Daniel Gruss

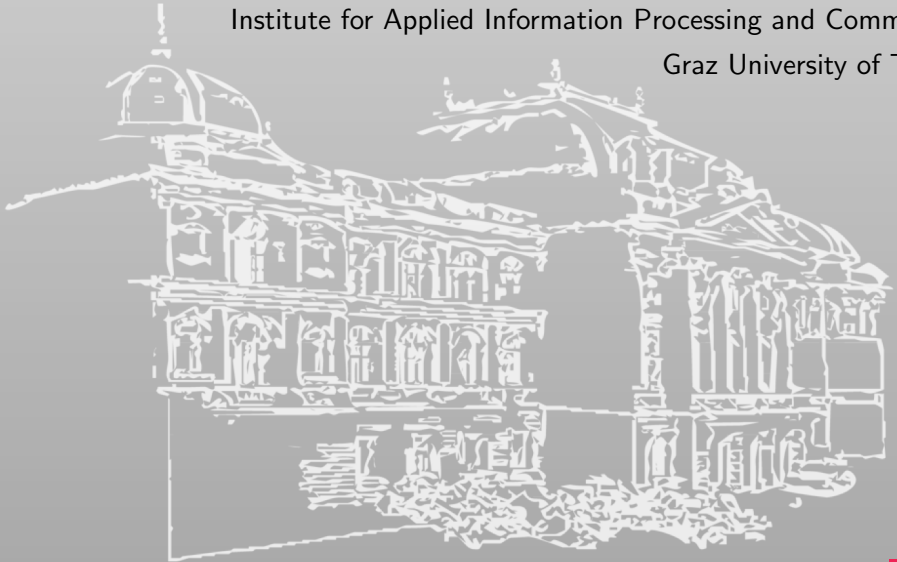
Software-based Microarchitectural Attacks Part I

PhD Thesis

Assessors: Stefan Mangard, Thorsten Holz

June 2017

Institute for Applied Information Processing and Communications
Graz University of Technology



SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Abstract

Modern processors are highly optimized systems where every single cycle of computation time matters. Many optimizations depend on the data that is being processed. Software-based microarchitectural attacks exploit effects of these optimizations. Microarchitectural side-channel attacks leak secrets from cryptographic computations, from general purpose computations, or from the kernel. This leakage even persists across all common isolation boundaries, such as processes, containers, and virtual machines.

Microarchitectural fault attacks exploit the physical imperfections of modern computer systems. Shrinking process technology introduces effects between isolated hardware elements that can be exploited by attackers to take control of the entire system. These attacks are especially interesting in scenarios where the attacker is unprivileged or even sandboxed.

In this thesis, we focus on microarchitectural attacks and defenses on commodity systems. We investigate known and new side channels and show that microarchitectural attacks can be fully automated. Furthermore, we show that these attacks can be mounted in highly restricted environments such as sandboxed JavaScript code in websites. We show that microarchitectural attacks exist on any modern computer system, including mobile devices (e.g., smartphones), personal computers, and commercial cloud systems.

This thesis consists of two parts. In the first part, we provide background on modern processor architectures and discuss state-of-the-art attacks and defenses in the area of microarchitectural side-channel attacks and microarchitectural fault attacks. In the second part, a selection of our papers are provided without modification from their original publications.[1] I have co-authored these papers, which have subsequently been anonymously peer-reviewed, accepted, and presented at renowned international conferences.

[1] Several of the original publications were in a two-column layout. Without any changes in content, we updated the layout for all included papers from the camera-ready versions to fit the layout and formatting of this thesis, including resizing some figures and tables, and changing the citation format.

Acknowledgements

First and foremost, I want to thank my advisor Stefan Mangard. You constantly challenged my mind with ambitious ideas like the one of a cache-based key logger at the beginning of my PhD. You gave me the freedom to research anything I'm interested in and supported me in all endeavors. Thank you for your guidance throughout the last years.

I also want to thank my assessor Thorsten Holz for valuable comments and interesting discussions at past conferences.

I really enjoyed my time at IAIK and in the Secure Systems group – you were the best colleagues one can wish for and I owe thanks to all of you! You all helped me directly or indirectly throughout my PhD.

I owe special thanks to some of my colleagues. Thank you Raphael for helping me in the first months of my PhD and in my first steps on cache attacks. Thank you Clémentine. I've learned so much from you – without you, the amazing last years would not have been possible. It was a great time and a lot of fun working with you, independent of linguistic opinions! Thank you Michael and Moritz. You are two of the most brilliant minds I know and I enjoy every day working with you!

I also want to thank Anders Fogh for fierce competition, great collaborations, great ideas and lots of fun!

Thank you Peter Lipp and David Derler. I enjoyed teaching during my PhD so much and not least because it's great working in a team with you!

Finally, I want to thank my fiancée Maria Eichlseder, my friends, my family, and my fiancée's family for supporting me during my PhD. Thank you Maria for supporting me in everything I do, helping me with my numerous questions on cryptography and maths, and tolerating what I call a healthy work-job balance.

Contents

Part I.

Introduction to Software-based Microarchitectural Attacks

1

Introduction

The idea of learning the secret code for a safe by listening to the clicking sounds of the lock, is likely as old as safes are. The clicking sound is an inadvertent influence on the environment revealing secret information. In 1996, Kocher [Koc96] described side-channel attacks, a technique that allows to derive secret values used in a computation from inadvertent influences the computation has on its environment. This seminal work was the beginning of an entire area of research on side channels. Kocher performed what we now describe as a timing attack, an attack exploiting differences in the execution time of an algorithm. In the following years, side-channel attacks have been demonstrated based on virtually any measurable environmental change caused by various types of computations, such as power consumption [MOP08], electro-magnetic radiation [RR01; KOP09], temperature [HS13], photonic emission [Sch+12; CSW17], acoustic emissions [Bac+10], and many more. These attacks have in common that they require an attacker to have some form of physical access to the target device.

In contrast to side-channel attacks, which do not cause any damage to the target device, there are also fault attacks [BDL97; BS97]. In a fault attack an attacker tries to manipulate computations of a device to either evade security mechanisms of the device or to leak its secrets. For this purpose, the attacker manipulates the environment in a way that influences the target device. Typically such fault-inducing environments are at the border of or beyond the specification range of the target device. Like for side-channel attacks, different environment manipulations have been investigated, such as exposure to voltage glitching [Aum+02], clock glitching [SMC09], extreme temperatures [HS13], or photons [SA02]. Again, to perform a fault attack, some form of physical access to the target device is required.

Modern computer systems are highly complex and highly optimized. Consequently, information leakage, the inadvertent influence of the environment in a secret-dependent way, is not only introduced on an algorithmic level. Optimizations are performed based on the specific data values that are processed, the location of the data, the frequency of accesses to locations, and many other factors. It is clear to see, that any adversary observing the effects of these optimizations through a side channel can make deductions on the specific cause of the optimizations. Through these deductions, the adversary learns information about the secret data values that are processed.

In this thesis, we investigate *software-based microarchitectural attacks*. Software-based microarchitectural side-channel attacks exploit timing and behavior differences that are (partially) caused through microarchitectural optimizations, *i.e.*, differences that are not architecturally documented. Software-based microarchitectural fault attacks induce faults through microarchitectural optimizations, *i.e.*, operate elements of modern computer systems at the border of or beyond their specification range. Generally, software-based microarchitectural attacks do not require physical access, but instead only some form of code execution on the target system.

Cache attacks are the most prominent class of software-based microarchitectural attacks. The possibility of timing differences induced through processor caches was first described by Kocher [Koc96]. Cache timing attacks have first mostly been applied on cryptographic algorithms in software-based attacks [Pag02; TSS03; Ber05; BM06].

Cache attacks in more recent works are usually instances of three generic cache attack techniques. These techniques have been used in targeted attacks on cryptographic algorithms [Ber05; Per05; GBK11] and were later on generalized by Osvik et al. [OST06] and Yarom et al. [YF14]. These generic techniques are independent of the specific cache and hardware on which they are performed. Osvik et al. [OST06] described two generalized cache attack techniques. First, *Evict+Time*, where an attacker measures how the execution time of an algorithm is influenced by evicting a chosen cache set. Second, *Prime+Probe*, where an attacker measures whether a victim computation influences how long it takes to access every way of a chosen cache set.

In both attacks the attacker learns that the chosen cache set was used by the victim. Yarom et al. [YF14] introduced the third generalized attack technique, *Flush+Reload*. In a *Flush+Reload* attack, the attacker flushes

a shared memory location from the cache and subsequently measures how long it takes to reaccess it. If the victim loaded the shared memory location back into the cache in the meantime, the reaccess is faster. In a *Flush+Reload* attack the attacker does not only learn which cache set was used by the victim, but even the specific memory location (at the granularity of cache lines).

Based on these three attack primitives various computations have been attacked, for instance cryptographic algorithms [YF14; Liu+15], web server function calls [Zha+14], user input [GSM15; Gru+16b; Ore+15], kernel addressing information [HWH13; Gru+16a].

Software-based fault attacks are considerably more difficult to build in practice as faults must be induced in hardware. Hence, software has to move the system component that is targeted to the border of or beyond its specification range. Only in 2014 software-based fault attacks have been found to be practical, in the so-called Rowhammer attack [Kim+14; SD15]. In concurrent work, Karimi et al. [Kar+15] demonstrated a second software-based fault attack. They showed that a carefully crafted instruction stream can deteriorate the processor stability and cause severe permanent damage to the processor if executed continuously for weeks. Rowhammer attacks have by now been demonstrated in JavaScript [GMM16; Bos+16], on supposedly safe DDR4 [Pes+16], on co-located virtual machines [Raz+16; Xia+16], and on mobile devices [Vee+16].

To develop and evaluate potential countermeasures against software-based microarchitectural attacks, it is necessary to map and understand the attack surface in detail. In this thesis, we aim to improve the general understanding of the attack surface of software-based microarchitectural attacks and to provide novel insights to software-based microarchitectural attacks and attack vectors. Our research includes the minimization of requirements, the automation of previous attacks, and the identification of previously unknown side channels. Figure 1.1 gives an overview how the papers relate to each other and where they are located in this exploration space.

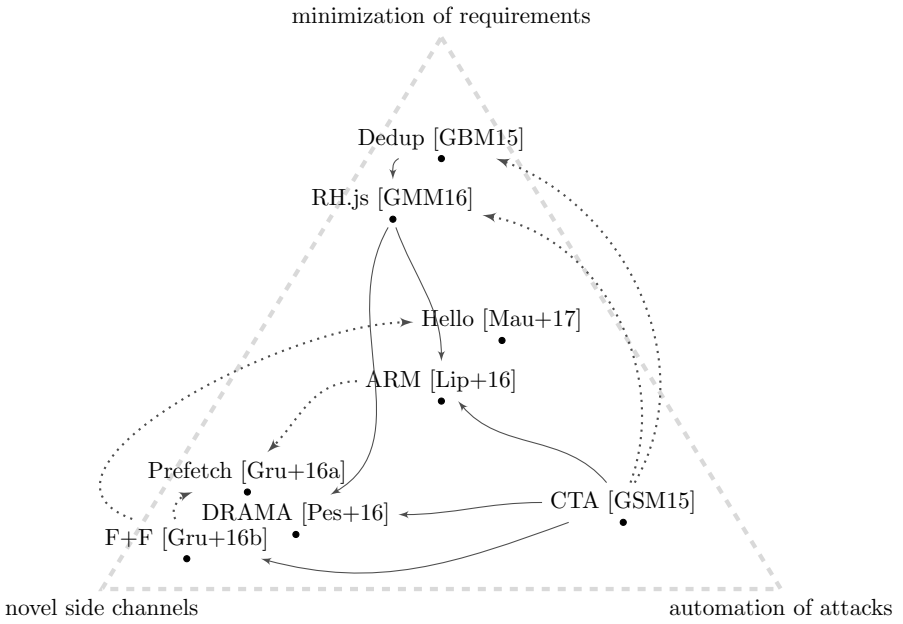


Figure 1.1.: Overview of the relation of the papers to each other. The dotted arrows illustrate where ideas from one paper facilitated the research conducted in the other. The continuous arrows illustrate where concepts from one paper where more directly applied in the other paper.

1.1. Main Contributions

We started the work on software-based microarchitectural attacks by enhancing the *Flush+Reload* cache attack technique [YF14] by an automated method to find and exploit vulnerabilities. This generic technique called Cache Template Attacks has been published at the USENIX Security 2015 conference [GSM15] in collaboration with Raphael Spreitzer and Stefan Mangard. We demonstrated that Cache Template Attacks can be used to identify and exploit leakage in old implementations of cryptographic algorithms, or automatically spy on user input events such as keystrokes or mouse movements. This publication is included as Chapter **??** of this thesis.

While caches buffer the comparably slow DRAM, the DRAM itself buffers the even slower hard disk. Hence, side-channel attacks are also possible on the DRAM level. Suzaki et al. [Suz+11] demonstrated a side-channel attack on page deduplication, as performed by the operating system or

hypervisor, which reveals whether specific data can be found in memory. We demonstrated that such attacks can even be performed from JavaScript integrated into a website. Our results have been published at the ESORICS 2015 conference [GBM15] in collaboration with David Bidner and Stefan Mangard. This publication is included as Chapter **??** of this thesis.

Based on these two works we investigated the possibility of Rowhammer attacks [Kim+14; SD15] from JavaScript integrated into websites. For such attacks to work it was necessary to evict data from caches using regular memory accesses fast enough to replace the comparably fast clflush instruction. Our investigations showed that cache eviction can be performed fast enough for such attacks to successfully be mounted. In our proof-of-concept implementation we were able to trigger bit flips in exploitable memory locations from JavaScript. The results of this research have been published at the DIMVA 2016 conference [GMM16] in collaboration with Clémentine Maurice, and Stefan Mangard. This publication is included as Chapter **??** of this thesis.

Related to the *Flush+Reload* attack we developed a new cache attack called *Flush+Flush*. This technique makes cache attacks faster and stealthier as it does not perform memory accesses itself. We evaluated the performance and stealthiness of the attack in comparison to other cache attacks such as *Flush+Reload* and *Prime+Probe* as well as to Rowhammer attacks. The results of our work are also published at the DIMVA 2016 conference [Gru+16b] in collaboration with Clémentine Maurice, Klaus Wagner, and Stefan Mangard. This publication is included as Chapter **??** of this thesis.

Our work on fast cache eviction motivated investigations on fine-grained access-driven cache attacks ARM Cortex-A systems, which usually has no user-space flush instruction. We demonstrated that all cache attack techniques can be performed on ARM Cortex-A systems as well. Based on these attack primitives we demonstrated that Cache Template Attacks provide a powerful means to find and exploit cache leakage on mobile devices. Our results have been published at the USENIX Security 2016 conference [Lip+16] in collaboration with Moritz Lipp, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. This publication is included as Chapter **??** of this thesis.

Besides the unified cache hierarchy there is also a second cache hierarchy in modern processors for page table entries. We found that prefetch instructions have a different execution time based on the state of these

page translation caches. Even worse, the x86 prefetch instructions allow unprivileged processes to prefetch privileged memory into the cache. We exploited these observations in order to defeat kernel address space-layout randomization (KASLR). Our results have been published at the CCS 2016 conference [Gru+16a] in collaboration with Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. This publication is included as Chapter **??** of this thesis.

1.2. Other Contributions

While working on this thesis, several contributions to other works that are not included as a part of this thesis. Nonetheless, we discuss them here to draw the complete picture of all contributions.

While working on the Rowhammer attack we observed timing differences caused by so-called row hits and row conflicts in the DRAM module. To get a better understanding of these timing differences we developed a fully automated method to reverse-engineer the mapping of physical addresses to DRAM cells in software. Using these reverse-engineered mappings reduces the runtime of Rowhammer attacks significantly. Investigating the timing differences in more detail, we found significant side-channel leakage that is comparable to that of cache attacks. These novel DRAM side-channel attacks have been published at the USENIX Security 2016 conference [Pes+16] in collaboration with Peter Pessl, Clémentine Maurice, Michael Schwarz, and Stefan Mangard.

