Michael Schwarz

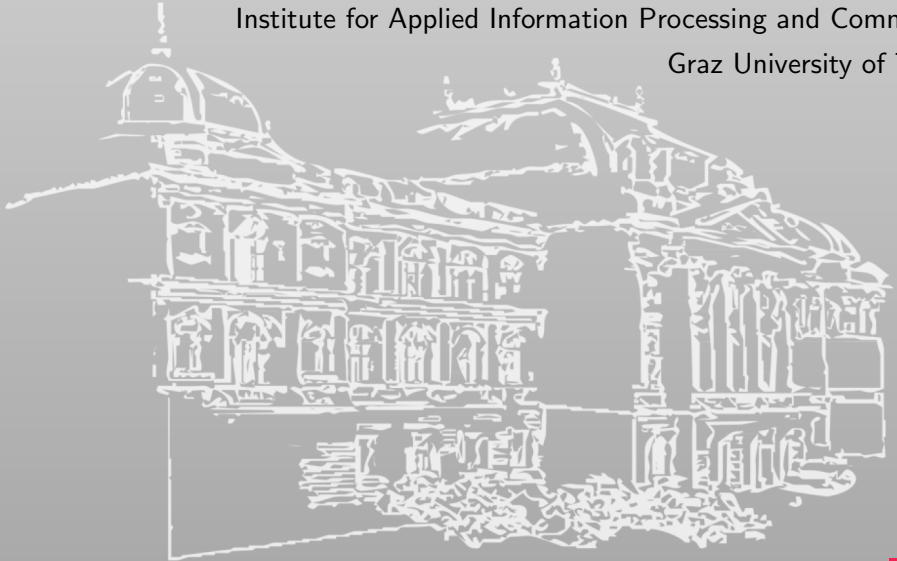
Software-based Side-Channel Attacks and Defenses in Restricted Environments Part 1

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

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

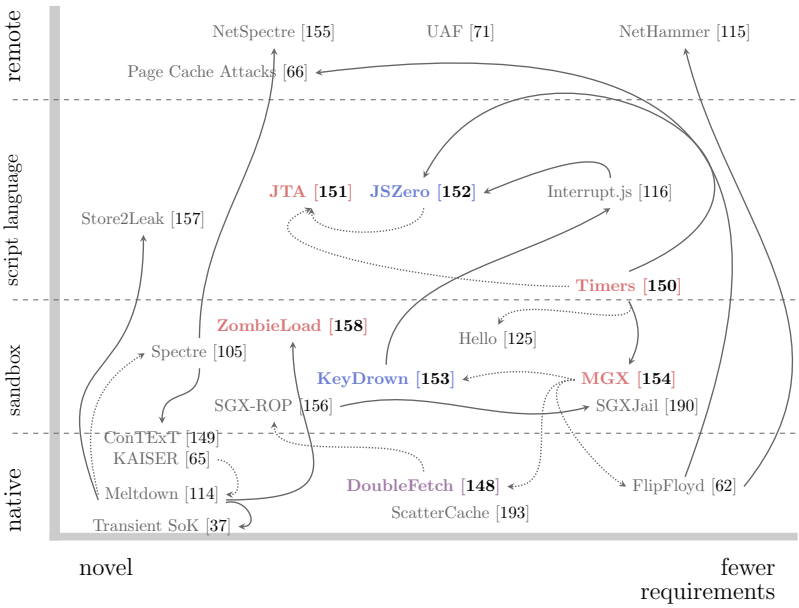


Figure 1.1: A relation between all the papers. Red and blue papers represent papers which are in the thesis, gray papers are co-authored papers which will not appear in the thesis. Blue represents defenses, and red represents attacks. Solid arrows indicate that ideas and techniques directly influenced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

