security.

Mounting a JavaScript Template Attack with the WebAPI Manager extension [66] activated leads to similar results. Again, we can detect that the extension is active as it modifies between 1472 and 2307 properties, depending on the protection level. We can also easily detect whether Chameleon or CyDec are active. Our JavaScript Template Attack identified 13 properties which are modified by Chameleon and 2365 properties which are modified or added by CyDec. Each of these properties can be used to detect that the user has Chameleon installed and activated. Interestingly, Ghostery is only detectable when installed in Firefox. Ghostery adds Ghostery-specific elements to every page in Firefox, revealing the usage of this extension. In Chrome, there are no differences, making Ghostery in Chrome not detectable with our automated method.

We can conclude that JavaScript Template Attacks are a valuable method for developers of privacy-enhancing extensions to test their extension. If extensions try to hide references instead of making them inaccessible, they can be easily revealed again, allowing an attacker to easily circumvent the extension. JavaScript Template Attacks can easily uncover such leaked references during development.

5.3 Private Mode Detection

Similarly to privacy-enhancing extensions, Firefox, Chrome, and Edge provide a built-in private-browsing mode. In this mode, the browser does not keep any tracks of visited websites, such as cookies or history. Furthermore, private browsing also includes some tracking protection [48].

We mounted a JavaScript Template Attack to detect whether there are any differences between normal mode and private-browsing mode. In Chrome, there are no detectable differences when using the browser in private-browsing mode. Similarly, we cannot detect differences between normal mode and guest mode, a feature similar to private-browsing mode.

For Firefox, however, there are properties revealing whether the current window is a private-browsing window or a normal window. For example, service workers are not available in private-browsing mode. Thus, all 73 properties corresponding to service workers are only detected in normal mode and not available in private-browsing mode.

An additional hint that a Firefox window is in private-browsing mode is the value of the `doNotTrack` property. Per default, this flag is set to "unspecified" and only gets an actual value if the user specifies one in the browser settings. In private-browsing mode, however, this flag is always set to "1" if not configured differently by the user. Thus, if this value is not "1", the window is probably not in private-browsing mode.

For Edge, we can also detect whether the window is in private-browsing mode or normal mode. We detected 72 properties corresponding to local databases and Microsoft-specific properties, such as `MSCredentials`. These features are only available in normal mode. Moreover, Edge handles the `doNotTrack` property in the same way as Firefox, providing another hint about the current mode.

5.4 Operating System Detection

If exploits interact with the environment, e.g., access operating-system specific resources, an attacker has to know which operating system is used. The same is true if an attacker tries to create fake system messages [11,

12]. Most browsers are available for all major platforms and provide the same functionality on all platforms. Thus, for a legitimate website, there is usually no reason to detect the operating system for any functionality except for statistics.

We mounted a JavaScript Template Attack to detect whether any property would reveal the underlying operating system. For Microsoft Edge, this would be trivial, as it only runs on Microsoft Windows. Thus, we did not include this browser in our tests. Furthermore, to eliminate influences which are not from the operating system, wherever possible, we mounted the attack on the same hardware for the different operating systems.

The Tor browser actively tries to eliminate all differences among operating systems. Still, some properties differ between operating systems. An interesting difference in properties we detected is the `window.innerWidth`/`window.innerHeight` pair. Although the Tor browser warns the user not to resize the window to prevent fingerprinting using these properties, they are not always the same. For example, `window.innerWidth` is 1000 on Linux (Kubuntu 16.04.4) but 1001 on Windows 10. The reason for this is that Windows 10 has native support for high-density displays and automatically scales application such that they have a usable size. For the browser, the screen appears to be smaller than the actual screen resolution. However, this scaling seems to introduce rounding errors, which results in this difference in the `window.innerWidth` property. On Android (with Orfox), this property is also different with a value of 980.

The font rendering causes another difference between operating systems. The list of installed fonts is already known to provide reliable fingerprints [6]. Due to different available fonts as well as differences in the font rendering code, the same text has different dimensions on different operating systems [50]. For example, in Tor, a default heading on Windows 10 is 1 pixel higher than on Linux. Such differences do not only exist for the Tor browser but also for Chrome.

For Firefox, we detected additional properties which give an even better indication about the underlying operating system. Firefox has experimental support for virtual-reality displays (e.g., `window.navigator.activeVRDisplays`). However, in the current version (61.0.1), only Windows is fully supported. Linux is not supported, and macOS is only partially supported. Thus, by detecting which functions are available for virtual-reality displays, the operating system can be detected.

Moreover, we detected differences in WebGL properties which allow distinguishing the operating system for both Firefox and Chrome. One property which reveals whether the underlying operating system is Windows is the `UNMASKED_RENDERER_WEBGL` property of the `WEBGL_debug_renderer_info` extension. This property contains the OpenGL renderer used for WebGL. On Windows, this string always contains `ANGLE`, which stands for *Almost Native Graphics Layer Engine*, the OpenGL compatibility layer on Windows [76]. The string `Iris` refers to Intel Iris Graphics, a GPU which is mostly found in MacBook Pros and iMacs, thus indicating that the browser is running on macOS.

The Android operating system can also be distinguished from other operating systems mostly by the lack of functions (and thus properties). For example, Firefox on Android does not support speech synthesis (e.g., `window.SpeechSynthesis`). Chrome on Android, for example, does not support support inline installation of extensions (e.g., `chrome.webstore.install`). Both browsers do not support shared workers (e.g., `window.SharedWorker`) and plugins on Android.

However, we detected one feature which is only available on Chrome for Android. The `window.MediaSession` allows a mobile website to show information about the currently played multimedia content in the notification bar. If this property is available, the underlying operating system is Android.

For some of the properties, the operating system can be directly inferred, and by combining the detected properties, we can reliably detect any of the major operating systems.

5.5 Architecture Detection

For exploits running binary code, it is vital to know the current ISA. Assuming a wrong ISA (e.g., x86 instead of ARM) results in an unsuccessful exploit. In both cases, the exploit attempt does not only fail, but it might also be detected.

As with all other properties, the Tor browser tries to provide the same functionality and properties on all architectures. On all desktop operating systems, the Tor browser reports the platform as `Win32`, independent of the actual operating system or ISA. However, we detected a difference when running a JavaScript Template Attack on Orfox, the official Android version of the Tor browser. There, the platform is not reported as `Win32` but the actual platform is reported (`armv8l` on an ARMv8 phone

and `armv7l` on an ARMv7 phone). We also disclosed this issue to the developers, and it will be fixed in one of the future versions.

Another property which indicates the underlying ISA is again the renderer information as well as the vendor information from WebGL. `Adreno`, `Mali`, and `Tegra` renderer are only available for ARM. Thus, if this string is contained in the renderer information, the underlying ISA is ARM. Similarly, on Linux, the renderer information can even contain the specific microarchitecture. For example, on a Lenovo T460s with an Intel Skylake CPU, the vendor string contains `Intel` and the renderer property value is `Mesa DRI Intel(R) HD Graphics 520 (Skylake GT2)`.

Finally, the artificial property presented in Section 4.1 can be used to distinguish 32-bit and 64-bit x86. We achieve a classification rate which is close to 100 %. Moreover, it has the huge advantage that it cannot easily be hidden from an attacker, whereas the values of properties can be anonymized by the browser vendors.

5.6 Virtual Machine Detection

Although virtual machines should not be distinguishable from native machines, we still detected one property which has a distinct value inside a virtual machine. In Firefox, the WebGL extension can reveal that Firefox is running inside a virtual machine.
The `UNMASKED_VENDOR_WEBGL` property of the `WEBGL_debug_renderer_info` extension is set to `VMWare, Inc.` when running inside VirtualBox or VMWare. For the Tor browser and Chrome, we could not detect any property which immediately reveals that the environment is a virtual machine.

However, there are two properties which can give a hint that the underlying environment is a virtual machine. First is the reported screen resolution (`window.screen.availWidth` / `window.screen.availHeight`). If the value is an odd value, *i.e.*, not one of the usually used resolutions of screens, it is a strong indicator that the browser is running in a virtual machine. For example, on our test machine, the screen resolution is 1920x1080, and the reported resolution inside the VM is 1920x944. Second, the number of reported CPUs can be easily queried using `navigator.hardwareConcurrency`. For a native environment, this value is usually a power of two on consumer hardware. A small number which is not a power of two (e.g., 3) is also an indicator that the browser is running inside a virtual machine.

Browser	MDN	JavaScript Template
Firefox	2247	15 709
Chrome	2698	13 570
Edge	1806	9666
Firefox Android	2104	15 612
Chrome Android	2676	13 119
Tor browser	2247†	15 639

† As the MDN does not distinguish between the Tor browser and Firefox, we used the Firefox numbers, as the Tor browser is based on Firefox.

Table 10.5: The number of properties documented in the MDN Web Docs compared to the number of properties found using a JavaScript Template Attack.

6 Coverage Analysis

In this section, we analyze the coverage of JavaScript Template Attacks. As a baseline, we parsed the MDN Web Docs [47]. We then compared all our detected properties to the properties extracted from the MDN Web Docs.

Table 10.5 shows the number of properties we parsed from the MDN Web Docs as well as the number of properties detected with a JavaScript Template Attack for Firefox, Chrome (both on Linux and Android), Edge, and the Tor browser (Linux only). Interestingly, the number of detected properties for every browser is much higher than the number of properties officially documented. One reason for this is that the documentation is apparently not complete. Moreover, we access several internal, undocumented properties. This is an interesting aspect, as our JavaScript Template Attack also allows to find completely new properties which might not have been considered for fingerprinting before as they are not documented. Another reason is that we access the same property for multiple objects, e.g., the `length` property. Properties from the prototype chain are not documented if they are already documented for the parent object. Thus, this property is counted twice although it is in principle the same property.

Still, we do not achieve a 100 % coverage for multiple reasons. The majority of the documented properties does not belong to static objects, *i.e.*, objects which always exists in the browser. Many objects have to be dynamically created, e.g., exceptions, or instances of elements. Thus, we cannot automatically explore the properties of these objects. It is, however, possible to create such objects and add them to the hierarchy manually.

Browser	Exploration	Without duplicates	Usable
Firefox	18 443	16 450	15 709
Chrome	15 585	13 604	13 570
Edge	13 752	11 850	9666
Firefox Android	18 214	16 296	15 612
Chrome Android	15 556	13 608	13 119
Tor browser	17 217	15 645	15 639

Table 10.6: The number of properties found using a JavaScript Template Attack and the number of properties which were left after the cleanup step of the analysis phase (cf. Section 3.2).

We showed this for WebGL (cf. Section 4.3) and the `toString` function (cf. Section 5.1). Future work has to research whether this step can be automated to achieve an even higher coverage. Nonetheless, as shown in Section 5, the coverage is already sufficient to find many properties which reveal information about the environment.

Another reason for missing properties is that some browser-specific properties are not referenced by the `window` root object and are thus not in the hierarchy illustrated in Figure 10.2.

Table 10.6 shows that most of the detected properties were actually usable for the property extraction step (cf. Section 3.2). The cleanup step (cf. Section 3.2) removed only a small percentage (<15 %) of the properties as they were duplicates. From the remaining properties, only a few (<9 %) had to be discarded as they changed their value when read multiple times. These properties were mostly timestamps.

For all browsers, we found around 10 000 usable properties. This massive number of automatically detected, partly undocumented and usable properties stresses the need for automated leakage detection.

7 Discussion

In this section, we discuss the differences between JavaScript Template Attacks and traditional fingerprinting, its limitations, and possible future improvements.

7.1 Difference to Fingerprinting

Although JavaScript Template Attacks look similar to fingerprinting, they have a different goal. In traditional fingerprinting, attackers try to

identify properties or combinations of properties which are unique for a *user*. For JavaScript Template Attacks, we try to identify properties or combinations of properties which are unique for an *environment*. In contrast to fingerprinting, it is preferable that the identified properties do not change for different users, but only for environments.

The overlap between JavaScript Template Attacks and fingerprinting lies in the fact that many detected properties can be used for fingerprinting. This makes JavaScript Template Attacks also a powerful method to automatically search for new fingerprinting sources. It detects differences in properties within seconds, without requiring any manual analysis. Thus, this also reduces the time to search for new fingerprints. As shown in Section 5, several of the properties we used to detect the environment are indeed useful for fingerprinting.

7.2 Limitations and Future Work

We currently focussed mainly on properties, and only added the `toString` function and the functions to query WebGL parameters. Thus, many properties which are hidden behind function calls are not identified. We expect that the results of function calls provide more information about the environment, similar to function calls in Android [69].

The most simple case are functions which do not take any argument. Still, adding these functions as artificial properties is not as straightforward as it seems at first glance. Several functions have to be blacklisted, as they would abort the script (e.g., `window.close()` or `document.location.reload()`) or pause the script until the user actively continues execution (e.g., `alert()`). Moreover, cycles have to be detected to not be stuck in endless loops (e.g., the result of `toString` is again a string which provides a `toString` function).

Future research has to investigate how this approach can be applied to functions with parameters. In contrast to Java [69], getting the number and types of arguments for a function in JavaScript is not straightforward. Moreover, choosing sane values is a hard problem. It would be interesting to combine techniques from fuzzing which select sane values with JavaScript Template Attacks to automatically test the return values of functions. However, fuzzing JavaScript APIs with a high coverage is still an open research problem [34].

An interesting direction would also be to target certain web standards, such as Web USB or Web NFC. To get useful results, a JavaScript Template Attack would require some manual initialization and possibly

user interaction to grant the corresponding permission. Thus, this is not in the scope of this paper, as it requires more research into automatically understanding the semantics of functions and calling them.

7.3 Countermeasures

Most browsers do not have the goal to prevent identification of the environment. While some properties which leak information about the environment cannot easily be removed, others can be anonymized as it is, e.g., done in the Tor browser. From our experiments, we have seen that Tor's anti-fingerprinting design [57] also prevents that an attacker can leak a lot of information about the environment. Thus, anti-fingerprinting techniques–if implemented correctly–are a viable method to also prevent the detection of the environment.