Our previous works on Rowhammer, cache eviction on ARM Cortex-A, and DRAM reverse-engineering systems, sparked the idea of performing Rowhammer attacks on Android devices. In a collaboration with Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, Cristiano Giuffrida we published our results on Rowhammer attacks on mobile devices at the ACM CCS 2016 conference [Vee+16].

Related work on software-based microarchitectural side channels typically discusses the capacity of a side channel based on the raw capacity of a covert channel built on top of it. Due to the nature of side channels, these covert channels are not error-free. Previous work claimed that straightforward application of error correcting codes is sufficient to eliminate all errors. Thus, to provide realistic estimates the error rate is taken into account to

compute a real-world capacity for the channel. Investigating how realistic these estimates are, we built an entirely error-free covert channel. We found that the application of error correcting codes is possible but has to be combined with other error detection techniques in a non-trivial way. Our channel is so reliable that we can even tunnel an SSH connection through it. Our results have been published at the NDSS 2017 conference [Mau+17] in collaboration with Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Carlo Alberto Boano, Kay Römer, and Stefan Mangard.

Many microarchitectural attacks could generally run in JavaScript, but require high-precision timers. We investigated high-precision timing sources in JavaScript and found techniques which allow to mount reliable attacks. We demonstrate this by building a covert channel through DRAM between JavaScript running on a website and an unprivileged application running inside a virtual machine. Our results have been published at the Financial Crypto 2017 conference [Sch+17c] in collaboration with Michael Schwarz, Clémentine Maurice, and Stefan Mangard.

A new feature in modern Intel processors is Intel SGX, an environment for secure execution on untrusted hardware and operating systems. SGX enclaves are highly secure and can generally not be inspected or monitored by the operating system. However, they are also restricted environments, which cannot perform any system calls directly. We investigated whether it is possible to exploit the security features to protect malicious software running inside an SGX enclave. We built cache side-channel attacks extracting cryptographic keys from the host or from co-located SGX enclaves. Our results will be published at the DIMVA 2017 conference [Sch+17b] in collaboration with Michael Schwarz, Samuel Weiser, Clémentine Maurice, and Stefan Mangard.

We investigated possible countermeasures against attacks on address-translation caches (cf. Chapter **??**). Our solution called KAISER is a practical extension for the Linux kernel, which eliminates the leakage entirely while having a low performance overhead on modern processors. Our results will be published at the ESSoS 2017 conference [Gru+17b] in collaboration with Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard.

Finally, we also investigated generic countermeasures against cache side-channel attacks. Modern Intel processors implement hardware-transactional memory on top of the cache hierarchy. Through creative instrumentation

we can use hardware-transactional memory to abort upon conflicting memory operations and cache misses. This effectively eliminates the leakage which is exploited in cache attacks. Our results will be published at the Usenix Security 2017 conference [Gru+17a] in collaboration with Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa.

1.3. Thesis Outline

This thesis consists of two parts. In the first part (Chapter 2 – Chapter 4), a we provide an overview on software-based microarchitectural attacks.

Chapter 2 explains the relevant background. We first detail in Section 2.1 how processors are organized. We explain the basic concept of virtual memory in Section 2.2 and discuss the idea of caching in Section 2.3. In Section 2.4, we provide an overview of how DRAM works.

Chapter 3 discusses the state of the art in microarchitectural attacks. We discuss software-based microarchitectural side-channel attacks in Section 3.1. We describe how cache attacks developed in the past decades to the current state of the art and provide a discussion of attacks on various caches. In Section 3.2 we introduce the concept of software-based microarchitectural fault attacks. Finally, we discuss countermeasures and defense mechanisms in Section 3.3.

Chapter 4 discusses future work and draws conclusions.

In the second part of this thesis (Chapter **??** – Chapter **??**), a list of all publications is provided, together with transcripts for a selection of papers constituting this thesis.

Chapter **??** consists of our USENIX Security 2015 conference paper [GSM15] on Cache Template Attacks. Chapter **??** consists of our ESORICS 2015 conference paper [GBM15] on page deduplication attacks in JavaScript. Chapter **??** consists of our DIMVA 2016 conference paper [GMM16] on Rowhammer.js. Chapter **??** consists of our DIMVA 2016 conference paper [Gru+16b] on *Flush+Flush*. Chapter **??** consists of our USENIX Security 2016 conference paper [Lip+16] on cache attacks on ARM-based mobile devices. Chapter **??** consists of our CCS 2016 conference paper [Gru+16a] on prefetch side-channel attacks.

2

Background

In this chapter, we provide background which is necessary to understand and discuss microarchitectural attacks. First of all, these attacks target microarchitectural elements, *i.e.*, elements of specific processor families and models. Hence, we discuss how processors are organized in Section 2.1. With virtual memory the processor provides process isolation for modern operating systems. It is deeply rooted in today's processor architectures and influences how various microarchitectural elements work. We discuss virtual memory in Section 2.2. Cache attacks are the most important class of microarchitectural attacks. They typically exploit timing differences introduced by the specific organization of the cache. We discuss caches in detail in Section 2.3. Finally, we discuss how the memory controller interacts with DRAM and exposes timing differences caused by the DRAM to software in Section 2.4.

2.1. Processor organization

Modern processors are highly parallelized machines operating at extremely high speeds. With shrinking process technology sizes, energy demands have decreased allowing to increase processor clock frequencies. Processor clock frequencies stayed roughly at the same level in the past decade. However, besides the processor clock frequency, there are other ways to improve the processing speed. Many optimizations reduce the execution time of specific instructions by a few cycles, sometimes depending on the data or the processor state.

Pipelining. Pipelining is one of the main contributors to performance improvements. Pipelining splits instructions into several stages. The num-

ber of stages and their purpose varies between processors. Modern processors have many pipelining stages, most importantly

- a fetch stage, loading the instruction opcode into the processor,
- a decode stage, decoding the instruction opcode to an internal representation of the instruction,
- and an execute stage, executing the instruction.

Hence, the processor can execute multiple instructions at once, each with a slight interleaving to the previous and the next. Modern processors can have multiple identical stages to perform certain computations in parallel and further improve performance.

Multi-core. Instead of optimizing the processing speed of a single execution core it is also possible to increase the number of execution cores. Especially in server environments multiple processors are installed in a single machine to multiply the performance for parallelizable tasks. Naturally, if a task is not parallelized, there is no performance difference. Hence, depending on the workload, a multi-processor system can provide a significant performance improvement. Workloads on personal computers have changed over the past decades. Today, these systems run several hundred tasks in parallel all the time and thus benefit from multiple processors. For this reason, multi-core processors have been introduced. These processors combine multiple execution cores into a single processor. Each of these execution cores has some private resources, e.g., registers and the execution pipeline, and some shared resources, e.g., the main memory interface.

Instruction Stream Optimization. Another idea to increase the processor performance is to execute branches speculatively. With this optimization the processor makes a guess and execute a branch before it knows whether this branch will be executed. If the guess was correct, the processor already has the information ready when it is required. Otherwise, the processor just discards the result. Another idea is to execute instructions out of order. The execution of instructions with pending data dependencies is delayed and other instructions without pending data dependencies are executed before, to optimize the CPU throughput.

2.2. Virtual memory

As multi-processing became more popular, proper isolation between different processes became more important. As a first step processor manufacturers introduced coarse-grained forms of *virtual memory*, such as Intel's x86 segmentation. Virtual memory regions are mapped to *physical memory* regions in large coherent blocks. At this level of virtual memory, the operating system can specify access privileges, an offset, and a length for each segment. Different processes use different segments and thus work on different physical memory. Consequently, we distinguish between two types of addresses: *virtual addresses*, which are specific to a process, and *physical addresses*, which are valid system-wide but not directly accessible for processes.

Instead of segmentation, modern processors employ a better form of virtual memory, called "paging", which works at a granularity of "pages", which are memory blocks of a fixed size. The entire virtual memory is sliced into virtual pages and the entire physical memory is sliced into physical pages. By numbering the pages we obtain *page numbers*. Paging can be seen as a process-specific map from virtual page numbers to physical page numbers. Modern processors typically support multiple page sizes where the smallest page size is often 4 KB or 1 KB. Larger page sizes are always multiples of smaller page sizes. Pages are aligned in physical memory and in virtual memory to their own size, *i.e.*, 4 KB pages are aligned to 4 KB boundaries in virtual and physical address space.

With 64-bit processors the address-width was increased significantly from 32 bits to 48 bits and an extension to 57 bits is already planned. Address translation is extremely critical to performance and the processor must be able to handle the data structure. Hence, the data structure must be similar to a simple array. An array that could possibly map every 4 KB page of the 48-bit virtual address space to a physical page would already consume 512 GB of physical memory just to store the map, at a map-entry size of 64 bits. Clearly, this is not a practical solution. The idea behind multi-level translation tables is that the virtual address space is usually mapped sparsely to physical memory. With multiple levels, the size of the map can be reduced to a negligible overhead.

To translate a virtual address into a physical address the processor first locates the top-level translation table by reading its address from a processor register. This register value is exchanged upon a context switch

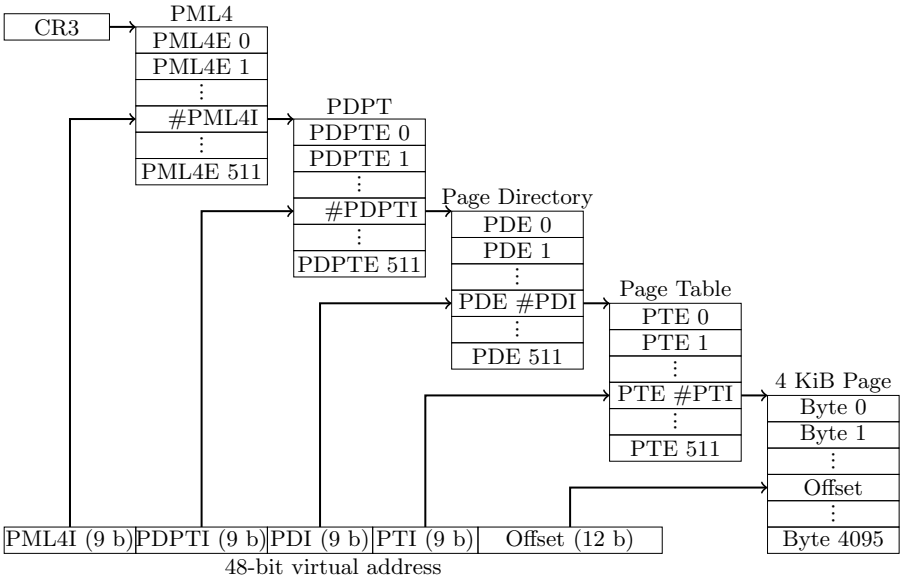


Figure 2.1.: Address translation for 4 KB pages on x86-64 processors. Starting with the PML4 base address from the CR3 register, the processor determines the physical address by gradually using parts of the virtual address.

to enable multi-processing. This is the basis for the process isolation we mentioned before. Each process has its own virtual address mappings and can only access to its own address space.

Modern Intel processors have 4 levels of translation tables as shown in Figure 2.2. The top-most level is the page map level 4 (PML4). It divides the 48-bit virtual address space into 512 memory regions of each 512 GB (PML4 entries). PML4 entries always map to page directory pointer tables (PDPT), *i.e.*, there is no possibility to map a 512 GB page. Each PML4 entry defines properties of the corresponding 512 GB memory region, e.g., whether the memory is mapped to physical memory, whether it is readable, writable, and whether it is accessible to user space. The lower levels are organized in the same way. Each PDPT again has 512 entries, with each entry defining the properties of a 1 GB virtual memory region. This 1 GB virtual page can directly be mapped to a so-called 1 GB page or to a page directory (PD). Modern operating systems use 1 GB virtual pages for instance for the large direct-physical mapping in kernel space which allows working on physical address directly although running in virtual

addressing mode. Each PD again has 512 entries, defining the properties of a 2 MB virtual memory region. Modern operating systems commonly use 2 MB pages to map files or large arrays. Alternatively, the PD entry can map a page table (PT). Each PT again has 512 entries, each controlling a 4 KB page. This is the default page size for most use cases.

For any operation the processor performs, one or more virtual addresses have to be translated into physical addresses. Consequently, the address translation latency must be very small. With translation tables being located in the main memory, this is not the case. Consequently, address translation caches have been introduced to hide the DRAM latency as we will see in the next section.

2.3. Caches

In this section we discuss caches and cache organization in detail. We will discuss the general organization of caches and the basic concepts in Section 2.3.1. Subsequently, we discuss cache replacement policies in Section 2.3.2. We describe the relation between virtual and physical addresses and caches in Section 2.3.3. Finally, we discuss how caches on modern Intel processors work in Section 2.3.4.

2.3.1. Cache Organization

The plain computation speed of processors was the bottleneck for a long time. However, with increasing processor frequencies, the latency of physical memory (DRAM) increasingly became a new bottleneck. While the bandwidth of DRAM has increased over the past decades, the latency is still very high. Processors caches are small and fast buffers intended to hide the latency of the slow DRAM. Modern processors have multiple cache hierarchies for different purposes with each multiple levels of different sized caches. Some caches are private to one execution core while other caches are shared among all cores.

Generally, all memory accesses go through the cache. If a memory access is served from the cache it is called a *cache hit*. Otherwise, it is a *cache miss* causing a fetch from the slow main memory.

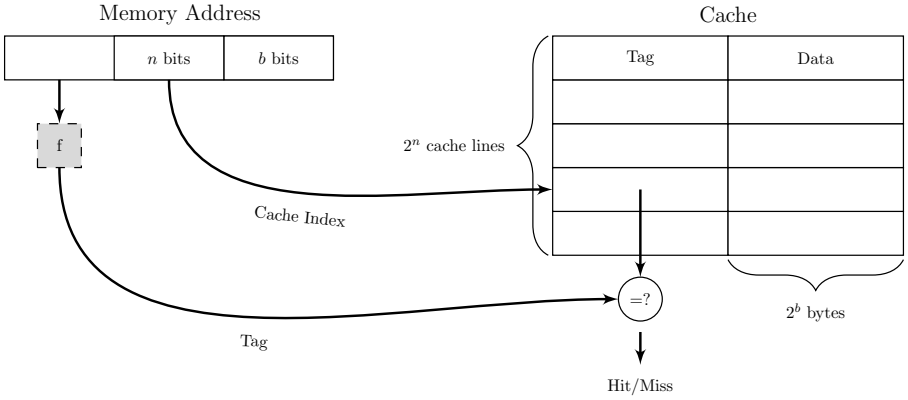


Figure 2.2.: A directly-mapped cache. Based on the middle n bits the cache index is computed to choose a cache line. The tag is used to check whether an address is cached. If it is cached (cache hit), the 2^b bytes data are returned to the processor.

Directly-mapped Caches. The most simple form of a cache is a *directly-mapped cache*, as illustrated in Figure 2.2. A cache consists 2^n *cache lines*, each consisting of a *tag* and 2^b bytes of associated data. The tag is computed from the corresponding memory address that is buffered in this cache line. It is used to determine whether or not a cache line currently buffers a specific memory address. The lowest b bits of the address are used as an offset within the cache line data. Most modern processors have a *cache line size* of 64 bytes, *i.e.*, $b = 6$. The middle n bits of the memory address are used as a *cache index*, telling the processor in which cache line to look for corresponding data. The size of the cache determines how many bits are used, *i.e.*, how many indices there are. Addresses with the same middle n bits are *congruent*, as they map to the same cache line. A significant problem of directly-mapped caches is that they can only store a single cache line out of all congruent cache lines. Hence, if the processor needs to work on two or more congruent cache lines, a directly-mapped cache would experience cache misses most of the time.