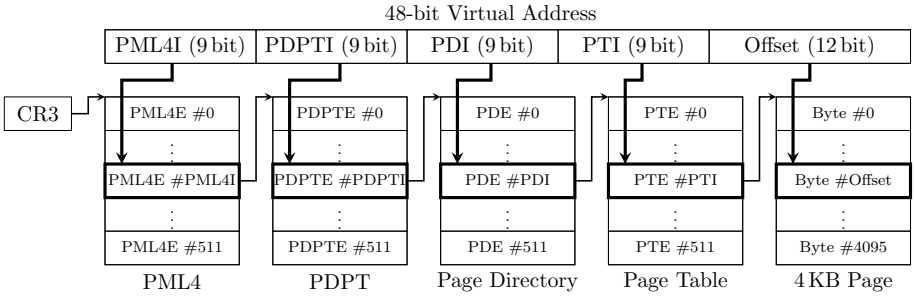


Figure 2.1: On x86_64, every process has a 4-level page-table hierarchy used for the translation from virtual to physical addresses by the MMU. The `CR3` register points to the first level of the page tables. The virtual address is split into parts which are used to index the page tables.

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.

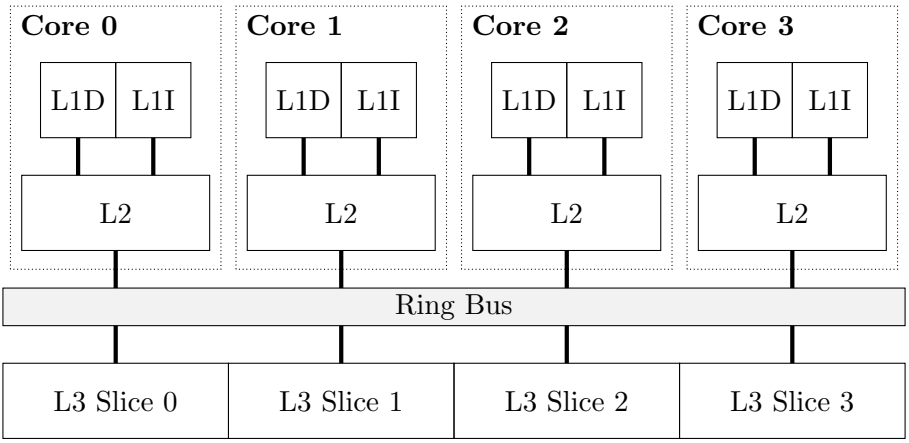


Figure 2.2: The cache hierarchy on modern Intel CPUs. Every CPU core has a private L1 cache which is statically split into an L1 instruction (L1I) and L1 data (L1D) cache, and a private unified L2 cache. The L3 cache (LLC) is split into slices. Every core can access one slice directly and the other slices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

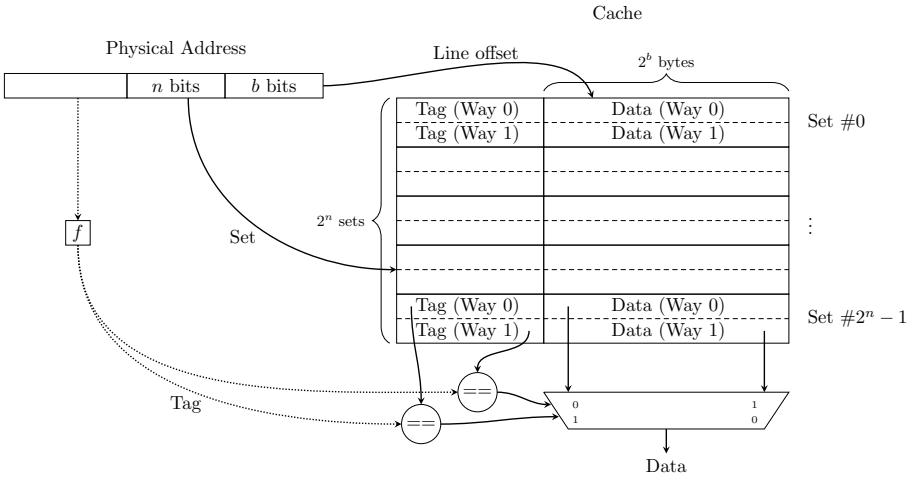


Figure 2.3: A simple example of a two-way set-associative cache. The cache set is determined by n bits of the memory address, the cache way is determined by the cache-replacement policy. The lowest b bits of the address are only used to address the byte inside a cache line. The highest bits of the address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