As shown in Section 5, JavaScript Template Attacks can detect leakage in privacy-enhancing browsers and extensions. Thus, the main use case of JavaScript Template Attacks is to provide an automated augmentation for the development process of defense mechanisms. If used in the development process of privacy-enhancing browsers and extensions, they can detect overlooked properties, as, e.g., in the case of the Orfox browser (cf. Section 5.5). This also shows shortcomings in the implementation of extensions, e.g., the original function references are still accessible (cf. Section 5.2).

8 Conclusion

In this paper, we presented JavaScript Template Attacks, a fully automated novel technique to detect subtle differences in browser engines caused by the environment. Furthermore, we showed two new side-channel attacks on browsers, allowing to detect the instruction-set architecture and the used memory allocator. Our techniques even work in the presence of anti-fingerprinting mechanisms in the browser. By leveraging the found differences in the browser engine, an attacker learns details about the environment and can get a clearer picture of a system for a targeted exploit. Moreover, our technique is applicable to identifying new fingerprints automatically.

We found environment-dependent properties in all major browsers, including Tor for Android, allowing us to reveal the underlying operating system, CPU architecture, used privacy-enhancing plugins, and the exact browser version. Furthermore, we showed that privacy-enhancing exten-

sions can provide a false sense of security as they can be circumvented semi-automatically using our technique if not implemented correctly. Thus, we stress that our method should be used in the development process of browsers and privacy extensions to automatically find flaws in the implementation.

Acknowledgments

We would like to thank our anonymous reviewers for their feedback. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. "FPDetective: dusting the web for fingerprinters." In: *CCS*. 2013.

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. "On the Power of Simple Branch Prediction Analysis." In: *AsiaCCS*. 2007.

[3] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. "Predicting secret keys via branch prediction." In: *CT-RSA 2007*. 2007.

[4] Patroklos Argyroudis and Chariton Karamitas. "Exploiting the jemalloc memory allocator: Owning Firefox's heap." In: *Blackhat USA* (2012).

[5] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. 2004. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[6] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. "User tracking on the web via cross-browser fingerprinting." In: *Nordic Conference on Secure IT Systems*. 2011.

[7] Billy Bob Brumley and Risto M Hakala. "Cache-timing template attacks." In: *International Conference on the Theory and Application of Cryptology and Information Security*. 2009.

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *arXiv:1811.05441* (2018).

[9] Yinzhi Cao, Song Li, and Erik Wijmans. "Browser Fingerprinting via OS and Hardware Level Features." In: *NDSS*. 2017.

[10] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. "Template attacks." In: *CHES*. 2002.

[11] George Chatzisofroniou. *Efficient Wi-Fi Phishing Attacks*. 2016. URL: `https://census-labs.com/media/effective_wifi_phishing_33c3.pdf`.

[12] George Chatzisofroniou. *Extra Phishing Pages*. 2018. URL: `https://github.com/wifiphisher/extra-phishing-pages`.

[13] Chromium. *Key Concepts in Chrome Memory*. 2018. URL: `https://chromium.googlesource.com/chromium/src/+/lkgr/docs/memory/key_concepts.md`.

[14] Chromium. *PartitionAlloc Design*. 2018. URL: https://chromium.googlesource.com/chromium/src/+/lkcr/base/allocator/partition_allocator/PartitionAlloc.md.

[15] Lin Clark. *A crash course in just-in-time (JIT) compilers*. 2017. URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/.

[16] Sarah Cortes. "Legalizing Domestic Surveillance: The Role of Mutual Legal Assistance Treaties in Deanonymizing TorBrowser Technology." In: (2015).

[17] Janis Dalins, Campbell Wilson, and Mark Carman. "Criminal motivation on the dark web: A categorisation model for law enforcement." In: *Digital Investigation* (2018).

[18] Alexis Deveria. *Can I use... Support tables for HTML5, CSS3, etc.* 2018. URL: http://caniuse.com/.

[19] Peter Eckersley. "How unique is your web browser?" In: *PETS*. 2010.

[20] Steven Englehardt and Arvind Narayanan. "Online tracking: A 1-million-site measurement and analysis." In: *CCS*. 2016.

[21] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR." In: *International Symposium on Microarchitecture (MICRO)*. 2016.

[22] Andreas Fobian and Carl-Benedikt Bender. "Firefox 0-day targeting Tor-users." In: (2016).

[23] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU." In: *IEEE S&P*. 2018.

[24] Barton Gellman, Craig Timberg, and Steven Rich. "Secret NSA documents show campaign against Tor encrypted network." In: *The Washington Post* (2013), p. 4.

[25] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. "Drive-by Key-Extraction Cache Attacks from Portable Code." In: *ACNS*. 2018.

[26] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. 2017.

[27] Khronos Group. *WebGL Extension Registry*. 2018. URL: https://www.khronos.org/registry/webgl/extensions/.

[28] Samuel Groß. *Exploiting a Cross-mmap Overflow in Firefox*. 2017. URL: https://saelo.github.io/posts/firefox-script-loader-overflow.html.

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[30] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[31] Daniel Gruss, David Bidner, and Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed JavaScript." In: *ESORICS*. 2015.

[32] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA*. 2016.

[33] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *S&P*. 2011.

[34] Renáta Hodován and Ákos Kiss. "Fuzzing JavaScript Engine APIs." In: *International Conference on Integrated Formal Methods*. 2016.

[35] Simon Kemp. *Digitnal In 2018: World's Internet users pass the 4 billion mark*. 2018. URL: https://wearesocial.com/blog/2018/01/global-digital-report-2018.

[36] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *S&P*. 2019.

[37] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems." In: *CRYPTO*. 1996.

[38] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints." In: *S&P*. 2016.

[39] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. "FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques." In: *ESSoS*. 2017.

[40] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. 2016.

[41] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space.” In: *USENIX Security Symposium*. 2018.

[42] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. “Nethammer: Inducing Rowhammer Faults through Network Requests.” In: *arXiv:1711.08002* (2017).

[43] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. “Practical Keystroke Timing Attacks in Sandboxed JavaScript.” In: *ESORICS*. 2017.

[44] Maryam Mehrnezhad, Ehsan Toreini, Siamak F Shahandashti, and Feng Hao. “Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript.” In: *Journal of Information Security and Applications* (2016).

[45] Alexei Miagkov. *Chameleon - Browser fingerprinting protection for everybody*. 2015. URL: `https://github.com/ghostwords/chameleon`.

[46] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. “Fingerprinting information in JavaScript implementations.” In: *W2SP*. 2011.

[47] Mozilla. *mdn-browser-compat-data*. 2018. URL: `https://github.com/mdn/browser-compat-data`.

[48] Mozilla. *Private Browsing - Use Firefox without saving history*. 2018. URL: `https://support.mozilla.org/en-US/kb/private-browsing-use-firefox-without-history`.

[49] Mozilla. *Vendor Prefix*. 2018. URL: `https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix`.

[50] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. “Cookieless monster: Exploring the ecosystem of web-based device fingerprinting.” In: *Security and privacy (SP)*. 2013.

[51] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. “Privaricator: Deceiving fingerprinters with little white lies.” In: *WWW*. 2015.

[52] Lukasz Olejnik. *Stealing sensitive browser data with the W3C Ambient Light Sensor API*. 2017. URL: `https://blog.lukaszolejnik.com/stealing-sensitive-browser-data-with-the-w3c-ambient-light-sensor-api/`.

[53] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. "Battery Status Not Included: Assessing Privacy in Web Standards." In: *Workshop on Privacy Engineering (IWPE)*. 2017.

[54] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS*. 2015.

[55] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA*. 2006.

[56] Colin Percival. "Cache missing for fun and profit." In: *BSDCan*. 2005.

[57] Mike Perry, Erinn Clark, Steven Murdoch, and Georg Koppen. *The Design and Implementation of the Tor Browser*. May 2018. URL: `https://www.torproject.org/projects/torbrowser/design/` (visited on 06/06/2018).

[58] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium*. 2016.

[59] Charlie Reis. *Mitigating Spectre with Site Isolation in Chrome*. 2018. URL: `https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html`.

[60] Dakota S Rudesill, James Caverlee, and Daniel Sui. "The Deep Web and the Darknet: A Look Inside the Internet's Massive Black Box." In: (2015).

[61] Bruce Schneier. "Attacking Tor: how the NSA targets users' online anonymity." In: *The Guardian* 4 (2013).

[62] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC*. 2017.

[63] Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks." In: *NDSS*. 2018.

[64] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *DIMVA*. 2017.

[65] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *arXiv:1807.10535* (2018).

[66] Peter Snyder, Cynthia Taylor, and Chris Kanich. "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security." In: *CCS*. 2017.

[67] Raphael Spreitzer. "Pin skimming: Exploiting the ambient-light sensor in mobile devices." In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014.

[68] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. "ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android." In: *AsiaCCS*. 2018.

[69] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. "SCANDroid: Automated Side-Channel Analysis of Android APIs." In: *11th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2018.

[70] Paul Stone. *Pixel Perfect Timing Attacks with HTML5*. Tech. rep. Context Information Security, June 2013. URL: `http://www.contextis.com/files/Browser_Timing_Attacks.pdf`.

[71] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses." In: *USENIX ATC*. 2018.

[72] Christof Ferreira Torres, Hugo Jonker, and Sjouke Mauw. "FP-block: Usable web privacy by controlling browser fingerprinting." In: *ESORICS*. 2015.

[73] V8 Team. *Launching Ignition and TurboFan*. 2017. URL: `https://v8project.blogspot.com/2017/05/launching-ignition-and-turbofan.html`.

[74] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium*. 2018.

[75] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. "The clock is still ticking: Timing attacks in the modern web." In: *CCS*. 2015.

[76] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. "FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies." In: *USENIX Security Symposium*. 2018.

[77] Pepe Vila and Boris Köpf. "Loophole: Timing Attacks on Shared Event Loops in Chrome." In: *USENIX Security Symposium*. 2017.

[78] Benjamin Vitaris. *Firefox Zero-Day Can Be Used To Deanonymize Tor Users*. 2016. URL: `https://www.deepdotweb.com/2016/12/11/firefox-zero-day-can-used-deanonymize-tor-users`.

[79] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries." In: *USENIX Security Symposium*. 2018.

[80] Ofir Weisse et al. "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution." In: *Technical report* (2018).

[81] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

11

ZombieLoad: Cross-Privilege-Boundary Data Sampling

Publication Data

Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *CCS*. 2019

Contributions

Main author.

ZombieLoad: Cross-Privilege-Boundary Data Sampling

Michael Schwarz[1], Moritz Lipp[1], Daniel Moghimi[2], Jo Van Bulck[3], Julian Stecklina[4], Thomas Prescher[4], Daniel Gruss[1]

[1] Graz University of Technology, Austria [2] Worcester Polytechnic Institute, USA [3] imec-DistriNet, KU Leuven, Belgium
[4] Cyberus Technology, Germany

Abstract

In early 2018, Meltdown first showed how to read arbitrary kernel memory from user space by exploiting side-effects from transient instructions. While this attack has been mitigated through stronger isolation boundaries between user and kernel space, Meltdown inspired an entirely new class of fault-driven transient-execution attacks. Particularly, over the past year, Meltdown-type attacks have been extended to not only leak data from the L1 cache but also from various other microarchitectural structures, including the FPU register file and store buffer.

In this paper, we present the ZombieLoad attack which uncovers a novel Meltdown-type effect in the processor's fill-buffer logic. Our analysis shows that faulting load instructions (*i.e.*, loads that have to be re-issued) may transiently dereference unauthorized destinations previously brought into the fill buffer by the current or a sibling logical CPU. In contrast to concurrent attacks on the fill buffer, we are the first to report data leakage of recently loaded and stored stale values across logical cores even on Meltdown- and MDS-resistant processors. Hence, despite Intel's claims [37], we show that the hardware fixes in new CPUs are not sufficient. We demonstrate ZombieLoad's effectiveness in a multitude of practical attack scenarios across CPU privilege rings, OS processes, virtual machines, and SGX enclaves. We discuss both short and long-term mitigation approaches and arrive at the conclusion that disabling hyperthreading is the only possible workaround to prevent at least the most-powerful cross-hyperthread attack scenarios on current processors, as Intel's software fixes are incomplete.

The original publication is available at `https://dl.acm.org/citation.cfm?doid=3319535.3354252`.

1 Introduction

In 2018, Meltdown [47] was the first microarchitectural attack completely breaching the security boundary between the user and kernel space and, thus, allowed to leak arbitrary data. While Meltdown was fixed using a stronger isolation between user and kernel space, the underlying principle turned out to be an entire class of transient-execution attacks [8]. Over the past year, researchers demonstrated that Meltdown-type attacks cannot only leak kernel data to user space, but also leak data across user processes, virtual machines, and SGX enclaves [72, 78]. Furthermore, leakage is not limited to the L1 cache but can also originate from other microarchitectural structures, such as the register file [71] and, as shown in concurrent work, the fill buffer [61], load ports [61], and the store buffer [55].

Instead of executing the instruction stream in order, most modern processors can re-order instructions while maintaining architectural equivalence. Instructions may already have been executed when the CPU detects that a previous instruction raises an exception. Hence, such instructions following the faulting instruction (*i.e.*, transient instructions) are rolled back. While the rollback ensures that there are no architectural effects, side effects might remain in the microarchitectural state. Most Meltdown-type attacks exploit overly aggressive optimizations around out-of-order execution.

For many years, the microarchitectural state was considered invisible to applications, and hence security considerations were often limited to the architectural state. Specifically, microarchitectural elements often do not distinguish between different applications or privilege levels [8, 13, 39, 47, 59, 64, 65].