Fully-associative Caches. The congruency problem does not exist in *fully-associative caches*, as illustrated in Figure 2.3. Fully-associative caches do not have cache indices and thus they do not have any cache lines. Instead they have multiple *cache ways* to store data. The tag is now used to determine whether the corresponding memory address is cached and

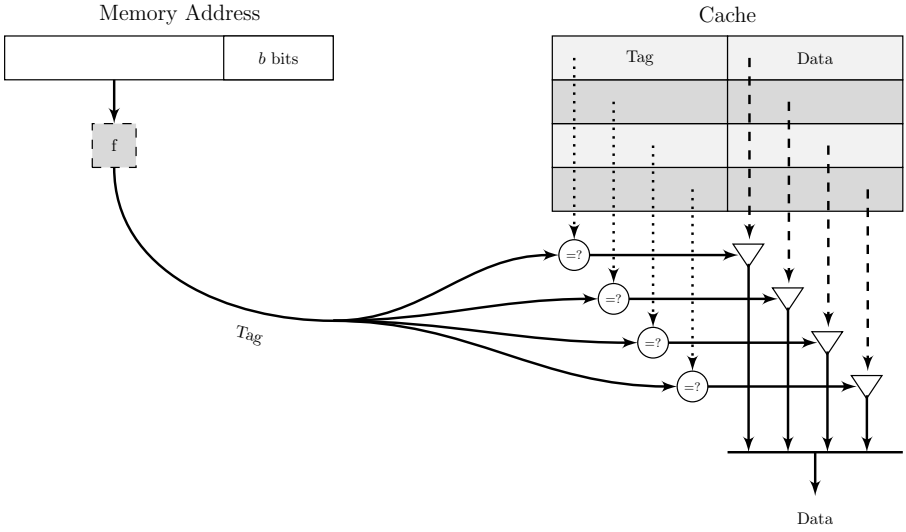


Figure 2.3.: A fully-associative cache. All cache ways are checked in parallel using the tag. The corresponding data value is selected based on the tag comparison.

which cache way contains the associated data. Fully-associative caches are increasingly expensive with the number of ways. Hence, they are typically restricted to a small number of ways, e.g., translation-lookaside buffers with 64 ways can be found in several modern processors.

Set-associative Caches. An elegant compromise are *set-associative caches*, which have *cache sets* instead of cache lines. These caches are widely used in modern processors, often referred to as m-way set-associative caches. Figure 2.4 shows an abstract model of a 2-way set-associative cache. The cache is divided into 2^n cache sets. The *cache set index* is determined from the middle n bits of the memory address. Each cache set has m ways to provide storage locations for m congruent addresses. Note that cache sets can also be seen as tiny fully-associative caches with m ways, for the set of congruent addresses. Hence, the tag is again used to determine which cache way buffers a specific memory address.

Data and instruction caches today are typically implemented as m-way set-associative caches, but processor manufacturers are also transitioning some address translation caches from fully-associative to set-associative.

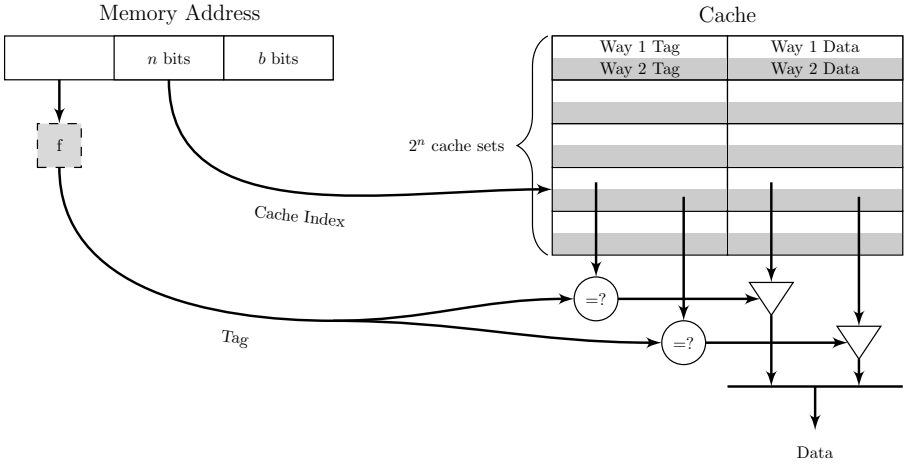


Figure 2.4.: A 2-way set-associative cache. The middle n bits are the cache index, selecting the cache set. The tag is used to check all ways simultaneously. The data in the matching cache way is returned to the execution core.

2.3.2. Cache Replacement Policies

Inevitably, with a limited number of ways and many congruent addresses mapping to the same cache set, the processor constantly has to evict cache ways to replace their content with newly requested data fetched from the main memory. Hence, a replacement policy is implemented to decide which cache way is replaced next when loading new data into a cache set. Processor manufacturers keep the details of their replacement policies mostly secret as they are a relevant contributing factor for the overall processor performance.

A wide-spread replacement policy commonly used by Intel for different caches is least-recently used (LRU). With an LRU replacement policy, every cache way has some form of last-usage timestamp. This is often an approximation to reduce the implementation complexity. Whenever the processor has to load new data into a cache set, LRU replaces the cache way with the oldest last-usage timestamp. LRU fails in cases where the processor works on a set of congruent addresses larger than the number of cache ways. In this case an LRU replacement policy yields a worst-case performance as every single memory access will be a cache miss.

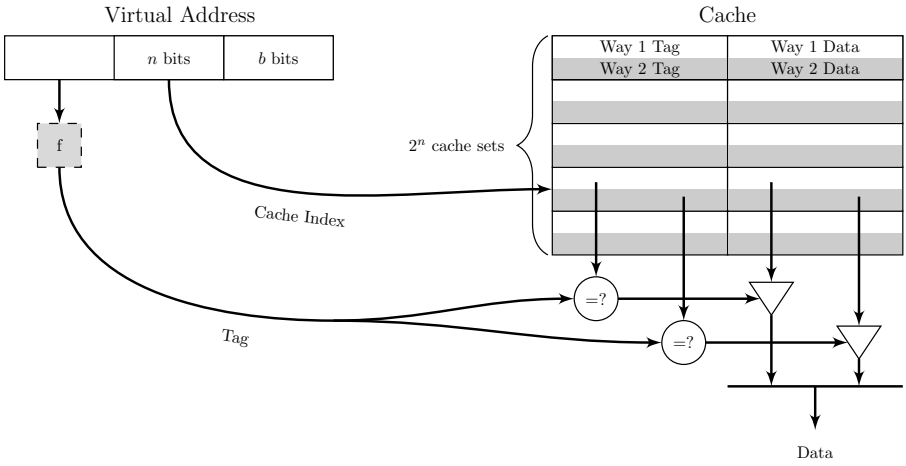


Figure 2.5.: A virtually-indexed virtually-tagged (VIVT) cache. The virtual address is used to compute both index and tag. The processor does not have to translate any addresses.

ARM processors commonly employs a pseudo-random replacement policy. Here the next cache way to be replaced is determined by a pseudo-random number generator. Random replacement policies are the easiest to implement in hardware [Sez93]. They additionally have the advantage of being energy-efficient [PS05]. In practice, random replacement policies have shown to deliver a high performance.

To overcome the limitations of simpler algorithms, since Ivy Bridge, Intel uses a bimodal insertion policy where the CPU can switch between the two strategies to achieve optimal cache usage [Qur+07]. For a group of cache sets the processor can either use a quad-age LRU strategy or a strategy that replaces more recent cache lines first unless they are accessed multiple times. This yields a significantly better performance when using slightly more congruent addresses than would fit in a cache set.

2.3.3. Addressing Modes

Caches can use either virtual addresses or physical addresses to compute the cache index and tag. Three designs have found their way into real-world processors.

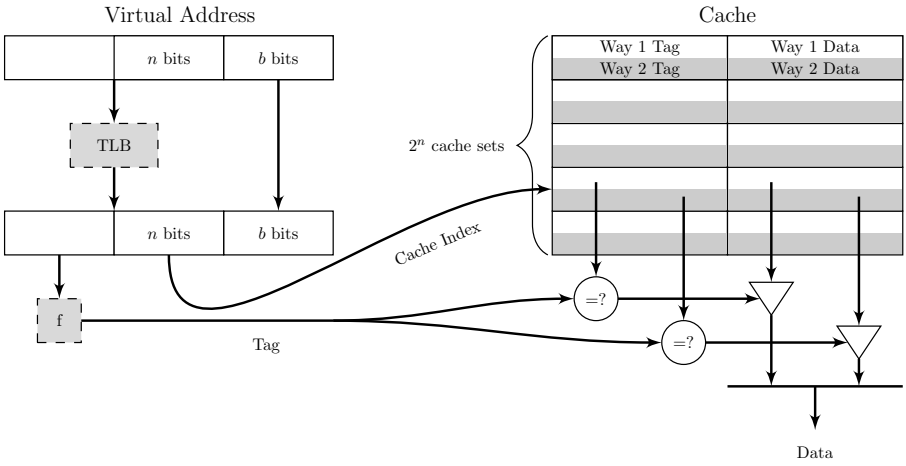


Figure 2.6.: A physically-indexed physically-tagged (PIPT) cache. The physical address is used to compute both index and tag. The processor has to translate the virtual address before the cache set lookup.

Virtually-indexed virtually-tagged (VIVT) caches (cf. Figure 2.5) use the virtual address for both index and tag. Consequently, they don't have to translate addresses at all and have the advantage of a low latency. However, this comes at the price that shared memory might not be shared in the cache, causing an unnecessary increase in cache utilization. Furthermore, upon a context switch it may be necessary to invalidate entries because the virtual tag is not unique. VIVT caches have been used for the smallest and fastest data and instruction caches in some ARM processors. Furthermore, address translation caches are typically VIVT caches.

Physically-indexed physically-tagged (PIPT) caches (cf. Figure 2.6) use the physical address for both index and tag. Consequently, they have a significantly higher latency than VIVT caches. However, shared memory will always be shared in the cache. Thus, there is no unnecessary cache utilization. Furthermore, the tag is physically unique and thus caches do not need to be invalidated upon context switches. Today, PIPT caches are widely used for data and instruction caches with the address translation latency mostly hidden in the address translation caches.

Virtually-indexed physically-tagged (VIPT) caches (cf. Figure 2.7) try to combine the advantages of both approaches by using the virtual address for the index which is required immediately. While the cache index is

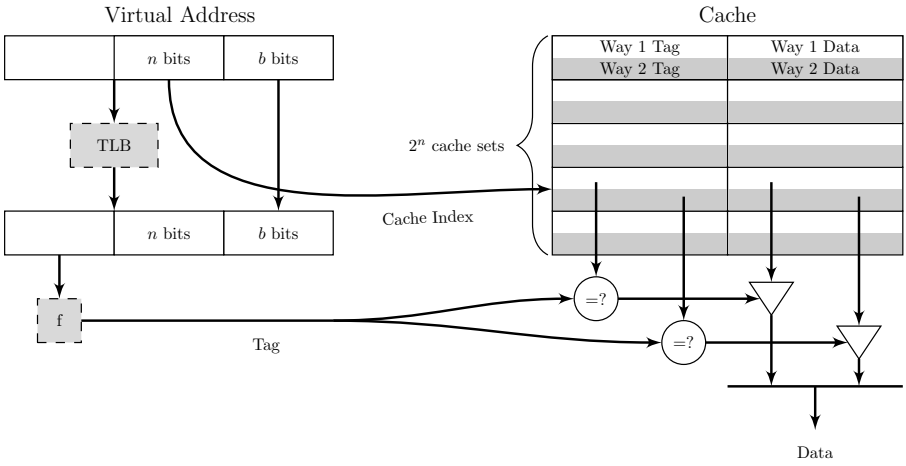


Figure 2.7.: A virtually-indexed physically-tagged (VIPT) cache. The physical address is used to compute both the tag, but the virtual address is used to compute the index. The cache set lookup is done in parallel to the address translation and tag computation.

looked up the tag is computed. This hides the latency of the address translation mostly and allows using a physical tag.

To avoid the disadvantages of VIVT caches, the cache index should not use address bits that are not part of the page offset in the virtual address. With a page size of 4 KB and a cache line size of 64 bytes, there are 6 bits which can be used as a cache index. Most Intel x86 processors from the past decade integrate two 8-way set-associative VIPT L1 caches per processor core, one for instructions and one for data. Consequently, the size of Intel's L1 caches is $2^6 \cdot 64 \cdot 8 = 32\,\text{KB}$ for most processors from the past decade.

2.3.4. Caches in Modern Intel Processors

For instructions and data, Intel x86 processors have a cache hierarchy consisting of L1, L2, and L3 cache. The instruction and data L1 caches are the fastest and smallest caches in this hierarchy. They are private per-core caches, *i.e.*, they are not shared with other cores. The L2 cache is a unified cache, storing both instruction and data cache lines. There is no strict relation between L1 and L2 cache, *i.e.*, cache lines can be presented

Address Bit		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
4 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕		
8 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕		
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕			⊕	⊕			⊕		⊕				⊕	⊕			⊕			

Table 2.1.: Complex addressing functions extrapolated from [Mau+15a].

in none, in one of the two caches, or in both caches. The L3 cache is a unified cache which is shared among all CPU cores. It is also commonly referred to as the *last-level cache*. The L3 cache is inclusive to L1 and L2 caches, *i.e.*, all cache lines in the L1 and L2 caches must also be present in the L3 cache. Both L2 and L3 cache are PIPT caches, enabling to share cache lines based on the physical address.

To enhance the performance, the last-level cache is divided into cache slices since the Intel Nehalem microarchitecture [Mau+15a]. On current Intel processors, each core has its own L3 cache slice. The slices are interconnected by a ring bus allowing all cores to access all L3 cache lines. The mapping from physical addresses to slices is not documented by Intel and referred is to as a *complex addressing* function. This function has recently been reversed-engineered by researchers [Mau+15a; Inc+15; Yar+15]. Table 2.1 lists the slices functions for different processors. Knowledge of the complex addressing functions facilitates cache side-channel attacks.

2.4. DRAM

The main memory of modern computer systems is typically DRAM. DRAM has a significantly higher latency than the various caches inside the processors. The reason for the latency is not only the low clock frequency of DRAM cells but also how DRAM is organized and how it is connected to the processor. As it is difficult to reduce the latency, hardware manufacturers instead focused on increasing the bandwidth of DRAM. The high bandwidth can be utilized to hide the latency, e.g., through speculative prefetching. Modern processors have an on-chip memory controller which communicates through the memory bus with the DRAM.

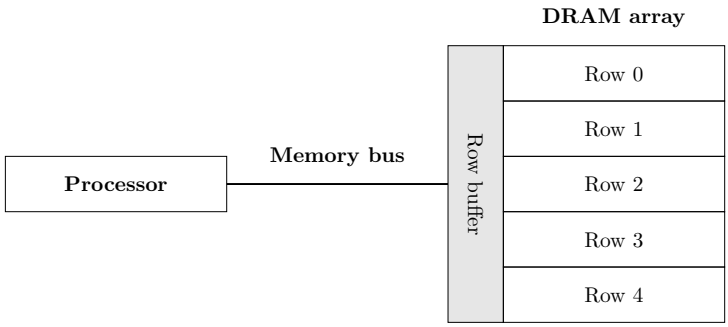


Figure 2.8.: A very simple computer system, with a single DRAM array, which is connected to the processor through a memory bus. The DRAM array consists of rows which are each 8 KB in size.

Figure 2.8 shows a very simple computer system which consists of a single DRAM array, which is connected to the processor through a memory bus. This DRAM array consists of DRAM *rows* and *columns* (typically 1024). Modern systems organize the memory in a way that a DRAM row typically has a *row size* of 8 KB. A row can either be *opened* or *closed*. If it is currently opened, the entire row is preserved in the *row buffer*.

To fetch a data value from DRAM, the processor sends a request to the integrated memory controller. The memory controller then determines a sequence of commands to send over the memory bus to the DRAM to retrieve the data. If the currently opened row contains the data to retrieve, the memory controller just fetches the data from the DRAM row buffer. As this situation is very similar to a cache hit, we call it a *row hit*. If the currently opened row does not contain the data to retrieve, we call it a *row conflict*. The memory controller then first closes the current row, *i.e.*, writes back the entire row to the actual DRAM cells. It subsequently activates the row with the data to retrieve, which is then loaded into the row buffer. Then the memory controller fetches the data value from the row buffer. Similar to cache misses, row conflicts incur an increased access latency.