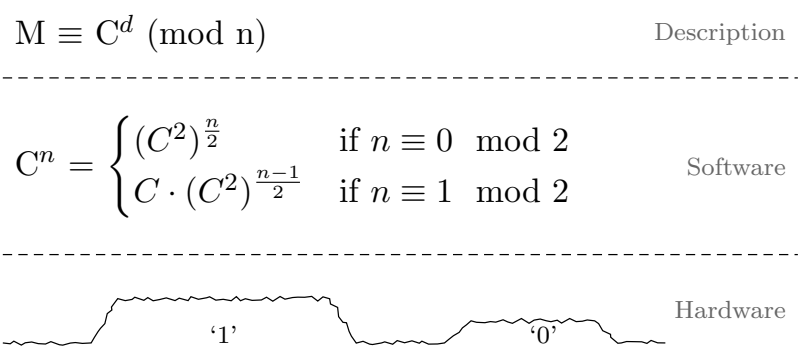


Figure 2.4: Different abstraction layers of a cryptographic algorithm (RSA decryption). The mathematical description is proven to be secure, however the software leaks timing information when e.g., implemented using the non-constant-time square-and-multiply algorithm. On the hardware level, there can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

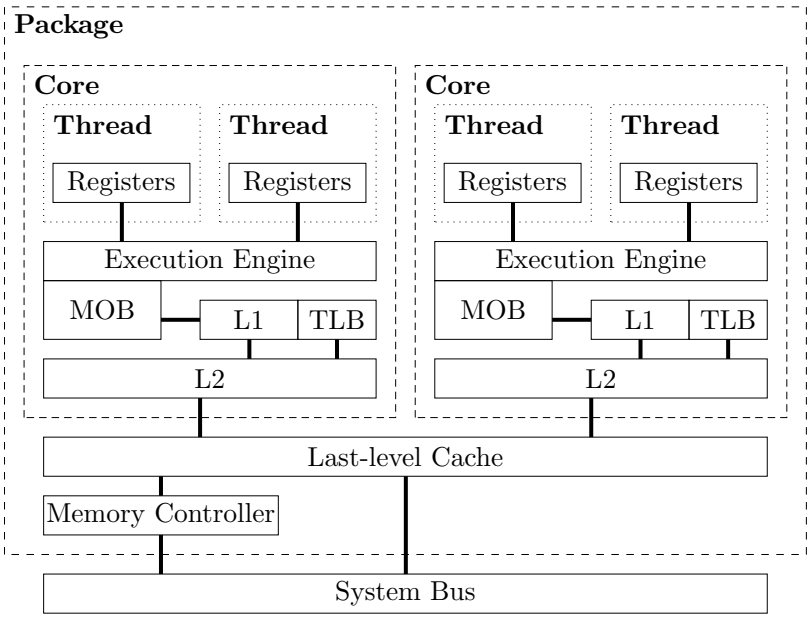


Figure 2.5: The multiple scopes of shared resources: private per thread, shared among threads, shared across cores, and global resources visible to the entire system.

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

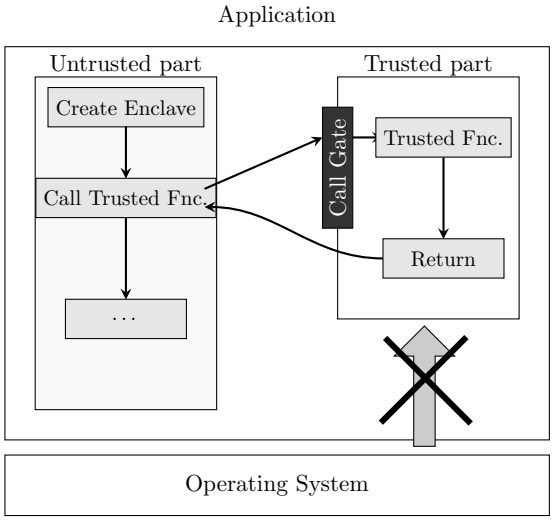


Figure 2.6: In the SGX model, applications are split into a trusted (enclave) and an untrusted (host) part. The hardware prevents any access to the trusted part. The only communication between enclave and host is via predefined ECALLs and OCALLs.