In this paper, we show that, first, there still are unexplored microarchitectural buffers, and second, both architectural and microarchitectural faults can be exploited. With our notion of "microarchitectural faults", *i.e.*, faults that cause a memory request to be re-issued internally without ever becoming architecturally visible, we demonstrate that Meltdown-type attacks can also be triggered without raising an architectural exception such as a page fault. Based on this, we demonstrate ZombieLoad, a novel, extremely powerful Meltdown-type attack targeting the fill-buffer logic.

ZombieLoad exploits that load instructions which have to be re-issued internally, may first transiently compute on stale values belonging to previous memory operations from either the current or a sibling hyperthread. Using established transient-execution attack techniques, adversaries can recover the values of such "zombie load" operations. Importantly, in con-

trast to all previously known transient-execution attacks [8], ZombieLoad reveals recent data values *without* adhering to any explicit address-based selectors. Hence, we consider ZombieLoad an instance of a novel type of *microarchitectural data sampling* (MDS) attacks. Unlike concurrent data sampling attacks like RIDL [61] or Fallout [55], our work includes the first and only attack variant that can leak data even on the most recent Intel Cascade Lake CPUs which are reportedly resistant against all known Meltdown, Foreshadow, and MDS variants. We present microarchitectural data sampling as the missing link between traditional memory-based side-channels which correlate data addresses within a victim execution, and existing Meltdown-type transient-execution attacks that can directly recover data values belonging to an explicit address. In this paper, we combine primitives from traditional side-channel attacks with incidental data sampling in the time domain to construct extremely powerful attacks with targeted leakage in the address domain. This not only opens up new attack avenues but also re-enables attacks that were previously assumed mitigated.

We demonstrate ZombieLoad's real-world implications in a multitude of practical attack scenarios that leak across processes, privilege boundaries, and even across logical CPU cores. Furthermore, we show that we can leak Intel SGX enclave secrets loaded from a sibling logical core, even on Foreshadow-resistant CPUs. We demonstrate that ZombieLoad attackers may extract sealing keys from Intel's architectural quoting enclave, ultimately breaking SGX's confidentiality and remote attestation guarantees. ZombieLoad is furthermore not limited to native code execution, but also works across virtualization boundaries. Hence, virtual machines can attack not only the hypervisor but also different virtual machines running on a sibling logical core. We conclude that disabling hyperthreading, in addition to flushing several microarchitectural states during context switches, is the only possible workaround to prevent this extremely powerful attack.

Contributions. The main contributions of this work are:

1. We present ZombieLoad, a powerful data sampling attack leaking data accessed on the same or sibling hyperthread.

2. We combine incidental data sampling in the time domain with traditional side-channel primitives to construct a targeted information flow similar to regular Meltdown attacks.

3. We demonstrate ZombieLoad in several real-world scenarios: cross-process, cross-VM, user-to-kernel, and SGX. ZombieLoad even works on Meltdown-resistant hardware.

4. We show that ZombieLoad breaks the security guarantees of Intel SGX, even on Foreshadow-resistant hardware.

5. We are the first to do post-processing of the leaked data within the transient domain to eliminate noise.

Outline. Sect. 2 provides background. Sect. 3 gives an overview of ZombieLoad, and introduces a novel classification for memory-based side-channel attacks. Sect. 4 describes attack scenarios and their attacker models. Sect. 5 introduces and evaluates the basic primitives required for mounting ZombieLoad. Sect. 6 demonstrates ZombieLoad in real-world attack scenarios. Sect. 7 discusses possible countermeasures. We conclude in Sect. 8.

Responsible Disclosure. We reported leakage of uncacheable-typed memory from a concurrent hyperthread on March 28, 2018, to Intel. We clarified on May 30, 2018 that we attribute the source of this leakage to the LFB. In our experiments, this works identically for Foreshadow, undermining the completeness of L1-flush-based mitigations. This issue was acknowledged by Intel and tracked under CVE-2019-11091 (MDSUM). We responsibly disclosed ZombieLoad Variant 1 to Intel on April 12, 2019. Intel verified and acknowledged our attack and assigned CVE-2018-12130 (MFBDS) to this issue. Both MDSUM and MFBDS were part of the Microarchitectural Data Sampling (MDS) embargo ending on May 14, 2019. We responsibly disclosed ZombieLoad Variant 2 (which is the only MDS attack that works on Cascade Lake CPUs) to Intel on April 24, 2019. This issue, which Intel refers to as Transactional Asynchronous Abort (TAA) is assigned CVE-2019-11135 and is part of an ongoing embargo ending on November 12, 2019. On May 16, 2019, we reported to Intel that their mitigations using `VERW` are incomplete and can be circumvented, which they verfied and acknowledged.

2 Background

In this section, we describe the background required for this paper.

2.1 Transient Execution Attacks

Today's high-performance processors typically implement an *out-of-order execution* design, allowing the CPU to utilize different execution units in parallel. The instruction stream is decoded *in-order* into simpler micro-operations (μOPs) [14] which can be executed as soon as the required operands are available. A dedicated reorder buffer stores intermediate results and ensures that instruction results are committed to the architectural state in-order. Any fault that occurred during the execution of an instruction is handled at instruction retirement, leading to a pipeline flush which squashes any outstanding μOP results from the reorder buffer.

In addition, modern CPUs employ *speculative execution* optimizations to avoid stalling the instruction pipeline until a conditional branch is resolved. The CPU predicts the outcome of the branch and continues execution along that direction. We refer to instructions that are executed speculatively or out-of-order but whose results are never architecturally committed as *transient instructions* [8, 47, 72].

While the results and the architectural effects of transient instructions are discarded, measurable microarchitectural side effects may remain and are not reverted. Attacks that exploit these side effects to observe sensitive information are called *transient execution attacks* [8, 44, 47]. Typically, these attacks utilize a cache-based covert channel to transmit the secret data observed transiently from the microarchitectural domain to an architectural state. In line with a recent exhaustive survey [8], we refer to attacks exploiting misprediction [28, 42, 44, 45, 51] as Spectre-type, whereas attacks exploiting transient execution after a CPU exception [8, 42, 47, 71, 72, 78] are classified as belonging to Meltdown-type.

2.2 Memory Subsystem

In this section, we overview memory loads in out-of-order CPUs.

Caches CPUs contain small and fast caches storing frequently used data. Caches are typically organized in multiple levels that are either private per core or shared amongst them. Modern CPUs typically use n-way set-associative caches containing n cache lines per set, each typically 64 B wide. Usually, Intel CPUs have a private first-level instruction (L1I) and data cache (L1D) and a unified L2 cache. The last-level cache (LLC) is shared across all cores.

Virtual Memory CPUs use virtual memory to provide memory isolation between processes. Virtual addresses are translated to physical memory locations using multi-level translation tables. The translation table entries define the properties, e.g., access control or memory type, of the referenced memory region. The CPU contains the translation-look-aside buffer (TLB) consisting of additional caches to store address-translation information.

Memory Order Buffer μOPs dealing with memory operations are handled by dedicated execution units. Typically, Intel CPUs contain 2 units responsible for loading and one for storing data. The *memory order* buffer (MOB), incorporating a *load buffer* and a *store buffer*, controls the dispatch of memory operations and tracks their progress to resolve memory dependencies.

Data Loads For every dispatched load operation an entry is allocated in the load buffer and the reorder buffer. To determine the physical address, the upper 36 b of the linear address are translated by the memory management unit. Concurrently, the untranslated lower 12 b are already used to index the cache set in the L1D [18]. If the address translation is in the TLB, the physical address is available immediately. Otherwise, the page miss handler (PMH) performs a page-table walk to retrieve the address translation as well as the corresponding permission bits. If the requested data is in the L1D (cache hit), the load operation can be completed.

If data is not in the L1D, it needs to be served from higher levels of the cache or the main memory via the line-fill buffer (LFB). The LFB serves as an interface to other caches and the main memory and keeps track of outstanding loads. Memory accesses to uncacheable memory regions, and non-temporal moves all go through the LFB.

On a fault, e.g., a physical address is not available, the page-table walk does not immediately abort [18]. An instruction in a pipelined implementation must undergo each stage and is simply reissued in case of a fault [2]. Only at the retirement of the faulting μOP, the fault is handled, and the pipeline is flushed [17, 18].

2.3 Processor Extensions

Microcode To support more complex instructions, *microcode* allows implementing higher-level instructions using multiple hardware-level in-

structions. This allows processor vendors to support complex behavior and even extend or modify CPU behavior through microcode updates [31]. Preferably, new architectural features are implemented as microcode extensions, e.g., Intel SGX [40].

While the execution units perform the fast-paths directly in hardware, more complex slow-path operations, such as faults or page-table modifications, are typically performed by issuing a *microcode assist* which points the sequencer to a predefined microcode routine [12]. To do so, the execution unit associates an event code with the result of the faulting micro-op. When the micro-op of the execution unit is committed, the event code causes the out-of-order scheduler to squash all in-flight micro-ops in the reorder buffer [12]. The microcode sequencer uses the event code to read the micro-ops associated with the event in the microcode [6].

Intel TSX Intel TSX is an x86 instruction set extension for hardware transactional memory [35] introduced with Intel Haswell CPUs. With TSX, particular code regions are executed transactionally. If the entire code regions completes successfully, memory operations within the transaction appear as an atomic commit to other logical processors. If an issue occurs during the transaction, a transactional abort rolls back the execution to an architectural state before the transaction, discarding all performed operations. Transactional aborts can be caused by different issues: Typically, a conflicting memory operation occurs where another logical processor either reads from an address which has been modified within the transaction or writes to an address which is used within the transaction. Further, the amount of read and written data within the transaction may not exceed the size of the LLC and L1 cache respectively [31]. In addition, some instructions or system event might cause the transaction to abort as well [35].

Intel SGX With the Skylake microarchitecture, Intel introduced Software Guard Extension (SGX), an instruction-set extension for isolating trusted code [31]. SGX executes trusted code inside so-called *enclaves*, which are mapped in the virtual address space of a conventional host application process but are isolated from the rest of the system by the hardware itself. The threat model of SGX assumes that the operating system and all other running applications could be compromised and, therefore, cannot be trusted. Any attempt to access SGX enclave memory in non-enclave mode results in a dummy value `0xff` [33]. Furthermore,

to protect against physical attackers probing the memory bus, the SGX hardware transparently encrypts the used memory region [12].

A dedicated `eenter` instruction redirects control flow to an enclave entry point, whereas `eexit` transfers back to the untrusted host application. Furthermore, in case of an interrupt or fault, SGX securely saves CPU registers inside the enclave's save state area (SSA) before vectoring to the untrusted operating system. Next, the `eresume` instruction can be used to restore processor state from the SSA frame and continue a previously interrupted enclave.

SGX-capable processors feature cryptographic key derivation facilities through the `egetkey` instruction, based on a CPU-level master secret and a secure measurement of the calling enclave's initial code and data. Using this key, enclaves can securely *seal* secrets for untrusted persistent storage, and establish secure communication channels with other enclaves residing on the same processor. Furthermore, to enable remote attestation, Intel provides a trusted *quoting enclave* which unseals an Intel-private key and generates an asymmetric signature over the local enclave identity report.

Over the past years, researchers have demonstrated various attacks to leak sensitive data from SGX enclaves, e.g., through memory safety violations [46], race conditions [77], or side-channels [56, 65, 73, 75]. More recently, SGX was also compromised by transient-execution attacks [10, 72] which necessitated microcode updates and increased the processor's security version number (SVN). All SGX key derivations and attestations include SVN to reflect the current microcode version, and hence security level.

3 Attack Overview

In this section, we provide an overview of ZombieLoad. We describe what can be observed using ZombieLoad and how that fits into the landscape of existing side-channel attacks. By that, we show that ZombieLoad is a novel category of side-channel attacks, which we refer to as *data-sampling attacks*, opening a new research field.

3.1 Overview

ZombieLoad is a transient-execution attack [8] which observes the values of memory loads and stores on the current CPU core. ZombieLoad exploits that the fill buffer is used by all logical CPUs of a CPU core and that it does not distinguish between processes or privileges.

Whenever the CPU encounters a memory load during execution, it reserves an entry in the load buffer. If the load was not an L1 hit, it requires a fill-buffer entry. When the requested data has been loaded, the memory subsystem frees the corresponding load- and fill-buffer entries, and the load instruction may retire. Similarly, if stores miss the L1 or are evicted from the L1, they are temporarily stored in a fill-buffer entry as well.

However, we observed that under certain complex microarchitectural conditions (e.g., a fault), where the load requires a microcode assist, it may first read stale values before being re-issued eventually. As with any Meltdown-type attack, this opens up a transient-execution window where this value can be used for subsequent calculations. Thus, an attacker can encode the leaked value into a microarchitectural element, such as the cache.

In contrast to previous Meltdown-type attacks, however, it is not possible to select the value to leak based on an attacker-specified address. ZombieLoad simply leaks any value which is currently loaded or stored by the physical CPU core. While this at first sounds like a massive limitation, we show that this opens a new field of data sampling-based transient-execution attacks. Moreover, in contrast to previous Meltdown-type attacks, ZombieLoad considers all privilege boundaries and is not limited to a specific one. Meltdown [47] can only leak data from the attacker's address space, Foreshadow [72] focussed exclusively on SGX enclaves, Foreshadow-NG [78] afterwards investigated cross-process and cross-VM leakage, and Fallout [55] can only leak kernel data on the same logical core. We show that ZombieLoad is an even more powerful attack in combination with existing side-channel techniques.

3.2 Microarchitectural Root Cause

For Meltdown, Foreshadow, Fallout, and RIDL, the source of the leakage is apparent. Moreover, for these attacks, there are plausible explanations on what is going wrong in the microarchitecture, *i.e.*, what the root cause of the leakage is [47, 55, 72, 78]. For ZombieLoad, however, this is not entirely clear.