We can see that the DRAM row buffer in our simple computer system (cf. Figure 2.8) behaves identically to a directly-mapped cache (cf. Figure 2.2). Just as congruent memory accesses constantly lead to cache misses in a directly-mapped cache, alternating row accesses constantly lead to row conflicts in DRAM. Unsurprisingly, similar approaches as in the case of caches have been implemented to increase DRAM performance and

CPU	Ch.	DIMM	BA0	BA1	BA2	BA3	Rank	DIMM	Channel
Sandy Bridge	1	1	13, 17	14, 18	15, 19	-	16	-	-
	2	1	14, 18	15, 19	16, 20	-	17	-	6
Ivy Bridge / Haswell	1	1	13, 17	14, 18	16, 20	-	15, 19	-	-
	1	2	13, 18	14, 19	17, 21	-	16, 20	15	-
	2	2	14, 18	15, 19	17, 21	-	16, 20	-	7, 8, 9, 12, 13, 18, 19
	2	4	14, 19	15, 20	18, 22	-	17, 21	16	7, 8, 9, 12, 13, 18, 19
Skylake	2	1	7, 14	15, 19	17, 21	18, 22	16, 20	-	8, 9, 12, 13, 18, 19
2x Haswell-EP (interleaved)	1	2	6, 22	19, 23	20, 24	21, 25	14	7, 17	-
	2	4	6, 23	20, 24	21, 25	22, 26	15	7, 17	8, 12, 14, 16, 18, 20, 22, 24, 26
2x Haswell-EP (non-interleaved)	1	2	6, 21	18, 22	19, 23	20, 24	13	-	-
	2	4	6, 22	19, 23,	20, 24	21, 25	14	-	7, 12, 14, 16, 18, 20, 22, 24, 26
Exynos 7420	2	-	14	15	16	8, 13	-	-	7, 12

Table 2.2.: DRAM addressing functions from [Pes+16].

eliminate bottlenecks. Modern computer systems organize the DRAM into *channels*, *DIMMs* (Dual Inline Memory Modules), *ranks*, and *banks*.

Our simple computer system (cf. Figure 2.8) had only 1 channel, 1 DIMM, 1 rank, and 1 bank. Modern DDR3 DRAM memory has 8 banks and DDR4 DRAM memory has even 16 banks (per rank). Each of these banks has an independent state and thus can have different rows opened at the same time. This reduces the chance of row conflicts for random memory accesses significantly. Modern DRAM modules typically have 1 to 4 ranks, multiplying the number of banks even further. Similarly, modern systems often allow to install multiple DIMM modules, again multiplying the number of banks with the number of DIMMs. Finally, the memory controller reorders memory accesses to reduce the number of row conflicts, e.g., accesses to the same row and bank are grouped together temporally.

This spatial parallelism reduces the number of row conflicts. Hence, it does not directly influence the bandwidth, but the average latency. To increase temporal parallelism and thus the bandwidth directly, modern computer systems employ multiple channels. Each channel is operated independently and in parallel over the DRAM bus. This effectively multiplies the bandwidth by the number of channels.

Only if two addresses map to the same DIMM, rank and bank they can be physically adjacent in the DRAM chip. In this case the two addresses are in the same bank. If two addresses map to the same bank number, but to a different rank or DIMM, they are not in the same bank and thus generally not physically adjacent in the DRAM chip.

Similar to the slice functions of the last-level cache there are functions mapping from physical addresses to channels, DIMMs, ranks, and banks.

While AMD publicly documents these addressing function [Adv13, p. 345], Intel does not. However, the addressing functions have recently been reverse-engineered for one architecture by Seaborn [Sea15] and with a generic software-based approach by us [Pes+16]. Table 2.2 lists the DRAM addressing functions for several common configurations. Knowledge of the DRAM addressing functions enables DRAM-based side-channel attacks.

3

State of the Art

In this chapter, we discuss the state-of-the-art microarchitectural attacks and defenses. We discuss software-based microarchitectural side-channel attacks in Section 3.1. We discuss software-based microarchitectural fault attacks in Section 3.2. Finally, we discuss defenses against software-based microarchitectural attacks in Section 3.3.

3.1. Software-based Microarchitectural Side-Channel Attacks

In this section, we discuss the most important microarchitectural side-channel attacks. We show that imperfections of the hardware, introduced by optimizations on a microarchitectural level, undermine system security and software security. The hardware leaks part of its internal state including potentially secret information through differences in behavior and timing. Software-based microarchitectural side-channel attacks exploit these differences entirely from software.

With the constantly growing field of microarchitectural attacks and software-based side-channel attacks several works aim to provide a systematization of attacks and defenses in this area of research [AK; Ge+16b; Ge+16a; BWM16; Sze16; DK16; Spr+16; Zha+16].

We discuss state-of-the-art cache attacks in Section 3.1.1, including *Evict+Time*, *Prime+Probe*, and *Flush+Reload* and variants of these attacks. We discuss branch-prediction attacks in Section 3.1.2. We discuss attacks on page-translation caches in Section 3.1.3, including prefetch side-channel attacks. We discuss exception-based attacks in Section 3.1.4, including page deduplication attacks. We discuss DRAM-based attacks in Section 3.1.5.

Finally, we discuss other microarchitectural side-channel attacks in Section 3.1.6.

3.1.1. Cache Attacks

The idea of caches is to hide the latency of the comparably slow physical main memory. Fetching data from the cache has a significantly lower latency. Kocher [Koc96] described the possibility of exploiting these timing differences in so-called cache-timing attacks. The idea is to deduce information on cryptographic secrets from the influence of the cache on the execution time of a cryptographic implementation. Cache timing attacks have been studied in many works [Kel+00; Pag02; TSS03; Ber05; BM06].

The first cache attacks have been cache-based timing attacks. More recent cache attacks are usually categorized into instances of three generalized cache attack techniques, which have first been performed on cryptographic algorithms [Ber05; Per05; GBK11] and were later on generalized by Osvik et al. [OST06] (*Evict+Time* and *Prime+Probe*) and Yarom et al. [YF14] (*Flush+Reload*). We discuss the different attack techniques in detail in the following.

Bernstein's attack. Bernstein [Ber05] described a remote cache-timing attack on an AES T-table implementation. T-tables are preprocessed S-box computations that directly follow the AES design [Nat01; DR13]. The entire AES algorithm can then be implemented as a fast sequence of T-table lookups. The T-table entries are accessed based on an algorithmically defined scheme. For instance in the first encryption round, the algorithm accesses the T-table entries $T_j[p_i \oplus k_i]$, where p_i is the i-th plaintext byte and k_i is the i-th key byte, with $i \equiv j \mod 4$ and $0 \leq i < 16$. Bernstein observed that these accesses may be cached and depending on whether they are cached a timing difference can be observed. By observing the timing difference, the attacker can deduce which T-table entry was accessed and thus, learns the upper 4 bits of the result of $p_i \oplus k_i$. In a chosen-plaintext attack the attacker can eliminate p_i from the equation and thus obtain the upper 4 bits for every key byte. Combining the information from not only the first round but from multiple rounds yields the full AES key. Bernstein's attack has been reproduced and evaluated in many works [NSW06; BM06; SP13b; Wei+14; SG14].

Evict+Time. Osvik et al. [OST06] described *Evict+Time* as a generic cache-timing attack technique. The attacker triggers several victim computations and measures the victim's execution time. To measure the influence of a specific cache set the attacker evicts the cache set before the computation for half of the victim runs. If there is a timing difference when evicting the cache set, the attacker can conclude that the cache set was used by the victim computation.

Similar as in Bernstein's attack, the timing differences observed this way yield which T-table entries were accessed. *Evict+Time* yields information on a cache-set granularity but suffers from various sources of noise on the execution time. Hence, several hundred thousand repetitions may be necessary to extract an entire AES key. *Evict+Time* requires the attacker to be able to measure the exact starting and end time of a victim computation. This might not be possible in asynchronous attacks where the attacker cannot trigger the computation or more generally in many cloud scenarios. An advantage of *Evict+Time* is that it does not require any shared memory. Complex addressing functions and replacement policies of modern processors make eviction more difficult and thus make *Evict+Time* attacks harder.

Evict+Time attacks have been investigated by Osvik et al. [OST06] in an attack on OpenSSL AES. Spreitzer et al. [SP13a; Lip+16] demonstrated that *Evict+Time* on OpenSSL AES is also applicable to mobile ARM-based devices. Hund et al. [HWH13] demonstrated that *Evict+Time* can be used to defeat kernel address space-layout randomization (KASLR).

Prime+Probe. The second technique presented by Osvik et al. [OST06] is much more powerful. In a *Prime+Probe* attack, the attacker continuously fills (primes) a cache set and measures how long it takes to refill the cache set, as illustrated in Figure 3.1. Osvik et al. described that the time it takes to refill the cache set is proportional to the number of cache ways that have been replaced by other processes. While this proportionality is not strictly present anymore for more recent microarchitectures, the general idea is still valid. A higher timing means that at least one cache way has been replaced. A lower timing means that likely no cache way has been replaced.

Prime+Probe has the same granularity as *Evict+Time*, *i.e.*, a cache set. The accuracy is higher than with *Evict+Time* as it measures the cache access times directly, whereas *Evict+Time* measures it indirectly through the

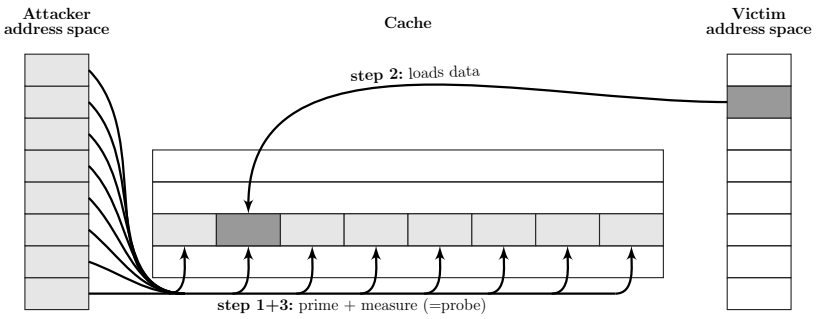


Figure 3.1.: A *Prime+Probe* attack illustrated in 3 steps. The attacker continuously primes a cache set using its own memory locations and measures the execution time of this step (step 1 and step 3). In step 2, the victim possibly accesses (non-shared) memory locations that map to the same cache set. If the victim accessed memory locations in the same cache set in step 2, the execution time of the priming (*i.e.*, the probe step) is high as one of the cache ways has been replaced. Otherwise, the execution time of the priming is low.

execution time. *Prime+Probe* does not require any measurement of the execution time and thus allows performing asynchronous attacks. Analogous to *Evict+Time*, complex addressing functions and replacement policies of modern processors also make *Prime+Probe* attacks more difficult.

The first *Prime+Probe* attacks targeted the L1 cache. However, the reverse-engineering of the last-level cache [Mau+15a; Inc+15; Yar+15] opened up the possibility to perform cross-core *Prime+Probe* attacks through the inclusive last-level cache. The first *Prime+Probe* attacks on the L1 cache have first been demonstrated by Percival [Per05] on RSA. Neve et al. [NS06] attacked an AES implementation, Osvik et al. [OST06] demonstrated an an attack on OpenSSL AES, Aciicmez et al. [AK06; Acı07b; AS08a; AS07] demonstrated attacks on OpenSSL AES and RSA exploiting the L1 instruction cache, Bonneau et al. [BM06] attack OpenSSL AES exploiting internal collisions, Brumley and Hakala [BH09] demonstrated and attack on ECDSA, Aciicmez et al. [ABG10] demonstrated an attack on DSA exploiting the L1 instruction cache, Zhang et al. [Zha+12] demonstrated an attack on ElGamal.

The first *Prime+Probe* attacks on the last-level cache have been performed by Ristenpart et al. [Ris+09] and later by Zhang et al. [Zha+11] to detect co-location in the cloud and eavesdrop on co-located virtual

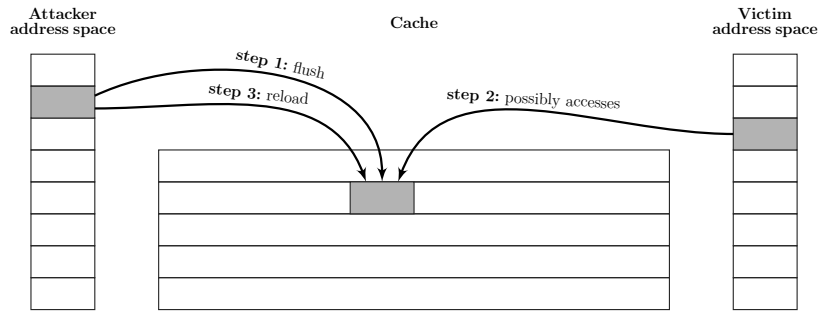


Figure 3.2.: A *Flush+Reload* attack illustrated in 3 steps. In step 1, the attacker flushes a shared memory location in the attacker virtual address space. In step 2, the victim possibly accesses the same shared memory location in the victim virtual address space. In step 3, the attacker reloads the shared memory location and measures the access latency. If the victim accessed the memory location in step 2, the access latency observed in the reload step is low. Otherwise, the access latency in the reload step is high.

machines. However, these attacks were performed on microarchitectures with a simpler organization, *i.e.*, before Intel Nehalem, without cache slices and without complex addressing functions [Mau+15a]. Maurice et al. [Mau+15b] presented a *Prime+Probe* covert channel through the last-level cache on a recent Intel processor. Liu et al. [Liu+15] demonstrated an attack on ElGamal, Irazoqui et al. [IES15] and later Kayaalp et al. [Kay+16] demonstrated attacks on AES. Oren et al. [Ore+15] performed a *Prime+Probe* attack in the browser to eavesdrop on user activities, Inci et al. [Inc+15] attacked ElGamal. Our *Prime+Probe* attack on BouncyCastle AES was the first last-level *Prime+Probe* attack on ARM-based mobile devices [Lip+16]. Finally, we presented a fast and robust *Prime+Probe* covert channel on the Amazon EC2 cloud which employs synchronization techniques and error correction codes to achieve a 0% error rate [Mau+17].

Brasser et al. [Bra+17] showed that *Prime+Probe* attacks can also be performed by a malicious operating system on Intel SGX enclaves. Intel SGX enclaves aim to provide a secure execution environment where software can be executed securely even if the operating system is compromised. We showed that *Prime+Probe* attacks can even be performed from inside an enclave [Sch+17b]. They use the protection features provided by Intel SGX to effectively hide the attack from the operating system. They demonstrate that they can steal most key bits of an RSA private key from a single RSA exponentiation running on the host or a co-located enclave.

Flush+Reload. *Flush+Reload* is often considered the most powerful cache attack. They work on a single cache line granularity and even more, they reveal to the attacker whether a very specific memory location was cached or not. The attack works by frequently flushing a cache line using the `clflush` instruction. The attacker then measures the time it takes to reload the data. If the reload time is low the attacker learns that another process (*i.e.*, the victim) must have reloaded the very same memory location into the cache. If the reload time is high the attacker learns that likely no other process accessed the memory location in the meantime. This general attack flow is illustrated in Figure 3.2. *Flush+Reload* exploits the availability of shared memory and especially shared libraries between the attacker and the victim program. Hence, in scenarios where shared memory is not available, *Flush+Reload* cannot be applied and an attacker has to resort to *Prime+Probe* instead.

There are many variants of *Flush+Reload*, most prominently *Evict+Reload* [GSM15; Lip+16] and *Flush+Flush* [Gru+16b; Lip+16], both of which we introduced. In an *Evict+Reload* attack the `clflush` instruction is replaced by a cache eviction as in a *Prime+Probe* attack. This makes *Evict+Reload* applicable to architectures that do not expose a flush instruction. *Flush+Flush* exploits a timing difference in the `clflush` instruction to determine whether a memory location is cached. Hence, the attacker can omit the reload step from *Flush+Reload* and build a significantly faster and stealthier cache attack that does not perform a single memory access.