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

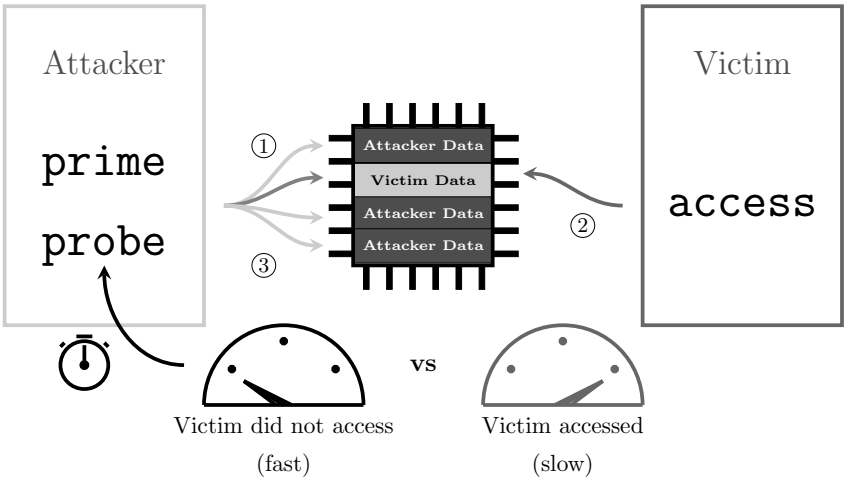


Figure 3.1: In a Prime+Probe attack, the attacker fills a target cache set with data ("prime"). If the victim accesses data in this cache set, the attacker data is evicted. The next time the attacker fills the cache set ("probe"), the time depends on whether the victim evicted the attacker data.

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

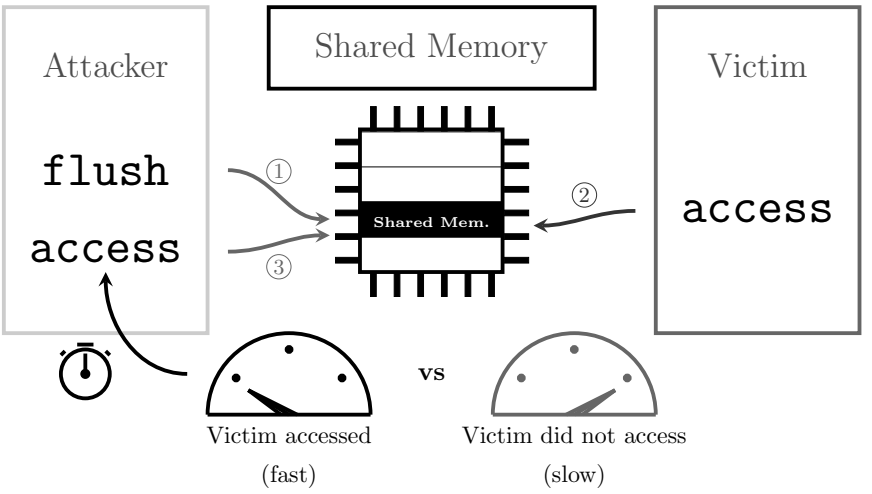


Figure 3.2: In a Flush+Reload attack, the attacker flushes a shared cache line out of the cache ("flush"). If the victim accesses the cache line, it is brought back into the cache. A subsequent access of the attacker ("reload") is then faster.