While we identified some necessary building blocks to observe the leakage (cf. Section 5), we can only provide a hypothesis on why the interaction of the building blocks leads to the observed leakage. As we could only observe data leakage on Intel CPUs, we assume that this is indeed an implementation issue (such as Meltdown) and not a design

issue (as with Spectre). For our hypothesis, we combined our observations with the little official documentation of the fill buffer [30, 31] and Intel's MDS analysis [29]. Ultimately, we could neither prove nor disprove our hypothesis, leaving the verification or falsification of our hypothesis to future work.

Stale-Entry Hypothesis. Every load is associated with an entry in the load buffer and potentially an entry in the fill buffer [30].

When a load encounters a complex situation, such as a fault, it requires a microcode assist [31]. This microcode assist triggers a machine clear, which flushes the pipeline. On a pipeline flush, instructions which are already in flight still finish execution [27].

As this has to be as fast as possible to not incur additional delays, we expect that fill-buffer entries are optimistically matched as long as parts of the physical address match. Thus, the load continues with a wrong fill-buffer entry, which was valid for a previous load or store. This leads to a use-after-free vulnerability [24] in the hardware. Intel documents the fill buffer as being competitively shared among hyperthreads [31], giving both logical cores access to the entire fill buffer (cf. Appendix F). Consequently, the stale fill-buffer entry can also be from a previous load or store of the sibling logical core. As a result, the load instruction loads valid data from a previous load or store.

Leakage Source. We devised 2 experiments to reduce the number of possible sources of the leaked data.

In our first experiment, we marked a page as "uncacheable" and flushed it from the cache. As a result, every memory load from the page circumvents all cache levels and goes directly to the fill buffer [31]. We then write the secret onto the uncacheable page to ensure that there is no copy of the data in the cache. When loading data from the page, we see leakage in the order of bytes per second, e.g., 5.91 B/s ($\sigma_{\bar{x}} = 0.18$, $n = 100$, where n is the number of experiments and $\sigma_{\bar{x}}$ is the standard error of the mean) on an i7-8650U. We can attribute this leakage to the fill buffer. This was also exploited in concurrent work [61]. Our hypothesis is further backed by the `MEM_LOAD_RETIRED.FB_HIT` performance counter, which shows multiple thousand line-fill-buffer hits (117 330 FB_HIT/s ($\sigma_{\bar{x}} = 511.57$, $n = 100$)).

Intel claims that the leakage is entirely from the fill buffer [29]. This is also what Van Schaik et al. [61] conclude for their RIDL attack. However, our second experiment shows that the line-fill buffer might not be the only source of the leakage for ZombieLoad. We rely on Intel TSX to

ensure that memory accesses do not reach the line-fill buffer as follows. Inside a transaction, we first write the secret value to a memory location which was previously initialized with a different value. The write inside the transaction ensures that the address is in the *write set* of the transaction and thus in L1 [30, 62]. Evicting data from the write set from the cache leads to a transactional abort [30]. Hence, any subsequent memory access to the data from the write set ensures that it is served from the L1, and therefore, no request to the line-fill buffer is sent [31]. In this experiment, we see a much higher rate of leakage, which is in the order of kilobytes per second. More importantly, we only see the value written inside the TSX transaction and not the value that was at the memory location before starting the transaction. Our hypothesis that the line-fill buffer is not the only source of the leakage is further backed by observing performance counters. The `MEM_LOAD_RETIRED.FB_HIT` and `MEM_LOAD_RETIRED.L1_MISS` performance counters do not increase significantly. In contrast, the `MEM_LOAD_RETIRED.L1_HIT` performance counter shows multiple thousand L1 hits.

While accessing the data to leak on the victim core, we monitored the `MEM_LOAD_RETIRED.FB_HIT` performance counter on the attacker core for 10 s. If the address was cached, we measured a Pearson correlation of $r_p = 0.02$ ($n = 100$) between the correct recoveries and line-fill buffer hits, indicating no association. However, while continuously flushing the data on the victim core, ensuring that a subsequent access must go through the LFB, we measure a strong correlation of $r_p = 0.86$ ($n = 100$). This result indicates that the line-fill buffer is not the only source of leakage. However, a different explanation might be that the performance counters are not reliable in such corner cases. Van Schaik et al. [61] reported that the RIDL attack can only leak data which is not served from the cache, *i.e.*, which has to go through the fill buffers. Hence, we conclude that RIDL indeed leaks from fill buffers, whereas the ZombieLoad leakage might not be entirely attributed to the fill buffer. Future work has to investigate whether other microarchitectural elements, e.g., the load buffer, are also involved in the observed data leakage.

Comparison to RIDL In concurrent work, Van Schaik et al. [61] presented the RIDL attack, which also leaks data from the fill buffers, as well as from the load ports. Table 11.1 shows a table which summarizes the main differences between RIDL and ZombieLoad. The most crucial difference between the attacks is that ZombieLoad still works on the newest generation of Intel CPUs (Cascade Lake with stepping B1) which

Table 11.1: Comparison between the RIDL attack [61] and ZombieLoad.

	RIDL	**ZombieLoad**
Leakage Source	Fill Buffer, Load Port	Fill Buffer
Leaked Loads	Uncached Loads Only (Fill Buffer)	All Loads (Fill Buffer)
Leaked Stores	All Stores (Fill Buffer)	All Stores (Fill Buffer)
Known Variants	1 or 2[†]	5
Exploited Fault	Page Fault	Microcode Assist, Page Fault
Fixed with Countermeasures	✓	✘
Works on MDS-resistant CPUs	✘	✓ (Variant 2)

[†] The RIDL paper [61] only describes one variant leaking from the fill buffers, but also mentions a variant leaking from the load ports without further description or evaluation.

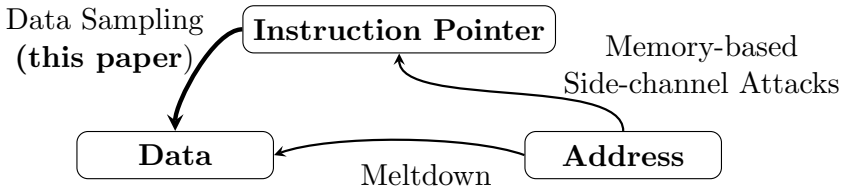


Figure 11.1: The 3 properties of a memory operation: instruction pointer of the program, target address, and data value. So far, there are techniques to infer the instruction pointer from target address and the data value from the address. With ZombieLoad, we show the first instance of an attack which infers the data value from the instruction pointer.

are not affected by RIDL or Fallout. RIDL can only leak loads which are not currently in the L1 cache. ZombieLoad can leak all loads, independent whether they are currently in the L1 cache or not. ZombieLoad has a thorough analysis of the microarchitectural root cause, which leads to more variants with unique features, such as leakage on an MDS-resistant CPU.

3.3 Classification

In this section, we introduce a way to classify memory-based side-channel and transient-execution attacks. For all these attacks, we assume a target program which executes a memory operation at a certain *address* with a specific data *value* at the program's current *instruction pointer*. Figure 11.1 illustrates these three properties as the corner of a triangle, and techniques which let an attacker infer one of the properties based on one or both of the other properties.

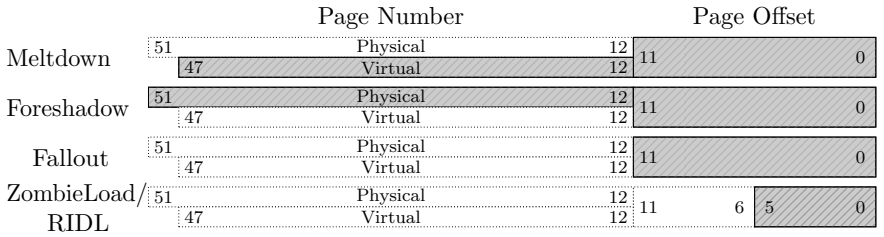


Figure 11.2: Meltdown-type attacks provide a varying degree of target control (gray hatched), from full virtual addresses in the case of Meltdown to nearly no control for ZombieLoad.

Traditional memory-based side-channel attacks allow an attacker to observe the location of memory accesses. The granularity of the location observation depends on the spatial accuracy of the used side channel. Most common memory-based side-channel attacks [19–21, 23, 39, 58, 59, 75, 80, 82] have a granularity between one cache line [20, 21, 23, 82] *i.e.*, usually 64 B, and one page [19, 39, 75, 80], *i.e.*, usually 4 kB. These side channels establish a connection between the time domain and the space domain. The time domain can either be the wall time or also commonly the execution time of the program which correlates with the instruction pointer. These classic side channels provide means of connecting the address of a memory access to a set of possible instruction pointers, which then allows reconstructing the program flow. Thus, side-channel resistant applications have to avoid secret-dependent memory access to not leak secrets to a side-channel attacker.

Since early 2018, with transient-execution attacks [8] such as Meltdown [47] and Spectre [44], there is a second type of attacks which allow an attacker to observe the value stored at a memory address. Meltdown provided the most control over target address. With Meltdown, the full virtual address of the target data is provided, and the corresponding data value stored at this address is leaked. The success rate depends on the location of the data, *i.e.*, whether it is in the cache or main memory. However, the only constraint for Meltdown is that the data is addressable using a virtual address [47]. Other Meltdown-type attacks [55, 72] also connect addresses to data values. However, they often impose additional constraints, such as that the data has to be cached in L1 [72, 78], the physical address has to be known [78], or that an attacker can choose only parts of the target address[55, 61].

Figure 11.2 illustrates which parts of the virtual and physical address an attacker can choose to target data values to leak. For Meltdown, the

virtual address is sufficient to target data in the same address space [47]. Foreshadow already requires knowledge of the physical address and the least-significant 12 bits of the virtual address to target any data in the L1, not limited to the own address space [72, 78]. When leaking the last writes from the store buffer, an attacker is already limited in choosing which value to leak. It is only possible to filter stores based on the least-significant 12 bits of the virtual address, a more targeted leakage is not possible [55].

Zombie loads, which are exploited by ZombieLoad and RIDL [61], provide no control over the leaked address to an attacker. The only possible target selection is the byte index inside the loaded data, which can be seen as an address with up to 6-bit in case an entire cache line is loaded. Hence, we do not count ZombieLoad and RIDL as an attack which leaks data values based on the address. Instead, from the viewpoint of the target control, ZombieLoad and RIDL are more similar to traditional memory-based side-channel attacks. With ZombieLoad and RIDL, an attacker observes the data value of a memory access. Thus, this side channel establishes a connection between the time domain and the data value. Again, the time domain correlates with the instruction pointer of the target address. ZombieLoad and RIDL are the first instances of a class of attacks which connects the *instruction pointer* with the *data value* of a memory access. We refer to such attacks as *data sampling attacks*. Essentially, this new class of data sampling attacks is capable of breaking side-channel resistant applications, such as constant-time cryptographic algorithms [26].

Following the classification scheme from Canella et al. [8], ZombieLoad is a Meltdown-type transient-execution attack, and we propose *Meltdown-MCA* as the canonical name for exploiting microcode assists (MCA, explained further) as exception type. We can further classify the different variants of ZombieLoad (cf. Section 5.1). We propose Meltdown-US-LFB for ZombieLoad Variant 1, as it exploits a page fault on a supervisor page to leak from the fill buffer. For ZombieLoad Variant 2, we propose Meltdown-MCA-TAA (microcode assist caused by transactional asynchronous abort), and for ZombieLoad Variant 3 Meltdown-MCA-AD (micorcode assist caused by modifying the accessed or dirty bit). The RIDL attack exploits non-present page faults caused by NULL-pointer accesses [61]. Thus, we propose the canonical name Meltdown-P-LFB for the RIDL attack.

4 Attack Scenarios & Attacker Model

Following most side-channel attacks, we assume the attacker can execute unprivileged native code on the target machine. We assume a trusted operating system if not stated otherwise. This relatively weak attacker model is sufficient to mount ZombieLoad. However, we also show that the increased attacker capabilities offered in certain scenarios, e.g., SGX and hypervisor attacks, may amplify the leakage while remaining within the respective threat model.

At the hardware level, we assume a ubiquitous Intel CPU with simultaneous multithreading (SMT, also known as hyperthreading) enabled. Crucially, we do not rely on existing vulnerabilities, such as Meltdown [47], Foreshadow [72, 78], or Fallout [55]. Hence, even the most recent Intel 9th generation processors with silicon-level Meltdown mitigations remain within our threat model.

User-Space Leakage In the cross-process user-space scenario, an unprivileged attacker leaks values loaded or stored by another concurrently running user-space application. We consider such a cross-process scenario most dangerous for end users. Many secrets are likely to be found in user-space applications such as browsers.

The attacker is co-located with the victim on the same physical but a different logical CPU core, a common case for hyperthreading.

Kernel Leakage ZombieLoad can also leak across the privilege boundary between user and kernel space. The values of loads and stores executed in kernel space are leaked to an unprivileged attacker, executing either on the same or a sibling logical core.

An unprivileged attacker performs a system call to the kernel, running on the same logical core. Importantly, we found that kernel load leakage may even survive the switch back from the kernel to user space. Hence, hyperthreading is *not* required for this scenario.

Intel SGX Leakage ZombieLoad can observe loads and stores executed inside an SGX enclave, even if the loads and stores target the encrypted memory region, *i.e.*, the enclave page cache. The attacker is executing outside of an SGX enclave on a sibling logical core, co-located with the victim enclave on the same physical core. In contrast to the kernel leakage, we did not observe leakage on the *same* logical core after exiting the enclave.

Intel [36] suggests that a remote verifier might reject attestations from a hyperthreading-enabled system "if it deems the risk of potential attacks from the sibling logical processor as not acceptable". Hence, hyperthreading can decidedly be enabled safely on recent Intel Cascade Lake CPUs which include hardware mitigations against Foreshadow [36], but even older SGX machines with up-to-date patched microcode may still run with hyperthreading enabled.