The first *Flush+Reload*-like attack has been demonstrated by Gullasch et al. [GBK11], attacking AES. Yarom and Falkner [YF14] demonstrated the first full *Flush+Reload* attack, attacking RSA. *Flush+Reload* attacks on AES have been demonstrated on Intel processors by Irazoqui et al. [IES15; Ira+15a] and Gülmezoğlu et al. [Gül+15]. We also demonstrated *Flush+Reload* attacks on AES on Intel processors [GSM15] and on mobile ARM-based devices [Lip+16]. *Flush+Reload* attacks have also been demonstrated on ECDSA by Benger et al. [Ben+14], van de Pol et al. [PSY15], and Yarom and Benger et al. [YB14]. Allan et al. [All+16] demonstrated that a combination with a denial-of-service attack to degrade the speed of the ECDSA algorithm can yield a more efficient *Flush+Reload* attack on ECDSA. Other attacks have been performed by Zhang et al. [Zha+14] on activities in co-located virtual machines and by Irazoqui et al. [Ira+15b] on TLS. Inci et al. [Inc+16] demonstrated that they can recover encryption keys used in co-located VMs on Amazon EC2 in a larger and automated

scale. Bruinderink et al. [Gro+16] demonstrated an attack on the BLISS signature scheme. Irazoqui et al. [Ira+14; IES16] demonstrated cross-VM and cross-CPU variants of the *Flush+Reload* attack. Another variation of the *Flush+Reload* attack combines it with return-oriented programming to attack cache designs that are less straightforward to attack [ZXZ16].

Beyond *Flush+Reload* attacks on cryptographic implementations, we demonstrated *Flush+Reload* attacks on user input [GSM15; Lip+16].

3.1.2. Branch-Prediction Attacks

Another set of caches are used for branch prediction. The branch-pattern table stores past results on branches and uses them to predict the outcome of future branches. The branch-target buffer caches branch targets from past branches to predict targets of future branches. Both caches are virtually-indexed and thus an adversary can target these caches without knowledge of physical addresses.

Software-based side-channel attacks based on branch prediction hits and misses have first been demonstrated by Aciicmez et al. [ASK07; Acı07a] in attacks on the RSA implementation of OpenSSL. The attacker primes the branch-target buffer by executing a sequence of branches. If the victim experiences a branch misprediction, an entry of the branch-target buffer will be replaced. The attacker subsequently observes a higher execution time due to a misprediction for one of its branches.

Bhattacharya et al. [BM15] show that branch prediction attacks based on hardware performance counters can be used to extract RSA keys from exponentiations running in other processes. Evtyushkin et al. [EPA15] demonstrate a covert channel between two processes manipulating the branch predictor. Evtyushkin et al. [EPA16] also demonstrate that KASLR can be defeated using the branch-target buffer. They infer where code has been executed in the kernel based on the mapping from virtual addresses to the branch-target buffer cache lines. Lee et al. [Lee+16] show that a malicious operating system can reverse-engineer the control flow of SGX enclaves through branch-prediction analysis. Ge et al. [Ge+16a] analyze besides other side channels also the capacity and error rate of a branch-prediction-based covert channel.

3.1.3. Page-translation Cache Attacks

Hund et al. [HWH13] presented the first attack exploiting timing differences caused by page-translation caches. Triggering page faults on inaccessible memory regions reveals whether the memory region would be valid for the kernel, as the valid page-translation entries are cached, independent of the current privilege level. This allows recovering which addresses are valid and even which addresses are used by specific parts of the kernel, *i.e.*, it defeats kernel address-space-layout randomization (KASLR). To run the attack, multiple page faults are triggered. When processing the first page fault, the processor walks through the page translation tables, caching every valid entry. For every subsequent page fault on the same address, the translation table entries are already cached and thus the time until the page fault is triggered is significantly lower.

Jang et al. [JLK16] exploited TSX transaction aborts upon page faults to optimize this attack. Their attack is significantly faster and more reliable and allows defeating KASLR within seconds. Furthermore, they observed a timing difference when trying to execute inaccessible kernel addresses. For executable kernel addresses the latency until the TSX transaction abort is lower than for non-executable kernel memory. Simultaneously to their work, we demonstrated that prefetch instructions leak the same timing difference and can be used to defeat KASLR as well [Gru+16a].

Van Schaik et al. [Sch+17a] showed that timing attacks allow to reverse-engineer the size and structure of the page translation caches. Gras et al. [Gra+17] showed that this information can be used in *Evict+Reload* attacks to defeat ASLR from sandboxed JavaScript.

3.1.4. Exception-based Attacks

Exception-based attacks deduce information from processor exceptions they trigger. Typical exceptions are scheduler interrupts, instruction aborts, page faults, but also behavioral differences, e.g., instructions providing the user with an error code. Through this behavior the CPU leaks direct information (*i.e.*, the behavior itself) and indirect information (*i.e.*, timing differences due to the behavior).

We include exception-based attacks as microarchitectural attacks, as they exploit both architecturally defined and undefined behavior. Especially the implementation of instructions depends on the specific microarchitectural

design. Unspecified cases may influence the processor state and operation on some microarchitectures whereas others ignore it.

Two decades ago, Warner et al. [War+96] presented the first covert channel based on page faults. Page faults can leak sensitive information in three ways: the location of the page fault, the execution time of the page fault, and the mere fact that a page fault occurred *i.e.*, the memory access was delayed or not successful. Page deduplication is a mechanism to share identical memory pages across boundaries of virtual machines to reduce the memory footprint of systems [Mił+09]. However, the fact that a page fault occurred reveals to a user process that a second copy of the same page exists somewhere on the same machine. Suzaki et al. [Suz+11] presented an attack exploiting this to detect programs running in co-located virtual machines. Owens et al. [OW11] demonstrated fingerprinting based on page deduplication attacks. Xiao et al. [Xia+12; Xia+13] demonstrated cross-VM covert channels based on the page deduplication side channel. We demonstrated page deduplication attacks from JavaScript running in a website [GBM15].

Xu et al. [XCP15] demonstrated that page faults can be used as a side-channel by a malicious operating system to spy on an application running in a secure enclave. Shinde et al. [Shi+16] later confirmed their results. The malicious operating system frequently changes the virtual memory mappings of the enclave from valid to invalid. This forces the enclave to experience page faults for almost every memory reference. Although the page fault address in Intel SGX is truncated to be page-aligned the operating system is able derive accurate information of what the application in the secure enclave is processing. Weichbrodt et al. [Wei+16] used a similar approach to interrupt the secure enclave frequently to exploit double-fetch vulnerabilities in enclaves reliably. Zhang et al. [ZW09] proposed an attack that uses system interrupts information to derive user input. Simon et al. [SXA16] demonstrated a similar attack on Android that allows an attacker to recover words and sentences.

3.1.5. DRAM-based Attacks

Modern cloud systems often have multiple processors installed in a multi-socket mainboard. The processor caches are kept coherent with an inter-processor coherency protocol. However, this only has an effect on shared memory cache lines. For co-located virtual machines that do not share

memory, the cache does not provide a communication channel in this setup.

Wu et al. [WXW12; WXW14] found that timing differences caused by memory bus locking can be exploited to build a covert channel between co-located virtual machines in this setup. Their covert channel works through channel contention and has a raw capacity of 13.5 KB/s at an error rate of 0.75% on the Amazon EC2 cloud. Inci et al. [İnc+16] found that memory bus locking can be used to verify co-location in the Microsoft Azure cloud.

More recently, we found found that the DRAM itself can also be exploited [Pes+16]. They described two attack primitives as so-called DRAM addressing (DRAMA) attacks. The row-conflict attack primitive works similar to *Prime+Probe*, the row-hit attack primitive is a side channel that is comparable to a *Flush+Reload* attack. Both side channels work without requiring any shared memory. DRAMA attacks exploit the DRAM row buffer which acts similarly as a directly-mapped per-bank cache for the DRAM rows.

In case of the row-conflict attack, the attacker and victim share a DRAM bank, but no DRAM row. The attacker continuously opens the same row in a bank. Whenever the victim accesses a different row in the same bank, the attacker observes a higher latency. The row-conflict covert channel achieves a performance of 74.5 KB/s with an error rate of 0.4% in a cross-VM cross-processor cloud setup.

In a row-hit attack, attacker and victim share a DRAM row in hardware. The attacker loads another row into the row buffer, comparable to the flush-step in a *Flush+Reload* attack. If the victim accesses the shared row again, it will be loaded into the row buffer. The attacker then reloads a memory location from the shared row and measures the access latency. If the victim accessed the shared row in the meantime, the access latency is low. Otherwise, the access latency is high.

Bhattacharya et al. [BM16] exploited the row-conflict side channel to locate the DRAM bank of a cryptographic secret exponent. They use this side-channel information to perform a Rowhammer attack on the secret exponent, leading to faulty signatures which allow a full key recovery. We demonstrated that the timing differences of the row-conflict side channel are large enough to be measured from JavaScript [Sch+17c]. Based on this observation, we demonstrated a transmission from an unprivileged process inside a virtual machine with no network access to JavaScript

code running inside a website. The covert channel achieves a raw capacity of 11 b/s and an error rate of 0%.

3.1.6. Other Microarchitectural Side-Channel Attacks

Besides these main categories of software-based microarchitectural side-channel attacks, some works have investigated other interferences in instructions and microarchitectural elements. These interferences originate in the throughput limitations of processors. Aciicmez et al. [AS08b] demonstrated that parallel execution of multiplication instructions can leak an RSA key used in a square-and-multiply exponentiation. Evtyushkin et al. [EP16] build a covert channel exploiting timing differences of the `rdseed` instruction depending on the state of the internal random number buffer.

3.2. Software-based Microarchitectural Fault Attacks

A common assumption in system security and software security is the security of the hardware and its error-free operation. However, hardware is not perfect and especially when operated outside the specification, faults can be induced by an attacker [BDL97; BS97; Aum+02; SMC09; HS13; SA02]. A unique feature of microarchitectural fault attacks is that they use effects caused by microarchitectural elements or operations implemented on a microarchitectural level. In the software-based variant these effects and operations are triggered from software.

The first software-based microarchitectural fault attack was the so-called Rowhammer bug. Kim et al. [Kim+14] found that it can be triggered from software and that this could have implications on system security. They execute a sequence of memory accesses and `clflush` operations to frequently open and close DRAM rows, as illustrated in Figure 3.3. If the DRAM rows are in physical proximity, a bit can flip in another DRAM row without accessing it. This other memory location might be inaccessible to the attacker and even belong to another security domain. Their work sparked a series of publications that investigated the security implications of the Rowhammer bug and the requirements to successfully trigger it.

In early 2015, Seaborn and Dullien [SD15] presented the first two practical Rowhammer exploits. The first exploit escapes from the Google NaCl

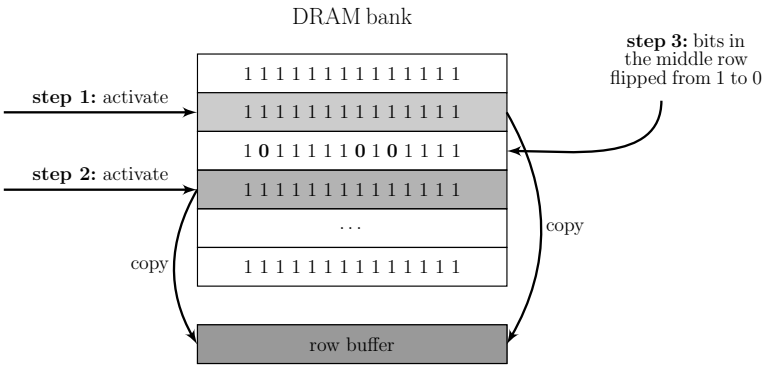


Figure 3.3.: To trigger the Rowhammer bug, memory locations in the same bank but in different rows are alternately accessed (steps 1 and 2) in a high frequency. Depending on the DRAM cell susceptibility to Rowhammer and the activation frequency, bits flip in step 3.

sandbox by causing a bit flip in dynamically generated indirect jumps. They spray the memory with indirect jumps, to maximize the probability to flip a bit in one of them. After a bit flipped in the jump instruction, the attacker gains control over the jump and can thus escape from the NaCl sandbox. The second exploit escalates from user mode to kernel privileges by causing a bit flip in a page table. Again they spray the memory, but this time with page tables, by memory-mapping the same file over and over again. Every mapping requires page table entries, which consist to more than 80% of address bits. The file contents are only kept in memory once. Hence, they fill almost the entire memory with address bits. When a bit flips in an address bit, the user memory mappings change from the file contents to another page. Hence, the user program has very likely gained access to its own page tables and thus has gained kernel privileges.

Seaborn and Dullien found 85% of all DDR3 modules they tested to be susceptible to the Rowhammer bug.

In July 2015, we demonstrated the first Rowhammer attack from sandboxed JavaScript [GMM16]. We triggered bit flips in page tables from JavaScript, by using cache eviction instead of the `clflush` instruction. We introduced a new hammering technique called amplified single-sided hammering, where an attacker hammers two DRAM rows in a 2 MB page to flip a bit in another 2 MB page.

Despite claims by DRAM vendors, we publicly reported the first bit flips on DDR4 memory in late 2015 and published our results in early 2016 [Pes+16]. For our attack we reverse-engineered the DRAM addressing functions for this purpose to perform an optimized attack. Our results on DDR4 were later on confirmed independently by Lanteigne [Lan16]. He observed a susceptibility rate of 67% of the DDR4 memory modules he tested. Aichinger [Aic15a; Aic15b] found that the Rowhammer bug also exists in ECC memory.

Also in 2016, Qiao and Seaborn [QS16] implemented a Rowhammer attack with non-temporal memory accesses, showing that prohibiting access to `clflush` is not sufficient. Bosman et al. [Bos+16] developed a reliable Rowhammer exploit in JavaScript exploiting page deduplication on Windows systems besides the Rowhammer bug. Bhattacharya et al. [BM16] demonstrated the first Rowhammer attack on cryptographic secrets. They trigger bit flips in an RSA key used in an exponentiation. Subsequently, they recover the RSA key through the Chinese remainder theorem as in first fault attacks on RSA [BDL97]. Xiao et al. [Xia+16] implemented a Rowhammer attack on Xen-PVM, triggering bit flips in hypervisor page tables and consequently obtaining hypervisor privileges.

Razavi et al. [Raz+16] demonstrated a Rowhammer attack on co-located tenants in the presence of page deduplication. They first scan the entire DRAM of the system for bit flips using the Rowhammer attack. Subsequently, they fill vulnerable pages with data that they suspect to be in a victim machine that will be co-located in the future. As soon as the victim is co-located, the identical memory pages are deduplicated. If the attacker now performs the Rowhammer attack again to trigger a bit flip in the page that is now also used by the victim. Through this attack they are able to manipulate installed certificates and update URLs used in the co-located machine to install malicious fake updates.

Besides personal computers and servers, mobile devices can also be attacked using the Rowhammer bug [Vee+16]. In our work with Van der Veen et al. we show that memory allocation techniques on Android devices expose uncached memory to user programs. Thus, the user can hammer DRAM rows efficiently and flip bits in kernel-level data structures.

In simultaneous work another software-based microarchitectural fault attack has been presented. Karimi et al. [Kar+15] demonstrated that software can artificially age circuits used in specific pipeline stages. A carefully crafted instruction stream increases the latency of the critical

path of circuits if executed for several weeks. When the latency of the critical path exceeds the specification the subsequent pipeline stages work with incorrect values. Consequently, any further computations on the processor may have erroneous results.

3.3. Defenses Against Software-based Microarchitectural Attacks

Defenses against software-based microarchitectural attacks can be implemented on the user-space layer, system layer, or hardware layer. System layer and user-space layer would allow for protecting commodity systems. However, on a hardware level, defenses may induce a smaller performance overhead.

Most generic defenses try to reduce the amount of resource sharing to mitigate specific side-channel attacks. A countermeasure that has been proposed against same-core attacks is to schedule processes in different security domains only to different CPU cores [Per05; Mar+10]. Similarly, to mitigate cross-core attacks one could schedule such processes to different CPUs. For cross-CPU attacks one could perform the entire computation on different physical systems. However, the central idea of the cloud is to share resources and cloud providers will not eliminate multi-tenancy. Furthermore, information leakage through a remote interface might still be possible [Ber05].