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

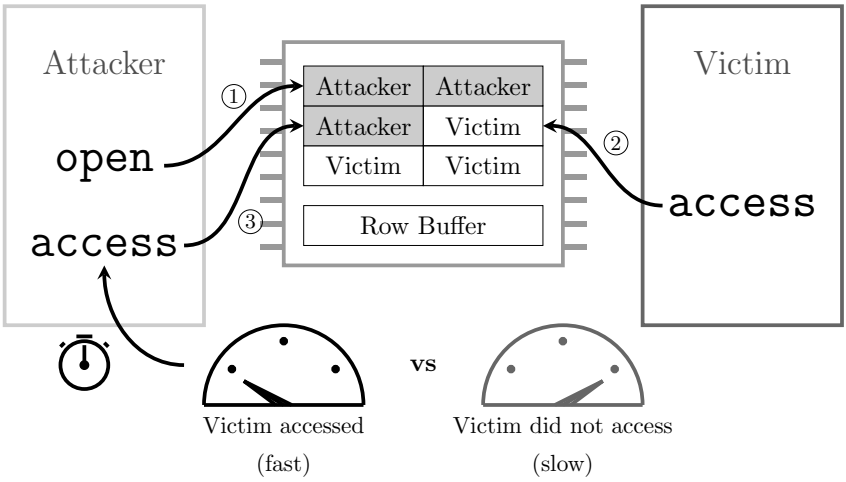


Figure 3.3: In a DRAMA attack, the attacker exploits that DRAM banks are shared among attacker and victim, and that the row buffer acts as a cache.

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

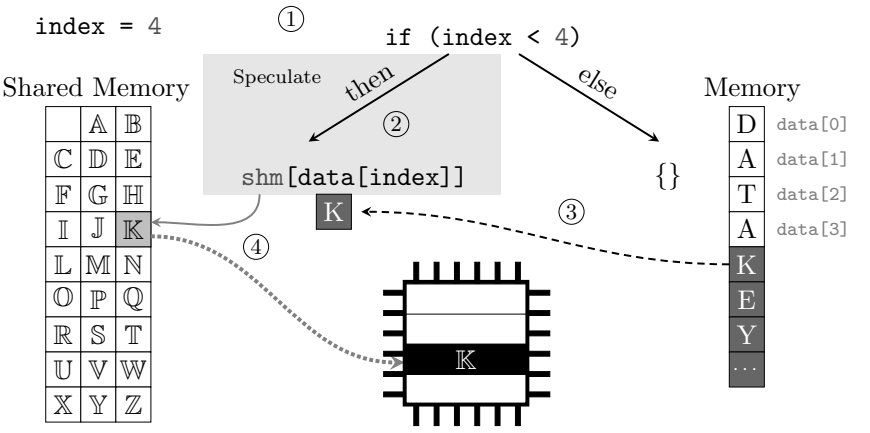


Figure 3.4: In a Spectre attack, an attacker manipulates a branch predictor such that the victim accesses and encodes data which is normally not accessed. Using a side-channel attack, the attacker recovers the encoded data.

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

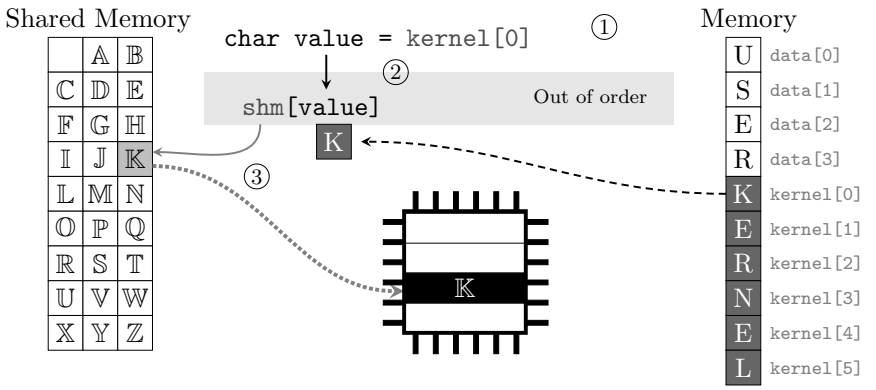


Figure 3.5: In a Meltdown attack, an attacker exploits that the CPU forwards inaccessible data to transient instructions caused by an exception. The data can then be encoded in a microarchitectural state, from which the attacker can recover it using a side-channel attack.

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

Note that this is only the first part of the thesis. For Part II, please download the full thesis.