Within the SGX threat model, an attacker can, e.g., modify page table entries [75], or precisely execute the victim enclave at most one instruction at a time [74].

Virtual Machine Leakage ZombieLoad can leak loaded and stored values across virtual-machine boundaries. An attacker running inside a virtual machine can leak values from a different virtual machine co-located on the same physical but different logical core.

As the attacker is running inside an untrusted virtual machine, the attacker is not restricted to unprivileged code execution. Thus, the attacker can, e.g., modify guest-page-table entries.

Hypervisor Leakage An attacker inside a virtual machine can use ZombieLoad to leak values of loads and stores executed by the hypervisor.

As the attacker is running inside an untrusted virtual machine, the attacker is not restricted to unprivileged code execution.

5 Building Blocks

In this section, we describe the building blocks for the attack.

5.1 Zombie Loads

The main primitive for mounting ZombieLoad is a load which triggers a microcode assist, resulting in a transient load containing wrong data. We refer to such a load as a *zombie load*. Zombie loads are loads which either architecturally or microarchitecturally fault and thus cannot complete, requiring a re-issue of the load at a later point. We identified multiple different scenarios (cf. Appendix G) to create such zombie loads required for a successful attack. Most variants have in common that they abuse the `clflush` instruction to reliably create the conditions required for leaking from a wrong destination (cf. Section 3.2). In this section, we describe 3 different variants that can be used to leak data (cf. Section 5.2)

Table 11.2: Overview of different variants to induce zombie loads in different scenarios.

Scenario \ Variant	1	2	3
Unprivileged Attacker	○ ◐	◐ ◐	● ○
Privileged Attacker (root)	● ●	◐ ◐	● ●

Symbols indicate whether a variant can be used in the corresponding attack scenario (●), can be used depending on the hardware configuration as discussed in Sect. 5.1 (◐), or cannot be used (○).

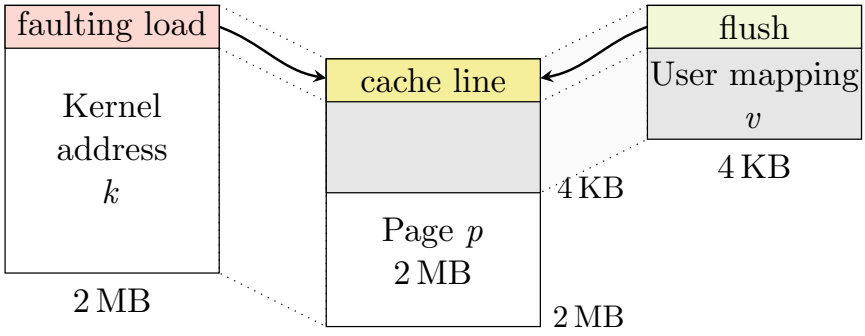


Figure 11.3: Variant 1: Using huge kernel pages for ZombieLoad. Page p is mapped using a user-accessible address (v) and a kernel-space huge page (k). Flushing v and then reading from k using Meltdown leaks values from the fill buffer.

depending on the adversary's capabilities. While there are more variants (cf. Appendix G and Van Schaik et al. [61] for more known variants), these 3 variants are fast, and each has a unique feature. Table 11.2 overviews which variants are applicable in which scenarios, depending on the operating system and underlying hardware configuration.

Variant 1: Kernel Mapping. The first variant is a ZombieLoad setup which does not rely on any specific CPU feature. We require a kernel virtual address k, *i.e.*, an address where the user-accessible bit is *not* set in the page-table entry. In practice, the kernel is usually mapped with huge pages (*i.e.*, 2 MB pages). Thus k refers to a 2 MB physical page p. Note that although we use such huge pages for our experiments, it is not strictly required, as the setup also works with 4 kB pages. We also require

the user to have read access to the content of the physical page through a different virtual address v.

Figure 11.3 illustrates such a setup. In this setup, accessing the page p via the user-accessible virtual address v provides an architecturally valid way to access the contents of the page. Accessing the same page via the kernel address k results in a zombie load similar to Meltdown [47] requiring a microcode assist. Note that while there are other ways to construct an inaccessible address k, e.g., by clearing the present bit [72], we were only able to exploit zombie loads originating from kernel mappings.

To create precisely the scenario depicted in Figure 11.3, we allocate a page p in the user space with the virtual address v. Note that p is a regular 4 kB page which is accessible through the virtual address v. We retrieve its physical address through `/proc/pagemap`, or alternatively using a side channel [23, 38, 67]. Using the physical address and the base address of the direct-physical map, we get an inaccessible kernel address k which maps to the allocated page p. If the operating system does not use stronger kernel isolation [22], e.g., KPTI [50], the direct-physical map in the kernel is mapped in the user space and uses huge pages which are marked as not user accessible. A privileged attacker (e.g., for hypervisor or SGX-enclave attacks) can easily create such pages if they do not exist.

The disadvantage of this approach is that it does not work on Meltdown-resistant machines. There, we have to use Variant 2.

Variant 2: Intel TSX With the second variant of inducing zombie loads, we eliminate the requirement of a kernel mapping. We only require a physical page p which is user accessible via a virtual address v. Any page allocated in user space fulfills this requirement.

Within a TSX transaction, we encode the value of v in a cache covert-channel likewise to Spectre or Meltdown. This ensures that v is in the read set of the transaction [30]. Note that we perform a legitimate load to the user-accessible address v which itself should not cause the TSX transaction to fail. However, by inducing conflicts in the read set (cf. Section 2.3), the TSX transaction "faults" and does not commit. There is no architectural fault but only a transient fault which results in a zombie load.

The main advantage of this approach is that it also works on machines with hardware fixes for Meltdown, which we verified on an i9-9900K and Xeon Gold 5218. However, in contrast to Variant 1, we require the Intel TSX instruction-set extension which is only available in selected CPUs since 2013.

Variant 3: Microcode-Assisted Page-Table Walk. A variant similar to Variant 1 is to trigger a microcode-assisted page-table walk. If a page-table walk requires an update to the access or dirty bit in the page-table entry, it falls back to a microcode assist [12].

In this setup, we require one physical page p which has 2 user-accessible virtual addresses, v and v_2. This can be easily achieved by using a shared-memory segment or memory-mapped file, which is mapped twice in the application. The virtual address v can be used to access the contents of p architecturally. For v_2, we have to clear the accessed bit in the page-table entry. On Linux, this is not possible in the case of an unprivileged attacker, and can thus only be used in attacks where we assume a privileged attacker (cf. Section 4). However, we experimentally verified that Windows 10 (1803 build 17134.706) periodically clears the accessed bits. We assume that the page-replacement algorithm is responsible for this. Thus, this variant enables the attack on Windows for unprivileged attackers if the CPU does not support Intel TSX.

When accessing the page through the virtual address v_2, the accessed bit of the page-table entry has to be set. This, however, cannot be done by the page-miss handler [12]. Instead, microarchitecturally, the load faults, and a micro-code assist is triggered which repeats the page-table walk and sets the accessed bit [12].

If the access to v_2 is done transiently, *i.e.*, behind a misspeculated branch or after an exception, the accessed bit cannot be set architecturally. Thus, the leakage is not only exploitable once but instead for every access.

5.2 Data Leakage

To leak data with any setup described in Section 5.1, we constantly flush the first cache line of p through the virtual address v. We achieve this by executing the unprivileged `clflush` instruction on the user-accessible virtual address v. For Variant 1, we leverage Meltdown to read from the kernel address k which maps to the cache line flushed before. As with Meltdown-US [47], there are various methods of preventing an architectural exception. We verified that ZombieLoad with Variant 1 works with exception prevention (*i.e.*, speculative execution), handling (*i.e.*, a custom signal handler), and suppression (*i.e.*, Intel TSX).

For Variant 2, the cache-line invalidation of the flush triggers a conflict in the read set of the transaction and aborts the transaction. As there is no architectural exception on a transactional conflict, there is no need to handle exceptions.

For Variant 3, we transiently, *i.e.*, behind a mispredicted branch, read from the address v_2. Similar to Variant 2, there is no architectural exception. Hence, there is no need to handle exceptions.

Counterintuitively, the resulting values leaked for all variants are not coming from page p. Instead, we get access to data which is currently loaded or stored on the current or sibling logical CPU core. Thus, it appears that we reuse fill-buffer entries, and leak the data which the entries references. For Variant 1 and Variant 3, this allowed us to access all bytes from the cache line that the fill-buffer entry references. However, for Variant 2, we are only able to recover the number of bytes of the victim's load or store operation and in contrast to Variant 1, not the entire cache line.

5.3 Data Sampling

Independent of the setup for ZombieLoad, we cannot directly control the address of the data to leak. Both the virtual addresses k and v, as well as the physical address of p is arbitrary and does not correlate with the leaked data. In any case, we simply get the value referenced by one fill-buffer entry which we cannot specify.

However, there is at least control within the fill-buffer entry, *i.e.*, we can target specific bytes *within* the 64 B fill-buffer entry. The least-significant 6 bits of the virtual address v refer to the byte within the fill-buffer entry. Hence, we can target a single byte at a specific position from the fill-buffer entry. While at first, this does not sound powerful, it allows leaking sensitive information, such as AES keys, byte-by-byte as shown in Section 6.1.

As described in Section 4, the leakage is not limited to the own process. With ZombieLoad, we observe values from all processes running on the same as well as on the sibling logical CPU core. Furthermore, we also observe leakage across privilege boundaries, *i.e.*, from the kernel, hypervisor, and Intel SGX enclaves. Thus, ZombieLoad allows sampling of all data which is loaded or stored by any application on the current physical CPU core.

5.4 Performance Evaluation

In this section, we evaluate ZombieLoad and the performance of our proof-of-concept implementations[1].

[1] Our proof-of-concept implementations can be found in a GitHub repository: `https://github.com/IAIK/ZombieLoad`

Table 11.3: Tested environments. A '✓' indicates that the version works, '✘' that it does not work, and '-' that TSX is disabled or not supported on this CPU.

			Variant		
Setup	**CPU (Stepping)**	**μ-arch.**	**1**	**2**	**3**
Lab	Core i7-3630QM (E1)	Ivy Bridge	✓	-	✓
Lab	Core i7-6700K (R0)	Skylake-S	✓	✓	✓
Lab	Core i5-7300U (H0)	Kaby Lake	✓	✓	✓
Lab	Core i7-7700 (B0)	Kaby Lake	✓	✓	✓
Lab	Core i7-8650U (Y0)	Kaby Lake-R	✓	✓	✓
Lab	Core i7-8565U (W0)	Whiskey Lake	✘	-	✘
Lab	Core i7-8700K (U0)	Coffee Lake-S	✓	✓	✓
Lab	Core i9-9900K (P0)	Coffee Lake-R	✘	✓	✘
Lab	Xeon E5-1630 v4 (R0)	Broadwell-EP	✓	✓	✓
Cloud	Xeon E5-2670 (C2)	Sandy Bridge-EP	✓	-	✓
Cloud	Xeon Gold 5120 (M0)	Skylake-SP	✓	✓	✓
Cloud	Xeon Platinum 8175M (H0)	Skylake-SP	✓	-	✓
Cloud	Xeon Gold 5218 (B1)	Cascade Lake-SP	✘	✓	✘

Environment We evaluated the different variants of ZombieLoad, described in Section 5.1, on different environments listed in Table 11.3. The tested CPUs range from Sandy Bridge (released 2012) to Cascade Lake (released 2019). While we were able to mount Variant 1 and Variant 3 on different microarchitectures except for Whiskey Lake, Coffee Lake-R, and Cascade Lake-SP, we successfully used Variant 2 on all systems where Intel TSX was available. Thus, Variant 2 also works on microarchitectures with hardware mitigations against Meltdown and Foreshadow.

Performance To evaluate the performance of each variant, we performed the following experiment on an i7-8650U. While reading a specific value on one logical core, we performed each variant of ZombieLoad on the sibling logical core for 10 s, recording the number of successful and unsuccessful recoveries. For Variant 1 using TSX to suppress the exception, we achieve an average transmission rate of 5.30 kB/s ($\sigma_{\bar{x}} = 0.076$, $n = 1000$) and a true positive rate of 85.74 % ($\sigma_{\bar{x}} = 0.0046$, $n = 1000$). For Variant 2, we achieved an average transmission rate of 39.66 kB/s ($\sigma_{\bar{x}} = 0.048$, $n = 1000$) and a true positive rate of 99.99 % ($\sigma_{\bar{x}} = 6.45^{-9}$, $n = 1000$). With Variant 3 in combination with signal handling, we achieved an average transmission rate of 0.08 kB/s ($\sigma_{\bar{x}} = 0.002$, $n = 1000$)

and a true positive rate of 52.7 % ($\sigma_{\bar{x}} = 0.0062$, $n = 1000$). Variant 3 in combination with TSX, achieves an average transmission rate of 7.73 kB/s ($\sigma_{\bar{x}} = 0.21$, $n = 1000$) and a true positive rate of 76.28 % ($\sigma_{\bar{x}} = 0.0055$, $n = 1000$).

6 Case Study Attacks

In this section, we present 5 attacks using ZombieLoad in real-world scenarios.

6.1 AES-NI Key Leakage

To demonstrate that data sampling is a powerful side channel, we extract an AES-128 key. The victim application uses AES-NI, which is resistant against timing and cache-based side-channel attacks [26].

However, even with the hardware-assisted AES-NI, the key has to be loaded from memory to a 128-bit XMM register. This is usually the case before invoking `AESKEYGENASSIST`, which is used to derive the AES round keys. The round-key derivation is entirely done in hardware using the XMM registers. Hence, there is no memory load required for the derivation of the 11 round keys used in AES-128. Thus, when the key is loaded from memory before the round-key derivation starts is the point where we can mount ZombieLoad to leak the value of the key. For OpenSSL (v3.0.0), this is in the function `aesni_set_encrypt_key` which is called by `EVP_EncryptInit_ex`. Note that instead of leaking the key, we can also leak the round keys loaded in the encryption process. However, to attack the round keys, an attacker needs to leak (and distinguish) more different values, making the attack more complex.