Eliminating resource sharing does not work at all in the case of personal computers. Users deliberately want to execute third-party code such as native binaries or JavaScript on a website. Hence, it is important to find defense mechanisms that do not eliminate sharing but provide protection through other means. Furthermore, defenses against Rowhammer, as a microarchitectural fault attack, often follow similar approaches as defenses against microarchitectural side-channel attacks in many cases. In the following, we discuss possible state-of-the-art defenses grouped by the three layers: user-space layer, system layer, and hardware layer.

3.3.1. User-Space Layer

Constant-time and data-obliviousness. In his seminal work on cache timing attacks, Bernstein [Ber05] already proposed several mitigation tech-

niques to protect AES computations. He emphasized that constant-time implementations are the most important protection mechanisms for cryptographic algorithms. Furthermore, he criticized the design of AES that suggests to perform data-dependent lookups in software. Similar defense mechanisms have been advised by later work [Cop+09; BLS12; Zha+12]. In his work on RSA cache timing attacks, Aciicmez et al. [Acı07a] proposed eliminating secret-dependent branches to defeat branch prediction attacks. Agosta et al. [Ago+07] suggested to either eliminate secret-dependent branches by replacing the branch by arithmetic operations or by converting branches to indirect jumps. Joye et al. [JT07] proposed an oblivious software exponentiation that performs no secret-dependent data or code accesses and thus is cache side-channel resistant.

Today, constant-time and oblivious computations are a standard countermeasure against attacks on cryptographic implementations. However, it has been found very hard to write truly constant-time implementations for some algorithms [BS13; YGH16; GBY16]. Still, microarchitectural side-channel attacks on cryptographic algorithms can generally be mitigated entirely in user-space implementations.

Preloading all data into the cache before running an algorithm has been investigated as a countermeasure against cache side-channel attacks. Hilton et al. [HLL16] demonstrated that this can improve performance of secure enclaves significantly while eliminating leakage based on cache misses. However, this approach is probabilistic and an attacker running on a second core might still be able to manipulate the cache state to restore the cache misses and thus, the leakage. However, experiments we performed showed that prefetching or preloading AES T-tables does not have any significant effect on asynchronous *Flush+Reload* and *Prime+Probe* cross-core cache attacks.

3.3.2. System Layer

Constant-time and data-obliviousness. Andrysco et al. [And+15] developed a library that allows constant-time fixed-point numeric computations. Ohrimenko et al. [Ohr+16] developed a framework that allows making any algorithm data- and code-oblivious. Through conditional CPU operations the control flow is always the same and memory fetches are always performed. They demonstrated the practicality of their solution on commonly used machine learning algorithms run in Intel SGX

enclaves. However, the framework does not protect against accidentally adding code that leaks side-channel information through timing or data accesses. From a more theoretical aspect Oblivious RAM (ORAM) could provide data-obliviousness in general. To perform an array lookup, a simple ORAM construction would access every element in the array. More complex ORAM constructions achieve a lower runtime overhead while still maintaining certain lower security bounds for distinguishing any two array lookups based on the access sequence. In practice, ORAM suffers from severe performance and latency penalties and is therefore not widely applicable.

Manipulation of timing sources. Most microarchitectural attacks require some form of accurate timing measurements. Consequently, simulating timing sources to reduce their usefulness for attackers has been proposed in several independent works [Avi+10a; Avi+10b; For12; Wu+15a; Wu+15b; LGR13; MDS12]. In modern cloud environments, this is typically already the case. Every virtual machine has its own timing offsets, including low-level timers like cycle counter registers. However, microarchitectural attacks seem widely unaffected from this [Ira+14; IES16; Liu+15].

Vattikonda et al. [VDS11] proposed to add additional fuzziness to timers to destroy any reliable timing information for the guest. However, microbenchmark measurements as used in modern cache attacks only require minimal timing differences [Gru+16b; Mau+17]. Furthermore, statistical methods allow to align traces and recover the secret information [Liu+15]. Furthermore, even in absence of any timing source an attacker can fall-back to a counting thread [Wra92; Lip+16] or build even more sophisticated and accurate timers [Sch+17c; Sch+17b]. Hence, it would be necessary to prevent any form of parallelism in the attacker process and also prevent any access to indirect timing sources such as interrupts from a preemptive scheduler [Dun+02; Avi+10a; Ste+13; Coc+14].

Timing differences in general can be made invisible to other processes, by making algorithms always consume the worst case time or by bucketing their computation time [KD09; AZM10; ZAM11; Coc+14]. This can be effective against timing attacks including the *Evict+Time* cache timing attack. However, it does not have a significant effect on other cache side-channel attacks such as *Flush+Reload* or *Prime+Probe*, as they can derive exactly when a memory access is performed and not only overall execution time differences. Many threat models, especially for multi-user environments, exclude direct measurement of the execution time of a

specific algorithm. Hence, *Evict+Time* is not possible in such a threat model, whereas *Flush+Reload* and *Prime+Probe* are.

Disabling cache-line sharing and shared memory. Disabling resource sharing can be applied on every level for different resources with different granularities. As the last-level cache is typically physically-indexed and physically-tagged, cache lines can only be shared across processes if they are part of a shared memory region. Disabling cache-line sharing by avoiding shared memory has first been suggested by Yarom et al. [YF14] against *Flush+Reload* attacks. However, this would increase memory utilization significantly and also increase the execution time due to higher cache miss rates.

Another source of shared memory is page deduplication. Suzaki et al. [Suz+11] recommended to disable page deduplication in cloud environments to mitigate page deduplication attacks. The importance of disabling page deduplication in cloud environments has been substantiated with more sophisticated attacks being demonstrated [OW11; Xia+12; Xia+13; Bar+15; Raz+16]. In our work on page deduplication attacks, we also recommended disabling page deduplication to prevent attacks on personal computers [GBM15]. Bosman et al. [Bos+16] came to the same conclusion in order to prevent Rowhammer attacks on Windows systems.

While the large cloud providers have already disabled page deduplication, cloud providers generally have a large interest in keeping page deduplication enabled to optimize resource utilization and increase cost efficiency. Ning et al. [Nin+16] designed a system that enables page deduplication between virtual machines belonging to the same group, while preventing cross-group attacks.

Zhou et al. [ZRZ16] proposed "copy-on-access", a more dynamic approach to disable cache-line sharing. They developed a system where the operating system or hypervisor dynamically creates copies of pages that are used simultaneously by multiple programs. However, in practice it only increases the amount of measurements an attacker has to perform, but does not fully mitigate any cache side-channel attack.

To mitigate microarchitectural attacks on KASLR, we proposed to use separate paging structures for kernel space and user space to avoid sharing cache lines in paging structure caches [Gru+16a]. This was also proposed in concurrent work by Jang et al. [JLK16]. Gruss et al. [Gru+17b] showed

that separating paging structures indeed eliminates the page-translation cache side channel.

Avoiding cache-set sharing. Assuming the problem of cache-line sharing is solved and thus *Flush+Reload* attacks are not possible anymore, cache-set sharing still allows performing *Prime+Probe* attacks. To prevent cache-set sharing, cache-coloring has been proposed as a countermeasure against side-channel attacks by Shi et al. [Shi+11]. Kim et al. [KPM12] implemented a protection mechanism for cryptographic implementations based on cache coloring on modern Intel CPUs. Godfrey [GZ14] implemented a similar protection mechanism in the Xen hypervisor. Cock [Coc+14] evaluated cache coloring on ARM-based devices. In all cases the authors measured only a small performance impact. However, the memory overhead is significant: colors are fixed to physical addresses and thus large portions of physical memory have to be assigned to the same virtual machine or process in order to provide strict cache coloring without other virtual machines or processes working in the same cache set. Costan et al. [CD16] proposed Sanctum, an alternative to Intel SGX that employs cache coloring to protect against cache side-channel attacks on enclaves.

The slices used in the last-level cache in modern Intel processors can be utilized for cache coloring [HWH13; Mau+15a; Inc+15; Yar+15]. They already implicitly make cache attacks more difficult as the attacker has to gain knowledge on how addresses map to slices. Additionally, slices facilitates implementing cache coloring schemes on a system level. With every slice the number of colors is multiplied by 2.

With Intel CAT (cache allocation technology) [Int14], system software can now directly control how the slices are used. Intel CAT allows restricting cores to a subset of slices of the last-level cache. By separating processes of different security domain to different cores and thus their data into different cache slices, any cache-set sharing is eliminated. Liu et al. [Liu+16] implemented CATalyst, a system that instruments Intel CAT to protect general purpose software and cryptographic algorithms. They use Intel CAT to pin cache lines in the cache by first restricting access to one slice to its core and subsequently removing all cores from this slice. Still, cached values are served from the cache, effectively pinning the values cached in this slice into the cache. A compiler could generate the code to protect secret-dependent operations with Intel CAT. The performance overhead for low-memory tasks is negligible.

Intel CAT likely can also be used to prevent DRAMA side-channel attacks. The attacker may not be able to flush or evict victim data. Consequently, the victim process has full control over all its memory accesses in terms of cache hits and cache misses and can avoid information leakage through cache misses.

Weiß et al. [Wei+14] developed a scheduler that reduces the amount of cache set sharing between different virtual machines. Moon et al. [MSR15] proposed frequent VM migration to avoid colocation for longer periods in time. A similar approach based on container migration has been proposed by Azab et al. [AE16].

Avoiding spatial proximity. To mitigate Rowhammer attacks it is not sufficient to avoid memory and cache-set sharing, but it is also necessary to avoid spatial proximity in DRAM. Brasser et al. [Bra+16] proposed to isolate processes running in different security domains in DRAM such that no security domain can flip bits in another security domain.

Cache cleansing. If we assume that attacker and victim cannot access any cache set simultaneously, the question is how to cope with leakage that remains in the cache after the victim was descheduled. Cache cleansing aims to protect against attacks in such a scenario. It tries to maintain the cache in a state that leaks no information to prevent cache attacks. Zhang et al. [ZR13] and Godfrey et al. [GZ14] proposed cache cleansing to prevent leakage in cloud scenarios. They flush the cache upon context switches to eliminate the secret information from the cache. Cock et al. [Coc13] implemented a scheduler that reduces the amount of flushes on context switches and while maintaining the same security properties. Varadarajan et al. [VRS14] proposed a minimum-runtime guarantee for virtual machines in the cloud to prevent frequent context switches between different virtual machines. They also use cache cleansing to mitigate leakage of sensitive data. While this does not degrade system performance significantly it increases latency for requests to other VMs by up to 17%. Braun et al. [BJB15] proposed to compile specially annotated functions to be constant-time and data-oblivious. These functions also do not access any shared cache sets. Furthermore, they employ cache cleansing before and after the secret-dependent execution.

With the advent of multi-core processors, cache cleansing lost some of its practical relevance. Although disabling hyperthreading might be viable, disabling multi-core or the last-level cache entirely is not a practical

solution. Even without the last-level cache, coherency protocols can keep cache lines coherent across processors and reintroduce the supposedly eliminated timing differences.

Detecting vulnerabilities. A different branch of countermeasures are mechanisms to detect vulnerabilities in software. Detected vulnerabilities can be eliminated by patching the software.

With Cache Template Attacks [GSM15], we presented a way to scan software for vulnerabilities. Doychev et al. [Doy+15] proposed to detect potential leakage in applications using static analysis techniques. A similar system has been proposed by Irazoqui et al. [IES17] to detect microarchitectural attacks including DRAMA and Rowhammer. Reparaz et al. [RBV16] proposed black-box leakage detection for cryptographic implementations and other algorithms. Zankl et al. [ZHS16] advised to incorporate automated leakage detection in the deployment process for cryptographic libraries. In line with these works are also approaches to quantify cache leakage using detailed abstract models of the cache [Dem+12; DK16; Cha+16]. A developer can use this information to eliminate the leakage through source code improvements.

To mitigate Rowhammer attacks, Kim et al. [Kim+14] proposed to detect vulnerable DRAM rows and remap them to spare DRAM cells. Brasser et al. [Bra+16] proposed to disable these DRAM regions in the boot loader.

Detecting and stopping ongoing attacks. Another form of detection mechanisms aim at detecting ongoing attacks. Following the idea of virus and malware scanners, a software runs continuously checking the system for malicious activity and subsequently stop the attacking processes or virtual machines. Zhang et al. [Zha+11] proposed HomeAlone, a system using a *Prime+Probe* covert channel to detect colocation. Their system allows detecting when a virtual machine is co-located with other virtual machines on the same physical machine although being billed for a dedicated machine. With Cache Template Attacks [GSM15], we presented a way to search and detect ongoing cache attacks. However, both approaches increase the system load significantly.

Cardenas et al. [CB12] used performance counters to detect microarchitectural denial-of-service attacks in cloud environments. Demme et al. [Dem+13] proposed the use of performance counters to detect abnormal cache behavior to detect malware and Tang et al. [TSS14] enhanced this idea by evaluating performance counters using unsupervised learning.

Chouhan et al. [CH16] proposed to use bloom filters on the cache miss traces to detect yet unknown cache side-channel attacks. Hunger et al. [Hun+15] proposed detecting side-channel attacks through measuring the performance variations in a program that mimics a typical victim application.

Herath and Fogh [HF15] proposed to monitor cache misses to detect *Flush+Reload* attacks and Rowhammer. Similar approaches have been chosen by Chiappetta et al. [CSY15] and Zhang et al. [ZZL16]. Both built systems that use cache hit and miss traces to detect *Flush+Reload* attacks in native and cloud environments respectively. We showed that detection mechanisms through performance counters might be an insufficient solution [Gru+16b]. We showed that performance counters fail to detect all variants of cache attacks, such as *Flush+Flush* and slowed-down variants of *Flush+Reload*. In response, Fogh [Fog15] developed a mechanism that uses performance-monitor interrupts on `rdtsc` instructions. This mechanism checks the program code around the current instruction pointer for suspicious instructions like `clflush`. If suspicious instructions are found, the potentially malicious program is slowed down or terminated. Payer [Pay16] developed a system called HexPADS, which detects cache attacks and Rowhammer at runtime. HexPADS uses different performance events like cache references, cache misses, and page faults to evaluate whether a process is malicious. HexPADS can easily be applied to commodity operating systems.

Chen et al. [Che+17] designed a framework to detect ongoing controlled-channel attacks on SGX enclaves at runtime. They use TSX to built a trusted in-enclave counting thread. This counting thread is used to measure the execution time of code sections. If the execution time is too high an interrupt must have occurred and thus the untrusted operating system interrupted the enclave, likely to perform an attack.

To specifically detect and stop ongoing Rowhammer attacks, several works propose the usage of performance counters to detect whether to induce or wait for row refreshes [Kim+14; Cor16; Awe+16].

3.3.3. Hardware Layer

Eliminating timing differences. Naturally, timing differences introduced by the hardware could be eliminated in hardware [Pag03; Ber05; Per05]. The best example might be the `clflush` instruction, which has

a small but exploitable timing difference when accessing cached and uncached memory locations. Making this instruction constant-time would likely not be noticed in practice [Gru+16b]. The instruction is rarely used and a timing penalty of less than 10 cycles is negligible for rarely used instructions. Similarly, prefetch instructions leak through timing differences that could be eliminated without a significant performance penalty [Gru+16a].

Timing differences due to hardware modification only occur if the software runs through an instruction stream that runs into these hardware optimizations. Leakage introduced this way could be avoided by providing constant-time instructions as building blocks for more complex algorithms, e.g., cryptographic implementations. Indeed, processor manufacturers are incorporating and increasing number of constant-time instructions for cryptographic primitives [Int08; AMD09; ARM12; ARM13], most prominently Intel AES-NI. Instruction set extensions like AES-NI have been proposed as countermeasures against various attacks [Pag03; Ber05; Per05]. Today, many cryptographic libraries use these instructions typically by default [Ope].

Wang et al. [WFS14] proposed to change memory controllers to eliminate timing side-channels. Most importantly, they suggest changes to the row-buffer policy. Instead of keeping the row buffer open, they immediately close the row buffer again, leaving the DRAM in a pre-charged state. This introduces a lower latency penalty than a row conflict, but still a significant performance penalty as compared to current implementations. This countermeasure would likely eliminate DRAMA side-channel attacks [Pes+16].

Timing differences are no problem if the execution time does not vary depending on secret information. Wang et al. [WL07] proposed to eliminate cache side-channel leakage with their so-called partition-locked caches (PLcache) and random-permutation caches (RPcache). The PLcache allows locking cache lines in the cache and prevent their eviction. Hence, an attacker cannot observe any cache misses from the victim process as the victim operates entirely on the cache. The RPcache approach introduces a different mapping from physical addresses to cache sets for every process at runtime. Hence, each program has its own cache sets which may be overlapping or disjoint but are never identical. Consequently, the attacker cannot prime a cache set of the victim and thus the attacker cannot perform a cache attack on the victim.

Kong et al. [Kon+08] showed that both the PLcache and the RPcache protections can be circumvented by an attacker. They proposed informing loads as an extension to the RPcache to protect against cache side channels. Informed loads are special instructions that do not only perform the memory load but also re-randomize data structures [Kon+09]. Liu et al. [LL14] proposed a cache design where the mapping from addresses to cache sets is dynamically randomized at runtime. While the randomized address mapping does not prevent that attacker and victim share a cache set, it does effectively prevent that attacker and victim share a cache set over a longer period in time. Consequently, many attacks are mitigated. Fuchs et al. [FL15] proposed to use specialized prefetching algorithms to mitigate side-channel attacks.

Disabling resource sharing. Also on the hardware level it is possible to disable or reduce the amount of resource sharing. Page [Pag05] suggested partitioned caches to avoid cache-set sharing across processes. These per-process partitions are maintained dynamically, avoiding any static cache mapping and cache sharing. Wang et al. [WL08] proposed a new cache architecture to mitigate cache side-channel attacks. Their cache architecture would prevent cache-set sharing between attacker and victim using dynamic mappings between addresses and cache sets. Tiwari et al. [Tiw+09] proposed a mechanism to execute untrusted code with tight upper limits on the leakage in terms of time and side effects, with a moderate performance impact. Sanchez et al. [SK10] proposed a faster cache design which decouples cache ways and cache associativity. This cache design is also likely to impact the applicability of eviction-based attacks like *Prime+Probe*. Domnitser et al. [Dom+11] proposed non-monopolizable caches as a defense against cache attacks. Non-monopolizable caches prevent that any process can allocate enough cache lines to observe cache collisions with another process. Domnitser et al. observed a low performance penalty for cryptographic algorithms. However, using non-monopolizable caches for larger parts of the software stack would severely impact the system performance, as it is equivalent to reducing the cache size per process.

To mitigate branch prediction attacks, Tan et al. [TWG14] proposed a new branch target buffer scheme which allows detecting potentially malicious activity. After detection, the hardware prevents that branch target buffer entries are shared with the suspected malicious processes.

Rowhammer countermeasures in hardware. Hardware countermeasures against Rowhammer are specific to the Rowhammer hardware defi-

ciency. Kim et al. [Kim+14] proposed several solutions to the Rowhammer bug, including usage of ECC memory, building more reliable DRAM cells, and increasing the refresh rate. They also showed that increasing the refresh rate is not always effective unless it is increased by a factor of 7 or more. Instead, they propose PARA, a mechanism which probabilistically opens adjacent rows. As Rowhammer attacks require a huge number of accesses, the adjacent rows are very likely refreshed early enough and no bit flip occurs.

A different mechanism is target-row refresh (TRR). TRR refreshes lines after a certain number of accesses to adjacent lines. Although TRR has been announced for DDR4 modules, it was removed from the final DDR4 standard [GMM16].

4

Future Work and Conclusions

We can draw conclusions on four different axis from this thesis and the corresponding publications.

First, microarchitectural attacks can be widely automated. We have shown this with our work on Cache Template Attacks [GSM15], but automation also played a significant role in our other works [Lip+16; Mau+17]. Automation provides any unsophisticated user with the ability to perform microarchitectural attacks. It also enables more large scale attacks. Future work will likely investigate automation of microarchitectural attacks in further detail.

Second, unknown and novel side channels are very likely to exist and to be found. We showed that modern microarchitectures expose several previously unknown side channels, such as the `clflush` instruction [Gru+16b], the DRAM [Pes+16], or prefetch instructions [Gru+16a]. While we found several new side channel, it is more difficult to find all microarchitectural side channels. Hence, we can expect to find more microarchitectural side channels and especially find new side channels with every new microarchitecture. Furthermore, many microarchitectural side channels have not been investigated in detail yet. For instance, it is likely that the prefetch side channel [Gru+16a] contains information that has not been used in any published attacks. Also microarchitectural elements at low levels, closer to the execution core, should be investigated for new side channels. Future work should investigate whether these hardware components, such as graphic adapters, can be instrumented in attacks.

Third, it is possible to reduce and minimize requirements of known attacks to a point where they can be performed in highly-restricted and sandboxed environments. We have shown that this is the case in our work on Rowhammer attacks in JavaScript [GMM16] and in our work on page deduplication attacks in JavaScript [GBM15]. In terms of software-based microarchitectural fault attacks, we are just starting to investigate various

hardware elements and how they can be accessed from unprivileged environments. Investigating the applicability of Rowhammer [Kim+14; SD15] and MAGIC [Kar+15] attacks in different scenarios will help to assess their risks. However, besides the DRAM and specific processor components, there is an abundance of other hardware elements in modern systems that could be attacked. Such novel attacks might reduce the requirements for fault attacks even further.

Fourth, constructing both effective and efficient countermeasures is a difficult task. Research often over-ambitiously aims to find universal countermeasures against microarchitectural attacks, ignoring that the various attacks have vastly different requirements and properties [Gru+16b; Pes+16]. At the core of microarchitectural attacks is usually a temporal or behavioral difference that is intended by the processor manufacturer to optimize the performance. Hence, it we cannot always find a universal countermeasure that does not degrade the performance as was the case for the prefetch side channel [Gru+16a]. Security and performance are contradicting each other to a growing extent. Countermeasures can only be practical if they provide useful and possibly dynamic trade-offs between security and performance. Instead of universal countermeasures, it may appear as a low-hanging fruit to protect specific scenarios, but it is also more likely to be practical. Especially cryptographic implementations are already being constantly improved to defend against new microarchitectural attacks.

References

[ABG10] O. Acıiçmez, B. B. Brumley, and P. Grabher. New Results on Instruction Cache Attacks. In: CHES'10. 2010 (p. 32).

[Acı07a] O. Acıiçmez. Advances in Side-Channel Cryptanalysis: MicroArchitectural Attacks. PhD Thesis. Oregon State University, 2007 (pp. 35, 43).

[Acı07b] O. Acıiçmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In: Proceedings of the 1st ACM Computer Security Architecture Workshop. 2007 (p. 32).

[Adv13] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. 2013. URL: `http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf` (p. 27).

[AE16] M. Azab and M. Eltoweissy. MIGRATE: Towards a Lightweight Moving-target Defense against Cloud Side-Channels. In: IEEE Security and Privacy Workshops (SPW). 2016 (p. 47).

[Ago+07] G. Agosta, L. Breveglieri, G. Pelosi, and I. Koren. Countermeasures against branch target buffer attacks. In: IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'07). 2007 (p. 43).

[Aic15a] B. Aichinger. DDR memory errors caused by Row Hammer. In: HPEC'15. 2015 (p. 41).

[Aic15b] B. Aichinger. Row Hammer Failures in DDR Memory. In: memcon'15. 2015 (p. 41).

[AK] O. Acıiçmez and C. K. Koç. Microarchitectural attacks and countermeasures. In: (p. 29).

[AK06] O. Acıiçmez and Ç. K. Koç. Trace-Driven Cache Attacks on AES (Short Paper). In: Proceedings of the 8th international conference on Information and Communications Security. 2006, pp. 112–121 (p. 32).

[All+16] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom. Amplifying Side Channels Through Performance Degradation. In: Proceedings of the 32th Annual Computer Security Applications Conference (ACSAC'16). 2016 (p. 34).

[AMD09] AMD. AMD I/O Virtualization Technology (IOMMU) Specification, rev 1.26. 2009 (p. 50).

[And+15] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In: S&P'15. 2015 (p. 43).

[ARM12] ARM Limited. ARM Architecture Reference Manual. ARM v7-A and ARMv7-R edition. ARM Limited, 2012 (p. 50).

[ARM13] ARM Limited. ARM Architecture Reference Manual ARMv8. ARM Limited, 2013 (p. 50).

[AS07] O. Acıiçmez and J.-P. Seifert. Cheap Hardware Parallelism Implies Cheap Security. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007) (Sept. 2007), pp. 80–91 (p. 32).

[AS08a] O. Acıiçmez and W. Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In: CT-RSA 2008. 2008 (p. 32).

[AS08b] O. Acıiçmez and W. Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In: CT-RSA 2008. 2008 (p. 39).

[ASK07] O. Acıiçmez, J.-P. Seifert, and Ç. K. Koç. Predicting secret keys via branch prediction. In: CT-RSA 2007. 2007 (p. 35).

[Aum+02] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In: CHES'02. 2002 (pp. 5, 39).

[Avi+10a] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating timing channels in compute clouds. In: Proceedings of the 2nd ACM Cloud Computing Security Workshop (CCSW'10). 2010, pp. 103–108 (p. 44).

[Avi+10b] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). 2010 (p. 44).

[Awe+16] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In: ACM SIGPLAN Notices 51.4 (2016), pp. 743–755 (p. 49).

[AZM10] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In: CCS'10. 2010 (p. 44).

[Bac+10] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. Acoustic Side-Channel Attacks on Printers. In: USENIX Security Symposium. 2010 (p. 5).

[Bar+15] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: Silently Breaking ASLR in the Cloud. In: WOOT'15. 2015 (p. 45).

[BDL97] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In: International Conference on the Theory and Applications of Cryptographic Techniques. Springer. 1997, pp. 37–51 (pp. 5, 39, 41).

[Ben+14] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In: CHES'14. 2014 (p. 34).

[Ber05] D. J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 6, 30, 42, 49, 50).

[BH09] B. Brumley and R. Hakala. Cache-Timing Template Attacks. In: ASIACRYPT'09. 2009 (p. 32).

[BJB15] B. A. Braun, S. Jana, and D. Boneh. Robust and Efficient Elimination of Cache and Timing Side Channels. In: arXiv:1506.00189 (2015) (p. 47).

[BLS12] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In: International Conference on Cryptology and Information Security in Latin America. 2012 (p. 43).

[BM06] J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. In: CHES'06. 2006 (pp. 6, 30, 32).

[BM15] S. Bhattacharya and D. Mukhopadhyay. Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms. In: Cryptology ePrint Archive, Report 2015/621 (2015) (p. 35).

[BM16] S. Bhattacharya and D. Mukhopadhyay. Curious case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis. In: CHES'16. 2016 (pp. 38, 41).

[Bos+16] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P'16. 2016 (pp. 7, 41, 45).

[Bra+16] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. In: arXiv:1611.08396 (2016) (pp. 47, 48).

[Bra+17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: arXiv:1702.07521 (2017) (p. 33).

[BS13] D. J. Bernstein and P. Schwabe. A word of warning. In: CHES'13 Rump Session. 2013 (p. 43).

[BS97] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In: Annual International Cryptology Conference. Springer. 1997, pp. 513–525 (pp. 5, 39).

[BWM16] J. Betz, D. Westhoff, and G. Müller. Survey on covert channels in virtual machines and cloud computing. In: Transactions on Emerging Telecommunications Technologies (2016) (p. 29).

[CB12] C. Cardenas and R. 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).

[CD16] V. Costan and S. Devadas. Intel SGX explained. Tech. rep. Cryptology ePrint Archive, Report 2016/086, 2016 (p. 46).

[CH16] M. Chouhan and H. Hasbullah. Adaptive detection technique for Cache-based Side Channel Attack using Bloom Filter for secure cloud. In: 3rd International Conference on Computer and Information Sciences (ICCOINS). IEEE. 2016 (p. 49).

[Cha+16] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the Information Leak in Cache Attacks through Symbolic Execution. In: arXiv:1611.04426 (2016) (p. 48).

[Che+17] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In: Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'17). 2017 (p. 49).

[Coc+14] D. Cock, Q. Ge, T. Murray, and G. Heiser. The last mile: an empirical study of timing channels on seL4. In: CCS'14. 2014 (pp. 44, 46).

[Coc13] D. Cock. Practical probability: Applying pGCL to lattice scheduling. In: International Conference on Interactive Theorem Proving. 2013 (p. 47).

[Cop+09] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: S&P'09 45-60 (2009) (p. 43).

[Cor16] J. Corbet. Defending against Rowhammer in the kernel. Oct. 2016. URL: `https://lwn.net/Articles/704920/` (p. 49).

[CSW17] E. Carmon, J.-P. Seifert, and A. Wool. Photonic Side Channel Attacks Against RSA. In: HOST'17. 2017 (p. 5).

[CSY15] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Cryptology ePrint Archive, Report 2015/1034. 2015 (p. 49).

[Dem+12] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In: ACM SIGARCH Computer Architecture News 40.3 (2012), pp. 106–117 (p. 48).

[Dem+13] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In: ACM SIGARCH Computer Architecture News 41.3 (2013), pp. 559–570 (p. 48).

[DK16] G. Doychev and B. Köpf. Rigorous Analysis of Software Countermeasures against Cache Attacks. In: arXiv:1603.02187 (2016) (pp. 29, 48).

[Dom+11] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. In: ACM Transactions on Architecture and Code Optimization (TACO) 8.4 (2011) (p. 51).

[Doy+15] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: a tool for the static analysis of cache side channels. In: ACM Transactions on Information and System Security (2015) (p. 48).

[DR13] J. Daemen and V. Rijmen. The Design of Rijndael: AES – The Advanced Encryption Standard. 2013 (p. 30).

[Dun+02] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In: ACM SIGOPS Operating Systems Review (2002) (p. 44).

[EP16] D. Evtyushkin and D. Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In: CCS'16. 2016 (p. 39).

[EPA15] D. Evtyushkin, D. Ponomarev, and N. 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, p. 5 (p. 35).

[EPA16] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: International Symposium on Microarchitecture (MICRO'16). 2016 (p. 35).

[FL15] A. Fuchs and R. B. Lee. Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In: Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15). 2015 (p. 51).

[Fog15] A. Fogh. Detecting stealth mode cache attacks: Flush+Flush. 2015. URL: `http://dreamsofastone.blogspot.co.at/2015/11/detecting-stealth-mode-cache-attacks.html` (p. 49).

[For12] B. Ford. Plugging side-channel leaks with timing information flow control. In: Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing. 2012 (p. 44).

[GBK11] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: IEEE Symposium on Security and Privacy – S&P. IEEE Computer Society, 2011, pp. 490–505 (pp. 6, 30, 34).

[GBM15] D. Gruss, D. Bidner, and S. Mangard. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In: 20th European Symposium on Research in Computer Security (ESORICS'15). 2015 (pp. 8, 9, 12, 37, 45, 53).

[GBY16] C. P. García, B. B. Brumley, and Y. Yarom. Make Sure DSA Signing Exponentiations Really are Constant-Time. In: CCS'16. 2016 (p. 43).

[Ge+16a] Q. Ge, Y. Yarom, F. Li, and G. Heiser. Contemporary Processors Are Leaky – and There's Nothing You Can Do About It. In: arXiv:1612.04474 (2016) (pp. 29, 35).

[Ge+16b] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In: Journal of Cryptographic Engineering (2016), pp. 1–27. DOI: 10.1007/s13389-016-0141-6 (p. 29).

[GMM16] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16. 2016 (pp. 7–9, 12, 40, 52, 53).

[Gra+17] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: (2017) (p. 36).

[Gro+16] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In: CHES'16. 2016 (p. 35).

[Gru+16a] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS'16. 2016 (pp. 7, 8, 10, 12, 36, 45, 50, 53, 54).