When leaking the key using ZombieLoad, we have first to detect which load corresponds to the key. Moreover, as we can only leak one byte at a time, we also have to combine the leaked bytes to the full AES-128 key correctly.

Side-Channel Synchronization. For the attack, we assume a shared library implementing the AES encryption, e.g., OpenSSL. Even though OpenSSL (v3.0.0) has a side-channel resistant AES-NI implementation, we can rely on classical memory-based side channels to monitor the control flow. With Flush+Reload, we detect when a specific code part is executed [15, 20]. This does not leak any secrets, but it is a synchronization primitive for ZombieLoad.

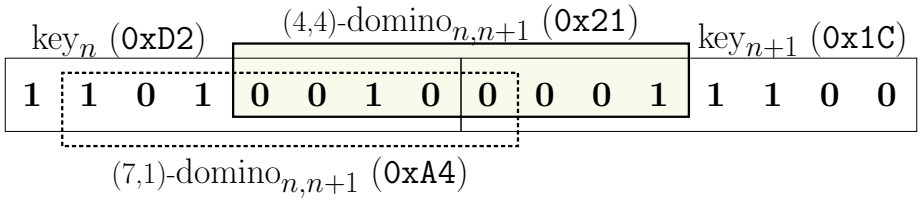


Figure 11.4: Additionally leaking domino bytes comprised of bits of different AES-key bytes to filter out unrelated loads.

We constantly monitor a cache line of the code which is executed right before the key is loaded from memory. In OpenSSL (v3.0.0), this is the second cache line of `aesni_set_encrypt_key`, *i.e.*, 64 B after the start of the function. Similarly to Schwarz et al. [62], we leverage the cache state of the cache line as a trigger for the actual attack. Only if we detect a cache hit on the monitored cache line, we start leaking values using ZombieLoad. Hence, we already filter out most bytes not related to the AES key. Note that the synchronization does not have to be perfect, as independent system noise cancels itself out over multiple measurements. Moreover, the key is always 16 B aligned, and we always leak an entire cache line. Hence, there can be no bytewise shift of the AES key – the first 16 B that we leak are always either from the key or from unrelated noise.

Note that if there is no cache line before the load which can be used as a trigger, we can still use a nearby cache line (*i.e.*, a cache line after the load) as a filter. In a parallel thread, we collect the timestamps of cache hits in the nearby cache line. If we also save the timestamps of the values leaked using ZombieLoad, in an offline post-processing step, we can filter out values which were leaked at a different instruction-pointer location.

To further reduce unrelated loads, it is also possible to slow down the victim using performance-degradation techniques such as flushing the code [3, 15]. For OpenSSL, we used performance degradation on the code directly following the load of the key.

Domino Attack. Inevitably, even when synchronizing ZombieLoad by using a cache-based trigger, we also leak values not related to the key. As the bytes in the AES key are independent of each other, we can only assume that the byte which we leak most often per byte position is the correct key byte. Thus, if there is a key byte suffering from noise from

unrelated loads, we may assume that the noise is the correct key byte, which leads to a wrong key.

Therefore, we propose the *Domino attack*, an innovative transient error-detection technique for reducing noise when leaking multi-byte loads. In addition to leaking every single key byte, we transmit a specially crafted *domino byte* composed by combining bits from two adjacent key bytes. Note that creating such a domino byte is possible, as the transient domain has access to the full AES key and can use it for arbitrary computations (as also shown with the transient error detection described in Section 6.3). Figure 11.4 illustrates the idea of the Domino attack. In this case, we leak (4,4) domino bytes consisting of 4 bits of two adjacent key bytes respectively. By combining the lower nibble of one key byte with the higher nibble of the next key byte, we transmit a domino byte which encodes partial information of two key bytes.

In a post-processing step, we consider two adjacent bytes as correct, if we not only leaked both of them often but additionally also the corresponding domino byte. Moreover, we do not look at two key bytes in isolation, but we look at the entire key as a chain of key bytes linked together by domino bytes. If all key bytes and the corresponding domino bytes occurred often in the leaked values, we can assume that the entire key is leaked correctly. Note that the selection of bits can be adapted to the noise measurable before leaking the key, e.g., multiple(7,1) domino bytes can be leaked that are shifted by only a single bit.

Results. We evaluated the attack in a cross-user-space attack (cf. Section 4) using Variant 1. We always ran the attack until the correct key was recovered, *i.e.*, until the key with the highest probability is the correct key. In a practical attack, the number of attacks can even be reduced, as typically it is easy to verify whether a key candidate is correct. Thus, an attacker can simply test all key candidates with a probability over a certain threshold and does not have to wait until the highest probability corresponds to the correct key.

On average, we recovered the entire AES-128 key of the victim in under 10 s using the cache-based trigger and the Domino attack. During this time, the victim loaded the key approximately 10 000 times.

6.2 SGX Sealing Key Extraction

In this section, we show that privileged SGX attackers can drastically improve ZombieLoad's temporal resolution and bridge from incidental data

sampling in the time domain to the targeted reconstruction of arbitrary enclave secrets (cf. Fig. 11.1). We first explain how state-of-the-art enclave execution control and transient post-processing techniques can be leveraged to reliably leak register values at any point during an enclave invocation. Then we demonstrate the impact of this attack by recovering a full 128-bit SGX sealing key, as used by Intel's trusted provision and quoting enclaves to decrypt the long-term EPID private attestation key.

Leaking Enclave Registers. We consider Intel SGX root attackers that co-locate with a victim enclave on the same physical CPU. As a system attacker, we can increase ZombieLoad's temporal resolution by leveraging previous research results exploiting page faults [75, 80] or interrupts [56, 73] to regulate the victim enclave's execution. We use the SGX-Step [74] framework to precisely single-step the victim enclave one instruction at a time, allowing the attacker to reach a code part where sensitive information is stored in CPU registers. At such a point, we switch to unlimited zero-stepping [72] by either setting the system timer interrupt to a very short interval or revoking code page execute permissions before resuming the victim enclave. This technique provides ZombieLoad attackers with a primitive to repeatedly force-reload CPU registers from the interrupted enclave's SSA frame (cf. Sect. 2.3). Our experiments show that even though the execution of the enclave instruction never completes, any direct operands plus SSA register file contents are loaded from memory each time. Importantly, since the enclave does not make progress, we can perform unlimited ZombieLoad attack attempts to reconstruct CPU register values from these implicit SSA memory accesses.

We further reduce noise from unrelated non-enclave loads on the victim CPU by opting for timer-based zero-stepping with a user-space interrupt handler [73] to avoid repeatedly invoking the operating system. Furthermore, we found that executing the ZombieLoad attack code in a separate address space avoids unnecessarily slowing down the spy through implicit TLB invalidations on enclave entry/exit [33].

Note that the SSA frame spans multiple cache lines. With ZombieLoad, we do not have explicit address-based control over which cache line is being leaked. Hence, leaked data might come from different saved registers that are at the same offset within a cache line. To filter out such noisy observations, we use the Domino transient error detection technique introduced in Sect. 6.1. Specifically, we implemented a "sliding window" that transmits 7 different domino bytes for each candidate key byte, stuffed with increasing bits from the next adjacent key byte candidate. Any noisy

observations that do not match the overlap can now efficiently be filtered out.

Attack on `sgx_get_key`. The Intel SGX design includes a secure key derivation facility through the `egetkey` instruction (cf. Section 2.3). Enclaves execute this instruction to query a 128-bit cryptographic key from the hardware, based on the calling enclave's code layout or developer identity. This is the underlying primitive used by Intel's trusted prebuilt quoting enclave to unseal a long-term private attestation key from persistent storage securely [12, 72].

The official Intel SGX SDK [33] offers a convenient `sgx_get_key` wrapper procedure that first executes `egetkey` with the necessary parameters, and eventually copies the retrieved key into a provided buffer. We reverse engineered the proprietary `intel_fast_memcpy` function and found that in this case, the key is copied using two 128-bit moves to/from the `xmm0` SSE register. We revert to zero-stepping on the last instruction of `memcpy`. At this point, the attacker-induced zero-step enclave resumptions will repeatedly reload a.o., the `xmm0` register containing the 128-bit key from the memory hierarchy.

Results. We evaluated the attack on a Kaby Lake i7-7700 CPU with an up-to-date Foreshadow-patched microcode revision 0x8e and ZombieLoad Variant 1.

In the first experiment, we implemented a benchmark enclave that uses `sgx_get_key` to generate a new report key with different random key IDs. We performed 100 key-recovery experiments on `sgx_get_key` with different random keys. Our results show that 30 % of the times (in 30 experiments) the full 128-bit key is among the key candidates with average remaining key space entropy of 8.8 bits. This entropy is calculated by averaging the entropy of these 30 cases where the full key is among the 128-bit candidates. Among these cases, 3 % of the times the exact full key has been recovered, and the worst-case entropy is about 14 bits. In the other 70 % of the cases where the full key is not among the key candidates, 31 % of the times, we have partial key bytes among the recovered key candidates. The average correct key bytes are 10 out of 16 bytes. In such cases, where some of the key bytes are part of the candidates, most of the failed key bytes reside in the first few bytes of the key. The reason is that the Domino attack has a stronger effect on key bytes in the middle that are surrounded by more key bytes. In the remaining 39 % of the times where the correct key is not among the key candidates, our attack

which uses the Domino technique with a sliding window did not reveal any candidates, which means an attacker can simply repeat the attack in such cases.

In the second experiment, we perform an attack on Intel's trusted quoting enclave. The quoting enclave performs a call to `sgx_get_key` to derive the sealing key which is used to decrypt the EPID provisioning blob. We executed the attack on a quoting enclave that is signed with debug keys, so we can use it as ground truth to easily verify that we have recovered the correct sealing key. We executed the attack multiple times on our setup, and we managed to recover the correct 128-bit sealing key after multiple executions of the attack and checking the candidates against each other. The recovered sealing key matches the correct key, and can indeed successfully decrypt the EPID blob for our debug signed quoting enclave. While we did not yet reproduce this attack on the official quoting enclave image signed by Intel, we believe that this experimental evaluation showcased all the required primitives to break Intel SGX's remote attestation guarantees, as demonstrated before by Foreshadow [72].

6.3 Cross-VM Covert Channel

To evaluate the performance of ZombieLoad, we implement a covert channel which can be used for all attack scenarios described in Section 4. However, in this section, we focus on the cross-VM covert channel. While covert channels are possible for Intel SGX, the kernel, and the hypervisor, these are somewhat artificial scenarios. Moreover, there are various covert channels available to user-space applications for stealthy inter-process communication [16, 53].

For VMs, however, there are not many known covert channels which can be used between two VMs. So far, all cross-VM covert channels either relied on Prime+Probe [48, 52, 53, 60, 81], DRAMA [59, 63], or bus locking [79]. We show that ZombieLoad can be used as a fast and reliable covert channel between VMs scheduled on the same physical core.

Sender. For the fastest result, the sender repeatedly loads the value to be transmitted from the L1 cache into a register. By not only loading the value from one memory address but instead from multiple memory addresses, the sender ensures that potentially multiple fill-buffer entries are used. In addition, this also thwarts an optimization of Intel CPUs which combines multiple loads from the same cache line to a single load [1].

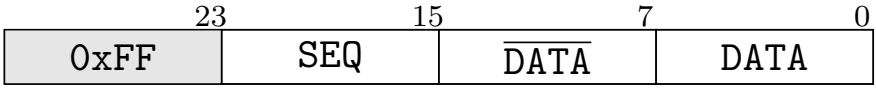


Figure 11.5: The packet format used in the covert channel. Every 32-bit packet consists of 8 data bits, 8-bit checksum (two's complement), 8-bit sequence number, and a constant prefix.

On a CPU supporting AVX2, the sender can encode up to 256 bits per load (e.g., using the `VMOVAPS` load).

Receiver. The receiver mounts ZombieLoad to leak the values loaded by the sender. However, as the receiver leaks the loads only in the transient domain, the leaked value have to be transferred into the architectural domain. We encode the leaked values into the cache and recover them using Flush+Reload. When encoding values in the cache, we require at least 2 cache lines, *i.e.*, 128 B, per bit to prevent the adjacent-cache-line prefetcher from interfering with the encoding. In practice, we require one physical page per possible value to prevent prefetcher interference. To reduce the bottleneck, we transfer single bytes from the transient to the architectural domain which already requires 256 runs of Flush+Reload.

As a result, our proof-of-concept limits the transmission of data to a single byte per leaked load. However, we can use the remaining bits in the load to ensure that the channel is free of errors.

Transient Error Detection. The transmission of the data between sender and receiver is free of any noise. However, the receiver does not only recover values from the sender, but also other loads from the current and sibling logical core. Hence, to get rid of this noise, we encode the data as shown in Figure 11.5. This allows the receiver to filter out data not originating from the sender.

Although we cannot transfer the entire packet into the architectural domain, we can compute on the packet in the transient domain. Thus, we run the error detection in the transient domain and only transmit valid packets to the architectural domain.

The challenge to run the error detection in the transient domain is that the number of instructions is limited, and not all instructions can be used. For reliable results, we cannot use instructions which speculate on either control or data flow. Hence, the error-detection code has to be as short as possible and branch free.

Our packet structure allows for extremely efficient error detection. We encode the data in the first byte and the two's complement of the data in the second byte as a checksum. To detect errors, we `XOR` the value of the first byte (*i.e.*, the data) onto the second byte (*i.e.*, the two's complement of the data). If both values are received correctly, the `XOR` ensures that the bits 8 to 15 of the packet are zero. Thus, for a correct packet, the least-significant 16 bits of the packet represent a value between 0 and 255, and for a wrong packet, these bits represent a value which is larger than 255. We use these resulting 16-bit value as an index into our oracle array, *i.e.*, an array consisting of 256 pages. Therefore, any value which is not a correct byte is out of bounds and has thus no effect on the cache state of the array. A correct byte is also a valid index into the oracle array and ensures that the first cache line of the corresponding page is cached. Finally, by applying a cache-based side-channel attack, such as Flush+Reload, we can recover the byte from the cache state of the oracle array [44, 47].