[Gru+16b] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA'16. 2016 (pp. 7–9, 12, 34, 44, 49, 50, 53, 54).

[Gru+17a] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: USENIX Security Symposium. (to appear). 2017 (p. 12).

[Gru+17b] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS'17. (to appear). 2017 (pp. 11, 45).

[GSM15] D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (pp. 7, 8, 12, 34, 35, 48, 53).

[Gül+15] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In: Constructive Side-Channel Analysis and Secure Design (COSADE). 2015 (p. 34).

[GZ14] M. M. Godfrey and M. Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. In: IEEE Transactions on Cloud Computing (2014) (pp. 46, 47).

[HF15] N. Herath and A. Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In: Black Hat 2015 Briefings. Aug. 2015 (p. 49).

[HLL16] A. Hilton, B. Lee, and T. Lehman. PoisonIvy: Safe speculation for secure memory. In: Proceedings of the 49th International Symposium on Microarchitecture (MICRO'16). 2016 (p. 43).

[HS13] M. Hutter and J.-M. Schmidt. The temperature side channel and heating fault attacks. In: International Conference on Smart Card Research and Advanced Applications. Springer. 2013, pp. 219–235 (pp. 5, 39).

[Hun+15] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. Understanding contention-based channels and using them for defense. In: IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). 2015 (p. 49).

[HWH13] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P'13. 2013 (pp. 7, 31, 36, 46).

[IES15] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P'15. 2015 (pp. 33, 34).

[IES16] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'16). 2016 (pp. 35, 44).

[IES17] G. Irazoqui, T. Eisenbarth, and B. Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. Cryptology ePrint Archive, Report 2016/1196. 2017. URL: `http://eprint.iacr.org/2016/1196` (p. 48).

[Inc+15] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. 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 (pp. 24, 32, 33, 46).

[Inc+16] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In: CHES'16. 2016 (p. 34).

[İnc+16] M. S. İnci, G. Irazoqui, T. Eisenbarth, and B. Sunar. Efficient, adversarial neighbor discovery using logical channels on Microsoft Azure. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM. 2016 (p. 38).

[Int08] Intel. Advanced Encryption Standard (AES) Instructions Set: White Paper. 2008 (p. 50).

[Int14] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. In: 253665 (2014) (p. 46).

[Ira+14] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14. 2014 (pp. 35, 44).

[Ira+15a] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. In: Proceedings on Privacy Enhancing Technologies (2015) (p. 34).

[Ira+15b] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Lucky 13 Strikes Back. In: AsiaCCS'15. 2015 (p. 34).

[JLK16] Y. Jang, S. Lee, and T. Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS'16. 2016 (pp. 36, 45).

[JT07] M. Joye and M. Tunstall. Securing OpenSSL against Micro-Architectural Attacks. In: SECRYPT. 2007 (p. 43).

[Kar+15] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri. MAGIC: Malicious aging in circuits/cores. In: ACM Transactions on Architecture and Code Optimization (TACO) 12.1 (2015) (pp. 7, 41, 54).

[Kay+16] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In: Proceedings of the 53rd Annual Design Automation Conference. 2016 (p. 33).

[KD09] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In: 22nd IEEE Computer Security Foundations Symposium. 2009 (p. 44).

[Kel+00] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. In: Journal of Computer Security 8.2/3 (2000), pp. 141–158 (p. 30).

[Kim+14] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA'14. 2014 (pp. 7, 9, 39, 48, 49, 52, 54).

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: Crypto'96. 1996 (pp. 5, 6, 30).

[Kon+08] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 2nd ACM Computer Security Architectures Workshop (2008) (p. 51).

[Kon+09] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA'09). 2009, pp. 393–404 (p. 51).

[KOP09] T. Kasper, D. Oswald, and C. Paar. EM side-channel attacks on commercial contactless smartcards using low-cost equipment. In: Information Security Applications. Springer, 2009, pp. 79–93 (p. 5).

[KPM12] T. Kim, M. Peinado, and G. Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security Symposium. 2012 (p. 46).

[Lan16] M. Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. Mar. 2016. URL: `http://www.thirdio.com/rowhammer.pdf` (p. 41).

[Lee+16] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In: arXiv:1611.06952 (2016) (p. 35).

[LGR13] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In: DNS'13. 2013 (p. 44).

[Lip+16] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (pp. 8, 9, 12, 31, 33–35, 44, 53).

[Liu+15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: IEEE Symposium on Security and Privacy – SP. IEEE Computer Society, 2015, pp. 605–622 (pp. 7, 33, 44).

[Liu+16] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: IEEE International Symposium on High Performance Computer Architecture (HPCA). 2016 (p. 46).

[LL14] F. Liu and R. B. Lee. Random Fill Cache Architecture. In: IEEE/ACM International Symposium on Microarchitecture (MICRO'14). 2014, pp. 203–215 (p. 51).

[Mar+10] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, and V. Bertocci. Security best practices for developing windows azure applications. In: Microsoft Corp (2010) (p. 42).

[Mau+15a] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID'15. 2015 (pp. 24, 32, 33, 46).

[Mau+15b] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA'15. 2015 (p. 33).

[Mau+17] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17. 2017 (pp. 8, 11, 33, 44, 53).

[MDS12] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: ACM SIGARCH Computer Architecture News (2012) (p. 44).

[Mił+09] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In: USENIX ATC'09. 2009 (p. 37).

[MOP08] S. Mangard, E. Oswald, and T. Popp. Power analysis attacks: Revealing the secrets of smart cards. Vol. 31. Springer Science & Business Media, 2008 (p. 5).

[MSR15] S.-J. Moon, V. Sekar, and M. K. Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In: CCS'15. 2015 (p. 47).

[Nat01] National Institute of Standards and Technology. Advanced Encryption Standard. NIST FIPS PUB 197. 2001 (p. 30).

[Nin+16] F. Ning, M. Zhu, R. You, G. Shi, and D. Meng. Group-Based Memory Deduplication against Covert Channel Attacks in Virtualized Environments. In: IEEE Trustcom. 2016 (p. 45).

[NS06] M. Neve and J.-P. Seifert. Advances on Access-Driven Cache Attacks on AES. In: Proceedings of the 13th international conference on Selected areas in cryptography (SAC'06). 2006 (p. 32).

[NSW06] M. Neve, J.-P. Seifert, and Z. Wang. A refined look at Bernstein's AES side-channel analysis. In: Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS'06). 2006 (p. 30).

[Ohr+16] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In: USENIX Security Symposium. 2016 (p. 43).

[Ope] OpenSSL. OpenSSL: The Open Source toolkit for SSL/TLS. URL: `http://www.openssl.org` (p. 50).

[Ore+15] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: ACM Conference on Computer and Communications Security – CCS. ACM, 2015, pp. 1406–1418 (pp. 7, 33).

[OST06] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In: Topics in Cryptology – CT-RSA. Vol. 3860. LNCS. Springer, 2006, pp. 1–20 (pp. 6, 30–32).

[OW11] R. Owens and W. Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In: 30th IEEE International Performance Computing and Communications Conference. Nov. 2011, pp. 1–8 (pp. 37, 45).

[Pag02] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. In: Cryptology ePrint Archive, Report 2002/169 (2002) (pp. 6, 30).

[Pag03] D. Page. Defending Against Cache Based Side-Channel Attacks. Tech. rep. Department of Computer Science, University of Bristol, 2003 (pp. 49, 50).

[Pag05] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. Cryptology ePrint Archive, Report 2005/280. 2005. URL: `http://eprint.iacr.org/2005/280` (p. 51).

[Pay16] M. Payer. HexPADS: a platform to detect "stealth" attacks. In: ESSoS'16. 2016 (p. 49).

[Per05] C. Percival. Cache missing for fun and profit. In: Proceedings of BSDCan. 2005 (pp. 6, 30, 32, 42, 49, 50).

[Pes+16] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 7, 8, 10, 26, 27, 38, 41, 50, 53, 54).

[PS05] K. Pagiamtzis and A. Sheikholeslami. Using cache to reduce power in content-addressable memories (CAMs). In: Proceedings of the IEEE Custom Integrated Circuits Conference. 2005 (p. 21).

[PSY15] J. van de Pol, N. P. Smart, and Y. Yarom. Just a little bit more. In: CT-RSA 2015. 2015 (p. 34).

[QS16] R. Qiao and M. Seaborn. A New Approach for Rowhammer Attacks. In: HOST'16. 2016 (p. 41).

[Qur+07] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In: ACM SIGARCH Computer Architecture News 35.2 (June 2007), p. 381 (p. 21).

[Raz+16] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In: USENIX Security Symposium. 2016 (pp. 7, 41, 45).

[RBV16] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? Cryptology ePrint Archive, Report 2016/1123. 2016. URL: `http://eprint.iacr.org/2016/1123` (p. 48).

[Ris+09] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: ACM Conference on Computer and Communications Security – CCS. ACM, 2009, pp. 199–212 (p. 32).

[RR01] J. R. Rao and P. Rohatgi. EMpowering Side-Channel Attacks. In: IACR Cryptology ePrint Archive 2001 (2001), p. 37 (p. 5).

[SA02] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In: CHES'02. 2002 (pp. 5, 39).

[Sch+12] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert. Simple Photonic Emission Analysis of AES. In: CHES'12. 2012 (p. 5).

[Sch+17a] S. van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida. Reverse Engineering Hardware Page Table Caches Using Side-Channel Attacks on the MMU. 2017. URL: http://www.cs.vu.nl/~herbertb/download/papers/revanc_ir-cs-77.pdf (p. 36).

[Sch+17b] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA'17. (to appear). 2017 (pp. 11, 33, 44).

[Sch+17c] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: Proceedings of the 21th International Conference on Financial Cryptography and Data Security (FC'17). 2017 (pp. 11, 38, 44).

[SD15] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat 2015 Briefings. 2015 (pp. 7, 9, 39, 54).

[Sea15] M. Seaborn. How physical addresses map to rows and banks in DRAM. Retrieved on July 20, 2015. May 2015. URL: http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html (p. 27).

[Sez93] A. Seznec. A case for two-way skewed-associative caches. In: ACM SIGARCH Computer Architecture News. Vol. 21. 2. ACM. 1993, pp. 169–178 (p. 21).

[SG14] R. Spreitzer and B. Gérard. Towards More Practical Time-Driven Cache Attacks. In: IFIP International Workshop on Information Security Theory and Practice. 2014 (p. 30).

[Shi+11] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W). 2011 (p. 46).

[Shi+16] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS'16). 2016 (p. 37).

[SK10] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In: 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10). 2010 (p. 51).

[SMC09] D. Saha, D. Mukhopadhyay, and D. R. Chowdhury. A Diagonal Fault Attack on the Advanced Encryption Standard. In: IACR Cryptology ePrint Archive 2009.581 (2009) (pp. 5, 39).

[SP13a] R. Spreitzer and T. Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In: Constructive Side-Channel Analysis and Secure Design (COSADE). 2013, pp. 200–214 (p. 31).

[SP13b] R. Spreitzer and T. Plos. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In: International Conference on Network and System Security. 2013 (p. 30).

[Spr+16] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. SoK: Systematic Classification of Side-Channel Attacks on Mobile Devices. In: arXiv:1611.03748 (2016) (p. 29).

[Ste+13] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In: Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS'13). 2013 (p. 44).

[Suz+11] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory Deduplication as a Threat to the Guest OS. In: Proceedings of the 4th European Workshop on System Security. 2011 (pp. 8, 37, 45).

[SXA16] L. Simon, W. Xu, and R. Anderson. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. In: Proceedings on Privacy Enhancing Technologies (2016) (p. 37).

[Sze16] J. Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. Cryptology ePrint Archive, Report 2016/479. 2016. URL: `http://eprint.iacr.org/2016/479` (p. 29).

[Tiw+09] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09). 2009 (p. 51).

[TSS03] Y. Tsunoo, T. Saito, and T. Suzaki. Cryptanalysis of DES implemented on computers with cache. In: CHES'03. 2003, pp. 62–76 (pp. 6, 30).

[TSS14] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In: RAID'14. 2014 (p. 48).

[TWG14] Y. Tan, J. Wei, and W. Guo. The Micro-architectural Support Countermeasures against the Branch Prediction Analysis Attack. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. 2014 (p. 51).

[VDS11] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW'11). 2011 (p. 44).

[Vee+16] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS'16. 2016 (pp. 7, 10, 41).

[VRS14] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based defenses against cross-vm side-channels. In: USENIX Security Symposium. 2014 (p. 47).

[War+96] A. Warner, Q. Li, T. Keefe, and S. Pal. The impact of multilevel security on database buffer management. In: Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS'96). 1996 (p. 37).

[Wei+14] M. Weiß, B. Weggenmann, M. August, and G. Sigl. On cache timing attacks considering multi-core aspects in virtualized embedded systems. In: International Conference on Trusted Systems. 2014 (pp. 30, 47).

[Wei+16] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In: Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS'16). 2016 (p. 37).

[WFS14] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In: IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14). 2014 (p. 50).

[WL07] Z. Wang and R. 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 (p. 50).

[WL08] Z. Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In: IEEE/ACM International Symposium on Microarchitecture (MICRO'08). 2008, pp. 83–93 (p. 51).

[Wra92] J. C. Wray. An analysis of covert timing channels. In: Journal of Computer Security 1.3-4 (1992), pp. 219–232 (p. 44).

[Wu+15a] W. Wu, E. Zhai, D. Jackowitz, D. I. Wolinsky, L. Gu, and B. Ford. Warding off timing attacks in Deterland. In: arXiv:1504.07070 (2015) (p. 44).

[Wu+15b] W. Wu, E. Zhai, D. Jackowitz, D. I. Wolinsky, L. Gu, and B. Ford. Warding off timing attacks in Deterland. In: arXiv:1504.07070 (2015) (p. 44).

[WXW12] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In: USENIX Security Symposium. 2012 (p. 38).

[WXW14] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: IEEE/ACM Transactions on Networking (2014) (p. 38).

[XCP15] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: S&P'15. 2015 (p. 37).

[Xia+12] J. Xiao, Z. Xu, H. Huang, and H. Wang. A covert channel construction in a virtualized environment. In: CCS'12. 2012 (pp. 37, 45).

[Xia+13] J. Xiao, Z. Xu, H. Huang, and H. Wang. Security implications of memory deduplication in a virtualized environment. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2013 (pp. 37, 45).

[Xia+16] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: USENIX Security Symposium. 2016 (pp. 7, 41).

[Yar+15] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser. Mapping the Intel Last-Level Cache. In: Cryptology ePrint Archive, Report 2015/905 (2015), pp. 1–12 (pp. 24, 32, 46).

[YB14] Y. Yarom and N. Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. Cryptology ePrint Archive, Report 2014/140. 2014. URL: `http://eprint.iacr.org/2014/140` (p. 34).

[YF14] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 6–8, 30, 34, 45).

[YGH16] Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: A timing attack on openssl constant time rsa. In: CHES'16. 2016 (p. 43).

[ZAM11] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In: CCS'11. 2011 (p. 44).

[Zha+11] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P'11. 2011 (pp. 32, 48).

[Zha+12] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In: CCS'12. 2012 (pp. 32, 43).

[Zha+14] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS'14. 2014 (pp. 7, 34).

[Zha+16] W. Zhang, X. Jia, C. Wang, S. Zhang, Q. Huang, M. Wang, and P. Liu. A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds. In: Information and Communications Security. Springer, 2016 (p. 29).

[ZHS16] A. Zankl, J. Heyszl, and G. Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In: International Conference on Smart Card Research and Advanced Applications. Springer. 2016 (p. 48).

[ZR13] Y. Zhang and M. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: CCS'13. 2013 (p. 47).

[ZRZ16] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In: CCS'16. 2016 (p. 45).

[ZW09] K. Zhang and X. Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security Symposium. 2009 (p. 37).

[ZXZ16] X. Zhang, Y. Xiao, and Y. Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In: CCS'16. 2016 (p. 35).

[ZZL16] T. Zhang, Y. Zhang, and R. B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: International Symposium on Research in Attacks, Intrusions, and Defenses. 2016 (p. 49).

Information on Part II

Note that Part II is not included in this PDF. Please download the full version for Part II.