The error detection in the transient domain has the advantage that we do not require computation time in the architectural domain. Instead of waiting for the exception to become architecturally visible by doing nothing, we already use this time to perform the required computation. An additional advantage is that while we are still in the transient domain, we can work on noise-free data. Thus, we do not require complex error correction [53].

Additionally, we also encode a sequence number into the packet. The sequence number allows ordering the received packets and is also recovered using the same method as the data value.

Results. We evaluate the covert channel in a lab environment and a public cloud. In the lab environment, we used 2 VMs running inside QEMU KVM on an i7-8650U. For the cloud scenario[1], we used 2 co-located virtual machines running CentOS 7.6.1810 with a Linux kernel version of 3.10.0-957 on a Xeon E5-2670 CPU.

Both on the cloud, as well as on our lab machine, we achieved an error-free transmission. On our lab machine, we observed transmission rates of up to 26.8 kb/s with Variant 1. As TSX was not available in the cloud scenario, we achieved a transmission rate of 1.99 kb/s ($\sigma_{\bar{x}} = 2.5\,\%$, $n = 1000$) with Variant 1 and signal handling.

Table 11.4 shows a comparison to the transmission rates of state-of-the-art cross-VM covert channels.

[1]The cloud provider asked us not to disclose its name at this point.

Covert channel	Speed	Error rate
Pessl et al. [59]	411 kb/s	4.11 %
Liu et al. [48]	600 kb/s	1 %
Maurice et al. [53]	362 kb/s	0 %
ZombieLoad (this)	26.8 kb/s	0 %
Maurice et al. [52]	751.2 b/s	5.7 %
Wu et al. [79]	746.8 b/s	0.09 %
Xu et al. [81]	215 b/s	5.12 %
Schwarz et al. [63]	11 b/s	0 %
Ristenpart et al. [60]	0.2 b/s	-

Table 11.4: Transmission rates of state-of-the-art cross-VM covert channels ordered by their transmission speed.

6.4 Browsing-Behavior Monitoring

ZombieLoad is also well suited for detecting specific byte sequences within loaded data. We demonstrate an attack for which we leverage ZombieLoad to fingerprint a web browser session. For this attack, we assume an unprivileged attacker running on one logical core and a web browser running on the sibling logical core. In this scenario, it is irrelevant whether the attacker and victim run on a native machine or whether they are in (different) virtual machines.

We present two different attacks, a keyword detection attack which can fingerprint website content, and an URL recovery attack to monitor a victim's browsing behavior.

Keyword Detection. The keyword detection allows an attacker to gain information on the type of content the victim is consuming. For this attack, we constantly sample data using ZombieLoad and match leaked values against a list of keywords defined by the attacker.

We leverage the fact that we have access to a full cache line and can do arbitrary computations in the transient domain (cf. Section 6.3). As a result, we only externalize a small integer indicating which keyword has matched via a cache side channel.

One limitation is the length of the keyword list, as in the transient domain, only a limited number of memory accesses are possible before the transient execution aborts. The most reliable solution is to store the keyword list entirely in CPU registers. Hence, the length of the keyword list is limited by the available registers. Moreover, the length is also

limited by the amount of code that is transiently executed to compare leaked values to the keyword list.

URL Recovery. In the second attack, we recover accessed websites from browser sessions without prior selection of interesting keywords. We take a more indirect approach that relies on modern websites performing many individual HTTP requests to the same domain, e.g., to load additional resources such as scripts and images.

In the transient domain, we again sample data using ZombieLoad. While still in the transient domain, we detect the substring "`www.`" inside the leaked data. When we discover a match, we leak the character following "`www.`" to the architectural domain using a cache side channel. This already results in a set of first characters of domain names which we refer to as the candidate set.

In the next iteration, for every domain in the candidate set, we take the last four leaked characters (e.g., "`ww.X`"). We use this string in the transient domain to filter leaked values, similar to the "`www.`" substring in the first iteration. If a match is found, we leak the next character, until the string ends with a top-level domain.

Note that this attack is not limited to URLs. Potentially all data which follows a predictable pattern, such as session cookies or credit-card numbers, can be leaked with this variant.

Results. We evaluated both attacks running an unmodified Firefox browser version 66.0.2 on the same physical core as the attacker. For both attacks, we used ZombieLoad Variant 2. Our proof-of-concept implementation of the keyword-checking attack can check four up to 8-byte long keywords. Due to excessive precomputations of browsers when entering an URL, a keyword is sometimes already matched during the autocompletion of the URL. For highly dynamic websites, such as *nytimes.com*, keywords reliably match on the first access of the website. Accessing mostly static websites, such as *gnupg.org*, have a 60 % probability of matching a keyword in this setup. We observed false positives after the first website access when continuing to use the browser. We hypothesize that memory locations containing the keywords get re-used and may thus leak at a later time again.

For the URL recovery attack, we simulated user behavior by accessing popular websites and refreshing them in a defined time interval. We counted the number of refreshes necessary until we recovered the entire

```
1 if (x < array_len) {
2     y = array[x];
3 }
```

Listing 6.1: A simple Spectre-PHT [44] prefetch gadget.

URL, including top-level domain. For each website, the experiment was repeated 100 times.

Table 11.5: Number of accesses required to recover a website name. The experiment was repeated 100 times per website.

Website	Minimal	Average	Maximum
nytimes.com	1	1	3
facebook.com	1	2	4
kernel.org	2	6	13
gnupg.org	2	10	34

The actual number of refreshes needed depends on the nature of the website that is visited. If it is a highly dynamic page, such as *facebook.com* or *nytimes.com*, a small number of reloads is sufficient to recover the entire name. For static pages, such as *gnupg.org* or *kernel.org*, the necessary reloads increase by approximately a factor of 10. See Table 11.5 for a detailed overview of required reloads.

6.5 Targeted Data Leakage

Inherently, ZombieLoad is a 1-dimensional side channel, *i.e.*, the leakage is only controlled by the time. Hence, leakage cannot be steered using specific addresses as is the case, e.g., for Meltdown [47]. While this data sampling is still sufficient for several real-world attacks, it is still a limiting factor for general attacks.

In this section, we show how ZombieLoad can be combined with *prefetch gadgets* [8] for targeted data leakage.

Speculative Data Leakage. Listing 6.1 illustrates such a gadget, which is a common pattern for accessing an array element [8]. First, the code checks whether the index lies within the bounds of the array. Only if this is the case, the element is accessed, *i.e.*, loaded. While it is evident that for a user-controlled index the corresponding array element can be loaded, such a gadget is more powerful.

On a CPU vulnerable to Spectre, an attacker can mistrain the branch predictor, e.g., by providing several valid values for the array index. Then, by providing an out-of-bounds index, the branch is misspeculated and speculatively accesses an out-of-bounds value. Alternatively, the attacker can alternate between valid and out-of-bounds indices randomly to achieve a high percentage of mispredictions without any prior branch predictor mistraining.

ZombieLoad cannot only leak architecturally accessed data but also speculatively accessed data. Hence, ZombieLoad can even see the value of loads which are never architecturally visible. Such loads include, among others, speculative memory loads and prefetches. Thus, any Spectre gadget which is not hardened, e.g., using a fence [4, 5, 8, 32] or a mask [8, 9], can be used to leak data.

Moreover, ZombieLoad does not require classic Spectre gadgets containing an indirect array access [44]. A simple out-of-bounds access (cf. Listing 6.1) is sufficient. While such gadgets have been demonstrated for breaking KASLR [66], they were considered as relatively harmless as they do not leak data [8]. Hence, most approaches for finding gadgets do not consider such gadgets [25, 76]. In the Linux kernel, however, such gadgets are patched if they are discovered, mainly as they can be used together with Foreshadow to leak arbitrary kernel memory [11, 70]. So far, 172 such gadgets were fixed in kernel 5.0 [8]. With ZombieLoad, we show that such gadgets are indeed powerful and require patching.

A huge advantage of ZombieLoad over Meltdown is that it circumvents KPTI. The targeted data is legitimately accessed in the kernel space by the prefetch gadget. Thus, in contrast to Meltdown, stronger kernel isolation [22] does not have any effect on the attack.

Potential Incompleteness of Countermeasures. Mainly, there are 2 methods to prevent exploitation of Spectre-PHT: memory fences after branches [4, 5, 8, 32], or constraining the index to a valid range using a bitmask [8, 9]. The variant using fences is implemented in the Microsoft compiler [43, 44], whereas the variant using bitmasks is implemented in GCC [49] and LLVM [9], and also used in the Linux kernel [49].

Both prevent exploitation of Spectre-PHT as the misspeculation cannot load any data, making it also effective against ZombieLoad.

However, even with these countermeasures in place, there is a remaining leakage which can be exploited using ZombieLoad. When architecturally loading an in-bounds value, ZombieLoad can leak up to 64 bytes of the load. Hence, with ZombieLoad, there is a potential leakage of up to

63 bytes which are out of bounds if the last in-bounds value is at the beginning of a cache line or the base of the array is at the end of a cache line.

Data Leakage. To demonstrate the feasibility of prefetch gadgets for targeted data leakage, we use an artificial prefetch gadget as given in Listing 6.1. For our evaluation, we used such a gadget in the system-call path of the Linux kernel 5.0.7. We execute ZombieLoad Variant 1 on one logical core and on the other, we execute system calls switching between out-of-bounds and in-bounds array indices to achieve a high frequency of mispredictions in the gadget.

This approach yields leaked values with a large noise component from unrelated loads. We repeat this setup without trying to generate mispredictions to generate a baseline of noise values. We generate frequency distributions for both runs and subtract the noise frequency from the misprediction run. We then choose the byte value that was seen most frequently. leverage We recover kernel memory at one byte per 10 s with 38 % accuracy. Probing bytes for 20 s improves the accuracy to 46 %.

As with Meltdown [47], common byte values such as `0x00` and `0xFF` occur too often and have to be removed from the leaked data for the recovery to work. Our approach is thus blind to these values.

The speed and accuracy can be improved if there is a priori knowledge of the target data. For example, a 7-bit ASCII string can be leaked with a probing time of 10 s per byte with 72 % accuracy.

7 Countermeasures

As ZombieLoad leaks loaded and stored values across logical cores, a straight-forward mitigation is disabling hyperthreading. Hyperthreading improves performance for certain workloads by 30 % to 40 % [7, 54], and as such disabling it may incur a significant performance impact.

Co-Scheduling. Depending on the workload, a more efficient mitigation is the use of co-scheduling [57]. Co-scheduling can be configured to prevent the execution of code from different protection domains on a hyperthread pair. Current topology-aware co-scheduling algorithms [68] are not concerned with preventing kernel code from running concurrently with user-space code. With such a scheduling strategy, leaks between user processes can be prevented but leaks between kernel and user space cannot. To prevent leakage between kernel and user space, the kernel

must additionally ensure that kernel entries on one logical core force the sibling logical core into the kernel as well [29]. This discussion applies in an analogous way to hypervisors and virtual machines.

Flushing Buffers. As ZombieLoad also works across protection boundaries on a single logical core, disabling hyperthreading or co-scheduling are not fully effective as mitigation. Flushing the L1 cache (using `MSR_IA32_FLUSH_CMD`) and issuing as many dummy loads as there are fill-buffer entries is not sufficient. Intel provided a microcode update [29] which added a side effect to the rarely used `VERW` instruction. Operating systems have to issue a dummy `VERW` instruction on every context switch. If the microcode update is installed, this clears the fill buffers and store buffer. Otherwise, the instruction has no side effect. While the microcode update (microcode 0xB4 on i7-8650U), in combination with a correct usage of the `VERW` instruction does reduce the leakage, it does not fully prevent it. We can still observe leakage from kernel values accessed on the same logical core. However, the leakage rate drops from multiple kilobytes per second to less than 0.1 B/s. Our hypothesis is that we can leak data which is evicted from L1 to L2 after issuing the `VERW` instruction. As the `VERW` instruction does not flush dirty L1-cache lines, these can be easily leaked if the attacker partly evicts the L1. Evicting the L1 cache forces the dirty L1-cache lines to go through the fill buffer to L2. Hence, to fully mitigate ZombieLoad, the operating system has to additionally flush the L1 cache. Our performance measurement showed that only flushing the L1 takes on average 1070 cycles (i7-8650U, $n = 1000$, $\sigma_{\bar{x}} = 1.08$). Therefore, we expect that flushing the L1 on every context switch would have a considerable performance impact.

If the microcode update is not available for a specific CPU, Intel provides code sequences to emulate that behaviour [29]. However, these code sequences do not fully work on all CPUs. For example, on the i7-8650U, we still observe leakage which we assume is caused by the replacement policy of the line-fill buffer.

Selective Feature Deactivation. Weaker countermeasures target individual building blocks (cf. Section 5). Intel SGX can be disabled if not required to disable the use of Variant 4 (cf. Appendix G) permanently. The operating system kernel can make sure always to set the accessed and dirty bits in page tables to impair Variant 3. To prevent Variant 2, Intel may offer a microcode update to disable TSX. Such a microcode update already exists for older microarchitectures with a faulty TSX im-

plementation [34]. On the Amazon EC2 cloud, we observed that all TSX transactions always fail, which indicates that such a microcode update might already be deployed there. Unfortunately, Variant 1 is always possible, if the attacker can identify an alias mapping of any accessible user page in the kernel. This is especially true if the attacker is running in or can create a virtual machine. Hence, we also recommend disabling VT-x on systems that do not need to run virtual machines.

Removing Prefetch Gadgets. To prevent targeted data leakage, prefetch gadgets need to be neutralized, e.g., using *array_index_nospec* in the Linux kernel. This function clamps array indices into valid values and prevents arbitrary virtual memory to be prefetched. Placing these functions is currently a manual task and due to the incomplete documentation of how Intel CPUs prefetch data, these mitigations cannot be complete. Note that Spectre mitigations might be incomplete against ZombieLoad (cf. Section 6.5).

Another way to prevent prefetch gadgets from reaching sensitive data is to unmap data from the address space of the prefetch gadget. Exclusive Page-Frame Ownership [41] (XPFO) partially achieves this for the Linux kernel's mapping of physical memory.

Instruction Filtering. For attacks inside of a single process (e.g., JavaScript sandbox), the sandbox implementation must make sure that the requirements for mounting ZombieLoad are not met. One example is to prevent generation and execution of the `clflush` instructions, which so far is a crucial part of the attack.

Secret Sharing. On the software side, we can also rely on secret sharing techniques used to protect against physical side-channel attacks [69]. We can ensure that a secret is never directly loaded from memory but instead only combined in registers before being used. As a consequence, observing the data of a load does not reveal the secret. For a successful attack, an attacker has to leak all shares of the secret. This mitigation is, of course, incomplete if register values are written to and subsequently loaded from memory as part of context switching.

8 Conclusion

With ZombieLoad, we showed a novel Meltdown-type attack targeting the processor's fill-buffer logic. ZombieLoad enables an attacker to leak

values recently loaded by the current or sibling logical CPU. We show that ZombieLoad allows leaking across user-space processes, CPU protection rings, virtual machines, and SGX enclaves. Furthermore, we show that ZombieLoad even works on MDS- and Meltdown-resistant processors, *i.e.*, even on the newest Cascade Lake microarchitecture. We demonstrated the immense attack potential by monitoring browser behaviour, extracting AES keys, establishing cross-VM covert channels or recovering SGX sealing keys. Finally, we conclude that disabling hyperthreading is necessary to fully mitigate ZombieLoad on current processors.

9 Acknowledgement

We thank Werner Haas (Cyberus Technology), Claudio Canella (Graz University of Technology), Jon Masters (Red Hat), Alex Ionescu (CrowdStrike), and Martin Schwarzl (Graz University of Technology). We would like to thank our anonymous reviewers and especially our shepherd, Yinqian Zhang, for their comments and suggestions that helped improving the paper. The research presented in this paper was partially supported by the Research Fund KU Leuven. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). Daniel Moghimi is supported by the National Science Foundation, under grant CNS-1814406. The project was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

F Fill-buffer Size

In this section, we analyze the size of the fill buffer in terms of fill-buffer entries usable per logical core. Intel describes the fill buffer as a "competitively-shared resource during HT operation" [31]. Hence, with 10 fill-buffer entries (Sandy Bridge and newer microarchitectures) [31], we expect that when hyperthreading is enabled, every logical core can use up to 10 entries.

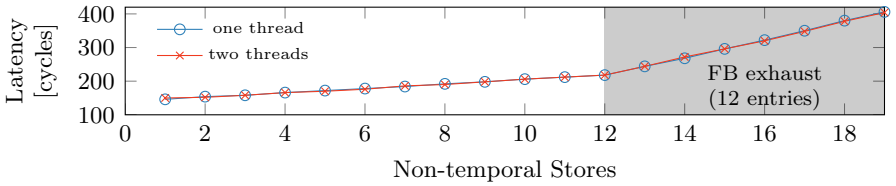


Figure 11.6: One logical core can leverage the entire fill buffer (12 entries). If both logical cores execute stores, the fill buffer is competitively shared, leading to an increased latency for both logical cores.

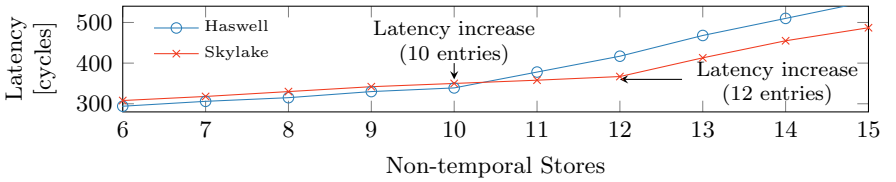


Figure 11.7: One pre-Skylake, we measure 10 fill-buffer entries, matching Intel's documentation. On Skylake and newer, we measure 12 fill-buffer entries.

Our experimental setup measures the time it takes to execute n stores to DRAM, for $n = 1, \ldots, 20$. We expect that the time increases linearly with the number of stores n as long as there are unused fill-buffer entries. To ensure that the stores occupy the fill buffer, we leverage non-temporal stores which bypass the cache and directly go to DRAM. We repeated our experiments 1 000 000 times, and we always measured the best case, *i.e.*, the minimum latency, to get rid of any noise.

Figure 11.6 shows that both logical cores can indeed leverage the entire fill buffer. When running the experiment on one (isolated) logical core, while the other (isolated) logical core does nothing, we get a latency increase when executing more than 12 stores. When we run the experiment on both logical cores in parallel, the latency increase is still after 12 stores.

Interestingly, the documented number of fill buffers does not match our experiments for Skylake and newer microarchitectures. While we measure 10 entries on pre-Skylake CPUs as it is documented, we measure 12 entries on Skylake and newer (cf. Figure 11.7).

From our experiments we conclude that both logical cores can leverage the entire fill buffer Therefore, every logical core can potentially use any entry in the fill buffer.

G Further Variants

As explained above, we hypothesized that load operations which require a microcode assist might first transiently dereference unauthorized fill buffer entries. Apart from the 3 main variants described in Section 5.1, we experimentally verified multiple approaches to provoke a microcode assist on attacker-controlled load operations.

Variant 4: SGX Abort Page Semantics. SGX-enabled processors trigger a microcode assist whenever an address translation resolves into SGX's "processor reserved memory" area and the CPU is outside enclave mode [12]. Next, the microcode assist replaces the address translation result with the address of the abort page which yields `0xff` for reads and silently ignores writes.

For this attack variant, we require a virtual address v mapping to a physical enclave page p. Whenever accessing v outside the enclave, abort page semantics apply, and a microcode assist will be invoked. While this ensures that the load instruction always reads `0xff` at the architectural level, we found however that unauthorized fill buffer entries accessed by the sibling logical core may still be transiently dereferenced before abort page semantics are applied.

In our experimental setup, much like Variant 2, we access v inside a TSX transaction and encode it in a cache-based covert channel. Interestingly, however, we found that for Variant 4 instead of flushing the first cache line of p, it suffices to simply *access* it before the TSX transaction. We conjecture that this is because abort page values never end up in the cache hierarchy.

Variant 5: Uncachable Memory. A variant closely-related to Variant 4 and CVE-2019-11091, yielding the same effect is to use a memory page that is marked as *uncacheable* instead of an enclave page. As the page miss handler issues a microcode assist when page tables are in uncacheable memory, we can leak data similar to the described SGX scenario where memory can also be marked as write-back [12].

References

[1] Jeffery M. Abramson, Haitham Akkary, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, Paul D. Madland, David B. Papworth, and Michael A. Fetterman. *Method and apparatus for dispatching and executing a load operation to memory*. US Patent 5,717,882. Feb. 1998.

[2] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. *Method and apparatus for dispatching and executing a load operation to memory*. US Patent 5,717,882. 1998.

[3] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. "Amplifying side channels through performance degradation." In: *ACSAC*. 2016.

[4] AMD. *Software Techniques for Managing Speculation on AMD Processors*. Revison 7.10.18. 2018.

[5] ARM Limited. *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*. 2018.

[6] Darrell D. Boggs and Scott D. Rodgers. *Microprocessor with novel instruction for signaling event occurrence and for providing event handling information in response thereto*. US Patent 5,625,788. Apr. 1997.

[7] James R Bulpin and Ian A Pratt. "Multiprogramming performance of the Pentium 4 with Hyper-Threading." In: *Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD)*. 2004.

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium*. 2019.

[9] Chandler Carruth. *RFC: Speculative Load Hardening (a Spectre variant #1 mitigation)*. Mar. 2018.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. "Leaking Enclave Secrets via Speculative Execution." In: *Euro S&P*. 2019.

[11] Jonathan Corbet. *Finding Spectre vulnerabilities with smatch*. Apr. 2018. URL: https://lwn.net/Articles/752408/.

[12] Victor Costan and Srinivas Devadas. "Intel SGX explained." In: (2016).

[13] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor." In: *ASPLOS'18*. 2018.

[14] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2016.

[15] Cesar Pereida García and Billy Bob Brumley. "Constant-time callees with variable-time callers." In: *USENIX Security Symposium*. 2017.

[16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2016).

[17] Andrew F Glew, Haitham Akkary, Robert P Colwell, Glenn J Hinton, David B Papworth, and Michael A Fetterman. *Method and apparatus for implementing a non-blocking translation lookaside buffer*. US Patent 5,564,111. Oct. 1996.

[18] Andrew F Glew, Haitham Akkary, and Glenn J Hinton. *Translation lookaside buffer that is non-blocking in response to a miss for use within a microprocessor capable of processing speculative instructions*. US Patent 5,613,083. 1997.

[19] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks." In: *USENIX Security Symposium*. 2018.

[20] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX Security Symposium*. 2015.

[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[22] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR." In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, pp. 161–176.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *CCS*. 2016.

[24] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. "Use-after-freemail: Generalizing the use-after-free problem and applying it to email services." In: *AsiaCCS*. 2018.

[25] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled Detection of Speculative Information Flows." In: *arXiv:1812.08639* (2018).

[26] Shay Gueron. *Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01*. 2012.

[27] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2017.

[28] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018. URL: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[29] Intel. *Deep Dive: Intel Analysis of Microarchitectural Data Sampling*. May 2019. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling`.

[30] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[31] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2016.

[32] Intel. *Intel Analysis of Speculative Execution Side Channels*. July 2018. URL: `https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf`.

[33] Intel. *Intel Software Guard Extensions SDK for Linux OS Developer Reference*. Rev 1.5. May 2016.

[34] Intel. *Intel Xeon Processor E3-1200 v3 Product Family Specification Update*. Aug. 2018. URL: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf.

[35] Intel. *Intel® C++ Compiler 19.0 Developer Guide and Reference*. Apr. 2019.

[36] Intel. *L1 Terminal Fault SA-00161*. Aug. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[37] Intel. *Side Channel Vulnerability MDS*. May 2019. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html.

[38] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks." In: *USENIX Security Symposium*. 2019.

[39] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX." In: *CCS*. 2016.

[40] Simon P. Johnson, Uday R. Savagaonkar, Vincent R. Scarlata, Francis X. McKeen, and Carlos V. Rozas. *Technique for Supporting Multiple Secure Enclaves*. US Patent 2012/0159184 A1. June 2012.

[41] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. "ret2dir: Rethinking kernel isolation." In: *USENIX Security Symposium*. 2014.

[42] Vladimir Kiriansky and Carl Waldspurger. "Speculative Buffer Overflows: Attacks and Defenses." In: *arXiv:1807.03757* (2018).

[43] Paul Kocher. *Spectre Mitigations in Microsoft's C/C++ Compiler*. 2018.

[44] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *S&P*. 2019.

[45] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer." In: *WOOT*. 2018.

[46] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves." In: *USENIX Security Symposium*. 2017.

[47] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium*. 2018.

[48] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *S&P*. 2015.

[49] LWN. *Spectre V1 defense in GCC*. July 2018. URL: `https://lwn.net/Articles/759423/`.

[50] LWN. *The current state of kernel page-table isolation*. Dec. 2017. URL: `https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/`.

[51] G. Maisuradze and C. Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers." In: *CCS*. 2018.

[52] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel." In: *DIMVA*. 2015.

[53] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. 2017.

[54] Michael Larabel. *Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS*. June 2018. URL: `https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4`.

[55] Marina Minkin et al. "Fallout: Reading Kernel Writes From User Space." In: *arXiv:1905.12701* (2019).

[56] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies The Power of Cache Attacks." In: *CHES*. 2017.

[57] John K Ousterhout et al. "Scheduling Techniques for Concurrent Systems." In: *ICDCS*. 1982.

[58] Colin Percival. "Cache missing for fun and profit." In: *BSDCan*. 2005.

[59] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *USENIX Security Symposium*. 2016.

[60] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds." In: *CCS*. 2009.

[61] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load." In: *S&P*. 2019.

[62] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features." In: *AsiaCCS* (2018).

[63] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC*. 2017.

[64] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks." In: *NDSS*. 2018.

[65] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *DIMVA*. 2017.

[66] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *ESORICS*. 2019.

[67] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs." In: *arXiv:1905.05725* (2019).

[68] J. H. Schönherr, B. Juurlink, and J. Richling. "Topology-aware equipartitioning with coscheduling on multicore systems." In: *6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*. 2013.

[69] Adi Shamir. "How to share a secret." In: *Communications of the ACM* (1979).

[70] Julian Stecklina. *[RFC] x86/speculation: add L1 Terminal Fault / Foreshadow demo*. Jan. 2019. URL: https://lkml.org/lkml/2019/1/21/606.

[71] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels." In: *arXiv:1806.07480* (2018).

[72] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium*. 2018.

[73] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic." In: *CCS*. 2018.

[74] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A practical attack framework for precise enclave execution control." In: *Workshop on System Software for Trusted Execution*. 2017.

[75] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution." In: *USENIX Security Symposium*. 2017.

[76] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. "oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis." In: *arXiv:1807.05843* (2018).

[77] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves." In: *ESORICS*. 2016.

[78] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. 2018.

[79] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud." In: *USENIX Security Symposium*. 2012.

[80] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *S&P*. May 2015.

[81] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. "An exploration of L2 cache covert channels in virtualized environments." In: *CCSW'11*. 2011.

[82] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature