Martin Schwarzl

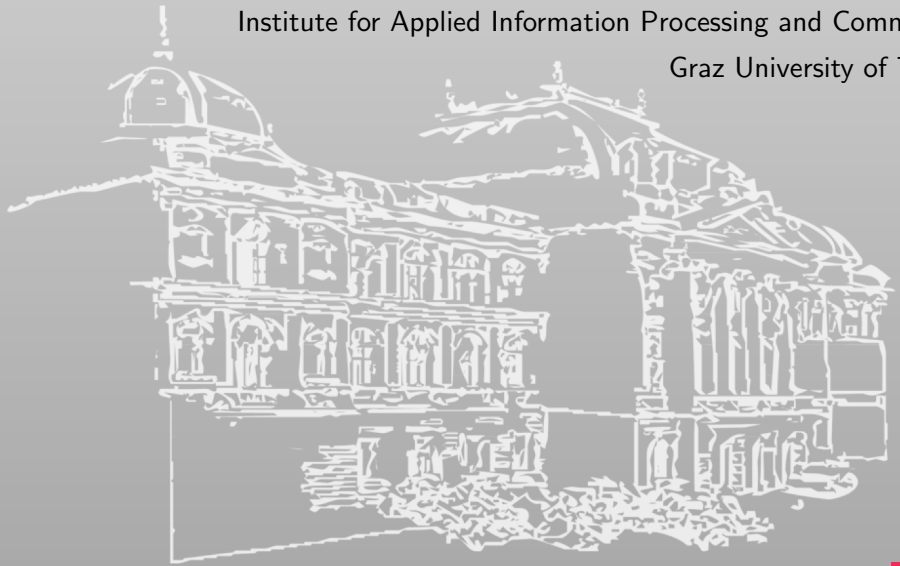
Remote Side-Channel Attacks and Defenses

PhD Thesis

Assessors: Daniel Gruss, Mathy Vanhoef

February 2023

Institute for Applied Information Processing and Communications
Graz University of Technology



SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature

Abstract

Modern software and hardware is highly optimized to meet the high performance demands of users and industry. Depending on user input, these optimizations leave measurable side effects reflected in the timing, power consumption, or electromagnetic radiation of the device. Side-channel attacks leverage the observed side effects to derive inaccessible processed information. Software-based side-channel attacks leak sensitive data from operating systems, cryptographic primitives, and real-world applications solely from software.

Many side-channel attacks run in a local setup, where the attacker has local code execution on the victim's system. However, in browsers and web applications, the attacker either has restricted code execution, e.g., sandboxed in JavaScript or only an API to interact with, making it more difficult to successfully mount attacks. Moreover, it is difficult to design efficient mitigations for commodity software and hardware.

In this thesis, we investigate remote side-channel attacks and defenses. We analyze existing software-based side-channel attacks and mount them in remote settings. Our research leads to the discovery of novel side-channel attacks in real-world applications such as browsers, databases, and web server applications. We investigate the precise requirements for an attacker to mount practical attacks across the internet. We create a remote attack on a cloud provider's production system and develop a high-performance detection and mitigation. Moreover, we develop new techniques to find side-channel leakage in real-world applications.

The first part of this thesis provides a summary of all contributions included in the thesis, the necessary background on optimizations in software and hardware, and software-based side-channel attacks. We discuss state-of-the-art side-channel attacks and defenses in both software and hardware. The second part consists of the peer-reviewed papers[1] accepted at renowned international security conferences.

[1]The content of the papers is unmodified from the camera-ready versions. The format of the included papers was modified to fit the layout of this thesis.

Acknowledgments

First, I want to thank Daniel Gruss, who gave me the opportunity to do a PhD. Thank you for all your support, the freedom you gave me with my research topics, and all the great discussions we had.

I want to especially thank Michael Schwarz for his never-ending support and for supervising both my bachelor's and master's thesis. You inspired me to do a PhD, and I really enjoyed working with you on all the fun projects. I wish you all the best for your research group and family.

Thank you, Mathy Vanhoef, for examining my thesis and providing valuable feedback. Moreover, thank you, Moritz Lipp and Claudio Canella, for valuable feedback on a draft of this thesis.

I want to thank Gerold Haynaly, an inspiring teacher who showed me the fun parts of IT security. A special thanks to Pietro Borrello for all the fantastic brainstorming sessions. It was awesome working with you, and I wish you all the best for your PhD.

Especially, I want to thank my colleagues and friends Moritz Lipp, Claudio Canella, Andreas Kogler, Catherine Easdon, Stefan Gast, and Jonas Juffinger. Thanks for all the inspiring discussions, debugging sessions, fun at conferences, and snack breaks. It was a pleasure working with you, I wish you all the best and hope we all stay in contact. Thank you, Roman Walch, for the pleasant morning chats and snack sessions. All the best for the finishing of your PhD thesis. Moreover, I want to thank my students Erik Kraft, Thomas Schuster and Hanna Müller for their motivation and fun projects we worked on together.

As music is an essential part of my life, I want to thank Alois Strohriegl and the people from MMK Grossklein for all the fun rehearsals, concerts, and jam sessions. Moreover, I want to thank my former piano teacher Anna Holler, for her dedication to music and all the fun jam sessions we had.

The most important part that helped me during my PhD were my family and friends. I want to thank my parents for supporting me during my school and studies. Especially, I want to thank my mum for all her love, patience, and support during my studies. A huge thanks to my siblings, their partners, and kids who kept me sane during my studies.

Thank you very much for all the fun parties, barbecues, and your never-ending support Maria, Walter, Dominik, Sarah, Andi, Sabine, Theodor, Benedikt, Thomas, Michi, Rudi, Babsi, Elisabeth, and Johannes. A big thanks to Lisa and Alex for your support and all the amazing barbecues, Buschenschank visits, and game nights. Thank you, Sabrina and Leo, on your excellent sense of humor, your happiness, and the fun we had together in the last years.

Finally, I want to thank my wonderful girlfriend Olivia. Although we met towards the end of my PhD you always supported me while writing this thesis. Without you, this thesis would not have been finished.

Table of Contents

Part I.

Remote Side-Channel Attacks and Defenses

1

Introduction

Software and hardware is highly optimized to fulfill the high performance demands of users and industry. These optimizations behave differently depending on user input and leave measurable side effects, reflected in the timing, power consumption or electromagnetic radiation of the device. Side-channel attacks derive the processed secret input from the measured side effects.

Already in 1992, Hu [102] described a cache attack that relies on the timing differences of accessing cached and uncached data. Kocher [132] discussed timing attacks on cryptographic algorithms in general by leveraging a hardware side channel, e.g., the cache. The attacks presented by Kocher [132] exploit timing differences in the RSA modular exponentiation to disclose private RSA keys. Later, cache attacks became more practical and were used to detect keystrokes or user interactions [151, 217, 257], break cryptographic primitives [24, 94–96, 107, 108, 144, 183, 185, 191, 246], to attack secure enclaves [33, 56, 79, 164, 215] and create hidden communication channels [94, 151, 160, 208, 279, 283]. More recently, with the discovery of Spectre [133] and Meltdown [153], a new research direction called transient-execution attacks emerged [44, 133, 153]. Meltdown leverages out-of-order execution and delayed exception handling to disclose sensitive data, e.g., kernel memory. Spectre [133] exploits speculative execution, following branch prediction, to disclose arbitrary data. Based on these initial findings, a wave of novel attacks was discovered in existing processors, disclosing sensitive data either based on Meltdown-type attacks [40, 44, 153, 199, 211, 212, 218, 248, 267] or Spectre-type attacks [44, 101, 131, 133, 135, 157]. Load Value Injection [249] showed that data can be transiently injected to mount attacks on secure enclaves. Different approaches were proposed to mitigate transient-execution attacks, such as new hardware and software mitigations [86, 111, 130, 281], process isolation [203], secret masking [46, 161], removing high-resolution timers [51, 52, 170], memory barriers [111] to stop speculation, and microcode patches [111]. However,

these mitigations introduce large overheads, focus only on a subset of attacks, or on a subset of scenarios [44, 47, 142, 143].

Besides optimizing the runtime performance, reducing the memory utilization is another challenge for both software and hardware developers. Memory deduplication is used in modern operating systems to decrease the memory utilization by detecting physical memory with the same content and merging duplicates. However, writing content to pages marked as copy-on-write leads to pagefaults [21, 32], which have a higher execution time. This timing side channel can be exploited to perform fingerprinting of operating systems and web browsers [84, 148, 184, 238], covert communication [273, 274], breaking ASLR and KASLR [21, 128], performing Rowhammer attacks in browsers [32, 54] or filesystems [186]. As an alternative to memory deduplication, memory compression was introduced to reduce the memory utilization in many applications such as web servers, file systems and operating systems. Kelsey [126] discovered the first memory-compression attack exploiting when plaintext is compressed and afterwards encrypted. In cases where both attacker-controlled content and secret content like browser cookies are compressed, the compression ratio forms an exploitable side-channel attack. The CRIME attack [206] was the first to exploit this compression-ratio side channel in web browsers. Based on their observation, further attacks on memory compression were discovered [25, 76, 124, 206, 251, 252].

Both optimizations on runtime and memory utilization introduced novel side channels. However, they were mainly exploited in scenarios with local code execution and in restricted environments such as browsers [63, 84, 182, 207, 211, 214, 218, 237]. Fully remote attacks have been shown on a small subset of side channels in a local network setup to attack cryptographic primitives [15, 120, 209, 295], breaking random-number generators [85], code diversification [227], transiently leak data [220], break the system's integrity [150, 241], or break post-quantum cryptography [264]. Therefore, for many existing side channels it remains unclear if they are exploitable in a remote setup. Moreover, to protect against side-channel attacks, different forms of isolation, such as language-level isolation, process isolation, and virtualization were introduced. However, it is unclear whether side-channel attacks can still be mounted within the same security domain or where user data and resources are still shared, e.g., for database applications. There is a large potential of undiscovered and practical remote attacks, which might become a serious threat in the future. In addition, practical

mitigations against existing side-channel attacks are required that meet the high performance demands of users and industry.

In this thesis, we analyze the practicality of side-channels attacks and defenses with a focus on real world applications. We discover novel side-channel attacks executed in restricted scenarios such as browsers, cloud environments, and fully remote scenarios where only a web interface is accessible for the attacker. Figure 1.1 shows all co-authored papers and their relations. The bold papers represent the papers included in this thesis. The x-axis shows whether a work was focusing on a novel attack or defense. In some cases, the papers included both attacks and defenses and are, therefore, placed in the center. The y-axis shows if the scope of the attack or defense is a local code-execution scenario or a remote setup, where the attacker has restricted access. Continuous arrows indicate a direct influence of one work to another. Dashed arrows indicate an indirect connection between the works.

In the direction of remote attacks, we examine state-of-the-art software-based side-channel attacks and mount them in remote scenarios. We analyze the address-translation attack presented by Gruss et al. [87] resolving virtual to physical addresses via the identity mapping in the Linux kernel. However, as we show in this thesis, this attribution was wrong as speculative execution caused the dereferencing of kernel addresses. For mitigations against memory-deduplication attacks, it is unclear whether attacks are possible within the same security domain. We show that, despite recent mitigations, remote memory-deduplication attacks are still possible, even across the internet. As attacks on memory compression mainly exploited the compression ratio, we investigate whether timing differences in compression algorithms are sufficient to reveal co-located sensitive data. Hence, we analyze timing leakage in common compression algorithms assisted by a fuzzer. The fuzzing approach optimizes the timing latency between correct and incorrect byte guesses to enable remote byte-by-byte leakage of sensitive data across the internet in server-based applications. Following a similar principle to automatically finding timing side channels, we improve the existing techniques to discover keystroke-related cache activity in shared libraries [90] by introducing a multi-layered approach. With this novel approach, we show that linker and compiler optimizations can introduce cache side channels in large software projects. We discover such a case in a recent version of the Chromium framework, which is widely used in the Chrome and Chromium browsers and Electron-based applications like Signal Desktop. In the direction of mitigating

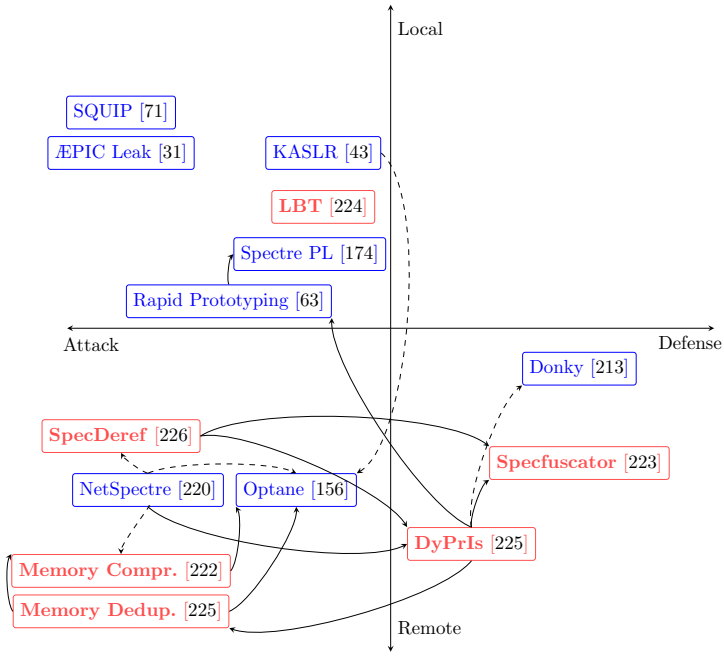


Figure 1.1.: Overview of all co-authored papers and their relations. Bold papers represents a paper included in the thesis. The x-axis indicates if a paper is in an attack or defense direction. The y-axis indicates if the scope an attack or defense was for a remote or local attacker. Papers that influenced other papers are connected via continuous arrows. Dashed arrows indicate an indirect influence of the papers.

side-channel attacks, we analyze existing mitigations against transient-execution attacks. Based on the insights gained from our own and existing attacks, we then develop novel mitigation techniques. Our research led to a novel Spectre mitigation which is integrated in the production system of Cloudflare Workers. This approach uses hardware-performance counters to detect Spectre attacks. Based on that detection, our dynamic approach isolates malicious tenants into separate processes. In contrast to previous mitigations against Spectre attacks, we verify the feasibility of branchless programs as a mitigation.

1.1. Main Contributions

We performed a root-cause analysis of the address-translation attack by Gruss et al. [87], enabling a translation of virtual to physical addresses, which was attributed to the x86 `prefetch` instruction. More specifically, we showed that this attribution is incorrect and the actual root cause of the leakage is speculative execution in the Linux kernel. Our analysis showed that userspace-controlled registers are dereferenced due to several Spectre-BTB gadgets in the syscall and interrupt handling. Based on the new insights, we mounted even stronger attacks than the address-translation attack including re-enabling the Foreshadow attack [248, 267], which enables leaking host-physical memory from a guest virtual machine. Moreover, we showed how the address-translation attack can be mounted from restricted environments such as the Firefox browser. The paper was published at Financial Crypto & Data Security 2021 [226] in collaboration with Thomas Schuster, Michael Schwarz, and Daniel Gruss.

To study potential novel remote attacks, we analyzed Cloudflare's edge-computing solution: Cloudflare Workers. Cloudflare Workers uses a JavaScript-based approach to intercept web requests and run small code snippets. The main performance advantage of their approach is a single-process design. However, a single-process JavaScript-sandboxed design running multiple tenants within the same process is susceptible to Spectre attacks, as Spectre can be performed in JavaScript [11, 43, 133, 207, 211, 237]. With a successful Spectre attack, the attacker can dump the entire virtual memory, including data of other tenants. As a hardening mechanism, access to local timers, execution time, and memory usage are restricted. However, amplification techniques and remote timers can be used to enable a remote Spectre attack on Cloudflare Workers leaking 120 bit/h. Using hardware performance counters, we developed, Dynamic Process Isolation, an efficient detection and process isolation mechanism against malicious scripts. The false-positive rate of our approach is only 0.61 % on the Cloudflare Workers. This work [221] was published at ESORICS 2022 and was joint work with Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz and Daniel Gruss.

We observed that all Spectre mitigations focus on introducing more branches and memory barriers to stop transient execution. However, none of the Spectre mitigations looked in the opposite direction of reducing the number of branches. Therefore, we proposed a new approach based on control-flow linearization and branch removal to mitigate Spectre

attacks. To achieve this, we used the *M/o/Vfuscator* [60] and optimized it in terms of binary size, compile time and run time. The overhead strongly varies for our test set of programs from, e.g., 5% up to an overhead of factor 1000. The paper [223] was published at Financial Crypto & Data Security 2021 in collaboration with Claudio Canella, Michael Schwarz, and Daniel Gruss.

Memory deduplication is used by cloud providers to reduce the memory footprint of applications by deduplicating memory with identical content. While several attacks were demonstrated in the past [21, 32, 84, 128, 148, 184, 238], mitigations prevent cross-security domain attacks. Afterwards, memory deduplication has been re-enabled on Windows and Ubuntu. Since attacks on the same security domain have not been demonstrated, we investigated this missing spot and analyzed common server applications and database systems for in-memory caching. We demonstrated multiple remote attacks in this scenario, which can remotely steal database records, fingerprint a system and break KASLR across the internet. The paper was published at NDSS 2022 [225] in collaboration with Erik Kraft, Moritz Lipp, and Daniel Gruss.

While working on memory-deduplication attacks, we were searching for similar side effects based on memory accesses leading to strong timing side channels. We discovered a novel timing side channel in several state-of-the-art lossless compression algorithms by exploiting sequence compression and specifically crafted memory layouts. To improve the exhaustive search for the crafted memory layouts, we developed an evolutionary side-channel fuzzer called Comprezzor to generate patterns that lead to byte-by-byte leakage. Based on the layouts found by Comprezzor, we demonstrated how timing latencies in compression algorithms can be exploited in databases to remotely steal database records. This work will appear at S&P 2023 [222] in collaboration with Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Daniel Gruss, and Michael Schwarz.

While looking for possible attack targets of page cache attacks [85], we developed a novel approach to speed up the runtime of templating large binaries. By dividing the templating into multiple layers, the runtime for scanning large software projects, e.g., browsers, can be improved by three orders of magnitude. Moreover, we discovered that even when the source code does not introduce cache side channels, compiler and linker optimizations can still introduce them. With Layered Binary Templating [224], we created a highly accurate cache-based keylogger for Chromium-based

applications. This work will appear at ACNS 2023 [224] in collaboration with Erik Kraft and Daniel Gruss.

1.2. Other Contributions

With NetSpectre, we analyze the preliminaries for Spectre to be exploited in a remote setup. We present two types of code snippets, so-called gadgets, required to perform NetSpectre. Based on the gadgets, we demonstrate a remote Spectre attack leaking data bitwise with 1.5 bit/hour. Moreover, we show that AVX instructions can be used instead of the cache to remotely leak data. This paper was published at ESORICS 2019 [220] in collaboration with Michael Schwarz, Moritz Lipp, Jon Masters, and Daniel Gruss.

While reverse-engineering the existing Meltdown mitigations on patched machines, we observed that loads from kernel addresses are only zeroed out. This observation led to a novel KASLR break, which works on Linux, SGX, Windows, and also in JavaScript. Based on the experience we got from developing Spectre PoCs in JavaScript, we mounted Meltdown in JavaScript on 32-bit Linux systems and evaluated it on vulnerable Intel CPUs. The paper was published at AsiaCCS 2020 [43] in collaboration with Claudio Canella, Michael Schwarz, Martin Haubenwallner, and Daniel Gruss.

As edge-computing providers partially rely on a single process design [52], there is the need for efficient in-process isolation to isolate tenants. Intel Memory Protection Keys (MPK) enable domain-based isolation for in-process architectures. Hence, we implemented an MPK-like mechanism in the RISC-V architecture. We additionally evaluated our design on the JavaScript engine V8 isolating the WASMEngine from the common V8-accessible memory. The paper was published at the USENIX Security Symposium 2020 [213] in collaboration with David Schrammel, Samuel Weiser, Stefan Steinegger, Michael Schwarz, and Daniel Gruss.

After working on several attack papers in the field of microarchitectural attacks, we concluded that a framework for fast prototyping is required. Therefore, we developed a novel framework enabling fast and efficient prototyping of proof-of-concept side channel attacks in native code and browsers. We demonstrated an LVI-NULL proof-of-concept and the first ZombieLoad proof-of-concept in an unmodified Firefox on Windows

leveraging memory deduplication. The paper was published at USENIX Security Symposium 2022 [63] in collaboration with Catherine Easdon, Michael Schwarz, and Daniel Gruss.

After working on the paper on fast prototyping of microarchitectural attacks [63], we looked in a similar direction to develop Spectre gadgets in different programming languages. We created a framework to mistrain and prototype Spectre-PHT gadgets in different languages and verify transient execution by checking the cache activity. In a systematic analysis, we verified that 26 out of 40 languages do not have any Spectre mitigations. We also demonstrated two case studies in Java and OCAML on cryptographic libraries, where we leaked key material by exploiting Spectre gadgets. This paper was published at ICISSP 2022 [174] in collaboration with Amir Naseredini, Stefan Gast, Pedro Miguel Sousa Bernardo, Amel Smajic, Claudio Canella, Martin Berger, Daniel Gruss.

Intel Optane [112] is a persistent memory that enables high-performance file accesses. Optane can be shared between multiple tenants in the cloud and is, therefore, a valuable target for side channels. Hence, we reverse-engineered the microarchitecture of Optane, such as caches, and the wear-leveling mechanism. We presented new side channels on persistent memory and demonstrated a local and a persistent remote covert channel and an inter-keystroke timing attack. The paper will appear at USENIX Security Symposium 2023 [156] in collaboration with Sihang Liu, Suraaj Kanniwadi, Andreas Kogler, Daniel Gruss, and Samira Khan.

While software vulnerabilities are well studied, we investigated whether software vulnerabilities are applicable to hardware. With ÆPIC Leak, we discovered that undefined regions in the APIC are improperly initialized. This enables data leakage of stale data by architecturally reading from the undefined regions. We demonstrated new attacks on Intel SGX leaking secret cryptographic keys from secure enclaves. ÆPIC Leak was published at USENIX 2022 [31] in collaboration with Pietro Borrello, Andreas Kogler, Moritz Lipp, Daniel Gruss, and Michael Schwarz.

Previous work [13] showed that exhaustion of execution ports of CPUs can be leveraged to attack cryptographic primitives. We investigated the separated scheduler queues per execution unit in AMD Zen processors and Apple's M1 processor and observed a similar resource exhaustion as in port contention. In the SQUIP attack, we demonstrate that contention in the scheduler queue can be exploited to create fast and stealthy covert channels and attack cryptographic primitives like the Square-and-Multiply

algorithm. SQUIP will appear at S&P 2023 [71] in collaboration with Stefan Gast, Jonas Juffinger, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl and Daniel Gruss.

1.3. Outline

In Chapter 2 we provide the necessary background on virtual memory, microarchitectural components, and software and hardware optimizations. Chapter 3 summarizes the state of the art of cache attacks, transient-execution attacks and defenses, attacks on memory optimizations, and remote timing attacks. Chapter 4 concludes the thesis and provides an outlook on future research directions.

2

Background

This chapter provides the necessary background for this thesis. Section 2.1 describes the fundamental concepts of virtual memory. In Section 2.2, we elaborate on computer organization and current optimizations. Section 2.3 describes optimizations to reduce the memory utilization in main memory and persistent storage devices. Section 2.4 defines side channels and provides a simple example of timing side channels.

2.1. Virtual Memory

Virtual memory was created with the motivation of simplifying memory management for programmers and to efficiently and safely share data from main memory between multiple processes. Every process has its own full *virtual address space* managed by the operating system, which cannot be accessed by other processes. A virtual address has a corresponding physical address. Therefore, the hardware has to resolve the virtual address to a physical address. To enable a fast translation of virtual to physical addresses, a concept called paging was introduced [98]. The physical memory is divided into smaller contiguous chunks of a fixed size (page frames). A typical design choice on the frame size (page size) on modern systems is 4 KiB [98]. For larger contiguous physical chunks, larger page sizes like 2 MiB and 1 GiB (huge pages) are also used, for instance, by the Linux kernel. A virtual address is associated with one or multiple pages of physical memory.

A page table stores multiple entries containing the mappings between virtual and physical addresses. A single page-table entry stores the page-frame number and additional metadata about the page frame, e.g., the present bit, indicating whether the memory is resident in memory or not. The page-frame number (PFN) is used to index the actual physical address

(PFN · page size + offset). Typically, paging is performed in multiple levels to reduce the memory utilization of storing the page tables [98]. For instance, in a 32-bit virtual address space with a page size of 4 KiB, and a page table entry size of 4 B, $2^{20} \cdot 4$ = 4MiB of memory are required for the page table alone. On a multi-process system, this is very expensive in terms of memory. Using 2-level paging, the 20 bit are divided into 10 bit each per page-table level. The first level is used to lookup an entry in the second level. There are 2^{10} second level page tables, whereas the entries of the second level point to the corresponding PFNs. With that split only $2^{10} \cdot 4$ = 4KiB of memory is required for the first level and another 4 KiB of memory for the corresponding second level page table.

Most currently in-use x86-64 CPUs use a 4-level paging structure with 48 bit (256 terabytes) of virtual address space. Figure 2.1 illustrates 4-level paging on x86-64 with 4KiB pages. A `CR3` register points to the base address of the first paging level Page Map Level 4 (PML4). The bit 39:47 of the virtual address index the entry of the PML4 table. The entry in the PML4 points to the Page Directory Pointer Table (PDPT). With the next 9 bit 38:30 of the virtual address, the entry in PDPT is selected, pointing to the Page Directory Table (PD). The bits 21:29 are used to select the PD entry. Using the next 9 bits of the virtual address 12:20, the entry in the Page Table (PT) is selected. With the page-table entry (PTE) containing the physical frame-number, the physical page can be determined. Translation caches [98], *i.e.*, translation lookaside buffers (TLB), speed up the lookups from virtual to physical addresses.

With the Ice Lake microarchitecture [141], another paging level was added, enabling 57-bit virtual address spaces (128 petabytes). Typically, the virtual memory space is divided by half for user space and kernel space. The Linux operating system has a full identity mapping (direct-physical map) of virtual addresses to physical addresses in the kernel space.

2.2. Computer Organization

In this section we discuss the most important components of modern CPUs. We introduce pipelining, caching, and branch prediction.

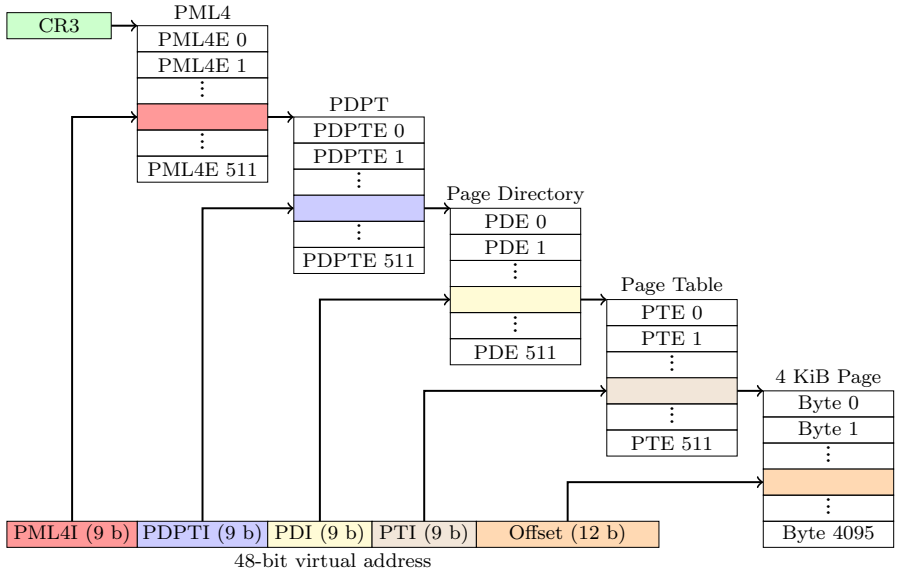


Figure 2.1.: Virtual to physical address-translation using 4-level paging on x86-64 [82, 149].

2.2.1. Instruction-Set Architecture

Similar to application programming interfaces (APIs), used in high-level applications between client and server applications, an abstraction layer for hardware and software is required. The Instruction-Set Architecture (ISA) is like a contract (interface) between software and hardware. The ISA not only defines the supported instructions of the CPU but also the available registers, the handling of exceptions and interrupts, or the memory model. Any program compiled for a certain ISA, i.e., x86, can run on processors that implement it. The microarchitecture defines how the ISA is implemented for a specific processor architecture [67]. This implementation includes several components like multi-stage pipelining, the memory subsystem, and branch prediction.

2.2.2. Pipelining

The main tasks of a simple Reduced Instruction Set (RISC) processor are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write-back (WB) [98]. Instruction pipelining increases

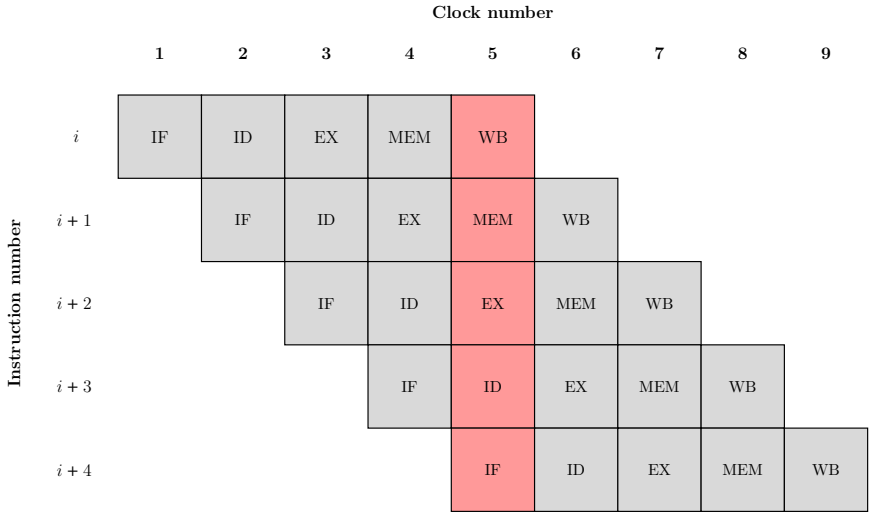


Figure 2.2.: Example of five stage RISC pipelining including the steps instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write-back (WB) [98]. On every new clock cycle, another instruction is fetched [98]. At clock cycle 5 (red), five instructions are at five different execution stages.

the performance of a processor by splitting and parallelizing the main tasks of a CPU into separate *stages*, similar to an assembly line in a factory [98]. In the IF phase, the next instruction is fetched. Afterwards, in the ID phase, the opcode of the instruction is decoded. Then the instruction is executed in the EX phase, e.g., an arithmetic operation is performed. If memory was accessed, it is handled in the MEM phase. The results of the instruction are written back and the registers updated accordingly in the WB phase. Figure 2.2 illustrates a simple five stage RISC pipeline. At the clock cycle 5, five instructions are at different pipeline stages. In modern processors, there are multiple and parallel pipelines that execute instructions in an out-of-order manner [98]. Pipelining suffers from the problem of hazards that might arise while handling different instructions. The different types of hazards in pipelined CPUs are data hazards, structural hazards, and control hazards. Hazards can lead to stalling, *i.e.*, waiting in the pipeline until a hazard is resolved.

Data hazards. *Data* hazards [98] take place if an instruction is data-dependent to a previous instruction result. One example is if an arithmetic instruction operates on the value of a register being changed by a pre-

vious arithmetic or logic instruction. Tomasulo's dynamic scheduling algorithm [244] resolves the three common data-dependency hazards Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). Instead of directly using the available registers, Tomasulo proposed to rename the registers to temporary registers. A common data bus is used to forward the results to the execution units. Between frontend and execution unit, there is a reservation station and a reorder buffer [244]. The reservation station buffers instructions and their operands. When operands become available in the reservation stations, the corresponding instruction is executed. WAR and WAW hazards are resolved with the introduction of register renaming, as there is no data dependency between the instructions as only the same register name is used. The reorder buffer keeps track of the order and state of the instructions. When an instruction is completed, the results are available to dependent instructions via the common data bus. Dynamic scheduling of instructions leads to out-of-order execution and, thus, a higher CPU performance. Note that instructions are committed in order to guarantee the program's correctness. However, a fault might occur, e.g., an invalid memory access, during the out-of-order execution, and then a pipeline stall occurs. Instructions that were executed during out-of-order execution but are not architecturally committed, e.g., due to a fault, are called *transient* instructions [44, 133].

Structural and Resource hazards. *Structural* and *Resource* hazards [98] can occur if two instructions are in the EX phase, but there is only one execution unit available to perform the operation. This can be considered already by the programmer and compiler to avoid scheduling instructions that need the same resource. Structural hazards can occur for special execution units such as the floating point unit. To overcome that issue, either multiple execution units of the same type can be introduced, which is, however, costly. Structural hazards can also arise in the IF and MEM phase. Separate caches were introduced on modern processors for instructions (IF) and data (MEM) to prevent such structural hazards [67, 98].

Control hazards. When the outcome of branches requires additional CPU cycles during execution, the pipeline stalls. To improve the runtime, branch prediction tries to predict the outcome of a branching instruction. *Control* hazards [98] occur due to mispredictions in the branch prediction and speculative execution, e.g., mispredicting a conditional branch. In case

of a misprediction, a pipeline stall occurs, the results of the speculation have to be discarded, and the correct branch has to be executed.

2.2.3. Speculative Execution

Branch instructions lead to pipeline stalls until the outcome of a branch is known at the execution phase. To overcome this performance bottleneck, speculative execution was introduced. Based on the predictions of one or multiple predictors, the predicted branch will be executed speculatively. If the prediction was correct, the results are committed in the reorder buffer. Conversely, if the prediction was incorrect, the results of the prediction are discarded. For conditional branches, the predictor needs to distinguish between taken and not taken states. Branches can also be predicted in a static and dynamic way based on the direction of the jump (forward and backward) and whether the branch was recently taken or not taken [98]. Similar to out-of-order execution, transient instructions might be executed if the prediction was incorrect [44, 133]. In Chapter 3, we elaborate how transient instructions can be exploited.

Static Prediction.

The assumption for static prediction is to assume that branches are always taken or not taken [231]. This strategy, however, assumes that programmers and compilers always write the more likely code in the taken branch part [231]. Another static approach was discussed to only take backward branches and to no take forward branches. This prediction strategy works well for loops, as the jump instruction is typically emitted at the end of the branch. Static prediction is not applicable for indirect branches as the outcome of branches will be computed at runtime.

Dynamic Prediction.

A more sophisticated strategy is to base the outcome of the prediction based on the history of taken and not taken branches at runtime [284]. In modern processor architectures, multiple components are used to store the history of branches. Every type of branch instruction has a corresponding prediction mechanism [98].

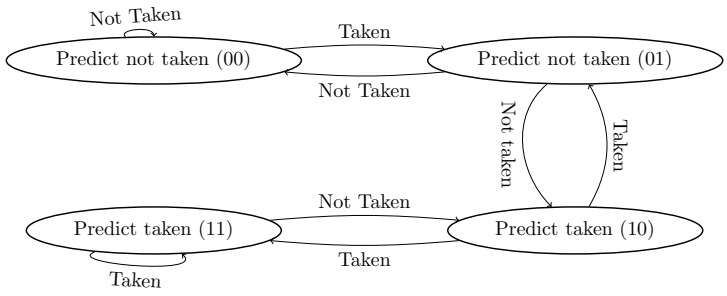


Figure 2.3.: State machine of a 2-bit predictor using a 2-bit counter [98]. If the counter is greater than 1, the branch will be predicted as taken.

To predict conditional branches, the predictor has to decide whether the branch will be taken or not and what the target address will be. Yeh [284] proposed a two-level prediction scheme for conditional branches consisting of a Branch History Register (BHR) and Pattern History Table (PHT). The first prediction level stores the history of the last k branches in the BHR. The value in the BHR is then used as an index for the PHT, which has 2^k entries. Each entry in the PHT [284] contains a table of 2-bit predictors e.g., for a 2 bit history it has four 2-bit counters. A 2-bit predictor predictor [284], is a 2-bit counter which increases if the branch was taken and decreases it if was not taken. Figure 2.3 illustrates the state machine of such a 2-bit counter. Using a 2-bit predictor, the branch will be taken if the most-significant bit is set. A branch will be predicted as taken if the 2-bit counter has a value greater equals 2. Predictors with more than 2 bit did not increase the prediction accuracy, as was shown by Yeh et al. [284]. The prediction outcome can also be globally correlated [67] for all conditional branches, e.g., if the last n branches were taken, then the next branch will also be taken.

A Branch Target Buffer (BTB) [67, 193] stores a mapping between the branch address, the target address, and whether the branch should be taken or not. Branch prediction can be performed locally for each individual branch. In addition to the target address of branches, the target return address can also be predicted. The Return Stack Buffer (RSB) was introduced to predict the target address of `ret` instructions. Over the time, many different branch prediction schemes evolved, such as bi-mode branch prediction with 2 PHTs, index-sharing predictors, variable path length predictors, and perceptron-based predictors [98, 121, 145, 235].

2.2.4. Caching

A cache is a small and fast memory buffer that is located between the CPU and the main memory. Caches reduce the latency for memory accesses by buffering recently accessed code and data. If data is resident in one of the caches it is called a *cache hit*. Conversely, if data is not resident in the caches and has to be loaded from main memory, it is called a *cache miss*. The smallest possible unit stored within a cache is called a *cache line*. Typically, a cache line is 64 B [98]. Additional metadata is stored together with the cache line, e.g., valid bit, and a tag. The *tag* is a unique identifier for a cache line. The *index* selects the cache line, and the *offset* is used to determine the exact location within the cache line. Both, tag and index, can be chosen based on parts of the virtual or physical address. A common choice for the L1 is to choose a virtually-indexed, and physically-tagged design, where the L1 lookup is performed directly, and the TLB lookup is performed concurrently [67].

Cache Organization. Caches can be organized in various ways. The simplest technique is to *directly map* each address to a corresponding cache line. This concept has the problem that only a single block of data can be stored at a certain location as addresses are congruent for different processes. In a *fully-associative* cache, any memory location can be mapped to any cache location. While there is no need to use an index to select a cache line, every tag has to be compared to locate a cache line.

A *set-associative* cache is a compromise between these two extremes. It has multiple *ways* per index, forming a cache set [110]. Therefore, data can be located in any of the ways of a certain set. On a cache lookup, the tag has to be compared in every way, *i.e.*, for an *n-way* associative cache, n tag comparisons have to be performed to find the matching cache line. AMD introduced a way predictor [16, 109], predicting which way will be selected on a cache lookup. This technique reduces the energy consumption as only the predicted way is first activated for the comparison.

Cache Hierarchy. Caches are typically organized in multiple levels [67, 100]. A modern CPU consists of multiple physical cores. A physical core can include multiple logical cores, e.g., hyperthreads on Intel CPUs. This principle is called simultaneous multithreading (SMT) [67], where one CPU core can run multiple threads concurrently. Each core contains a

private L1 and L2 cache and a shared last-level cache (LLC) cache between the cores. L1 caches are divided into separate instruction and data caches. A typical size for L1 caches on x86 CPUs with 4 KiB pages is between 16 and 64 KiB [67]. The Apple M1 CPU uses 16 KiB pages and hence, also has the size of 128 KiB for the L1 cache of energy-efficient cores. L2 caches contain instructions and data and are typically multiple hundred kilobytes large, e.g., 256 KiB on an Intel i5-8250U [268]. The LLC is the slowest and largest cache with a size up to a few megabytes [268]. On Intel CPUs, slice functions are used to split the LLC into simpler and smaller separate caches per CPU Core [159].

Inclusiveness As modern CPUs have a multi-core design and preemptive multithreading, the caches need to keep data coherent. The *inclusiveness* decides whether a hierarchically higher cache level, e.g., L2 is higher than L1, stores all data resident in the lower cache level. Inclusiveness speeds up cache coherency, as only the higher level has to be looked up. A major disadvantage is that data is redundantly stored. A cache is called *inclusive*, if the data is fully present in the lower cache level. In an *exclusive* cache, data is only resident in a single cache level and not redundantly stored. *Non-inclusive* caches can hold data from higher cache levels, but do not guarantee that all of the data is present. Therefore, data could be stored multiple times or in a single level in a non-inclusive cache. On recent Intel CPUs [98, 110] the L2 is non-inclusive to the L1 and the LLC is inclusive to L1 and L2.

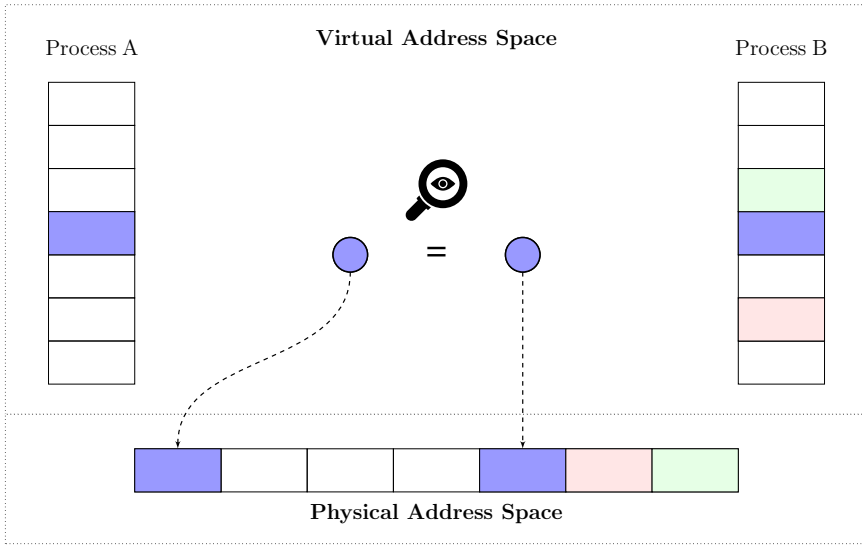
Replacement Policies As caches are limited in their size, the replacement policy decides in which way of the set to place the data. A simple heuristic is to replace cache lines randomly. However, this ignores the principle of locality [98], since data recently accessed and data not recently accessed should be treated differently. Another heuristic would be to replace data that was *least-recently* used (LRU). As the age has to be tracked, performing an actual LRU policy is too costly. Instead, an approximation of LRU is implemented in hardware (Pseudo-LRU) [98]. Abel et al. [1] and Vila et al. [253] reverse-engineered the replacement policies of different cache levels of Intel CPUs.

2.3. Memory Usage Optimizations

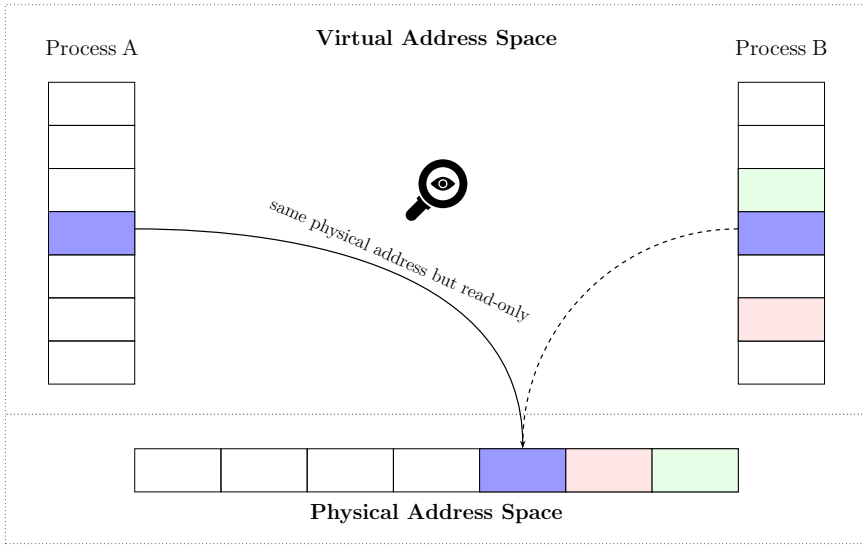
In this section, we will explore two fundamental techniques to reduce the memory utilization in both main memory and disk storage: memory deduplication and memory compression.

2.3.1. Memory Deduplication

Depending on the tasks and workload of a machine, identical memory is mapped multiple times in the main memory. Operating systems actively save memory by loading shared libraries only once into main memory and creating a virtual mapping for each process. With memory deduplication, an additional optimization was introduced in operating systems to reduce the memory footprint. Figure 2.4 illustrates memory deduplication between two processes. A kernel thread actively scans for identical memory pages and deduplicates them by modifying the page-table entries and mapping the merged page read-only (following the copy-on-write semantics). When a process attempts to modify a merged page, the content is first copied to a new memory location and then modified.



(a) Step 1: Find identical pages



(b) Step 2: The content is deduplicated, by modifying the page-table entry to point to the same physical memory location and mapping the page as read-only.

Figure 2.4.: Overview of memory deduplication. Identical pages are deduplicated and mapping with copy-on-write semantics.

2.3.2. Memory Compression

Memory compression is a technique to reduce memory utilization by modifying its representation so that the size decreases. There are two main techniques to compress memory. The first technique is *lossless* compression which ensures full reversibility for data. *Lossy* compression allows information loss, which enables higher compression rates. In this thesis, we focus on lossless compression algorithms.

Deflate

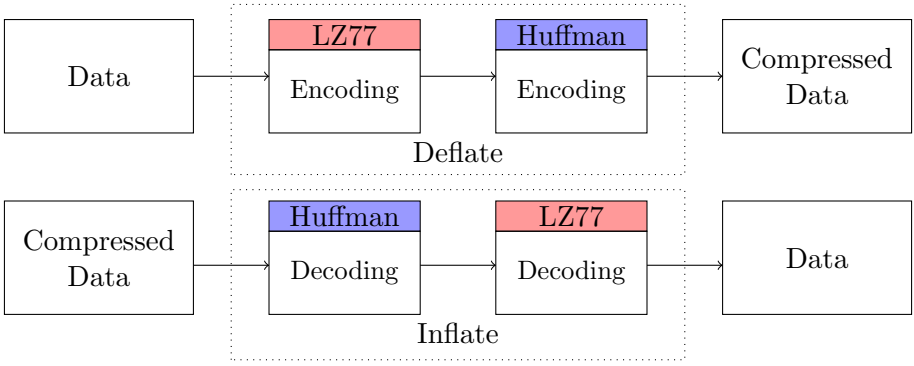


Figure 2.5.: ZLib compression using LZ77 and Huffmann encoding.

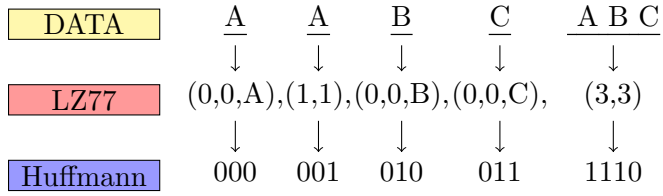


Figure 2.6.: Example for sequence compression using LZ77 and Huffmann encoding.

A commonly used lossless compression algorithm in gzip/zlib is *Deflate* [49, 58]. Deflate compresses data into a smaller lossless representation. Deflate

consists of two main components, the Lempel-Ziv77 [58] (LZ77) encoding and Huffmann encoding. Huffman codes are optimal and prefix-free codes that encode the most frequent symbols with the smallest code. LZ77 works by encoding sequences into a pair containing both length and distance to the first occurrence, *i.e.*, length-distance pair.

Figure 2.5 illustrates Deflate/Inflate in the zlib library. When compressing data with Deflate, the raw data is first encoded using LZ77, and then the LZ77 encoded data is encoded per symbol using Huffman encoding. By default, zlib uses a sequence length of 3 characters. A look-ahead and search buffer are used to find the longest match within the sliding window [49].

In the Inflate algorithm, the steps are performed in reverse order by first Huffmann decoding the compressed data and then decoding the LZ77 encoded symbols back to raw data. Figure 2.6 illustrates how a simple sequence can be compressed using LZ77 and Huffman encoding. First, the duplicate sequences are encoded as length-distance pairs. Then the data is encoded into small binary codes using Huffman codes. Some algorithms only rely on sequence encoding and skip the symbol encoding part, e.g., Snappy [78].

2.4. Side-Channel Attacks

A *side channel* is an information channel that transmits metadata correlated to the execution of a program under a given input to an adversary. Metadata can be any form of side effect during the computation of a program, e.g., timing [132], power consumption [134], electromagnetic radiation [12] or thermal information [106, 129]. Note that the input is not directly accessible to an adversary. *Side-channel attacks* use the metadata to derive information about the actual data being processed, e.g., cryptographic keys. Formalized definitions of side-channel attacks can be found in the works of Molnar et al. [167], Yuan et al. [287] and Gruss [83].

Timing Attacks. One form of side-channel attacks are timing attacks. Listing 2.4.1 provides a code snippet vulnerable to a timing attack. As the PIN code is not directly accessible for an attacker, the attacker would need to guess the correct PIN. In the worst case of this example, an attacker

would have to try 10 000 different PINs to get the correct PIN. The `strcmp` function used in the libc introduces a timing side channel by optimizing the functions runtime. If the first compared digit is wrong, the loop will be terminated. Conversely, if the the first digit is correct, the next byte will be compared. This optimization introduces a timing difference for correct and incorrect guesses, which can be exploited by the attacker to recover the correct digits successfully. Using this timing side channel, only 40 guesses are required in the worst case for the attacker. Another real-world example of an algorithm susceptible to timing attacks is the square-and-multiply algorithm used for modular exponentiation in RSA [132]. In this algorithm, a branch performs different arithmetic operations for the key stream. For a '1'-bit, a square and a multiply operation is performed, and for a '0' bit, only a square operation is performed. Consequently, the '1'-bit has higher latencies than a '0' or consume more energy.

Covert channels. In a covert channel, two parties want to establish communication over a medium that is not intended to be used for communication. The parties are not permitted to directly communicate over a channel. Both parties agree on a medium, e.g., the cache [265], to communicate and and a transmission protocol to establish communication. As a protocol the parties could agree to transmit data in binary form and in a certain time frame per bit. In contrast to a side channel, the two parties have full control of the sending and receiving side. Moreover, covert channels are often used to determine the true capacity of a side channel. The true channel capacity for a binary symmetric channel is defined as [242]:

$$Cap = RC * (1 + (p_{err} * log_2(p_{err}) + (1 - p_{err}) * log_2(1 - p_{err})))$$

RC is the raw channel capacity, *i.e.*, the amount of information being rawly transmitted and p_{err} is the probability of wrongly transmitted bits.

Scope of a side-channel attack. Side-channel attacks can be performed by physically collecting metadata or purely via software interfaces, e.g., by observing the file size of an encrypted file. In this thesis, we consider a side-channel attack to be **remote**, if the attacker has no local software or hardware access, and any attacker-driven activities are induced over the network.

```
int strcmp(const char *p1, const char *p2)
{
  const unsigned char *s1 = (const unsigned char*) p1;
  const unsigned char *s2 = (const unsigned char*) p2;
  unsigned char c1, c2;
  do
  {
    c1 = (unsigned char) *s1++;
    c2 = (unsigned char) *s2++;
    if (c1 == '\0')
      return c1 - c2;
  }
  while (c1 == c2);
  return c1 - c2;
}

void check_pin_code(char* user_input)
{
    char* pin = "4321";
    if(!strcmp(pin,user_input)) {
        printf("Correct");
    } else {
        printf("Incorrect");
    }
}
```

Listing 2.4.1.: Example for a simple timing channel caused by the strcmp function used in the Glibc [74].

Side channels are not only introduced by algorithms but also by compilers and the underlying hardware, *i.e.*, the microarchitecture. Side channels based on microarchitectural side effects are referred to as *microarchitectural* side channels. Optimizations often introduce side channels in the underlying microarchitecture, such as caches, branch prediction, and out-of-order execution. We provide more details on microarchitectural side-channel attacks and timing attacks in Chapter 3.

3

State of the Art

Microarchitectural attacks and defenses have been well studied over the last decades [9, 27, 72, 82, 83, 149, 214, 233, 239, 277]. In this chapter, we discuss the state-of-the-art on cache attacks, transient-execution attacks and defenses in Section 3.1. We discuss attacks and defenses on memory deduplication and compression in Section 3.3. Finally, Section 3.4 provides an overview of existing remote timing attacks.

3.1. Cache Attacks

Cache attacks exploit the timing difference between memory accessed in the cache compared to memory accessed from DRAM. The attacker measures the latency of memory accesses to derive whether data has been accessed by a victim application recently. To measure the subtle timing difference between cached and uncached data, an attacker requires a high-resolution *timer*. With a high-resolution timer, e.g., on x86, the `rdtsc` instruction, the attacker can measure the access times of the different cache levels and the DRAM. Schwarz et al. [219] created a high-resolution timer by leveraging a timing thread.

Hu [102] mentioned that cache covert channels can be used to extract sensitive information. Kocher [132] described timing differences in different microarchitectural components, including caches and branch prediction. Based on that assumption, Kocher [132] presents an attack on RSA that exploits timing differences in modular exponentiation caused by performing only a single multiplication or a squaring operation depending on the key bits of the private key. Kelsey et al. [127] discussed potential cache side-channel attacks on cryptographic algorithms. Tsunoo et al. [246] demonstrated cache attacks on DES.

3.1.1. Cache Eviction.

In a noisy real-world environment, repeated measurements are required for cache attacks. Therefore, to successfully perform cache attacks, an attacker requires a primitive to invalidate target cache lines efficiently. On x86 and ARM, there are special instructions for directly evicting a cache line, e.g., `clflush` on x86. However, such instructions are only helpful for cache eviction if attacker and victim share the same physical addresses, e.g., a shared library.

A naive and costly alternative would be to occupy the entire LLC and evict the entire data. Schwarz et al. [220] showed that remote attackers can evict the LLC by thrashing the cache via file downloads. Streamline [208] uses self eviction using a large shared array as an alternative to the flush instruction. A more efficient approach is to evict the target data by using multiple congruent addresses pointing to the same cache set. An *eviction set* is a set of congruent addresses, that when accessed in specific order, evicts a target address from the cache. Finding eviction sets can be challenging as there are complex addressing functions used in modern processors [159]. Moreover, knowledge of physical addresses can be required for the attacker, as different cache levels may use physical addresses for set indexing. Static and dynamic approaches can be used to create efficient and minimal eviction sets [34, 88, 115, 151, 159, 183, 232, 254].

3.1.2. Evict+Time

Bernstein [24] attacked AES with the idea of evicting data from the shared L1 cache and then timing the execution time of the encryption. Based on the time, the attacker learns which data was accessed by the target application. Concurrently, Percival [191] showed that cache misses can be used to recover parts of an RSA secret key. Osvik et al. [183] generalized the idea of cache attacks on cryptographic primitives and called this technique Evict+Time.

First, the attacker starts measuring the runtime and triggers an encryption using the targeted cryptographic primitive, e.g., AES. Then, the attacker evicts a certain cache set by accessing an eviction set. Finally, the attacker computes the overall runtime of the execution. If the attacker observes a slower encryption time, the data has been reaccessed and had to be

fetched from main memory. Evict+Time was used to steal cryptographic key material [6, 80, 119, 137, 144, 162, 183, 234] and breaking ASLR [80, 103].

3.1.3. Prime+Probe

Osvik et al. [183] further described another cache attack called Prime+Probe. In Prime+Probe, the attacker tries to examine the cache state after the encryption. Prime+Probe requires the attacker to create an eviction set.

In the prime step, the attacker accesses the eviction set, to fully occupy the target cache sets. Then, the attacker triggers an encryption for a known plaintext. The encryption evicts data that was cached. Finally, the attacker probes the access time over all elements in the eviction set. The attacker observes whether the encryption replaced any part of the eviction set and can infer information about the secret encryption key.

Prime+Probe was first used to attack the L1D and L1I caches [2–5, 8, 29, 35, 175, 183, 191, 245, 291]. By reverse-engineering the complex mapping functions, attacks on the LLC became possible [72, 73, 96, 107, 108, 115, 125, 155, 159, 202, 258]. Besides attacking cryptographic primitives in software and hardware, Prime+Probe was also used to create covert channels and to detect co-location in the cloud [205, 286], spy on user behavior in the browser [182, 217, 230], attack or perform attacks from secure enclaves [33, 56, 79, 139, 164, 215], and GPUs [61, 62].

3.1.4. Flush+Reload

The flush instructions provided by the ISA can be used to directly evict data from the cache. Gullasch [94] developed a first attack based on the x86 `clflush` instruction. Yarom et al. [283] generalized such flush-based attacks to Flush+Reload, where the attacker uses the `clflush` instruction to evict cache lines from shared memory, e.g., shared libraries, to observe cache activity from other applications.

The idea of Flush+Reload is as follows. First, the attacker *flushes* the shared target cache line out of the cache. Then, a victim application runs and causes cache activity. Finally, by *reloading* the shared cache line, the attacker can distinguish if the shared cache line was accessed or

not. One major requirement for Flush+Reload is, that there is shared memory between attacker and victim. The shared mapping enables the attacker to directly flush the victim's data as shared memory is shared in the LLC. Flush+Reload is a very accurate and reliable cache attack [283] that was used in various works to attack cryptographic primitives, spy on keystrokes and create covert communication [14, 15, 23, 39, 70, 81, 90, 94, 95, 107, 116–118, 192, 195, 282, 283, 289, 290].

3.1.5. Further Cache Attacks

The flush instruction can also be replaced in Flush+Reload with eviction in restricted environments, which results in an Evict+Reload attack [90, 151]. In Flush+Flush [89] only the `clflush` is used as a cache side channel. The `clflush` instruction takes more time for cached data that was accessed by the victim than for non-cached data. The cache replacement policy was also exploited to mount powerful attacks on cryptographic primitives and in the browser [34, 237, 276]. Purnal et al. [197] presented Prime+Scope improving the time resolution of Prime+Probe. Disselkoen et al. [59] showed that the Probe step in Prime+Probe can be replaced with an asynchronous ABORT in Intel's TSX. Lipp et al. [152] demonstrated that the mispredictions in the L1 way predictor can also be exploited. Similar to hardware caches, software caches, e.g., the page cache [85] can be exploited based on whether data is resident in the cache or not.

3.2. Transient-Execution Attacks and Defenses

Speculative execution and branch prediction are another microarchitectural component that is susceptible to side-channel attacks. Attackers exploited branch prediction to spy on cryptographic primitives [7, 10, 38], perform ASLR breaks [64], recover the control-flow of an Intel SGX enclave [146], and to leak secrets within SGX enclaves [65, 104].

Transient instructions are executed during out-of-order and speculative execution. These instructions are never architecturally committed but can influence the microarchitectural state, e.g., the cache. In 2018, Kocher et al. [133] discovered a powerful attack named Spectre, which leaks data by exploiting speculative execution. Concurrently, the Meltdown attack [153] was published, which exploits delayed exception handling during out-of-order execution. Meltdown enables an attacker to access an arbitrary

memory location, including data from kernel space, during transient execution. Meltdown and Spectre formed the new research direction of *transient-execution attacks* [44, 133].

Canella et al. [44] introduced a systematization of transient-execution attacks. They classified the attacks based on their root causes. One class is Meltdown-type attacks, where delayed exception handling during out-of-order execution leads to transient execution. The other class is Spectre-type attacks, exploiting speculative execution. The time window, *i.e.*, a number of instructions executed transiently, is referred to as *transient window* and bound by the size of the reorder buffer [44].

3.2.1. Spectre-type Attacks

Analogous to Return-Oriented-Programming (ROP) [228], small code snippets, *i.e.*, *gadgets*, are used to describe situations where a certain branch prediction component can be transiently exploited.

Spectre-PHT. Listing 3.2.1 illustrates a Spectre-PHT gadget [44, 133]. In this example, the attacker controls the `index` variable and can trigger a certain branch repeatedly. The attacker can mistrain the PHT by sending multiple *in-bounds* indices until the branch prediction predicts the branch to be always taken. On the subsequent access, the attacker uses an `out-of-bounds` index to access the secret data. The second load encodes the leaked byte of `data` into the cache by accessing the lookup table at a certain offset. In this example, the `lookup_table` is large enough to cover 256 different cache lines. While the branch will not be taken architecturally, the transient execution still performs the cache access.

Using a cache attack like Flush+Reload, the attacker can now recover the secret. The mistraining of the target branches can occur within the same address space or across address spaces [44, 133]. Moreover, due to the fact that only parts of the addresses are used for the branch prediction, there are congruent addresses, which can be used to mistrain the branch prediction as well [44, 133]. Göktas et al. [77] demonstrated how speculative control-flow hijacking in the Linux kernel can be used to break KASLR and to perform a privilege escalation.

```
1     char* secret = "This is a SECRET.";
2     unsigned char* data = "data";
3     if (index < strlen(data))
4     {
5         return lookup_table[data[index] << 12];
6     }
```

Listing 3.2.1.: Example for a Spectre-PHT gadget [44, 133].

Spectre-BTB. Spectre-BTB targets the BTB, which is to predict outcome of indirect call or jump instructions. The branch instruction jumps to a dereference of a general-purpose register, e.g., `jmp* rax`. Typical susceptible gadgets for Spectre-BTB are function pointers, e.g., jump tables and vtables in C++. Similar to ROP [133] the attacker tries to re-uses small code chunks that will be transiently executed to leak data.

In a typical Spectre-BTB scenario, the attacker can actively call different targets of a jump table and controls the parameter of a register. The register is dereferenced, and the value is encoded into the cache. Again, the attacker can use out-of-bounds values and a cache attack to leak arbitrary data within the process. Horn [101] demonstrated that eBPF can be leveraged to create indirect branch instructions in the kernel to leak arbitrary memory. Spectre attacks have also been demonstrated in browsers [11, 133, 157, 161, 176, 221, 237, 269] and SGX [178]. In Chapter 5, we show how a Spectre-BTB gadget in the Linux kernel can be used to revive the Foreshadow attack. Barberis et al. [20] demonstrated that the BHB can be used in combination with a crafted Spectre-BTB gadget in the Linux kernel to leak data across security domains and even break state-of-the-art mitigations.

Spectre-RSB. The RSB can also be exploited by mistraining instructions related to function calls, e.g., `call`, `ret`, and `pop` [157]. The x86-64 architecture uses the stack to store the return address and old base pointer, when calling a function. If the attacker has control over the stack, the attacker can always mistrain the RSB as follows.

Listing 3.2.2 illustrates how manipulating the return address on the stack leads to speculative execution caused by a misprediction in the RSB. First, there is a call to `manipulate` (Line 1) which pushes the address of `speculation` to the RSB. However, `manipulate` overwrites the return

```
1     call   0x40137 <manipulate> ; push target to RSB
2   speculation:
3     mov ecx, 0x12341234 ; runs in speculation
4     mov rax, QWORD PTR [rcx] ; read from arbitrary address
5     jmp end
6   manipulate:  ; 0x40137
7     lea eax, ds:0x40114f ; load address of end
8     mov DWORD PTR [esp],eax ; manipulate stack pointer
9     ret
10  end: ; 0x40114f
11    ret
12
```

Listing 3.2.2.: Example of Spectre-RSB (specpoline) [44, 135]. The `manipulate` block actively manipulates the return address and causes transient-execution of the `speculation` block.

address located on the stack and returns. The return target will be speculatively executed, leading to transient-execution at Line 3. Since the return address was modified, the results of the speculative execution will be discarded and the end branch will be executed. This primitive can be used to create a trampoline into speculation, *i.e.*, *specpoline*.

The RSB can underflow, thus, there is a fallback to the BTB [133, 135]. Wikner et al. [269] reverse-engineered the RSB on current Intel CPUs and demonstrated Spectre-RSB on Firefox. Retbleed [270] exploits Spectre-RSB on AMD CPUs and showed that active mitigations in the Linux can be circumvented.

Spectre-STL. Store-to-Load (STL) forwarding is a hardware optimization, which speculatively forwards previous stores to a load. The memory disambiguator predicts whether loads can be speculatively executed [44, 101]. However, the CPU only uses the lower 12 bit for this optimization. Horn [44, 101] showed that the memory disambiguation can exploited to speculatively bypass store instructions. Again, using a side channel, the transiently loaded data could be leaked. Listing 3.2.3 illustrates how a store operation can be bypassed and arbitrary memory speculatively accessed.

```
/* Based on
https://github.com/IAIK/transientfail/blob/master/pocs/
spectre/STL/main.c
*/
uint8_t* data = target_area;
uint8_t** data_slowptr = &data;
uint8_t*** data_slowslowptr = &data_slowptr;

// trigger memory disambiguation with store
// but the store will be circumvented
(*(*data_slowslowptr))[index] = 0;

/* Leak data at target index */
lookup_table[target_area[index] << 4096];

```

Listing 3.2.3.: Example of Spectre-STL using pointer chaining [44, 45, 101].

Further Spectre Attacks. Speculative execution of direct jumps, *i.e.*, straight-line speculation, was also exploited [18, 189]. We mounted Spectre attacks to different programming languages like Java and Go [174]. Instead of the cache, other side channels can be used to transiently encode data into a side channel transiently. In NetSpectre [220], we used vector instructions (AVX) to transiently leak data. Bhattacharyya et al. [26] and Fustos and Yun [68] used port contention to encode secret data. Weber et al. [266] found a novel side channel in the timing of the floating-point unit. Lipp et al. [152] demonstrated a Spectre attack encoding data through timing differences caused by collisions in the AMD way predictor. Ren et al. [204] used a side channel in the cache for micro-operations to perform Spectre attacks. Xu et al. [278] exploited a timing side channel in the frontend bus of Intel processors and demonstrated a Spectre attack.

Finding Spectre Gadgets. Finding Spectre gadgets is a challenging task as compilers can generate different instructions depending on the optimization levels and gadgets can be versatile [44, 122]. There are two main challenges in finding Spectre gadgets. First, the form of Spectre gadgets must be determined. Second, once a gadget that fits the form has been found, it must be determined whether it is exploitable in an attack. Kocher [133] provided 15 main examples how Spectre-PHT gadgets can look like. Static tools were developed to find Spectre gadget code

patterns [53, 201]. However, static approaches suffer from a large number of false positives.

Spectector [92] leverages symbolic execution to find Spectre-PHT gadgets in binaries. This approach formally proofs that there are no Spectre gadgets inside the binary. For this purpose, Spectector leverages symbolic execution and tracking of memory accesses and jump targets. Several further approaches are based on formal methods, symbolic execution, and taint tracking [28, 48, 57, 91–93, 216, 259, 260]. The problem with such approaches is that for large software projects, the symbolic execution suffers from path explosion and does not scale. Fuzzing-based approaches [105, 122, 180, 198] improved the runtime of gadget detection significantly. FastSpec [243] encodes assembly snippets with Google's BERT framework and trains a GAN with mutated Spectre gadgets. The GAN generated about 1 million different gadgets [243]. Various works used hardware performance counters to actively detect side-channel attacks [50, 89, 97, 99, 114, 158, 172, 190, 261, 262, 288, 292].

Spectre Mitigations. Researchers and vendors proposed to fix Spectre at the software, kernel, firmware, and hardware level [44]. The most expensive mitigation is to disable speculative execution [133]. For conditional branches, Intel proposed to use memory barriers, e.g., `lfence` instructions. Moreover, Intel offered interfaces to clear predictor states, e.g., clear the BTB [113]. *Retpoline* [247] uses a similar code sequence as in Listing 3.2.2 but with the addition of a speculation barrier. The compiler community proposed pointer masking and automated insertion of memory barriers to mitigate Spectre [173, 177, 179, 187, 229, 293]. Several works developed hardware-software co-designs to mitigate Spectre attacks [69, 136, 147, 216, 240, 256, 294]. Specfuscator (cf. Chapter 7) mitigates Spectre attacks by linearizing the control flow. Similarly, Borrello et al. [30] used control-flow linearization to protect cryptographic libraries. However, such an approach also mitigates Spectre attacks if the control flow is linearized. Browser vendors implemented process isolation [203] to efficiently mitigate in-process Spectre attacks. A more in-depth systematization of Spectre defenses were performed by various studies [41, 44, 47, 277].

```
1 // handle null-pointer exception
2 char data = *( char *) 0xffffffff8ca000fe;
3 *(volatile char*) NULL;
4 lookup_table[data * 4096] = 0;
```

Listing 3.2.4.: Meltdown [44, 153] example. A kernel address is dereferenced and the result encoded into the cache, which can be recovered with Flush+Reload.

3.2.2. Meltdown-type Attacks

Meltdown-type attacks exploit transient execution caused by delayed exception handling during out-of-order execution [153]. During transient execution the accessed data is loaded from the L1 and can be leaked via a side channel [113].

The first Meltdown variant, *Meltdown-US* [44], successfully leaks userspace-inaccessible kernel data. Listing 3.2.4 illustrates how Meltdown-US leaks arbitrary kernel memory. First, the attacker dereferences a null-pointer to create a large transient window. Then, the attacker actively loads data a kernel address and encodes it into the cache. Again, using a cache attack like Flush+Reload, the attacker can recover the leaked value. Another important Meltdown-type attack was Meltdown-P (cleared present bit), also known as Foreshadow. Foreshadow enables attacks on SGX and also leakage of hypervisor data [248, 267]. Meltdown-type effects have been found at different microarchitectural locations, e.g., FPU registers and the read-only bit [44, 131, 236]. Mainly Intel CPUs were susceptible to most Meltdown-type attacks. However, AMD, IBM, ARM and Samsung CPUs were also susceptible to Meltdown-type attacks caused by delayed exception handling [44, 171]. Based on the Meltdown effect, microarchitectural data-sampling attacks, have been found on various contexts such as browsers, virtualization software, and secure enclaves leaking data from various buffers and load ports [40, 44, 166, 200, 207, 210–212, 218, 275]. LVI turned the Meltdown effect around by transiently injecting values into a victim to achieve transient data-flow hijacking [249]. Similar to Spectre, LVI requires certain gadgets to perform a successful attack.

Mitigations against Meltdown-type Attacks. Meltdown attacks exploit transient execution caused by delayed exception handling. The first

mitigation to Meltdown-US was the KAISER patch which unmaps the kernel space when a thread runs in the context of a user to prevent transient leakage of kernel memory [75, 86]. Intel patched the Meltdown vulnerability in hardware, starting with the Cascade lake architecture. Canella et al. [43] showed that zeroing out the values on the Cascade Lake architecture is insufficient to fully mitigate against Meltdown-type effects. MDS-attacks like RIDL [211] and Zombieload [218] showed that Meltdown-resistant CPUs can still have transient data leakage. As these attacks exploit private components on a CPU core, another suggested mitigation was to disable SMT completely [140]. While MDS attacks have also been mitigated in hardware with the Ice Lake architecture, Moghimi [165] showed that MDS effects can still occur. A thorough analysis of the mitigations of Meltdown-type attacks was performed by Canella et al. [42, 44] and Cauligi et al. [47].

3.3. Attacks on Memory Usage Optimizations

In this section, we will discuss existing attacks on memory deduplication and memory compression. Moreover, we will discuss the state-of-the-art defenses against memory-deduplication attacks and memory-compression attacks.

3.3.1. Memory Deduplication

Memory deduplication is a technique used in operating systems to reduce the amount of duplicated memory. This technique marks pages with identical contents and maps all entries to the same physical location. As the memory is marked as read-only, a timing side-channel exists when overwriting a single deduplicated memory location as a copy-on-write pagefault is triggered. Memory-deduplication attacks were used to detect co-location in data centers [238, 271], covert communication [273, 274], OS and website fingerprinting via native code and JavaScript [84, 148, 184], break ASLR [21], perform browser sandbox escapes via Rowhammer and password leakage on web servers [32]. Palfinger et al. [186] showed that file systems are susceptible to deduplication attacks. Kim et al. [128] presented a KASLR break on VMWare ESXi using techniques similar to the KASLR break in remote memory-deduplication attacks (Chapter 8). Barua et al. [22] demonstrated a Rowhammer attack combined with

memory-deduplication attacks on industrial control systems to inject false commands into programmable logic units. DUPEFS [19] demonstrated remote leakage of OAuth tokens by exploiting memory deduplication on file systems. In concurrent work to our remote memory-deduplication attacks (Chapter 8), Costi et al. [54] evaluated the security of same-domain memory deduplication in a client-server setup and exploited a cross-tab scenario on Firefox using JavaScript.

The most simple mitigation to prevent memory-deduplication attacks, is to disable memory deduplication. Windows offers an option to disable memory deduplication per process [163]. Another alternative proposed by Bosman et al. [32] is to deduplicate only zero-initialized pages. VMWare TPS [255] uses a unique key per virtual machine to mitigate cross-VM deduplication attacks. Wang et al. [263] focus on hooking the `rdtsc` instruction of the virtual machine and monitoring the number of pagefaults. Oliverio et al. [181] enforce a fake merging of pages such that an attacker cannot distinguish a correctly guessed page from an incorrectly guessed page.

3.3.2. Memory Compression

Compression is widely used in operating systems, the web, and various applications [17, 37, 66, 169, 194, 196, 285]. Similar to memory deduplication, compression algorithms can be used to reduce memory utilization. In 2002, Kelsey et al. [126] presented a compression attack based on the compression ratio of plaintext that is first compressed and then encrypted. If attacker-controlled data is compressed together with secret data, an attacker can guess the content of the secret. Based on the compression ratio, the attacker can derive whether the attack was correct or incorrect. There are scenarios, e.g., in HTTP, where both attacker-controlled input and secret input, e.g., web cookies, are compressed. Thus, if the attacker performs a guess and monitors the packet length of a TLS packet, an attacker can derive the secret cookie. Several attacks exploited the compression ratio side channel [25, 76, 124, 206, 251, 252]. We show that various compression algorithms also reveal a timing side channel due to corner cases in Chapter 9.

The most simple mitigation to mitigate memory-compression-based attacks, is to avoid compressing sensitive data together with attacker-controlled data. Mutexion [168] mutually excludes co-location of sen-

sitive data and attacker-controlled data for HTTP by using automated secret annotations. SafeDeflate [296] prevents compression-ratio attacks by blocking certain keywords. Taint tracking was used to find HTTP responses containing both secret and attacker-controlled data [188]. Another mitigation is to disable the LZ77 part [123], leading to an additional runtime overhead in HTTP of up to 500 %.

3.4. Remote Timing Attacks

Brumley and Boney et al. [36] demonstrated remote timing attacks extracting SSL private keys by remotely inferring cache timings. Based on Bernstein's idea of attacking AES [24] several remote-timing attacks on AES have been demonstrated [15, 120, 209, 295]. Acıiçmez et al. [6] remotely attacked AES via a cache side-channel attack. Crosby et al. [55] presented the box test, a method to effectively determine the number of required packets to distinguish timing differences over the network. Irazoqui et al. [117] demonstrated an attack on TLS in a LAN which exploits cache timing differences. The TIME [25] attack abused the compression ratio to modify the size of TCP windows and amplify the timing differences between correct and incorrect cookie guesses. With the Heist attack, Vanhoef et al. [252] exploited HTTP/2 features to observe the exact size of a cross-origin resource. Van Goethem et al. [250] showed that the packet order in HTTP/2 concurrent requests can be used to perform remote timing attacks without relying on timing information. Based on that result, they successfully attacked HTTP/2 web servers, Tor onion services, and Wi-Fi authentication methods. Gruss et al. [85] demonstrated a remote timing attack by exploiting the page cache on Windows and Linux. Schwarz et al. [220] mounted the remote Spectre attack and transiently leaked arbitrary data across the network. Kurt et al. [138] presented a remote attack spying on keystrokes entered in an SSH session by exploiting a remote cache side-channel attack. Wang et al. [264] showed that dynamic voltage frequency scaling can be exploited to perform remote timing attacks on a post-quantum key encapsulation mechanism.

To mitigate remote timing attacks, the first approach is to eliminate the root cause of the timing leakage by mitigating the timing side channel. If this is not possible, network countermeasures can be used to add artificial noise to the response packets round-trip time. Moreover, as still many

request packets are required to just leak a single bit [220], distributed denial-of-service attack (DDoS) protection systems might detect such attacks. Thus, there is a tradeoff between the leaked bits and the number of packets sent per second.

4

Conclusion and Outlook

In this thesis, we took a deep dive in how attackers can mount remote side-channel attacks and how side-channel attacks can be actively mitigated. We draw the following conclusions from our results.

Amplification of Side-Channel Attacks. Mounting remote side-channel attacks is practical if attackers are able to amplify the latency of side channels. We demonstrated that memory deduplication and compression algorithms can reveal latencies that are distinguishable across several hops across the internet [222, 225]. Both techniques are broadly applicable, and the impact of the remote attacks is unclear yet. For many side-channel attacks, it is still unclear whether they can be amplified to mount successful and remote attacks. Recently, Xiao [272] showed that timing differences can be amplified by leveraging port contention and cache replacement policies. We expect future research to discover further techniques to amplify side channels, perform remote attacks and overcome low-resolution timers, e.g., used in browsers [237, 272]. Moreover, while the throughput of the attacks we demonstrated is in the range of multiple hundred bytes across the internet [222, 225], we expect remote side-channel attacks to become more dangerous with faster internet connections.

Spectre Mitigations. As speculative execution is a design choice, CPU vendors cannot simply eliminate this optimization without large performance losses [133, 143, 161]. Especially in single-process applications, it is hard to mitigate Spectre attacks [161]. While companies like Google already gave up on mitigating Spectre for single-process applications [161, 203], we demonstrated that there is still space left open for creating efficient and performant mitigations. Together with Cloudflare, we created a dynamic detection and mitigation that dynamically isolated malicious

scripts. With hardware-assisted features like protection keys [213], in-process isolation can be enabled. Future work should research isolation for complex software such as operating systems and browsers to mitigate against transient-execution attacks. There is still huge potential to create novel mitigations like Specfuscator [223] by changing the perspective on a certain problem. While there has been ongoing research in finding Spectre gadgets, there is still potential to create more accurate detection mechanisms.

Automatic Side-Channel Discovery and Root-Cause Analysis. Automatically finding and detecting side channels helps improving the security of software and hardware. Moreover, automatic discovery can also help to determine the root cause of side channels. We showed that the address-translation attack attributed to the `prefetch` instruction was actually possible due to speculative execution [226]. Using a fuzzer, we discovered novel timing side channels in various compression algorithms [222]. By using an automated cache templating technique [224], we discovered that keystrokes entered in Chromium-based applications can be leaked with a cache side channel. We showed that compilation and linking can introduce cache-side channels.

Outlook. In general, there is an ongoing trend to reduce hardware's energy consumption. This reduction might introduce novel side channels. Moreover, research already showed that timing differences caused by frequency throttling can introduce a timing side channel [154, 264]. Therefore, we expect future research to investigate the security impact of energy optimizations further.

Moreover, the market share for software-as-a-service is still increasing [280]. While this has many advantages in terms of costs, performance, and scalability, the customers' data has to be protected from remote attackers. Especially if resources such as hardware or software are shared between tenants, more side-channel attacks can be expected. As we demonstrated remote attacks in both sandboxed applications like browsers and solely via remote APIs, we expect more side-channel attacks on such systems. Therefore, we expect future research to discover remote timing attacks on the network, application, and hardware layers.

Complex optimizations in software and hardware are likely to introduce new side channels. Moreover, software-based side-channel attacks have

become more practical and dangerous during the last years. Automated discovery of side channels could improve the security for both software and hardware development. Therefore, we expect research on automated discovery of side channels within commodity software and hardware. Future work should investigate how to integrate such tools into continuous software and hardware development.

References

[1] Andreas Abel and Jan Reineke. “Reverse engineering of cache replacement policies in intel microprocessors and their evaluation.” In: *ISPASS*. 2014.

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

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

[4] Onur Acıiçmez and Çetin Kaya Koç. “Trace-Driven Cache Attacks on AES.” In: (2006), pp. 112–121.

[5] Onur Acıiçmez and Werner Schindler. “A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL.” In: *CT-RSA*. 2008.

[6] Onur Acıiçmez, Werner Schindler, and Cetin K. Koc. “Cache Based Remote Timing Attack on the AES.” In: *CT-RSA*. 2006.

[7] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. “New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures.” In: *IMACC*. 2007.

[8] Onur Acıiçmez. “Yet Another MicroArchitectural Attack: Exploiting I-cache.” In: *CSAW*. 2007.

[9] Onur Acıiçmez and Cetin Kaya Koç. “Microarchitectural attacks and countermeasures.” In: *Journal of Cryptographic Engineering*. 2009.

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

[11] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution." In: *S&P*. 2022.

[12] National Security Agency. *TEMPEST: A Signal Problem*. 1972.

[13] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit." In: *S&P*. 2019.

[14] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. "Amplifying Side Channels Through Performance Degradation." In: *ACSAC*. 2016.

[15] Hassan Aly and Mohammed ElGayyar. "Attacking aes using bernstein's attack on modern processors." In: *AfricaCrypt*. 2013.

[16] AMD. *Software Optimization Guide for AMD Family 17th Processors*. 2022.

[17] Apple Insider. 2013. URL: `https://appleinsider.com/articles/13/06/13/compressed-memory-in-os-x-109-mavericks-aims-to-free-ram-extend-battery-life`.

[18] ARM. *Straight-line Speculation*. Version 1.0. 2020.

[19] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. "DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels." In: *USENIX Security Symposium*. 2022.

[20] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks." In: *USENIX Security Symposium*. 2022.

[21] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. "CAIN: Silently Breaking ASLR in the Cloud." In: *WOOT*. 2015.

[22] Anomadarshi Barua, Lelin Pan, and Mohammad Abdullah Al Faruque. "BayesImposter: Bayesian Estimation Based .bss Imposter Attack on Industrial Control Systems." In: *ACSAC*. 2022.

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

[24] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[25] Tal Be'ery and Amichai Shulman. "A Perfect CRIME? Only TIME Will Tell." In: *Black Hat Europe*. 2013.

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

[27] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. "A survey of timing channels and countermeasures." In: *CSUR* 50.1 (2017), pp. 1–39.

[28] Roderick Bloem, Swen Jacobs, and Yakir Vizel. "Efficient Information-Flow Verification Under Speculative Execution." In: *ATVA*. 2019.

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

[30] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. "Constantine: Automatic side-channel resistance using efficient control and data flow linearization." In: *CCS*. 2021.

[31] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture." In: *USENIX Security Symposium*. 2022.

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

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

[34] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks." In: *USENIX Security Symposium*. 2020.

[35] Billy Brumley and Risto Hakala. "Cache-Timing Template Attacks." In: *AsiaCrypt*. 2009.

[36] David Brumley and Dan Boneh. "Remote Timing Attacks Are Practical." In: *USENIX Security Symposium*. 2003.

[37] BTRFS. *Compression*. 2021. URL: https://btrfs.wiki.kernel.org/index.php/Compression#Why_does_not_du_report_the_compressed_size.3F.

[38] Yuriy Bulygin and David Samyde. "Chipset based approach to detect virtualization malware." In: 2008.

[39] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. "Cache-Timing Attacks on RSA Key Generation." In: *Cryptology ePrint Archive* (2019).

[40] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs." In: *CCS*. 2019.

[41] Claudio Canella, Khaled N Khasawneh, and Daniel Gruss. "The evolution of transient-execution attacks." In: *GLSVLSI*. 2020.

[42] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. "Evolution of defenses against transient-execution attacks." In: *GLSVLSI*. 2020.

[43] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. "KASLR: Break It, Fix It, Repeat." In: *AsiaCCS*. 2020.

[44] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/. 2019.

[45] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. *Transient Fail (GitHub)*. 2020. URL: https://github.com/IAIK/transientfail.

[46] Chandler Carruth. *Speculative Load Hardening*. 2020. URL: https://llvm.org/docs/SpeculativeLoadHardening.html.

[47] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. "SoK: Practical Foundations for Software Spectre Defenses." In: *S&P*. 2022.

[48] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. "A formal approach to secure speculation." In: *CSF*. 2019.

[49] Euccas Chen. *Understanding zlib*. 2021. URL: https://www.euccas.me/zlib/#deflate.

[50] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. *Real time detection of cache-based side-channel attacks using Hardware Performance Counters*. Applied Soft Computing. 2016.

[51] Chromium. *Issue 821270: Re-enable SharedArrayBuffer + Atomics*. 2019. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=821270.

[52] Cloudflare. *Cloudflare Workers*. 2019. URL: https://www.cloudflare.com/products/cloudflare-workers/.

[53] Jonathan Corbet. *Finding Spectre vulnerabilities with smatch*. 2018. URL: https://lwn.net/Articles/752408/.

[54] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. "On the Effectiveness of Same-Domain Memory Deduplication." In: *EuroSys*. 2022.

[55] Scott Crosby, Dan Wallach, and Rudolf Riedi. "Opportunities and limits of remote timing attacks." In: *TISSEC* 12.3 (2009), pp. 1–29.

[56] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks." In: *CHES*. 2018.

[57] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse." In: *NDSS*. 2021.

[58] Peter Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification Version 1.3*. USA, 1996.

[59] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX." In: *USENIX Security Symposium*. 2017.

[60] Christopher Domas. *M/o/Vfuscator*. 2015. URL: https://github.com/xoreaxeaxeax/movfuscator.

[61] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems." In: *ISCA*. 2021.

[62] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." In: *arXiv preprint arXiv:2203.15981* (2022).

[63] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. "Rapid Prototyping for Microarchitectural Attacks." In: *USENIX Security Symposium*. 2022.

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

[65] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor." In: *ASPLOS*. 2018.

[66] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: `http://www.rfc.net/rfc2616.html`.

[67] Agner Fog. *The microarchitecture of Intel, AMD, and VIA CPUs*. 2022. URL: `https://www.agner.org/optimize/microarchitecture.pdf`.

[68] Jacob Fustos, Michael Bechtel, and Heechul Yun. "SpectreRewind: Leaking Secrets to Past Instructions." In: *ASHES*. 2020.

[69] Jacob Fustos, Farzad Farshchi, and Heechul Yun. "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks." In: *DAC*. 2019.

[70] Cesar Pereida García and Billy Bob Brumley. "Constant-Time Callees with Variable-Time Callers." In: *USENIX Security Symposium*. 2017.

[71] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. "SQUIP: Exploiting the Scheduler Queue Contention Side Channel." In: *S&P*. 2023.

[72] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[73] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. "Drive-by Key-Extraction Cache Attacks from Portable Code." In: *ACNS*. 2018.

[74] glibc Github. *strcmp.c*. 2022. URL: `https://github.com/zerovm/glibc/blob/master/string/strcmp.c`.

[75] Thomas Gleixner. *x86/kpti: Kernel Page Table Isolation (was KAISER)*. 2017. URL: `https://lkml.org/lkml/2017/12/4/709`.

[76] Yoel Gluck, Neal Harris, and Angelo Prado. "BREACH: reviving the CRIME attack." In: *Unpublished manuscript* (2013).

[77] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era." In: *CCS*. 2020.

[78] Google. *Snappy*. 2022. URL: `https://github.com/google/snappy`.

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

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

[81] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme." In: *CHES*. 2016.

[82] Daniel Gruss. "Software-based Microarchitectural Attacks." PhD thesis. Graz University of Technology, 2017.

[83] Daniel Gruss. "Transient-Execution Attacks and Defenses (Part I only)." PhD thesis. Graz University of Technology, 2020.

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

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

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

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

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

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

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

[91] Roberto Guanciale, Musard Balliu, and Mads Dam. "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis." In: *CCS*. 2020.

[92] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled Detection of Speculative Information Flows." In: *S&P*. 2020.

[93] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-software contracts for secure speculation." In: *S&P*. 2021.

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

[95] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES." In: *COSADE*. 2015.

[96] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross-VM cache attacks on AES." In: *IEEE TMCS* 2.3 (2016), pp. 211–222.

[97] Berk Gülmezoğlu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning." In: *arXiv preprint arXiv:1907.03651*. 2019.

[98] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 5th Edition*. 2019.

[99] Nishad Herath and Anders Fogh. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security." In: *Black Hat Briefings*. 2015.

[100] Mark D Hill and Alan Jay Smith. "Evaluating associativity in CPU caches." In: *IEEE Transactions on Computers* 38.12 (1989), pp. 1612–1630.

[101] Jann Horn. *Speculative execution, variant 4: Speculative Store Bypass*. 2018.

[102] Wei-Ming Hu. "Lattice Scheduling and Covert Channels." In: *S&P*. 1992.

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

[104] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX." In: *CHES*. 2020.

[105] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. "SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities." In: *CCS*. 2022.

[106] Michael Hutter and Jörn-Marc Schmidt. "The temperature side channel and heating fault attacks." In: *CARDIS*. 2013.

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

[108] Mehmet Sinan Inci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud." In: *Cryptology ePrint Archive* (2015).

[109] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. "Way-predicting set-associative cache for high performance and low energy consumption." In: *Low Power Electronics and Design*. 1999.

[110] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

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

[112] Intel. *Intel Optane DC Persistent Memory*. 2021. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[113] Intel. *Intel Speculative Execution Side Channel Mitigations*. 2018. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf.

[114] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "MASCAT: Preventing microarchitectural attacks before distribution." In: *CODASPY*. 2018.

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

[116] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Know Thy Neighbor: Crypto Library Detection in Cloud." In: *PETS* 2015.1 (2015), pp. 25–40.

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

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

[119] Himanshi Jain, D Anthony Balaraju, and Chester Rebeiro. "Spy Cartel: Parallelizing Evict+ Time-Based Cache Attacks on Last-Level Caches." In: *Journal of Hardware and Systems Security* 3.2 (2019), pp. 147–163.

[120] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. "Remote cache timing attack on advanced encryption standard and countermeasures." In: *ICIAFs*. 2010.

[121] Daniel Jiménez and Calvin Lin. "Dynamic branch prediction with perceptrons." In: *HPCA*. 2001.

[122] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "KASPER: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel." In: *NDSS*. 2022.

[123] Dimitris Karakostas, Aggelos Kiayias, Eva Sarafianou, and Dionysis Zindros. “CTX: Eliminating BREACH with context hiding.” In: 2016.

[124] Dimitris Karakostas and Dionysis Zindros. “Practical new developments on BREACH.” In: 2016.

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

[126] John Kelsey. “Compression and Information Leakage of Plaintext.” In: *Fast Software Encryption*. 2002.

[127] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. “Side Channel Cryptanalysis of Product Ciphers.” In: *ESORICS*. 1998.

[128] Taehun Kim, Taehyun Kim, and Youngjoo Shin. “Breaking KASLR Using Memory Deduplication in Virtualized Environments.” In: *Electronics* 10.17 (2021), pp. 2174–2185. URL: https://www.mdpi.com/2079-9292/10/17/2174.

[129] Taehun Kim and Youngjoo Shin. “ThermalBleed: A Practical Thermal Side-Channel Attack.” In: *IEEE Access* 10 (2022), pp. 25718–25731.

[130] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. “DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors.” In: *MICRO*. 2018.

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

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

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

[134] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis.” In: *CRYPTO*. 1999.

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

[136] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation." In: *S&P*. 2020.

[137] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs." In: *EuroS&P*. 2020.

[138] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "NetCAT: Practical Cache Attacks from the Network." In: *S&P*. 2020.

[139] Ben Lapid and Avishai Wool. "Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis." In: *SAC*. 2018.

[140] Michael Larabel. *Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS*. 2018. URL: `https://www.phoronix.com/review/intel-ht-2018/4`.

[141] Michael Larabel. *Linux 5.5 To Enable Intel's 5-Level Paging Support By Default*. 2019. URL: `https://www.phoronix.com/news/Linux-5.5-5-Level-Paging`.

[142] Michael Larabel. *The Brutal Performance Impact From Mitigating The LVI Vulnerability*. 2020. URL: `https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf`.

[143] Michael Larabel. *The Current Spectre / Meltdown Mitigation Overhead Benchmarks On Linux 5.0*. 2020. URL: `https://www.phoronix.com/review/linux50-spectre-meltdown`.

[144] Nate Lawson. "Side channel attacks on cryptographic software." In: *S&P*. 2009.

[145] Chih-Chieh Lee, I-CK Chen, and Trevor N Mudge. "The bi-mode branch predictor." In: *MICRO*. IEEE. 1997.

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

[147] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. "Conditional Speculation: An effective approach to safeguard out-of-order execution against spectre attacks." In: *HPCA*. 2019.

[148] Jens Lindemann and Mathias Fischer. “A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines.” In: *SAC*. 2018.

[149] Moritz Lipp. “Exploiting Microarchitectural Optimizations from Software.” PhD thesis. Graz University of Technology, 2021.

[150] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. “Nethammer: Inducing Rowhammer Faults through Network Requests.” In: *SILM Workshop*. 2020.

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

[152] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take a Way: Exploring the Security Implications of AMD’s Cache Way Predictors.” In: *AsiaCCS*. 2020.

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

[154] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. “Frequency throttling side-channel attack.” In: *CCS*. 2022.

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

[156] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. “Side-Channel Attacks on Optane Persistent Memory.” In: *USENIX Security Symposium*. 2023.

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

[158] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. “Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations.” In: *ACSAC*. 2019.

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

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

[161] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. "Spectre is here to stay: An analysis of side-channels and speculative execution." In: *arXiv:1902.05178* (2019).

[162] Vahid Meraji and Hadi Soleimany. "Evict+ Time Attack on Intel CPUs without Explicit Knowledge of Address Offsets." In: *ISeCure* (2021).

[163] Microsoft. *PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY structure (winnt.h)*. 2021. URL: `https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_side_channel_isolation_policy`.

[164] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX amplifies the power of cache attacks." In: *CHES*. 2017.

[165] Daniel Moghimi. "Data Sampling on MDS-resistant 10th Generation Intel Core (Ice Lake)." In: 2020.

[166] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis." In: *USENIX Security Symposium*. 2020.

[167] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. "The program counter security model: Automatic detection and removal of control-flow side channel attacks." In: *ICISC*. 2006.

[168] Taegeun Moon, Hyoungshick Kim, and Sangwon Hyun. "Mutexion: Mutually Exclusive Compression System for Mitigating Compression Side-Channel Attacks." In: *TWEB* 16.4 (2022), pp. 1–20.

[169] Mozilla. *Compression in HTTP*. 2021. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Compression`.

[170] Mozilla. *performance.now resolution*. 2019. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`.

[171] Saidgani Musaev and Christof Fetzer. "Transient Execution of Non-Canonical Accesses." In: *arXiv preprint arXiv:2108.10771* (2021).

[172] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. "WHISPER: A Tool for Run-time Detection of Side-Channel Attacks." In: *IEEE Access* 8 (2020), pp. 83871–83900.

[173] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. "Swivel: Hardening {WebAssembly} against Spectre." In: *USENIX Security Symposium*. 2021.

[174] Amir Naseredini, Stefan Gast, Martin Schwarzl, Pedro Miguel Sousa Bernardo, Amel Smajic, Claudio Canella, Martin Berger, and Daniel Gruss. "Systematic Analysis of Programming Languages and Their Execution Environments for Spectre Attacks." In: *ICISSP*. 2022.

[175] Michael Neve and Jean-Pierre Seifert. "Advances on Access-Driven Cache Attacks on AES." In: *SAC*. 2006.

[176] Lucas Noack and Tobias Reichert. "Exploiting speculative execution (spectre) via javascript." In: *Advanced Microkernel Operating Systems* (2018), p. 11.

[177] Ejebagom John Ojogbo, Mithuna Thottethodi, and TN Vijaykumar. "Secure automatic bounds checking: prevention is simpler than cure." In: *CGO*. 2020.

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

[179] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. "You shall not bypass: Employing data dependencies to prevent bounds check bypass." In: *arXiv preprint arXiv:1805.08506* (2018).

[180] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. "SpecfuzzBringing Spectre-type vulnerabilities to the surface." In: *USENIX Security Symposium*. 2020.

[181] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Secure Page Fusion with VUsion." In: *SOSP*. 2017.

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

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

[184] Rodney Owens and Weichao Wang. "Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines." In: *International Performance Computing and Communications Conference*. 2011.

[185] Dan Page. "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel." In: *Cryptology ePrint Archive, Report 2002/169* (2002).

[186] Gerald Palfinger, Bernd Prünster, and Dominik Ziegler. "Prying CoW: Inferring Secrets across Virtual Machine Boundaries." In: *SECRYPT*. 2019.

[187] Marco Patrignani and Marco Guarnieri. "Exorcising Spectres with Secure Compilers." In: *CCS*. 2021.

[188] Brandon Paulsen, Chungha Sung, Peter AH Peterson, and Chao Wang. "Debreach: mitigating compression side channels via static analysis and transformation." In: *ASE*. 2019.

[189] Pawel Wieczorkiewicz. *The AMD Branch (Mis)predictor: Just Set it and Forget it!* 2022. URL: `https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it`.

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

[191] Colin Percival. "Cache Missing for Fun and Profit." In: *BSDCan*. 2005.

[192] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really Are Constant-Time." In: 2016.

[193] Chris H Perleberg and Alan Jay Smith. "Branch target buffer design and optimization." In: *IEEE Transactions on Computers*. 1993.

[194] php.net. *memcached.constants.php*. 2021. URL: `https://www.php.net`.

[195] Joop van de Pol, Nigel P Smart, and Yuval Yarom. "Just a little bit more." In: *CT-RSA*. 2015.

[196] postgresql.com. *TOAST Compression*. 2021. URL: https://www.postgresql.org/docs/current/storage-toast.html.

[197] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks." In: *CCS*. 2021.

[198] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. "SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets." In: *NDSS*. 2021.

[199] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real." In: *S&P*. 2021.

[200] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CROSSTALK: Speculative Data Leaks Across Cores Are Real." In: *S&P*. 2021.

[201] RedHat. *Spectre Gadget Scanner*. https://people.redhat.com/~nickc/Spectre_Scanner/scanner.tar.xz. 2018.

[202] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. "Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack." In: *SBCCI*. 2016.

[203] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *USENIX Security Symposium*. 2019.

[204] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. "I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches." In: *ISCA*. 2021.

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

[206] Juliano Rizzo and Thai Duong. "The CRIME attack." In: *ekoparty security conference*. 2012.

[207] Stephen Röttger. *Escaping the Chrome Sandbox with RIDL*. 2020. URL: https://googleprojectzero.blogspot.com/2020/02/escaping-chrome-sandbox-with-ridl.html.

[208] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion." In: *ASPLOS*. 2021.

[209] Vishal Saraswat, Daniel Feldman, Denis Foo Kune, and Satyajit Das. "Remote Cache-timing Attacks Against AES." In: *Workshop on Cryptography and Security in Computing Systems*. 2014.

[210] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *SGAxe: How SGX Fails in Practice*. `https://sgaxeattack.com/`. 2020.

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

[212] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. "CacheOut: Leaking Data on Intel CPUs via Cache Evictions." In: *S&P*. 2021.

[213] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. "Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86." In: *USENIX Security Symposium*. 2020.

[214] Michael Schwarz. "Software-based Side-Channel Attacks and Defenses in Restricted Environments." PhD thesis. Graz University of Technology, 2019.

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

[216] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre." In: *NDSS*. 2020.

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

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

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

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

[221] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. “Robust and Scalable Process Isolation Against Spectre in the Cloud.” In: *ESORICS*. 2022.

[222] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. “Practical Timing Side Channel Attacks on Memory Compression.” In: *S&P*. 2023.

[223] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. “Specfuscator: Evaluating Branch Removal as a Spectre Mitigation.” In: *FC*. 2021.

[224] Martin Schwarzl, Erik Kraft, and Daniel Gruss. “Layered Binary Templating.” In: *ACNS*. 2023.

[225] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. “Remote Memory-Deduplication Attacks.” In: *NDSS*. 2022.

[226] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. “Speculative Dereferencing of Registers: Reviving Foreshadow.” In: *FC*. 2021.

[227] Jeff Seibert, Hamed Okhravi, and Eric Söderström. “Information leaks without memory disclosures: Remote side channel attacks on diversified code.” In: *CCS*. 2014.

[228] Hovav Shacham. “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).” In: *CCS*. 2007.

[229] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O’Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. “Spectre Declassified: Reading from the Right Place at the Wrong Time.” In: *Cryptology ePrint Archive* (2022).

[230] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. “Robust Website Fingerprinting Through The Cache Occupancy Channel.” In: *USENIX Security Symposium*. 2019.

[231] James Smith. “A study of branch prediction strategies.” In: *ISCA*. 1998.

[232] Wei Song and Peng Liu. "Dynamically Finding Minimal Eviction Sets Can be Quicker Than You Think for Side-Channel Attacks Against the LLC." In: *RAID*. 2019.

[233] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. "Systematic classification of side-channel attacks: a case study for mobile devices." In: *IEEE Communications Surveys & Tutorials* (2017).

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

[235] Jared Stark, Marius Evers, and Yale N Patt. "Variable length path branch prediction." In: *ASPLOS*. 1998.

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

[237] Stephen Roettger and Artur Janc. *A Spectre proof-of-concept for a Spectre-proof web*. 2021. URL: `https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html`.

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

[239] Jakub Szefer. "Survey of microarchitectural side and covert channels, attacks, and defenses." In: *Journal of Hardware and Systems Security* (2019).

[240] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. "Context-sensitive fencing: Securing speculative execution via microcode customization." In: *ASPLOS*. 2019.

[241] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses." In: *USENIX ATC*. 2018.

[242] Cover Thomas and Joy Thomas. In: *Elements of Information Theory*. 2006.

[243] M Caner Tol, Berk Gülmezoğlu, Koray Yurtseven, and Berk Sunar. "Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings." In: *EuroS&P*. 2021.

[244] Robert M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units.” In: *IBM Journal of research and Development* (1967).

[245] Eran Tromer, Dag Arne Osvik, and Adi Shamir. “Efficient Cache Attacks on AES, and Countermeasures.” In: *Journal of Cryptology* (2010).

[246] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. “Cryptanalysis of DES implemented on computers with cache.” In: *CHES*. 2003.

[247] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: `https://support.google.com/faqs/answer/7625886`.

[248] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.” In: *USENIX Security Symposium*. 2018.

[249] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection.” In: *S&P*. 2020.

[250] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. “Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections.” In: *USENIX Security Symposium*. 2020.

[251] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. “Request and conquer: Exposing cross-origin resource size.” In: *USENIX Security Symposium*. 2016.

[252] Mathy Vanhoef and Tom Van Goethem. “HEIST: HTTP Encrypted Information can be Stolen through TCP-windows.” In: *Black Hat US Briefings*. 2016.

[253] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Koepf. “CacheQuery: Learning Replacement Policies from Hardware Caches.” In: *PLDI*. 2020.

[254] Pepe Vila, Boris Köpf, and Jose Morales. “Theory and Practice of Finding Eviction Sets.” In: *S&P*. 2019.

[255] Vmware. *Additional Transparent Page Sharing management capabilities and new default settings*. 2021. URL: https://kb.vmware.com/s/article/2097593.

[256] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. "BRB: Mitigating Branch Predictor Side-Channels." In: *HPCA*. 2019.

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

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

[259] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. "KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution." In: *TOSEM* 29.3 (2020), pp. 1–31.

[260] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. "oo7: Low-overhead Defense against Spectre attacks via Program Analysis." In: *Transactions on Software Engineering* 47.11 (2019), pp. 2504–2519.

[261] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks." In: *ICCAD*. 2020.

[262] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, Tinoosh Mohsenin, and Houman Homayoun. "Comprehensive Evaluation of Machine Learning Countermeasures for Detecting Microarchitectural Side-Channel Attacks." In: *GLSVLSI*. 2020.

[263] S. Wang, Weizhong Qiang, H. Jin, and Jinfeng Yuan. "CovertInspector: Identification of Shared Memory Covert Timing Channel in Multi-tenanted Cloud." In: *International Journal of Parallel Programming* 45.1 (2017), pp. 142–156.

[264] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86." In: *USENIX Security Symposium*. 2022.

[265] Z. Wang and R. B. Lee. “Covert and Side Channels Due to Processor Architecture.” In: *ACSAC*. 2006.

[266] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. “Osiris: Automated Discovery Of Microarchitectural Side Channels.” In: *USENIX Security Symposium*. 2021.

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

[268] Wikichip. *Core i5-8250U - Intel*. 2021. URL: `https://en.wikichip.org/wiki/intel/core_i5/i5-8250u`.

[269] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks.” In: *WOOT*. 2022.

[270] Johannes Wikner and Kaveh Razavi. “RETBLEED: Arbitrary Speculative Code Execution with Return Instructions.” In: *USENIX Security Symposium*. 2022.

[271] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D Corner. “Memory buddies: exploiting page sharing for smart colocation in virtualized data centers.” In: 43.3 (2009), pp. 27–36.

[272] Haocheng Xiao and Sam Ainsworth. “Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers.” In: *arXiv preprint arXiv:2211.14647* (2022).

[273] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. “A covert channel construction in a virtualized environment.” In: *CCS*. 2012.

[274] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. “Security implications of memory deduplication in a virtualized environment.” In: *DSN*. 2013.

[275] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. “SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities.” In: *NDSS*. 2020.

[276] Wenjie Xiong and Jakub Szefer. “Leaking Information Through Cache LRU States.” In: *HPCA*. 2020.

[277] Wenjie Xiong and Jakub Szefer. "Survey of Transient Execution Attacks and Their Mitigations." In: *ACM Computing Surveys* 54.3 (2021), pp. 1–36.

[278] Ke Xu, Ming Tang, Han Wang, and Sylvain Guilley. "Reverse-engineering and Exploiting the Frontend Bus of Intel Processor." In: *IEEE Transactions on Computers* (2022), pp. 1–14.

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

[280] Yahoo. *Software as a Service (SaaS) Market Size (2022)*. 2022. URL: `https://finance.yahoo.com/news/software-saas-market-size-2022-065000389.html`.

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

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

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

[284] Tse-Yu Yeh and Yale N Patt. "Alternative implementations of two-level adaptive branch prediction." In: 20.2 (1992), pp. 124–134.

[285] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals Part 1*. 7th ed. Microsoft Press, 2017.

[286] Younis A Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. "A new prime and probe cache side-channel attack for cloud computing." In: *CIT/IUCC/DASC/PICOM*. IEEE. 2015.

[287] Yuanyuan Yuan, Qi Pang, and Shuai Wang. "Automated side channel analysis of media software with manifold learning." In: *USENIX Security Symposium*. 2022.

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

[289] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. “Return-oriented flush-reload side channels on arm and their implications for android devices.” In: *CCS*. 2016.

[290] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-Tenant Side-Channel Attacks in PaaS Clouds.” In: *CCS*. 2014.

[291] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-VM Side Channels and Their Use to Extract Private Keys.” In: *CCS*. 2012.

[292] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. “See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer.” In: *AsiaCCS*. 2021.

[293] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. “Breaking and Fixing Speculative Load Hardening.” In: *Cryptology ePrint Archive* (2022).

[294] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. “A lightweight isolation mechanism for secure branch predictors.” In: *DAC*. 2021.

[295] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. “Cache Timing Attacks on Camellia Block Cipher.” In: *Cryptology ePrint Archive* 2009 (2009), pp. 354–371.

[296] Michał Zielinski. *SafeDeflate: compression without leaking secrets*. Tech. rep. Cryptology ePrint Archive, 2016.

Part II.

Publications

5

Speculative Dereferencing of Registers:Reviving Foreshadow

Publication Data

Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. "Speculative Dereferencing of Registers: Reviving Foreshadow." In: *FC*. 2021

Contributions

Main author.

Speculative Dereferencing: Reviving Foreshadow

Martin Schwarzl[1], Thomas Schuster[1], Michael Schwarz[2], Daniel Gruss[1]

[1] Graz University of Technology, Austria [2] CISPA Helmholtz Center for Information Security, Germany

Abstract

In this paper, we provide a systematic analysis of the root cause of the prefetching effect observed in previous works and show that its attribution to a prefetching mechanism is incorrect in all previous works, leading to incorrect conclusions and incomplete defenses. We show that the root cause is speculative dereferencing of user-space registers in the kernel. This new insight enables the first end-to-end Foreshadow (L1TF) exploit targeting non-L1 data, despite Foreshadow mitigations enabled, a novel technique to directly leak register values, and several side-channel attacks. While the L1TF effect is mitigated on the most recent Intel CPUs, all other attacks we present still work on all Intel CPUs and on CPUs by other vendors previously believed to be unaffected.

5.1. Introduction

For security reasons, operating systems hide physical addresses from user programs [33]. Hence, an attacker requiring this information has to leak it first, e.g., with the *address-translation attack* by Gruss et al. [16, §3.3 and §5]. It allows user programs to fetch arbitrary kernel addresses into the cache and thereby to resolve virtual to physical addresses. As a mitigation against e.g., the address-translation attack, Gruss et al. [15, 16] proposed the KAISER technique.

Other attacks observed and exploited similar prefetching effects. Meltdown [41] practically leaks memory that is not in the L1 cache. Xiao et al. [73] show that this relies on a prefetching effect that fetches data from the L3 cache into the L1 cache. However, Van Bulck et al. [67] observe no such effect for Foreshadow.

We systematically analyze the root cause of the prefetching effect exploited in these works. We show that, despite the sound approach of these papers, the attribution of the root cause, *i.e.*, why the kernel addresses are cached, is incorrect in all cases. The root cause is unrelated to software prefetch instructions or hardware prefetching effects due to memory accesses and instead is caused by speculative dereferencing of user-space registers in the kernel. While there are many speculative code paths in the kernel, we focus on code paths with Spectre [6, 34] gadgets that can be reliably triggered on both Linux and Windows.

These new insights correct several wrong assumptions from previous works, also leading to new attacks. Most significantly, the difference that Meltdown can leak from L3 or main memory [41] but Foreshadow (L1TF) can only leak from L1 [67, Appendix A], was never a limitation in practice. The same effect that allowed Meltdown to leak data from L3, enables our slightly modified Foreshadow attack to leak data from L3 as well, *i.e.*, L1TF was in practice never restricted to the L1 cache. Worse still, we show that for the same reason Foreshadow mitigations [67, 70] are still incomplete. We reveal that Foreshadow attacks are unmitigated on many kernel versions even with all mitigations and even on the most recent kernel versions. However, retpoline affects the success rate, but it is only enabled on some kernel versions and some microarchitectures.

We present a new technique that uses dereferencing gadgets to directly leak data without an encoding attack step. We show that we can leak data from registers, e.g., cryptographic key material, from SGX and that the assumptions in previous works were incorrect, making certain attacks only reproducible on kernels susceptible to speculative dereferencing, including, e.g., results from Gruss et al. [16, §3.3 and §5], Lipp et al. [41, §6.2], and Xiao et al. [73, §4-E]. This also allowed us to improve the performance of address-translation attacks and to mount them in JavaScript [16]. We demonstrate that the address-translation attack also works on recent Intel CPUs with the latest hardware mitigations with all mitigations enabled. Finally, we also demonstrate the attack on CPUs previously believed to be unaffected by the prefetch address-translation attack, *i.e.*, ARM, IBM Power9, and AMD CPUs.

Contributions The main contributions of this work are:

1. We discover an incorrect attribution of the root cause in previous works to prefetching effects [16, 41, 73].

2. We show that the root cause is speculative execution, leaving CPUs from other vendors equally affected and the effect exploitable from JavaScript.
3. We discover a novel way to exploit speculative dereferences, enabling direct leakage data in registers.
4. We show that this effect, responsible for Meltdown from non-L1 data, can be adapted to Foreshadow and show that Foreshadow attacks on data from the L3 cache are possible, even with Foreshadow mitigations enabled.

Outline Section 5.2 provides background. Section 5.3 analyzes the root cause. Section 5.4 improves and extends the attacks. Section 5.5 presents cross-VM data leakage. Section 5.6 presents a new leakage method. Section 5.7 presents a JavaScript-based attack. Section 5.8 discusses implications. Section 5.9 concludes.

5.2. Background and Related Work

In this section, we provide relevant details regarding virtual memory, CPU caches, Intel SGX, and transient execution attacks and defenses.

Virtual Memory In modern systems, each process has its own virtual address space, divided into user and kernel space. Many operating systems map physical memory directly into the kernel [29, 39], e.g., to access paging structures. Thus, every user page is mapped at least twice: in user space and in the kernel direct-physical map. Access to virtual-to-physical address information requires root privileges [33]. The prefetch address-translation attack [16, §3.3 and §5] obtains the physical address for any user-space address via a side-channel attack.

Caches and Prefetching Modern CPUs have multiple cache levels, hiding latency of slower memory levels. Software prefetch instructions hint the CPU that a memory address should already be fetched into the cache early to improve performance. Intel and AMD x86 CPUs have 5 software `prefetch*` instructions.

Prefetching attacks Gruss et al. [16] observed that software prefetches appear to succeed on inaccessible memory. Using this effect on the kernel direct-physical map enables the user to fetch arbitrary physical memory into the cache. The attacker guesses the physical address for a user-space address, tries to prefetch the corresponding address in the kernel's direct-physical map, and then uses Flush+Reload [74] on the user-space address. On a cache hit, the guess was correct. Hence, the attacker can determine the exact physical address for any virtual address, re-enabling various mircorarchitectural attacks [31, 43, 49, 60].

Intel SGX Intel SGX is a trusted execution mechanism enabling the execution of trusted code in a separate protected area called an enclave [22]. Although enclave memory is mapped in the virtual address space of the host application, the hardware prevents access to the code or data of the enclave from any source other than the enclave code itself [27]. However, as has been shown in the past, it is possible to exploit SGX via memory corruption [37, 53], ransomware [58], side-channel attacks [5, 54], and transient-execution attacks [51, 55, 67, 68].

Transient Execution Modern CPUs execute instructions *out of order* to improve performance and then retire *in order* from reorder buffers. Another performance optimization, speculative execution, predicts control flow and data flow for not-yet resolved conditional control- or data-flow changes. Intel CPUs have several branch predictors [21], e.g., the Branch History Buffer (BHB) [3, 34], Branch Target Buffer (BTB) [11, 34, 38], Pattern History Table (PHT) [12, 34], and Return Stack Buffer (RSB) [12, 35, 42]. Instructions executed out-of-order or speculatively but not architecturally are called *transient instructions* [41].

These *transient instructions* can have measurable side effects, e.g., modification of TLB and cache state, that can be exploited to extract secrets in so-called transient-execution attacks [6, 26]. Spectre-type attacks [7, 18, 32, 34, 35, 42, 57] exploit misspeculation in a victim context. By executing along the misspeculated path, the victim inadvertently leaks information to the attacker. To mitigate Spectre-type attacks several mitigations were developed [25], such as retpoline [24], which replaces indirect jump instructions with `ret` instructions.

In Meltdown-type attacks [41], such as Foreshadow [67], an attacker deliberately accesses memory across isolation boundaries, which is possible

due to deferred permission checks in out-of-order execution. Foreshadow exploits a cleared present bit in the page table-entry to leak data from the L1 cache or the line fill buffer [51, 55]. A widely accepted mitigation is to flush the L1 caches and line fill buffers upon context switches and to disable hyperthreading [23].

5.3. From Address-Translation Attack to Foreshadow-L3

In this section, we systematically analyze the properties of the address-translation attack erroneously attributed to the software prefetch instructions [16, §3.3 and §5]. We identify the root cause to be unmitigated misspeculation in the kernel, leading to a new Foreshadow-L3 attack that works despite mitigations [67].

In the address-translation attack [16] the attacker tries to find a direct physical map address $\bar{p}$ for a virtual address p. The attacker flushes the user-space address p, and prefetches the inaccessible direct physical map address $\bar{p}$. If Flush+Reload [74] determines that p was reloaded via $\bar{p}$, the physical address of p is $\bar{p}$ minus the known direct-physical-map offset. We measure the attack performance in *fetches per second*, *i.e.*, how often per second p was cached via $\bar{p}$.

The prefetching component of the original attack's proof-of-concept [19] runs a loop, `for (size_t i = 0; i < 3; ++i) { sched_yield(); prefetch(direct_phys_map_addr); }`.

The compiled and disassembled code can be found in Listing 5.3.1. We extracted the following hypotheses(H1-H5) from the original attack (cf. Section A for quotes):

H1 the `prefetch` instruction (to instruct the prefetcher to prefetch);
H2 the value stored in the register used by the `prefetch` instruction (to indicate which address the prefetcher should prefetch);
H3 the `sched_yield` syscall (to give time to the prefetcher);
H4 the use of the `userspace_accessible` bit (as kernel addresses could otherwise not be translated in a user context);
H5 an Intel CPU – other CPU vendors are claimed to be unaffected.

We test each of the above hypotheses in this section.

```
1    41 0f 18 06   prefetchnta (%r14) ; replace with nop for testing, r14 = direct
       phys. addr.
2    41 0f 18 1e   prefetcht2 (%r14)  ; replace with nop for testing, r14 = direct
       phys. addr.
```

Listing 5.3.1.: Disassembly of the prefetching in the address-translation attack.

5.3.1. H1: Prefetch instruction required

The first hypothesis is that the `prefetch` instruction is necessary for the address-translation attack. We replace the `prefetch` instructions in the original code [19] with same-size `nop`s (cf. Listing 5.3.1). Surprisingly, we observe no change in the number of cache fetches, *i.e.*, we measure 60 cache fetches per second (i7-8700K, Ubuntu 18.10, kernel `4.15.0-55`), without any `prefetch` instruction. We also exclude the hardware prefetcher by disabling them via the model-specific register `0x1a4` [69] during the experiment. We still observe ≈ 60 cache fetches per second.

Documented prefetchers are not required for the address-translation attack.

5.3.2. H2: Values in registers required

The second hypothesis is that providing the direct-physical map address via the register is necessary. The registers that must be used vary across kernel versions. We identified the registers `r12,r13,r14` (Ubuntu 18.10, kernel *4.18.0-17*), *r9,r10* (Debian 8, kernel *4.19.28-2* and Kali Linux, kernel *5.3.9-1kali1*) and `rdi,rdx` (Linux Mint 19, kernel `4.15.0-52`). Gruss et al. [16] used recompiled binaries that used different registers for the kernel address (cf. Section A).

A referenced location is only fetched into the cache if the absolute virtual address is stored in one of these registers.

We additionally verified that only the absolute virtual address causes this effect. Any other addressing mode for the prefetch instruction does not leak. By loading the address into most general-purpose registers, we observe leakage across all Linux versions, even with KPTI enabled, meaning that the KAISER technique [15] never protected against this attack. Instead, the implementation merely changed the required registers, hiding the effect for a specific binary-kernel combination. On an Intel Xeon Silver 4208 CPU with in-silicon patches against Meltdown [41],

Foreshadow [67], and ZombieLoad [55], we still observe about 30 cache fetches per second on Ubuntu 19.04 (kernel `5.0.0-25`). On Windows 10 (build `1803.17134`), which has no direct physical map, we fill all registers with a kernel address and perform the syscall `SwitchToThread`. We observe ≈ 15 cache fetches per second for our kernel address.

5.3.3. H3: sched_yield required

The third hypothesis is that the `sched_yield` syscall is required. We observe that other syscalls e.g., `gettid`, expose a similar number of cache fetches. This shows that `sched_yield` is not required and can be replaced with other syscalls. To test whether syscalls in the main attack loop are required, we run a address-translation attack without context switches or interrupts and without `sched_yield` on an isolated core. Here, we do not observe any cache fetches (i7-8700K, kernel 4.15.0-55) when running this attack for 10 hours. However, when inducing a large number of context switches using interrupts, we observe about 15 cache fetches per second if the process filling the registers gets interrupted continuously. These hits occur during speculative execution in the interrupt handler, as we validated manually via code changes and fencing in interrupt handlers.

We conclude that the essential part is performing syscalls or interrupts while specific registers are filled with an attacker-chosen address.

5.3.4. H4: userspace_accessible bit required

The fourth hypothesis is that user-mapped kernel pages are required, *i.e.*, access is prevented via the `userspace_accessible` bit. We constructed an experiment where we allocate several pages of memory. We choose cache lines A and B on different pages. In a loop, we dereference a register pointing to A and use Flush+Reload to detect whether A was cached. In the last loop iteration, we speculatively exchange the register value to point to either B or the direct-physical map address of B. Hence, both the architectural and speculative dereferences happen at the same instruction pointer value and in the same register. With a register-value-based hardware prefetcher, we would expect B to be cached. When dereferencing the direct-physical-map address of B architecturally, B is usually cached after the loop. However, when we dereference the register with its value speculatively changed from A to either B or the direct-physical map address of B, B is never cached after the final run. In a second

```
1  ;<do_syscall_64+106>                    ; with retpoline
2  => 0xffffffff81802000: jmpq   *%rax     callq  0xffffffff8180200c
3  => 0xffffffff8180200c:                  mov    %rax,(%rsp)
4  => 0xffffffff81802010:                  retq
```

Listing 5.3.2.: The kernel performs indirect jumps, e.g., to syscall handlers. With retpoline [64], the kernel uses a `retq` instead of the indirect jump.

experiment, we show that the effect originates from the kernel. While prefetching direct-physical-map addresses works, user-space addresses are only fetched when SMAP (supervisor-mode access prevention) is disabled. Thus, the root cause of the address-translation attack adheres to SMAP.

Hence, we can conclude that the root cause is code execution in the kernel.

5.3.5. H5: Effect only on Intel CPUs

The fifth hypothesis is that the "prefetching" effect only occurs on Intel CPUs. We run our experiments (cf. Section 5.3.4) on an AMD Ryzen Threadripper 1920X (Ubuntu 17.10, kernel `4.13.0-46generic`), an ARM Cortex-A57 (Ubuntu 16.04.6 LTS, kernel `4.4.38-tegra`), and an IBM Power9 (Ubuntu 18.04, kernel `4.15.0-29`). On the AMD Ryzen Threadripper 1920X, we achieve up to 20, on the Cortex-A57 up to 5, and on the IBM Power9 up to 15 speculative fetches per second.

Any Spectre-susceptible CPU is also susceptible to speculative dereferencing.

5.3.6. Speculative Execution in the Kernel

From the previous analysis, we conclude that the leakage is due to speculative execution in the kernel. While this might not be suprising with the knowledge of Spectre, Spectre was only discovered one year after the original prefetch paper [16] was published. We show that the primary leakage is caused by Spectre-BTB-SA-IP (training in same address space, and in-place) [6].

During a syscall, the kernel performs multiple indirect jumps (cf. Listing 5.3.2), which are generally susceptible to Spectre-BTB-SA-IP. The address-translation attack succeeds because misspeculated branch targets dereference registers without sanitization. With retpoline, the kernel uses

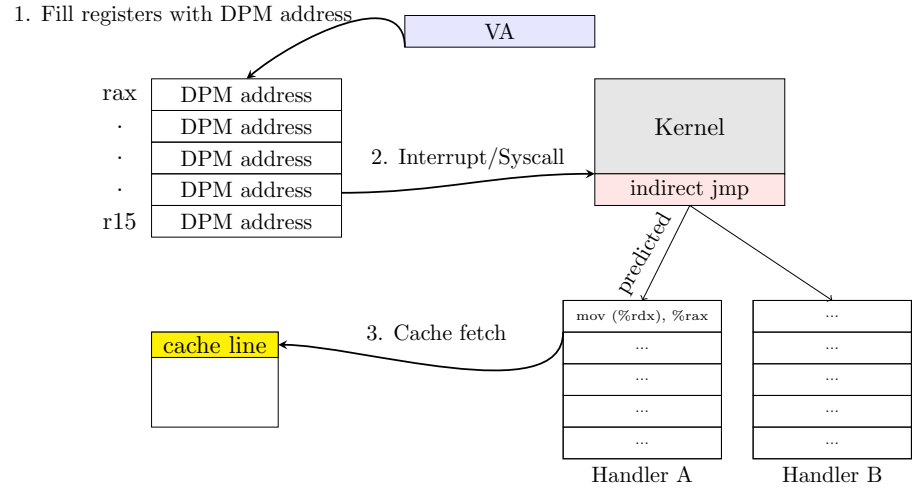


Figure 5.1.: The kernel speculatively dereferences the direct-physical map address. Flush+Reload detects cache hits on the corresponding user-space address.

a `retq` instead of the indirect jump to trap the speculative execution to a fixed branch. Thus, during speculative execution, the CPU might use an incorrect prediction from the branch-target buffer (BTB) and speculate into the wrong syscall while registers contain attacker-chosen addresses (cf. Figure 5.1). In the misspeculated syscall, registers containing attacker-chosen addresses are used. On recent kernels (4.19 or newer), retpoline eliminates the leakage. We provide a full analysis of the `sched_yield` gadget causing speculative dereferences in Section B. Even worse, cloud providers still use older kernel versions (e.g., the first option on AWS at the time of writing is Amazon Linux 2 AMI with kernel 4.14) where retpoline does *not* fully eliminate the leakage. On the other hand, recent systems such as Ice Lake do not use retpoline anymore due to improved hardware mitigations, which unfortunately have *no* effect on our speculative dereferencing attack. Hence, our attack remains unmitigated on many systems, and is most importantly not mitigated by KAISER (KPTI) [15], or LAZARUS [13] as claimed in previous works. The Spectre-BTB-SA-IP leak from Listing 5.3.2 is only one of many, e.g., we still observe $\approx$ 15 speculative fetches per second on an i5-8250U (kernel `5.0.0-20`) if we eliminate this specific leak. However, any prefetch gadget [6], based on PHT, BTB, or RSB mispredictions, can be used for an address-translation attack [16] and thus would also re-enable Foreshadow-VMM attacks [67,

```
1 movzbl (%rax,%rdi,1),%eax
2 <op> (%rcx,%rax,1),%dl
3 ; gadget in Linux kernel
4 98d4be:       0f b6 34 06             movzbl (%rsi,%rax,1),%esi
5 98d4c2:       45 01 3c b3             add    %r15d,(%r11,%rsi,4)
```

Listing 5.3.3.: If the attacker controls three register values, it is possible to leak arbitrary kernel memory.

70]. Concurrent work showed that there are kernel gadgets to fetch data into the L1D cache in Xen [72] and an artificial gadget was exploited by Stecklina for that purpose [63].

We also analyzed the interrupt handling in the Linux kernel version 4.19.0 and observed that the register values from `r8-r15` are cleared but stored on the stack and restored after the interrupt. In between, stack dereferences in misspeculated branches can still access these values. On recent Ice Lake processors, retpoline is replaced by enhanced IBRS. Unfortunately, this is a security regression, re-enabling Spectre-BTB in-place attacks and, thus, moves our focus on a set of previously overlooked gadgets, where the user only controls certain register values in the transient domain. We measure the performance of our attack by exploiting such a Spectre-BTB gadget in a kernel module and evaluate it on our Ice Lake CPU. Listing 5.3.3 illustrates an eIBRS-bypassing Spectre-BTB gadget containing only two instructions, where the attacker controls, e.g., three registers. The smallest eIBRS-bypassing Spectre-BTB gadget we found contains only 7 bytes.

We demonstrate that on Ice Lake, this regression re-enables transient leakage of kernel memory like the original Spectre attack paper described [34], *i.e.*, measured by leaking a 1024 B secret key. We observe a completely noise-free leakage rate of 30 B/s ($n = 1000, \sigma_{\bar{x}} = 0.1429$). By shifting the byte *i.e.*, binary searching via two consecutive cache lines, we then can recover the exact byte value [34]. We analyzed the Linux kernel 5.4.0-48 (vmlinux binary) and looked for similar opcodes and found a gadget at offset `0x984dbe`(see Listing 5.3.3 line `3` and `4`).

5.3.7. Meltdown-L3 and Foreshadow-L3

The speculative dereferencing was noticed but also misattributed to the prefetcher in subsequent work. The Meltdown paper [41] reports that data is fetched from L3 into L1 while mounting a Meltdown attack. Van

Bulck et al. [67] confirmed the effect for Meltdown but did not observe this prefetching effect for Foreshadow. Based on this observation, further works also mentioned this effect without analyzing it thoroughly [6, 30, 47, 51]. Xiao et al. [73] state that a Meltdown-US attack causes data to be repeatedly prefetched from L1 to L3 [73].

We used similar Meltdown-L3 setups as SpeechMiner [73] (kernel 4.4.0-134 with boot flags `nopti,nokaslr` and Meltdown [41] (Ubuntu 16.10, kernel 4.8.0, no mitigations existed back then). In our Meltdown-L3 experiment, one physical core constantly accesses a secret to ensure that the value stays in the L3, as the L3 is shared across all physical cores. On a different physical core, we run Meltdown on the direct-physical map. On recent Linux kernels with full Spectre v2 mitigations implemented, we could not reproduce the result. With the `nospectre_v2` flag, our Meltdown-L3 attack works again by triggering the prefetch gadget in the kernel on the direct-physical map address. In the SpeechMiner [73] and Meltdown [41] experiment, no mitigation (including retpoline) eliminates the leakage fully. Without our new insights that the prefetching effect is caused by speculative execution, it is almost inevitable to not misdesign these experiments, inevitably leading to incomplete or incorrect observations and conclusions on Meltdown and Foreshadow and their mitigations. We confirmed with the authors that their experiment design was not robust to our new insight and therefore lead to wrong conclusions. `Foreshadow-L3`. The same prefetching effect can be used to perform Foreshadow [67]. If a secret is present in the L3 cache and the direct-physical map address is derefenced in the hypervisor kernel, data can be fetched into the L1. This reenables Foreshadow even with Foreshadow mitigations enabled. We demonstrate this attack in KVM in Section 5.5.

5.4. Improving the Leakage Rate

We can leverage our insights to increase the leakage by using syscalls other than `sched_yield`, and executing additional syscalls to mistrain the branch predictor.

Setup We tested our attacks on an Intel i5-8250U (Linux kernel 4.15.0-52), an i7-8700K (Linux kernel 4.15.0-55), an ARM Cortex-A57 (Linux kernel 4.4.38-tegra), and an AMD Threadripper 1920X (Linux kernel 4.13.0-46). As retpoline is not available on all machines, we run the tests

without retpoline. By performing syscalls before filling the registers with the direct-physical map address, we can mistrain the BTB, triggering the CPU to speculatively execute this syscall. The mistraining analysis of `sched_yield` can be seen in the extended version of the paper [59].

Evaluation We evaluated different syscalls for branch prediction mistraining by executing a single syscall before and after filling the registers with the target address. We observe that effects occur for different syscalls and both on AMD and ARM CPUs, with similar success rates (extended version Appendix G) [59]. Alternating syscalls additionally mistrains the branch prediction and increases the success rate, e.g., with syscalls like `stat`, `sendto`, or `geteuid`. However, not every additional syscall increases the number of cache fetches. On recent Linux kernels (version `5`), we observe that the number of speculative cache fetches decreases, due to a change in syscall handling. Our results show that the `pipe` syscall much more reliably triggers speculative dereferencing ($\geq$99.9 %), but the execution time of `sched_yield` is much lower and thus despite the lower success rate (around 66.4 % in the most basic case) it yields a higher attack performance.

Capacity Measurement in a Cross-Core Covert Channel We measure the capacity of our attack in a covert channel by using the speculative dereferencing effect ('1'-bit) or not ('0'-bit). The receiver uses Flush+Reload to measure whether the cache state of cache line dereferenced in the kernel. We evaluated the covert channel on random data and across physical CPU cores. Our test system was equipped with an Intel i7-6500U CPU Linux 4.15.0-52 with the `nospectre_v2` boot flag. We achieved the highest capacity at a transmission rate of 10 bit/s. At this rate, the standard error is, on average, 0.1 %. This result is comparable to related work in similar scenarios [49, 71]. To achieve an error-free transmission, error-correction techniques [43] can be used. I/O interrupts, *i.e.*, syncing the NVMe device, create additional speculative dereferences and can thus further improve the capacity.

5.5. Speculative Dereferences and Virtual Machines

In this section, we examine speculative dereferencing in virtual machines. We demonstrate a successful end-to-end attack using interrupts from a

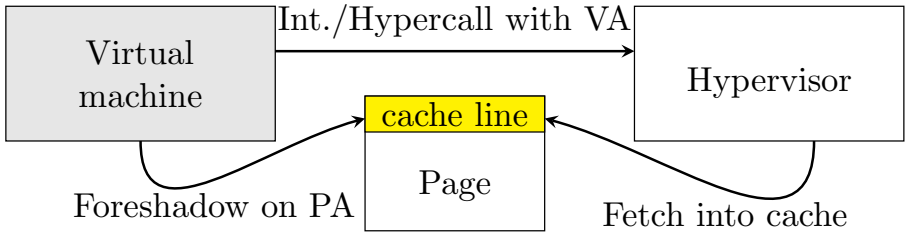


Figure 5.2.: If a guest-chosen address is speculatively fetched into the cache during a hypercall or interrupt and not flushed before the virtual machine is resumed, the attacker can perform a Foreshadow attack to leak the fetched data.

virtual-machine guest running under KVM on a Linux host [36]. We leak data (e.g., cryptographic keys) from other virtual machines and the hypervisor, like the original Foreshadow attack. We do not observe any speculative dereferencing of guest-controlled registers in Microsoft's Hyper-V HyperClear Foreshadow mitigation which additionally uses retpoline, or on more recent kernel versions with retpoline. We provide a thorough analysis of this negative result. However, the attack succeeds even with the recommended Foreshadow mitigations enabled and with kernel versions before 4.18 (e.g., as used by default on AWS Amazon Linux 2 AMI) with all default mitigations enabled, *i.e.*, including retpoline. We investigate whether speculative dereferencing also exists in hypercalls. The attacker targets a specific host-memory location where the host virtual address and physical address are known but inaccessible.

Foreshadow Attack on Virtualization Software If an address from the host is speculatively fetched into the L1 cache on a hypercall from the guest, it has a similar speculative-dereferencing effect. With the speculative memory access in the kernel, we can fetch arbitrary memory from L2, L3, or DRAM into the L1 cache. Consequently, Foreshadow can be used on arbitrary memory addresses provided the L1TF mitigations in use do not flush the entire L1 data cache [63, 65, 72]. Figure 5.2 illustrates the attack using hypercalls or interrupts and Foreshadow. The attacking guest loads a host virtual address into the registers used as hypercall parameters and then performs hypercalls. If there is a prefetching gadget in the hypercall handler and the CPU misspeculates into this gadget, the

host virtual address is fetched into the cache. The attacker then performs a Foreshadow attack and leaks the value from the loaded virtual address.

5.5.1. Foreshadow on Patched Linux KVM

Concurrent work showed that prefetching gadgets in the kernel, in combination with L1TF, can be exploited on Xen and KVM [63, 72]. The default setting on Ubuntu 19.04 (kernel `5.0.0-20`) is to only conditionally flush the L1 data cache upon VM entry via KVM [65], which is also the case for Kali Linux (kernel `5.3.9-1kali1`). The L1 data cache is only flushed in nested VM entry scenarios or in situations where data from the host might be leaked. Since Linux kernel `4.9.81`, Linux's KVM implementation clears all guest clobbered registers to prevent speculative dereferencing [10]. In our attack, the guest fills all general-purpose registers with direct-physical-map addresses from the host.

End-To-End Foreshadow Attack via Interrupts In Section 5.3.3, we observed that context switches triggered by interrupts can also cause speculative cache fetches. We use the example from Section 5.3.3 to verify whether the "prefetching" effect can also be exploited from a virtualized environment. In this setup, we virtualize Linux buildroot (kernel `4.16.18`) on a Kali Linux host (kernel `5.3.9-1kali1`) using qemu (`4.2.0`) with the KVM backend. In our experiment, the guest constantly fills a register with a direct-physical-map address and performs the `sched_yield` syscall. We verify with Flush+Reload in a loop on the corresponding host virtual address that the address is indeed cached. Hence, we can successfully fetch arbitrary hypervisor addresses into the L1 cache on kernel versions before the patch, *i.e.*, with Foreshadow mitigations but incomplete Spectre-BTB mitigations. We observe about 25 speculative cache fetches per minute using NVMe interrupts on our Debian machine. The attacker, running as a guest, can use this gadget to prefetch data into the L1. Since data is now located in the L1, this reenables a Foreshadow attack [67], allowing guest-to-host memory reads. 25 fetches per minute means that we can theoretically leak up to $64 \cdot 25 = 1600$ bytes per minute (or 26.7 bytes per second) with a Foreshadow attack despite mitigations in place. However, this requires a sophisticated attacker who avoids context switches once the target cache line is cached. We develop an end-to-end Foreshadow-L3 exploit that works despite enabled Foreshadow mitigations. In this attack the host constantly performs encryptions using a secret key on a

physical core, which ensures it remains in the shared L3 cache. We assign one isolated physical core, consisting of two hyperthreads, to our virtual machine. In the virtual machine, the attacker fills all registers on one logical core (hyperthread) and performs the Foreshadow attack on the other logical core. Note that this is different from the original Foreshadow attack where one hyperthread is controlled by the attacker and the sibling hyperthread is used by the victim. Our scenario is more realistic, as the attacker controls both hyperthreads, *i.e.*, both hyperthreads are in the same trust domain. With this proof-of-concept attack implementation, we are able to leak 7 bytes per minute successfully [1]. Note that this can be optimized further, as the current proof-of-concept produces context switches regardless of whether the cache line is cached or not. Our attack clearly shows that the recommended Foreshadow mitigations alone are not sufficient to mitigate Foreshadow attacks, and retpoline must be enabled to fully mitigate our Foreshadow-L3 attack.

No Prefetching gadget in Hypercalls in KVM We track the register values in hypercalls and validate whether the register values from the guest system are speculatively fetched into the cache. We neither observe that the direct-physical-map address is still located in the registers nor that it is speculatively fetched into the cache. However, as was shown in concurrent work [63, 72], prefetch gadgets exist in the kernel that can be exploited to fetch data into the cache, and these gadgets can be exploited using Foreshadow.

5.5.2. Negative Result: Foreshadow on Hyper-V HyperClear

We examined whether the same attack also works on Windows 10 (build `1803.17134`), which includes the latest patch for Foreshadow. As on Linux, we disabled retpoline and tried to fetch hypervisor addresses from guest systems into the cache. Microsoft's Hyper-V HyperClear Mitigation [45] for Foreshadow claims to only flush the L1 data cache when switching between virtual cores. Hence, it should be susceptible to the same basic attack we described at the beginning of this section. For our experiment, the attacker passes a known virtual address of a secret variable from the host operating system for all parameters of a hypercall. However, we could not find any exploitable timing difference after switching from the guest

[1] Demonstration video can be found here: https://streamable.com/8ke5ub

to the hypervisor. The extended version discusses the negative result in Appendix F [59].

5.6. Leaking Values from SGX Registers

In this section, we present a novel method, *Dereference Trap*, to leak register contents via speculative register dereference. Leaking the values of registers is useful, e.g., to extract parts of keys from cryptographic operations.

5.6.1. Dereference Trap

The setup for *Dereference Trap* is similar as in Section 5.3.6. We exploit transient code paths inside an SGX enclave that speculatively dereference a register containing a secret value. Such a gadget is easily introduced in an enclave, e.g., when using polymorphism in C++. The extended version contains a minimal example of such a gadget (Appendix C, Listing 5) [59]. However, there are also many different causes for such gadgets [24], e.g., function pointers or (compiler-generated) jump tables. The basic idea of *Dereference Trap* is to ensure that the entire virtual address space of the application is mapped. Thus, if a register containing a secret is speculatively dereferenced, the corresponding virtual address is cached. The attacker can detect which virtual address is cached and infer the secret. However, it is infeasible to back every virtual address with unique physical pages and mount Flush+Reload on every cache line, as that takes 2 days on a 4 GHz CPU [53].

Instead of mapping every page in the virtual address space to its own physical pages, we only map 2 physical pages $p1$ and $p2$, as illustrated in Figure 5.3. By leveraging shared memory, we can map one physical page multiple times into the virtual address space. The maximum number of mappings per page is $2^{31} - 1$, which makes it possible to map $1/16^{th}$ of the user-accessible virtual address space. If we only consider 32-bit secrets, *i.e.*, secrets which are stored in the lower half of 64-bit registers, 2^{20} mappings are sufficient. Out of these, the first 2^{10} virtual addresses map to physical page $p1$ and the second 2^{10} addresses map to page $p2$. Consequently the majority of 32-bit values are now valid addresses that either map to $p1$ or $p2$. Thus, after a 32-bit secret is speculatively dereferenced inside the enclave, the attacker only needs to probe the 64 cache lines of each of the

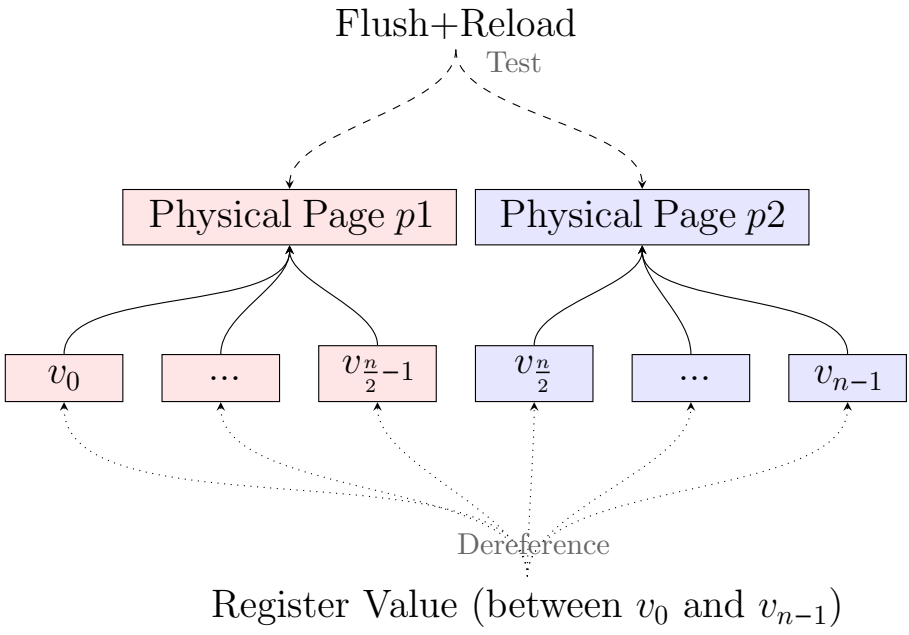


Figure 5.3.: Leaking the value of an x86 general-purpose register using *Dereference Trap* and Flush+Reload on two different physical addresses. v_0 to v_{n-1} represent the memory mappings on one of the shared memory regions.

two physical pages. A cache hit reveals the most-significant bit (bit 31) of the secret as well as bits 6 to 11, which define the cache-line offset on the page. To learn the remaining bits 12 to 30, we continue in a fashion akin to binary-search. We unmap all mappings to $p1$ and $p2$ and create half as many mappings as before. Again, half of the new mappings map to $p1$ and half of the new mappings map to $p2$. From a cache hit in this setup, we can again learn one bit of the secret. We can repeat these steps until all bits from bit 6 to 31 of the secret are known. As the granularity of Flush+Reload is one cache line, we cannot leak the least-significant 6 bits of the secret. On our test system, we recovered a 32-bit value (without the least-significant 6 bits) stored in a 64-bit register within 15 minutes with *Dereference Trap*.

5.6.2. Generalization of Dereference Trap

Dereference Trap is a generic technique that applies to any scenario where the attacker can set up the hardware and address space accordingly. *Dereference Trap* applies to all Spectre variants. Many in-place Spectre-v1 gadgets that are not the typical encoding array gadget are still entirely unprotected with no plans to change this. For instance, Intel systems before Haswell and AMD systems before Zen do not support SMAP, and more recent systems may have SMAP disabled. On these systems, we can also `mmap` memory regions and the kernel will dereference 32-bit values misinterpreted as pointers (into user space). Using this technique the attacker can reliably leak a 32-bit secret which is speculatively dereferenced by the kernel. Cryptographic implementations often store keys in the lower 32 bits of 64bit registers (*i.e.*, specderef:OpenSSL AES round key u32 *rk) [62]. Hence, these implementations might be susceptible to *Dereference Trap*. We evaluated the same experiment on an Intel i5-8250U, ARM Cortex-A57, and AMD ThreadRipper 1920X with the same result of 15 minutes to recover a 32-bit secret (without the least-significant 6 bits). Thus, retpoline and SMAP must remain enabled to mitigate attacks like *Dereference Trap*.

5.7. Leaking Physical Addresses from JavaScript using WebAssembly

In this section, we present an attack that leaks the physical address (cache-line granularity) of a JavaScript variable. This shows that the "prefetching" effect is much simpler than described in the original paper [16], *i.e.*, *it does not require native code execution*. The only requirement for the environment is that it can keep a 64-bit register filled with an attacker-controlled 64-bit value. In contrast to the original paper's attempt to use native code in browser, we create a JavaScript-based attack to leak physical addresses from Javascript variables and evaluate its performance in Firefox. We demonstrate that it is possible to fill 64-bit registers with an attacker-controlled value via WebAssembly.

Attack Setup JavaScript encodes numbers as 53-bit double-precision floating-point values [46]. It is not possible to store a full 64-bit value into a register with vanilla JavaScript. Hence, we leverage WebAssembly, a

binary instruction format which is precompiled for the JavaScript engine and not further optimized [66]. On our test system (i7-8550U, Debian 8, kernel`5.3.9-1kali1`) registers `r9` and `r10` are speculatively dereferenced in the kernel. Hence, we fill these registers with a guessed direct-physical-map address of a variable. The WebAssembly method `load_pointer` (Appendix F [59]) takes two 32-bit values that are combined into a 64-bit value and populated into as many registers as possible. To trigger interrupts, we use web requests, as shown by Lipp et al. [40]. Our attack leaks the direct-physical-map address of a JavaScript variable. The attack works analogously to the native-code address-translation attack [16].

1. Guess a physical address p for the variable and compute the corresponding direct-physical map address $d(p)$.
2. Load $d(p)$ into the required registers (`load_pointer`) in an endless loop, e.g., using endless-loop slicing [40].
3. The kernel fetches $d(p)$ into the cache when interrupted.
4. Use Evict+Reload on the target variable. On a cache hit, the physical address guess p from Step 1 was correct. Otherwise, continue with the next guess.

Attack from within Browsers We mount an attack in an unmodified Firefox 76.0 by injecting interrupts via web requests. We observe up to 2 speculative fetches per hour. If the logical core running the code is constantly interrupted, e.g., due to disk I/O, we achieve up to 1 speculative fetch per minute. As this attack leaks parts of the physical and virtual address, it can be used to implement various microarchitectural attacks [14, 17, 34, 48, 49, 52, 56]. Hence, the address-translation attack is possible with JavaScript and WebAssembly, without requiring the NaCl sandbox as in the original paper [16]. Upcoming JavaScript extensions expose syscalls to JavaScript [8]. Hence, as the second part of our evaluation, we investigate whether a syscall-based attack would also yield the same performance as in native code. To simulate the extension, we expose the `sched_yield` syscall to JavaScript. We observe the same performance of 20 speculative fetches per second with the syscall function.

Limitations of the Attack We conclude that the bottleneck of this attack is triggering syscalls. In particular, there is currently no way to directly perform a single syscall via Javascript in browsers without high overhead. We traced the syscalls of Firefox using `strace`. We observed that syscalls

such as `sched_yield, getpid, stat, sendto` are commonly performed upon `window` events, e.g., opening and closing pop-ups or reading and writing events on the JavaScript console. However, the registers `r9` and `r10` get overwritten before the syscall is performed. Thus, whether the registers are speculatively dereferenced while still containing the attacker-chosen values strongly depends on the engine's register allocation and on other syscalls performed. As Jangda et al. [28] stated, not all registers are used in JIT-generated native code [28].

5.8. Discussion

The "prefetching" effect was first observed by Gruss et al. [16] in 2016. In May 2017, Jann Horn discovered that speculative execution can be exploited to leak arbitrary data, later on published in the Spectre [34] paper. Our results indicate that the address-translation attack was the first inadvertent exploitation of speculative execution, albeit in a much weaker form where only metadata, *i.e.*, information about KASLR, is leaked rather than real data as in a full Spectre attack. Even before the address-translation attack, speculative execution was well known [50] and documented [22] to cause cache hits on addresses that are not architecturally accessed. Currently, the address-translation attack and our variants are mitigated on both Linux and Windows using the retpoline technique to avoid indirect branches. Another possibility upon a syscall is to save user-space register values to memory, clear the registers to prevent speculative dereferencing, and later restore the user-space values after execution of the syscall. However, as has been observed in the interrupt handler, there might still be some speculative cache accesses on values from the stack. The retpoline mitigation for Spectre-BTB introduces a large overhead for indirect branches. The performance overhead can in some cases be up to 50 % [61]. This is particularly problematic in large scale systems, e.g., cloud data centers, that have to compensate for the performance loss and increased energy consumption. Furthermore, retpoline breaks CET and CFI technologies and might thus also be disabled [4]. As an alternative, randpoline [4] could be used to replace the mitigation with a probabilistic one, again with an effect on Foreshadow mitigations. And indeed, mitigating memory corruption vulnerabilities may be more important than mitigating Foreshadow in certain use cases. Cloud computing concepts that do not rely on traditional isolation boundaries are already being explored [1, 9, 20, 44]. On current CPUs, retpoline must remain enabled,

which is not the default in many cases. Other Spectre-BTB mitigations, including enhanced IBRS, do not mitigate our attack. On newer kernels for ARM Cortex-A CPUs, the branch prediction results can be discarded, and on certain devices branch prediction can be entirely disabled [2]. Our results suggest that these mechanisms are required for context switches or interrupt handling. Additionally, the L1TF mitigations must be applied on affected CPUs to prevent Foreshadow. Otherwise, we can still fetch arbitrary hypervisor addresses into the cache. Finally, our attacks also show that SGX enclaves must be compiled with the retpoline flag. Even with LVI mitigations, this is currently not the default setting, and thus all SGX enclaves are potentially susceptible to *Dereference Trap*.

5.9. Conclusion

We showed that the underlying root cause of prefetching effects was misattributed in previous works [6, 15, 30, 41, 47, 51, 67] and speculative dereferencing of a user-space register in the kernel actually causes the leakage. As a result, we were able to mount a Foreshadow (L1TF) attack on data from the L3 cache, even with the latest mitigations enabled. Furthermore, we were able to improve the performance of the original attack, apply it to AMD, ARM, and IBM and exploit the effect via JavaScript in browsers. Our novel technique, *Dereference Trap*, leaks the values of registers used in SGX (or privileged contexts) via speculative dereferencing.

Acknowledgments

We want to thank Moritz Lipp, Clémentine Maurice, Anders Fogh, Xiao Yuan, Jo Van Bulck, and Frank Piessens of the original papers for reviewing and providing feedback to drafts of this work and for discussing the technical root cause with us. Furthermore, we want to thank Intel and ARM for valuable feedback on an early draft. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from Cloudflare, Intel and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

Appendix

A. Extracting Hypotheses from Previous Works

The hypotheses are extracted from previous works as detailed in this section. The footnotes for each hypothesis provide the exact part of the previous work that we reference.

H1 the `prefetch` instruction (to instruct the prefetcher to prefetch);[1]
H2 the value stored in the register used by the `prefetch` instruction (to indicate which address the prefetcher should prefetch);[2]
H3 the `sched_yield` syscall (to give time to the prefetcher);[3]
H4 the use of the `userspace_accessible` bit (as kernel addresses could otherwise not be translated in a user context);[4]
H5 an Intel CPU – the "prefetching" effect only occurs on Intel CPUs, and other CPU vendors are not affected.[5]

The original paper also describes that "delays were introduced to lower the pressure on the prefetcher" [16]. In fact, this was done via recompilation. Note that recompilation with additional code inserted may have side effects such as a different register allocation, which we find to be an important influence factor to the attack.

B. Actual Spectre V2 gadget in Linux kernel

We analyzed the Linux kernel 4.16.18 and used the GNU debugger(GDB) to debug our kernel. As our target syscall we analyzed the path of the `sched_yield` syscall. We used the same experiment, which fills all general-purpose registers with the corresponding DPM address, perform

[1] "Our attacks are based on weaknesses in the hardware design of prefetch instructions" [16].

[2] "2. Prefetch (inaccessible) address $\bar{p}$. 3. Reload p. [...] the *prefetch of $\bar{p}$ in step 2 leads to a cache hit* in step 3 with a high probability." [16] with emphasis added.

[3] "[...] delays were introduced to lower the pressure on the prefetcher." [16]. These delays were implemented using a different number of `sched_yield` system calls, as can also be seen in the original attack code [19].

[4] "Prefetch can fetch inaccessible privileged memory into various caches on Intel x86." [16] and corresponding NaCl results.

[5] "[...] we were not able to build an address-translation oracle on [ARM] Android. As the prefetch instructions do not prefetch kernel addresses [...]" [16] describing why it does not work on ARM-based Android devices.

`sched_yield` and verify the speculative dereference with Flush+Reload. We repeat this experiment 10 000 000 times. We analyzed each indirect branch in this code path and replaced the indirect call/jump with a retpolined version. Furthermore, we analyzed all general-purpose registers and traced their content if the DPM-address is still valid in some registers. By systematically retpolining the indirect branches, we observed that the indirect call `current->sched_class->yield_task(rq);` in the function `sys_sched_yield` causes the main leakage. We set a breakpoint to this function and observed that four general-purpose registers (`%rcx,%rsi,%r8,%r9`) still contain the kernel address we set in our experiment.

In the function `put_prev_task_fair`, the `%rsi` register is dereferenced. To check whether this dereference cause the leakage, we add an `lfence` instruction at the beginning of the function. We run the same experiment again and observe almost no cache fetches on our address. The `%rsi` register is dereferenced in line 48

C. Mistraining BTB for `sched_yield`

We evaluate the mistraining of the BTB by calling different syscalls, fill all general-purpose registers with direct-physical map address and call `sched_yield`. Our test system was equipped with Ubuntu 18.04 (kernel 4.4.143-generic) and an Intel i7-6700K. We repeated the experiment by iterating over various syscalls with different parameters (valid parameters,NULL as parameters) 10 times with 200 000 repetitions. Table 5.1 lists the best 15 syscalls to mistrain the BTB when sched_yield is performed afterwards. On this kernel version it appears that the read and getcwd syscalls mistraing the BTB best if sched_yield is called after the register filling.

D. Speculative Dereference Gadget in SGX

Listing 5.9.1 shows a minimal example of introducing a speculative-dereference gadget that can be used for *Dereference Trap* (cf. Section 5.6). The virtual functions are implemented using *vtable*s for which the compiler emits an indirect call in Section D. The branch predictor for this indirect call learns the last call target. Thus, if the call target changes because the type of the object is different, speculative execution still executes the function of the last object with the data of the current object.

Table 5.1.: Table of syscalls which achieve the highest numbers of cache fetches, when calling sched_yield after the register filling.

Syscall	Parameters	Avg. # cache fetches
readv	readv(0,NULL,0);	13766.3
getcwd	syscall(79,NULL,0);	7344.7
getcwd	getcwd(NULL,0);	6646.9
readv	syscall(19,0,NULL,0);	5541.4
mount	syscall(165,s_cbuf,s_cbuf,s_cbuf,s_ulong,(void*)s_cbuf);	4831.6
getpeername	syscall(52,0,NULL,NULL);	4600.0
getcwd	syscall(79,s_cbuf,s_ulong);	4365.8
bind	syscall(49,0,NULL,0);	3680.6
getcwd	getcwd(s_cbuf,s_ulong);	3619.3
getpeername	syscall(52,s_fd,&s_ssockaddr,&s_int);	3589.3
connect	syscall(42,s_fd,&s_ssockaddr,s_int);	2951.2
getpeername	getpeername(0,NULL,NULL);	2822.4
connect	syscall(42,0,NULL,0);	2776.4
getsockname	syscall(51,0,NULL,NULL);	2623.4
connect	connect(0,NULL,0);	2541.5

```
1  class Object {
2  public:  virtual void print() = 0;
3  };
4  class Dummy : public Object {
5  private: char* data;
6  public:  Dummy() { data = "TEST"; }
7           virtual void print() { puts(data); }
8  };
9  class Secret : public Object {
10 private: size_t secret;
11 public:  Secret() { secret = 0x12300000; }
12          virtual void print() { }
13 };
14 void printObject(Object* o) { o->print(); }
```

Listing 5.9.1.: Speculative type confusion which leaks the secret of `Secret` class instances using *Dereference Trap*.

In this code, calling `printObject` first with an instance of `Dummy` mistrains the branch predictor to call `Dummy::print`, dereferencing the first member of the class. A subsequent call to `printObject` with an instance of `Secret` leads to speculative execution of `Dummy::print`. However, the dereferenced member is now the secret (Section D) of the `Secret` class.

The speculative type confusion in such a code construct leads to a speculative dereference of a value which would never be dereferenced architecturally. We can leak this speculatively dereferenced value using the *Dereference Trap* attack.

E. WebAssembly Register Filling

```
1  extern void yield_wrapper();
2  uint64_t G1 = 5, G2 = 5, G3 = 5, G4 = 5, G5 = 5, value = 0;
3  void spec_fetch() {
4    for (uint64_t i = G1+5; i > G1; i--)
5      for (uint64_t k = G3+5; k > G3; k--)
6        for (uint64_t j = G2-5; k < G2; j++)
7          for(uint64_t l = G4; i < G4;l++)
8            for(uint64_t m = G5-5;m<G5;m++)
9              value = l + j + k + i;
10     yield_wrapper();
11 }
12 int load_pointer(int high, int low) {
13   uint64_t a = (((uint64_t)high) << 32ull) | ((uint64_t)(unsigned
       int)low);
14   G1 = a;
15   G2 = a;
16   G3 = a;
17   G4 = a;
18   G5 = a;
19   spec_fetch();
20   return a;
21 }
22 int main() {
23   load_pointer(0x12345678,0x9abcdef0);
24 }
```

Listing 5.9.2.: WebAssembly code to speculatively fetch an address from the kernel direct-physical map into the cache. We combine this with a state-of-the-art Evict+Reload loop in JavaScript to determine whether the guess for the direct-physical map address was correct.

The WebAssembly method `load_pointer` of Listing 5.9.2 takes two 32-bit JavaScript values as input parameters. These two parameters are loaded

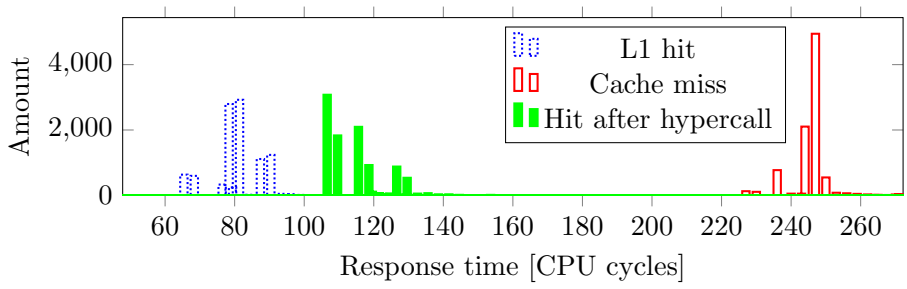


Figure 5.4.: Timings of a cached and uncached variable and the access time after a hypercall in a Ubuntu VM on Hyper-V.

into a 64-bit integer variable and stored into multiple global variables. The global variables are then used as loop exit conditions in the separate loops. To fill as many registers as possible with the direct-physical-map address, we create data dependencies within the loop conditions. In the `spec_fetch` function, the registers are filled inside the loop. After the loop, the JavaScript function `yield_wrapper` is called. This tries to trigger any syscall or interrupt in the browser by calling JavaScript functions which may incur syscalls or interrupts. Lipp et al. [40] reported that web requests from JavaScript trigger interrupts from within the browser.

F. No Foreshadow on Hyper-V HyperClear

We set up a Hyper-V virtual machine with a Ubuntu 18.04 guest (kernel `5.0.0-20`). We access an address to load it into the cache and perform a hypercall before accessing the variable and measuring the access time. Since hypercalls are performed from a privileged mode, we developed a kernel module for our Linux guest machine which performs our own malicious hypercalls. We observe a timing difference (see Figure 5.4) between a memory access which hits in the L1 cache (dotted), a memory access after a hypercall (grid pattern), and an uncached memory access (crosshatch dots). We observe that after each hypercall, the access times are $\approx$ 20 cycles slower. This indicates that the guest addresses are flushed from the L1 data cache. In addition, we create a second experiment where we load a virtual address from a process running on the host into several registers when performing a hypercall from the guest. On the host system, we perform Flush+Reload on the virtual address in a loop and verify whether the virtual address is fetched into the cache. We do not observe any cache hits on the host process when performing hypercalls from the

guest system. Thus we conclude that either the L1 cache is always flushed, contradicting the documentation, or creating a situation where the L1 cache is not flushed requires a more elaborate attack setup. However, we believe that speculative dereferencing is the reason why Microsoft adopted the retpoline mitigation despite having other Spectre-BTB mitigations already in place.

G. Evaluation Framework for Speculative Dereferencing in Syscalls

Table 5.2.: F1-Scores for speculative cache fetches with different syscalls on different CPU architectures.

Syscall	Syscall executed before	i5-8250U	i7-8700K	Threadripper 1920X	Cortex-A57
sched_yield	None	66.40 %	91.49 %	99.29 %	76.61 %
	send-to	56.42 %	4.60 %	52.94 %	44.88 %
	geteuid	46.62 %	1.90 %	63.94 %	48.82 %
	stat	77.37 %	57.44 %	69.28 %	63.57 %
pipe	None	100.00 %	99.35 %	100.00 %	100.00 %
	send-to	99.90 %	99.60 %	100.00 %	100.00 %
	geteuid	99.90 %	99.61 %	100.00 %	100.00 %
	stat	99.90 %	99.55 %	99.90 %	100.00 %
read	None	10.42 %	0.09 %	8.50 %	57.95 %
	send-to	14.47 %	21.26 %	1.90 %	78.86 %
	geteuid	15.32 %	56.73 %	2.35 %	73.73 %
	stat	28.32 %	24.07 %	9.70 %	23.32 %
write	None	7.69 %	91.24 %	76.46 %	58.95 %
	send-to	14.29 %	9.88 %	11.00 %	45.68 %
	geteuid	15.49 %	32.21 %	52.94 %	49.47 %
	stat	9.16 %	9.70 %	52.83 %	12.03 %
nanosleep	None	21.20 %	27.43 %	52.61 %	87.40 %
	send-to	46.59 %	13.43 %	76.23 %	82.83 %
	geteuid	29.93 %	96.05 %	89.62 %	69.63 %
	stat	59.84 %	99.14 %	89.68 %	77.67 %

We created a framework that runs the experiment from Section 5.3.4 with 20 different syscalls (after filling the registers) and computes the F1-score. We perform different syscalls before filling the registers to mistrain the branch prediction. One direct-physical-map address has a corresponding mapping to a virtual address and should trigger speculative fetches into the cache. The other direct-physical-map address should not produce any cache hits on the same virtual address. If there is a cache hit on the correct virtual address, we count it as a true positive. Conversely, if there is no hit when there should have been one, we count it as a false negative. On the second address, we count the false positives and true negatives. For syscalls with parameters, e.g., `mmap`, we set the value of all parameters to the direct-physical-map address, *i.e.*, `mmap(addr, addr, addr, addr, addr, addr)`. We repeat this experiment 1000 times for each syscall on each system and compute the F1-Score. Table 5.2 lists the results of our evalution. As can be seen the `pipe` syscall achieves the highest F1-Score.

References

[1] Amazon AWS. *AWS Lambda@Edge*. 2019. URL: `https://aws.amazon.com/lambda/edge/`.

[2] ARM. *ARM: Whitepaper Cache Speculation Side-channels*. 2018. URL: `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper`.

[3] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. "Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls." In: *Cryptology ePrint Archive, Report 2017/968* (2017).

[4] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. *Efficient mitigation of side-channel based attacks against speculative execution processing architectures*. US Patent App. 16/023,564. 2019.

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

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/. 2019.

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution." In: *EuroS&P*. 2019.

[8] Chromium. *Mojo in Chromium*. 2020. URL: `https://chromium.googlesource.com/chromium/src.git/+/master/mojo/README.md`.

[9] Cloudflare. *Cloudflare Workers*. 2019. URL: `https://www.cloudflare.com/products/cloudflare-workers/`.

[10] Elixir bootlin. 2018. URL: `{https://elixir.bootlin.com/linux/latest/source/arch/x86/kvm/svm.c#L5700}`.

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

[12] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2016.

[13] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. "LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization." In: *RAID*. 2017.

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

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

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

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

[18] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[19] IAIK. *Prefetch Side-Channel Attacks V2P*. 2016. URL: {https://github.com/IAIK/prefetch/blob/master/v2p/v2p.c}.

[20] IBM. 2019. URL: https://cloud.ibm.com/functions/.

[21] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[22] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.

[23] Intel. *Intel Analysis of Speculative Execution Side Channels*. Revision 4.0. 2018.

[24] Intel. *Retpoline: A Branch Target Injection Mitigation*. Revision 003. 2018.

[25] Intel. *Speculative Execution Side Channel Mitigations*. Revision 3.0. 2018.

[26] Intel Corporation. *Refined Speculative Execution Terminology*. 2020. URL: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology.

[27] Intel Corporation. "Software Guard Extensions Programming Reference, Rev. 2." In: (2014).

[28] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code." In: *USENIX ATC*. 2019.

[29] kernel.org. *Virtual memory map with 4 level page tables (x86_64)*. 2009. URL: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[30] Taehyun Kim and Youngjoo Shin. "Reinforcing Meltdown Attack by Using a Return Stack Buffer." In: *IEEE Access* 7 (2019).

[31] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors." In: *ISCA*. 2014.

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

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

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

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

[36] KVM contributors. *Kernel-based Virtual Machine*. 2019. URL: https://www.linux-kvm.org.

[37] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves." In: *USENIX Security Symposium*. 2017.

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

[39] Jonathan Levin. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

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

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

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

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

[44] Microsoft. *Azure serverless computing*. 2019. URL: https://azure.microsoft.com/en-us/overview/serverless-computing/.

[45] Microsoft Techcommunity. *Hyper-V HyperClear Mitigation for L1 Terminal Fault*. 2018. URL: https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429.

[46] Mozilla. *JavaScript data structures*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures.

[47] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. *A Survey of Published Attacks on Intel SGX*. 2020.

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

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

[50] Chester Rebeiro, Debdeep Mukhopadhyay, Junko Takahashi, and Toshinori Fukunaga. "Cache timing attacks on Clefia." In: *International Conference on Cryptology in India*. 2009.

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

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

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

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

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

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

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

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

[59] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. *Speculative Dereferencing of Registers: Reviving Foreshadow*. 2021. URL: https://martinschwarzl.at/media/files/spec_deref_extended.pdf.

[60] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges." In: *Black Hat Briefings*. 2015.

[61] Slashdot EditorDavid. *Two Linux Kernels Revert Performance-Killing Spectre Patches*. 2019. URL: https://linux.slashdot.org/story/18/11/24/2320228/two-linux-kernels-revert-performance-killing-spectre-patches.

[62] specderef:OpenSSL. *specderef:OpenSSL: The Open Source toolkit for SSL/TLS*. 2019. URL: http://www.openssl.org.

[63] Julian Stecklina. *An demonstrator for the L1TF/Foreshadow vulnerability*. 2019. URL: {https://github.com/blitz/l1tf-demo}.

[64] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: https://support.google.com/faqs/answer/7625886.

[65] Ubuntu Security Team. *L1 Terminal Fault (L1TF)*. 2019. URL: https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/L1TF.

[66] V8 team. *v8 - Adding BigInts to V8*. 2018. URL: https://v8.dev/blog/bigint.

[67] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *USENIX Security Symposium*. 2018.

[68] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *S&P*. 2020.

[69] Vish Viswanathan. *Disclosure of Hardware Prefetcher Control on Some Intel Processors*. URL: `https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors`.

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

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

[72] xenbits. *Cache-load gadgets exploitable with L1TF*. 2019. URL: `https://xenbits.xen.org/xsa/advisory-289.html`.

[73] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities." In: *NDSS*. 2020.

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

6

Robust and Scalable Process Isolation Against Spectre in the Cloud

Publication Data

Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. "Robust and Scalable Process Isolation Against Spectre in the Cloud." In: *ESORICS*. 2022

Contributions

Main author.

Robust and Scalable Process Isolation against Spectre in the Cloud

Martin Schwarzl[1], Pietro Borrello[2], Andreas Kogler[2], Kenton Varda[3], Thomas Schuster[1], Daniel Gruss[1], Michael Schwarz[4]

[1]Graz University of Technology, Austria [2]Sapienza University of Rome, Italy [3] Cloudflare Inc. [4]CISPA Helmholtz Center for Information Security, Germany

Abstract

In the quest for efficiency and performance, edge-computing providers replace process isolation with sandboxes, to support a high number of tenants per machine. While secure against software vulnerabilities, microarchitectural attacks can bypass these sandboxes.

In this paper, we present a Spectre attack leaking secrets from co-located tenants in edge computing. Our remote Spectre attack, using amplification techniques and a remote timing server, leaks 2 bit/min. This motivates our main contribution, *DyPrIs*, a scalable process-isolation mechanism that only isolates suspicious worker scripts following a lightweight detection mechanism. In the worst case, DyPrIs boils down to process isolation. Our proof-of-concept implementation augments real-world cloud infrastructure used in production at large scale, *Cloudflare Workers*. With a false-positive rate of only 0.61 %, we demonstrate that DyPrIs outperforms strict process isolation while statistically maintaining its security guarantees, fully mitigating cross-tenant Spectre attacks.

6.1. Introduction

With the recent discovery of transient-execution attacks [7], such as Spectre [34] or Meltdown [37], attackers even leak data, not only metadata. As most transient-execution attacks work across logical CPUs, *i.e.*, hyperthreads, many cloud providers do not assign logical CPUs to different tenants. With the introduction of edge computing [2, 10], where resources are dynamically provided on a machine that is close to the customer, virtualization-based security was replaced by more efficient solutions.

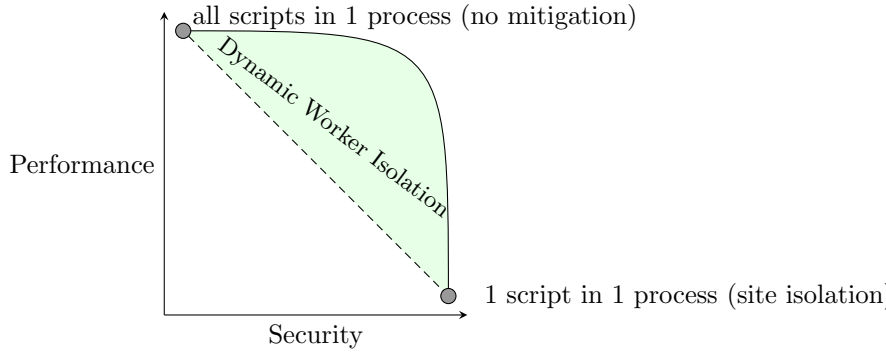


Figure 6.1.: Strict process isolation choses the security and performance trade-off via the number of scripts inside one process (dashed line). DyPrIs improves this trade-off while never being worse than strict process isolation.

Cloud providers either rely on strict process isolation [2, 42], *i.e.*, one process per tenant, or language-level isolation [10, 16, 17], *i.e.*, code is written in a sandboxed language such as JavaScript. While language-level isolation has the least overhead [11], it does not protect against Spectre within the same process [30, 34, 41, 57], necessitating process or site isolation [48]. To avoid these costly countermeasures, *Cloudflare Workers* rely on a modified JavaScript sandbox [10] that disables all known timers and primitives that can be abused to build timers [22, 54]. A similar design using language-level isolation WebAssembly is used by Fastly [17]. As *Cloudflare* is one of the top three edge computing providers, with millions of requests daily, this raises the following scientific question:

Can edge computing without strict process isolation, as is already deployed and widely used today, offer the same security levels with respect to microarchitectural attacks as edge computing with strictly isolated processes?

This paper has an offensive and a defensive contribution: First, we demonstrate that it is possible to steal secrets on *Cloudflare Workers* with 2 bit/min using an amplified Spectre attack [58] relying on an external time server. This proof-of-concept attack shows that language-level isolation is insufficient.

Second, we propose, *DyPrIs (Dynamic Process Isolation)*, a technique that relies on a probabilistic Spectre detection and process-isolates suspicious workloads. DyPrIs is a middle ground between the two extremes of strict process isolation and language-level isolation. Hence, DyPrIs keeps the

performance benefits of language-level isolation for the majority of benign workloads and provides the security guarantees of process isolation against malicious workloads. Even if every workload was classified as Spectre, DyPrIs only boils down to strict process isolation with a the small overhead of 2 % for the detection, but on average, it results in far higher performance (cf. Figure 6.1).

Our detection uses hardware performance counters (HPC) for mispredicted and retired branches. We show that HPC usage, as suggested in prior work [32, 43, 46, 70] has too much overhead for efficiency-driven edge systems. However, we demonstrate that even with a limited set of performance counters, we detect running Spectre attacks with a small performance overhead of 2 %.

We evaluated DyPrIs in a production environment in the cloud. Our result is a false-positive rate of 0.61 %, while detecting all attack attempts with all state-of-the-art techniques. DyPrIs blocks our attack without interrupting any of our own or other workloads.

Contributions. The main contributions of this work are:

1. We demonstrate a remote Spectre attack on the restricted *Cloudflare Workers*, showing that current mitigations are insufficient.
2. We propose a novel, low-overhead probabilistic detection for Spectre attacks.
3. We introduce DyPrIs, a technique with, on average, lower overhead than state-of-the-art strict process isolation.

6.2. Background and Related Work

In modern processors, instructions are divided into multiple micro-operations (μOPs) that are executed out of order. To improve the performance of branch instructions, CPUs leverage speculative execution. For example, the branch prediction unit (BPU) tries to predict whether a branch is taken or not using different data structures, e.g., the Pattern History Table (PHT) [34]. If the prediction was correct, the results of the execution are retired. Otherwise, the speculatively executed instructions are discarded, and the correct code path is executed. Mistakenly executed instructions are called *transient instructions* [7, 37]. They still have an effect on the microarchitecture, e.g., measurable timing differences in the cache that

can be extracted with cache attacks [7, 34, 37]. Cache attacks are even possible in JavaScript [44].

Spectre attacks [34] exploit speculative execution. Spectre-PHT [7] (also known as Spectre V1) exploits the Pattern History Table, which predicts the outcome of a conditional branch [34]. A typical Spectre-PHT gadget is a bounds check, e.g., `if (x < array1_size) y = array2[array1[x] * 4096];`. The attacker controls the index `x`, which is bounds-checked. By mistraining the branch prediction with in-bounds values, speculation follows the in-bounds path with out-of-bounds values, allowing out-of-bounds reads. Spectre variants exploit different prediction mechanisms, e.g., the Branch Target Buffer, memory disambiguation, or the Return-Stack Buffer [29, 34, 35, 38] and have been demonstrated over the network [55] and in JavaScript [34, 41, 57].

Many cache side-channel defenses have been proposed, e.g., focusing on *detection* using HPCs [8, 28, 32, 46, 66, 67, 70, 71]. To detect Spectre-type attacks, static code analysis and patching, taint tracking, symbolic execution, and detection via HPCs were proposed [13, 25–27, 40, 43, 65]. However, these proposals focus on attack detection but do not propose and evaluate mechanisms to respond to detected attacks. Detection methods suffer from false positives but terminating a detected attack is not acceptable for *Cloudflare Workers*.

Cloudflare Workers is an edge computing service to intercept web requests and modify their content using JavaScript, handling millions of HTTP requests per second across tens of thousands of web sites. *Cloudflare Workers* support multiple thousand workers from up to 2000 tenants running inside the same process. Each worker is single-threaded and stateless. This design leads to a high-performant solution based on language-level isolation. To impede microarchitectural attacks, *Cloudflare Workers* restricts the available JavaScript timing functions to only update after a request is performed. Additionally, JavaScript worker threads are disabled to prevent counting threads [22, 34, 54].

6.3. Remote Spectre Attacks on Cloudflare Workers

In this section, we show that the single-address-space design of *Cloudflare Workers* enables remote Spectre attacks. First, we define the Spectre building blocks and overview how a remote adversary can mount a Spectre

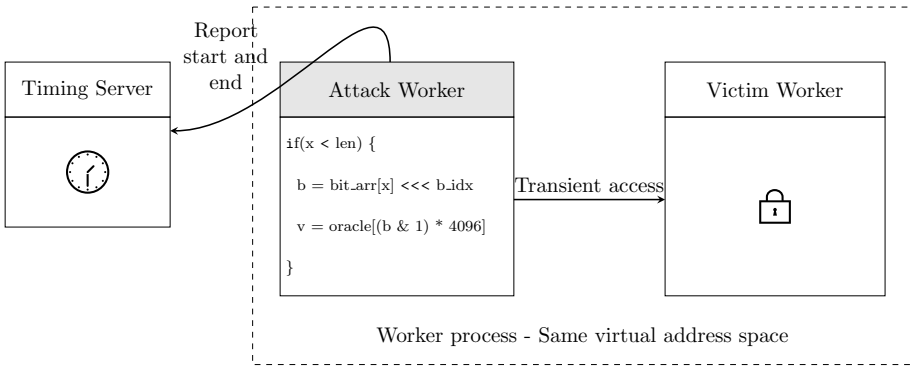


Figure 6.2.: Overview of the *Cloudflare Workers* remote Spectre attack.

attack. Since there is no local timing primitive, a common requirement for microarchitectural attacks [18, 53], we have to resort to a remote timing primitive. Our proof-of-concept implementation running on *Cloudflare Workers* leaks 2 bit/min, even if address space layout randomization (ASLR) is active.

6.3.1. Threat Model & Attack Overview

In our threat model, the attacker can run *Cloudflare Workers* executing JavaScript code but no native code. Furthermore, the attacker controls a remote server to record high-resolution timestamps, e.g., using `rdtsc`,and a low-latency network connection. We also assume a powerful attacker with a worker co-located with the victim worker, e.g., by spawning multiple *Cloudflare Workers* and detecting co-location. An attacker spawning its instances close in time to the victim's one can maximize the probability of co-location [49]. *Cloudflare Workers* architecture aims to serve the same application from every location. A high number of tenants per machine is possible. Physical co-location of the attacker server is not required. However, this leads to the strongest possible attacker. We assume no exploitable software bugs, e.g., memory safety violations, in the JavaScript engine and no sandbox escapes. Thus, *architectural* exploits to leak data from other tenants or processes are not possible.

The typical requirements for state-of-the-art Spectre attacks on the timer and memory are listed in Table 6.1, showing the differences to our attack.

Figure 6.2 provides an overview of our attack. In the *Cloudflare Workers* setup, each worker runs in the same process, and thus, shares the virtual address space. The attacker runs a malicious JavaScript file containing a self-crafted Spectre-PHT gadget that performs a Spectre attack on its own process. As the victim and attacker share the same process, the attacker can leak sensitive data from a victim worker, without having an existing Spectre gadget in the victim.

Spectre attacks in JavaScript rely on speculative out-of-bounds accesses of objects. Assuming the attacker can either trigger a victim worker's secret allocation, delay it, or just manages to execute before the victim, we can use heap-grooming techniques [21] to bring the process memory into a predictable state before both the leaking object and the victim data are allocated. Alternatively, the attacker worker can predict the offset between the leaking object and the victim worker's data, target a certain range of the virtual memory, e.g., regions where V8 places similar objects [61], or break ASLR using speculative probing [19]. Hence, ASLR does not mitigate the attack. Furthermore, Agarwal [1] demonstrated that it is possible to leak over the full address space using a JavaScript Spectre attack in the V8 engine.

For our attack, we rely on a Spectre-PHT [34] gadget, as this is the simplest gadget to introduce in JIT-compiled code. Moreover, Spectre-BTB [34] can be prevented by the JIT compiler [59]. In contrast to the original Spectre attack [34], we do not encode the data bytewise but bitwise. The advantage of such a *binary Spectre gadget* is that it is easier to distinguish two states compared to 256 states using a side channel [4, 55]. While such a gadget might not be commonly *found* in real applications, it is easy to *introduce*.

As there are no high-resolution timers to distinguish microarchitectural states directly, we have to amplify the timing difference between a cache hit and a miss, *i.e.*, between a leaked '0' and '1' bit. We combine the amplification techniques by McIlroy et al. [41] with the remote measurement methods by Schwarz et al. [55]. With this semi-remote Spectre attack, we show that it is indeed feasible to leak data from co-located *Cloudflare Workers* in such a restricted setting. Our Spectre attack is the only one not requiring native code execution, a local timer, or an existing gadget. Moreover, microcode cannot prevent it (cf. Table 6.1).

Table 6.1.: Requirements and leakage rate of Spectre attacks.

Spectre attack (variant)	Gadget	Native	HR Timer	Memory	Leakage Rate	Error	Channel
Kocher et al. [34] (PHT)	Yes	Yes	Yes (ns)	2.40 MB	4420.46 B/s ± 6.75 %	0.07 %	Cache-L3
Canella et al. [7] (PHT)	Yes	Yes	Yes (ns)	3.54 MB	3.13 B/s ± 113.79 %	0.00 %	Cache-L3
Safeside [20] (PHT)	Yes	Yes	Yes (ns)	7.00 MB	4384.03 B/s ± 7.75 %	0.00 %	Cache-L3
Canella et al. [7] (BTB)	Yes	Yes	Yes (ns)	6.91 MB	0.71 B/s ± 2.43 %	0.00 %	Cache-L3
SafeSide [20] (BTB)	Yes	Yes	Yes (ns)	7.01 MB	269.53 B/s ± 0.85 %	0.00 %	Cache-L3
Canella et al. [7] (STL)	Yes	Yes	Yes (ns)	3.54 MB	14.37 B/s ± 211.95 %	0.00 %	Cache-L3
Safeside [20] (STL)	Yes	Yes	Yes (ns)	7.00 MB	272.46 B/s ± 0.22 %	0.00 %	Cache-L3
Canella et al. [7] (RSB)	Yes	Yes	Yes (ns)	20.08 MB	30.67 B/s ± 195.59 %	0.00 %	Cache-L3
Safeside [20] (RSB)	Yes	Yes	Yes (ns)	7.00 MB	116.70 B/s ± 0.58 %	0.00 %	Cache-L3
Google [57] (PHT)	No	No	Yes (μs)	15.00 MB	335.02 B/s ± 23.50 %	0.26 %	Cache-L1
Google [57] (PHT)	No	No	Yes (ms)	15.00 MB	9.46 B/s ± 31.40 %	2.71 %	Cache-L1
Agarwal et al. [1] (PHT)	No	No	Yes (μs)	N/A	533.00 B/s ± N/A	0.32 %	Cache-L3
Schwarz et al. [55] (PHT)	Yes	Yes	No	N/A	7.50 B/h ± N/A	0.58 %	AVX unit
Our work (PHT)	**No**	**No**	**No**	27.54 MB	15.00 B/h ± 2.67 %	0.00 %	Cache-L3

Gadget: Spectre gadget must be in victim; Native: native code execution; HR Timer: High-resolution timer

6.3.2. Building Blocks

As our attack uses the cache as the covert-channel part of the Spectre attack, we require building blocks for measuring the timing of cache accesses in JavaScript. While this can be done using a high-resolution timer in some browsers [34], the required primitives are not available on *Cloudflare Workers*. Hence, in addition to a different timing primitive with a lower resolution, we have to amplify the signal such that we can reliably distinguish '0' and '1' bits.

Remote Timer On *Cloudflare Workers*, there are no local timers or known primitives to build timers [54]. We verified that, indeed, no technique from Schwarz et al. [54] resulted in a timer with a resolution higher than 100 ms. Thus, there is no possibility to accurately measure the time directly in JavaScript, and, therefore, it is not possible to perform a local Spectre attack [34].

In this setup, the attacker sends a network request to a remote server to start a timing measurement. The remote server stores a local high-resolution timestamp, e.g., using `rdtsc`, associated with the request. To stop the timing measurement and receive the time delta, the attacker sends another request to the remote server, which sends back the time difference from the current to the stored timestamp. Hence, the attacker has a high-resolution time difference that is only impacted by the network latency between the attacker's worker and the remote server. We evaluated this timing primitive on *Cloudflare Workers*. For the best case, *i.e.*, same physical machine, we achieve a resolution of 0.47 ns on a 2.1GHz CPU,

```
if (secret_bit) {
  read A; //transiently leak bit
} else
{
  read B;
}
read A; //perform architectural access
```

Listing 6.1: Amplified Spectre-PHT gadget [41].

with a jitter of 1.67 %. With a resolution of 0.47 ns, we can distinguish a cache hit from a miss for the cache covert channel. However, this case is unlikely in reality, as the latency is typically in the microsecond range [63].

Amplification In our attack scenario, the attacker has no high-resolution timer but full control over the Spectre gadget. Hence, to mount a successful attack with the remote timer, we have to rely on amplification techniques that amplify the latency between a cache hit and miss [41]. One such technique is to transiently access multiple cache lines for a single bit instead of a single cache line and probe over these to increase the latency between a cache hit and a miss. However, this technique is quite memory-consuming and limited by the number of cache lines.

A way to arbitrarily amplify the latency between cache hits and misses is to either access a memory location which encodes a '0' or '1' bit transiently and then accesses the memory location for a '1' again architecturally [58]. listing 6.1 illustrates an *arbitrary amplification* [58] gadget. If the Spectre gadget is optimal in terms of mistraining, we have twice as many cache misses for a '0' bit as for a '1' bit. With a loop over the gadget, we can create arbitrarily large timing differences between hits and misses. We evaluate the amplification idea on an Intel Xeon Silver 4208, running Ubuntu 20.04 (kernel 5.4.0) in native code. We increase the number of amplification iterations and run each iteration 1000 times to get stable results. This leads to a linear growth with the increase of the number of loop iterations (amplification factor). Depending on how much runtime is given to the worker, it is possible to arbitrarily increase the delay. Hence, we can also see that there are no strict requirements for the resolution of the remote timer. For lower resolutions, we can increase the amplification, resulting in a reduced leakage rate, no prevention of the attack, as also shown in related work [57].

Eviction To repeat our amplification and reset the cache state, cache eviction is required. One way to evict certain addresses from the cache is by building eviction sets [23, 44, 64]. While a targeted eviction set leads to a fast eviction, building the eviction set is costly. Even with a local timer, the currently fastest approach takes more than 100 ms [64]. In our remote scenario, this would require a lot of network requests to find the eviction set for our encoding oracle, as building the eviction set requires constant timing measurements. Furthermore, eviction sets cannot be reused due to address-space-layout randomization on each run. Instead of using eviction sets, we iterate over a large eviction array (multiple MB, depending on the cache size) in cache-line steps (64 byte) and access the values. If enough addresses are accessed, the cached value is evicted [23, 34].

We evaluate the eviction directly on the V8 engine used in *Cloudflare Workers* on an Intel Xeon Silver 4208, running Ubuntu 20.04 (kernel 5.4.0). We access a certain index v of a large array to cache it, iterate over the eviction set, and verify if v is still cached. We observe that an eviction array of 2 MB always evicts v on our Intel Xeon Silver 4208 ($n = 1000$).

Note that address randomization can be deterministically circumvented using engineering. Göktas et al. [19] introduced the concept of a speculative probing primitive that leverages Spectre to break classical and fine-grained ASLR. Gras et al. [22], Schwarz et al. [51], and Lipp et al. [36] demonstrated that microarchitectural attacks in JavaScript can break memory randomization.

6.3.3. Attack on Cloudflare Workers

Using the building blocks, we mount an attack on *Cloudflare Workers* to extract secret bits from a worker at a known location to estimate the best possible attack. For that, we send an initial request with a sequence number to a timing server. The timing server stores a local, high-resolution timestamp on this request. We perform a Spectre attack on a target address and send another request to the server. The timing server computes the delta between the current and the stored timestamp to distinguish between a cache hit or miss. As the attacker controls both the attacking worker and the timing server, there is no need to send the leaked information back to the worker.

There are different challenges when creating a JavaScript Spectre PoC, as the V8 JIT compiler optimizes code based on assumptions. If such

assumptions are invalidated, the function is de-optimized. We thus avoid triggering any de-optimization points in our generated code, as that ruins the training achieved. Therefore, we place the out-of-bound access behind a mispredicted guard branch, preventing the JIT compiler from de-optimizing the code when detecting out-of-bound accesses. Moreover, during the garbage collection phase, objects move between different heap spaces of the same worker to reduce the memory footprint. By forcing garbage collection phases, we stabilize an object's location.

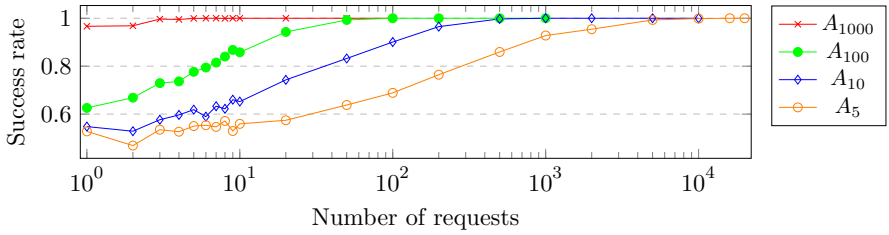
Evaluation. To develop and evaluate a proof-of-concept attack, we obtained a local developer copy of *Cloudflare Workers* to not interfere with any worker of other customers. We ensured that the configuration on our local system is identical to the configuration running on the cloud. As *Cloudflare Workers* mostly use server CPUs, we also focus our attack on an Intel server CPU, specifically an Intel Xeon Silver 4208, running Ubuntu 20.04 (kernel 5.4.0).

We create a Spectre-PHT PoC that leaks bits from a victim `ArrayBuffer` by transiently reading out-of-bounds. We describe the technical implementation details for optimal leakage in the extended version [56] (Appendix B).

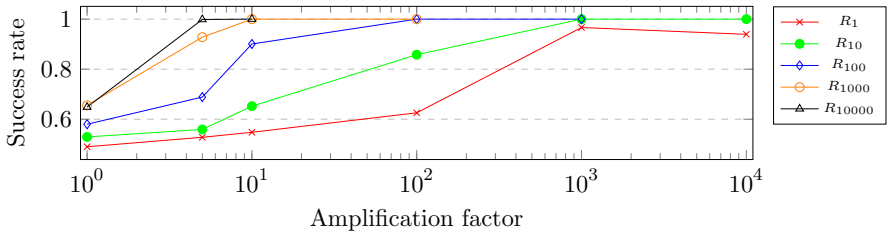
We call the function performing a Spectre attack 10 000 times and repeat the experiment 1000 times, observing a success rate of 54.31 % (n = 1000, σ = 23.16 %). We assume that the attacker is capable of creating a stable exploit with 100 % success rate. From now on, we evaluate our metrics with a 100 % success rate to estimate the best possible attack, where the attacker knows where the secret array is located.

We evaluate a set of different amplification factors (number of loop iterations) in native code between 1 and 1000 , and sample each loop length 100 000. We implement the box test [14] to determine the number of required requests [6, 14, 63]. Figure 6.3a illustrates the number of requests required to achieve a certain success rate for different amplification factors. The higher the amplification factor is, the fewer requests are required to achieve high success rates. As Figure 6.3b illustrates, with small amplification factors but enough requests, we can also achieve a high success rate of more than 95 %. We refer to the work of Van Goethem et al. [63] and Schwarz et al. [55] for the required requests in a network with multiple hops.

We evaluate our attack locally, *i.e.*, with a timing server on the same machine. We first evaluate an optimal attack in native code. Ideally, an



(a) Success over the number of requests per number of amplification factor.



(b) Success of different amplification factors for different number of requests.

attacker chooses the number with the highest success rate and the lowest number of requests required, minimizing the execution time. We choose a random 16-bit secret. As amplification factor, we choose 100 000 loop iterations and perform just one request. With this setup, leaking one bit takes on average 2.5 s (n = 100, $\sigma_{\bar{x}}$ = 0.05 %). We repeat the experiment 100 times and observe a leakage rate of 23 bit/s (n = 100, $\sigma_{\bar{x}}$ = 2.8 %). Using an outlier filter, this error can be reduced towards 0. As these values are from a native-code attack, we consider these numbers as the maximum achievable leakage rate for JavaScript. A JavaScript attacker is more restricted in terms of evicting certain addresses from the cache and thus requires additional time for the eviction. Furthermore, the code is JIT-compiled, requiring a warmup to stabilize the JIT-compiled code. We evaluate the amplification in JavaScript in the V8 engine with an amplification factor of 250 000, a native timestamp counter to measure the response times, and a random 16 bit secret. One script execution takes about 30 s, which is the maximum execution time for *Cloudflare Workers* [12]. All evaluated numbers are shown in Table 2 in the extended version [56] (Appendix A). With a success rate of 100 % we determine an optimal leakage rate of 2 bit/min leading to a leakage rate of 120 bit/h.

6.4. DyPrIs

In this section, we present an approach to dynamically isolate malicious *Cloudflare Workers* to benefit both from the security of process isolation and the performance of language-level isolation. The basic idea is to use HPCs to detect potential Spectre attacks and isolate suspicious *Cloudflare Workers* using process isolation (Figure 6.4). While a detection mechanism typically suffers from false positives, DyPrIs can cope even with high false-positive rates. In the worst case, a Spectre attack is detected for every worker, leading to the worst-case scenario of one worker per process, *i.e.*, strict process isolation, as currently used in browsers plus the 2 % detection overhead. As workers are stateless, they can also be suspended or migrated at any time. Thus, even if many worker are considered malicious, the resources of *Cloudflare* are not exhausted. Every false-positive rate below 100 % performs better than strict process isolation.

We discuss how to reliably detect Spectre attacks using performance counters (cf. Section 6.4.1). We integrate our approach into *Cloudflare Workers* and measure the performance overhead of reading performance counters on a real-world cloud system (cf. Section 6.4.2). We show that there is a small performance overhead of 2 % for reading performance counters.

6.4.1. Detecting Spectre Attacks

In this section, we discuss the detection of Spectre attacks using HPCs. While the common use of HPCs is finding bottlenecks, researchers used HPCs for detecting malware, rootkits, CFI violations, ROP, Rowhammer, or cache-side channel attacks [5, 8, 24, 28, 39, 68, 69, 73].

Detecting Attacks using Normalized Performance Counters

Our second approach tries to detect Spectre attacks using normalized performance counters. At first we collect data from different performance counters. We collect the following hardware events (`PERF_COUNT_HW_*`): `CACHE_iTLB`, `BRANCH_MISSES`, `BRANCH_INSTRUCTIONS`, `CACHE_REFERENCES`, `CACHE_MISSES`, `CACHE_L1D/READ_MISSES` and `CACHE_L1D/READ_ACCESSES`. We normalize the values using iTLB performance counters (iTLB accesses) which was also used by Gruss et al. [24] to detect Rowhammer and cache

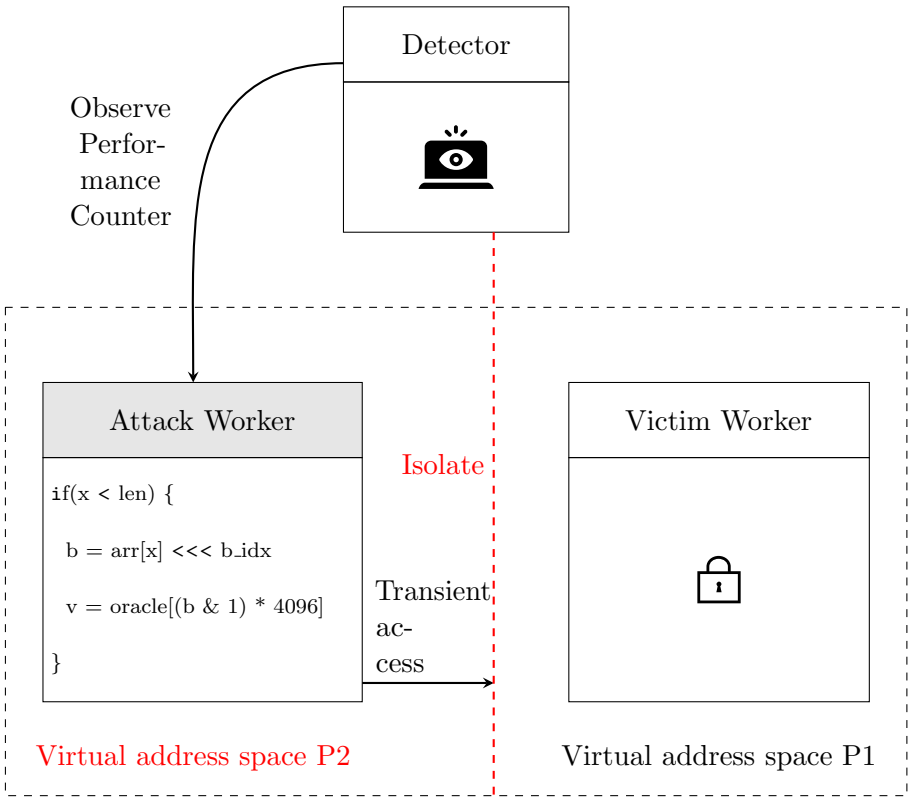


Figure 6.4.: DyPrIs isolating a malicious worker based on performance counters.

attacks. Similarly to Rowhammer and cache attacks, the main attack code for Spectre has a small code footprint with a high activity in the branch-prediction unit.

The iTLB counter normalizes the branch-prediction events with respect to the code size by dividing the performance counter value by the number of iTLB accesses. We integrate the monitor into *Cloudflare Workers*, to read the performance counters before and after each script execution. The averaged per-execution numbers are updated in a 1-second interval (Note that a single script runs up to 30 s [12]). While reducing the interval does not directly impact the performance of a worker, it potentially leads to more false positives as outliers are not filtered. We collect data from the benign workload and compare it to a worker executing a Spectre attack. Based on the performance numbers, we find a threshold to distinguish between an attack and normal workload. We evaluate this approach in Section 6.5.1.

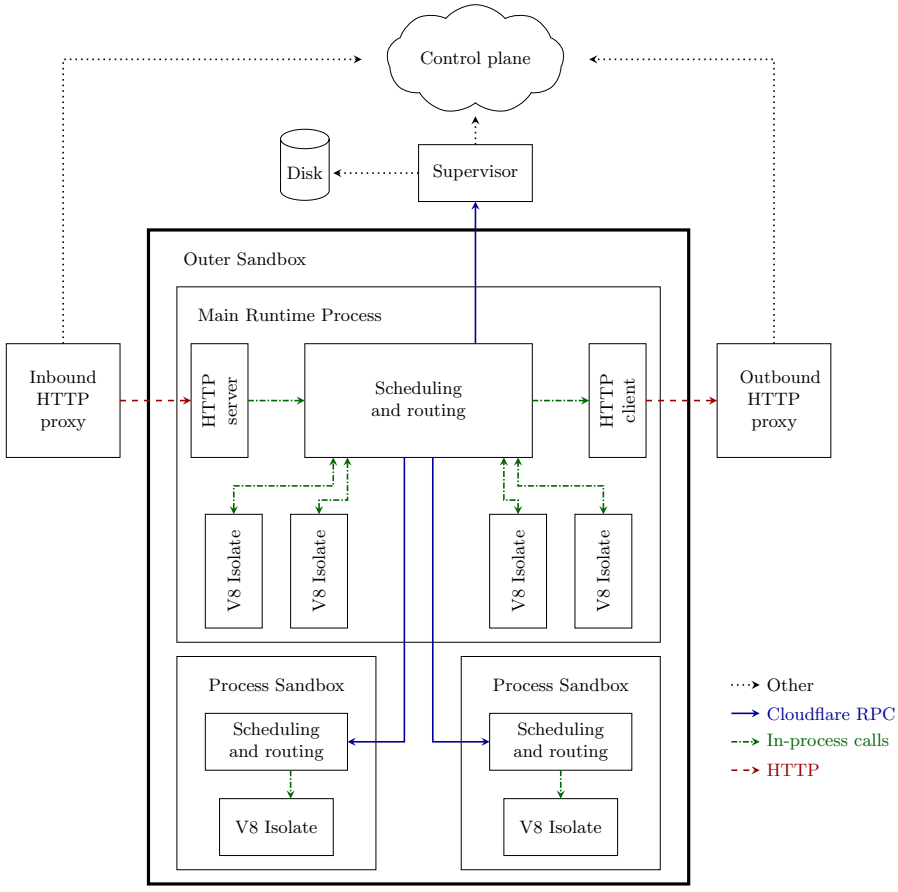


Figure 6.5.: DyPrIs overview.

6.4.2. Process Isolation

For DyPrIs, we fundamentally rely on process isolation. A well-known implementation of process isolation is site isolation, where every page in a browser runs in its own process to prevent memory safety violations as well as Spectre attacks [48]. However, in contrast to full site isolation, we only isolate potentially malicious *Cloudflare Workers* if the Spectre detection mechanism flags them. Hence, DyPrIs only falls back to full site isolation in the worst case, while reducing the overhead caused by process isolation in the average case.

Related work proposes efficient in-process isolation mechanisms using Intel Memory Protection Keys (MPK) [45, 50, 62]. However, Intel MPK is

only available on selected CPUs since Skylake-SP, limited to 16 protection keys and thus not practical for *Cloudflare Workers* [62], running multiple thousand workers per process. Furthermore, the threat model of these approaches does not include side-channel or transient-execution attacks. For DyPrIs, we modify the *Cloudflare Workers* software to isolate a potentially malicious worker, *i.e.*, a worker that was flagged by the performance-counter-based detection, into a separate process. We implement process isolation in *Cloudflare Workers* from scratch (cf. Figure 6.5). For that, we start process sandboxes by forking from a zygote process, and talk to the new process over an RPC protocol [3, 9]. All communication between the main process and the isolated process are over this RPC connection, communications between the process sandbox and the outside have to go through the main process. Since the runtime of a worker is, on average, less than 1 ms, the isolation must not introduce a high performance overhead. Thus, one instance of a worker frequently reads out the performance counters per script execution and computes a moving average. From our results in Section 6.5, we observed that the normalized iTLB performance provides the best detection tradeoff in terms of performance overhead and accuracy. We first run an attack and collect its performance-counter data. Additionally, we collect anonymized per-CPU-core performance-counter data of real scripts running in production. Based on our evaluation in Section 6.5.1, we use a threshold of 4096 retired branches per iTLB access to distinguish between a suspicious and a benign script. If a script exceeds this threshold, we flag it as a potential Spectre attack and isolate it into a separate process. In contrast to, e.g., browser tabs, worker are stateless. Thus, a worker can simply be migrated. Isolating instead of terminating ensures that the worker can continue running, e.g., in case the detection was a false positive, while it cannot access data of any other worker.

6.5. Evaluation

In this section, we evaluate the accuracy and performance overhead of our detection methodology. We choose a threshold of 4096 branch accesses per iTLB access, which allows distinguishing a Spectre attack from a benign script. We use a large set of different programs to sample the number of mispredicted and retired branches. For our set, we observe that out of 141 programs, which includes the 13 Spectre gadgets from Kocher [33], we cannot distinguish 4 benign programs from a Spectre gadget, resulting in

a false-positive rate of 2.83 % with a small performance overhead of 2 %. Using our normalized counters approach, we observe a negligible overhead of 2 % in our production environment.

6.5.1. Normalized Performance Counters

We evaluated our approach on 5 Intel Xeon server CPUs (Broadwell, Skylake 4116, Skylake 6162, Skylake 6162, Cascade Lake 6262) and one AMD Epyc Rome CPU. To decide whether a script is susceptible or not, we collect performance data from the production system running our Spectre attack. We recorded the performance counters on the production environment and sampled over 50 000 times as a baseline. Figure 6.6 shows the normalized performance counters of our cloud machines. For last-level-cache accesses, misses, and branch misses, the numbers of the attack script are below the average script. For the number of L1-cache accesses and retired branches, we can clearly distinguish average script from attack. Especially for the retired branches, the distance between an attack script and the average regular script is 34 times the standard deviation of a benign script. We collected our numbers from real-world worker production machines to calculate the false-positive rate. We choose the number of normalized retired branch instructions as an indicator for a Spectre attack and run it on our cloud machines. First, we run a Spectre attack to verify whether their number is in a similar range on each test machine. We then evaluate different threshold boundaries for the number of normalized retired branch instructions and report the number of false positives. Figure 6.7 shows the number of false positives depending on the threshold on our cloud machines in the production environment. For a strict threshold, *i.e.*, 1024, the false-positive rate is 21.41 %. However, this threshold is set higher to reduce the number of false positives. The numbers of false positives are in a similar range on each of the tested machines. Setting the threshold to 4096, results in an average false positive rate of 0.61 % on our devices. For a threshold of 8192 the average false-positive rate decreases to 0.26 %, and at a threshold of 65 536 , we do not observe any false positives.

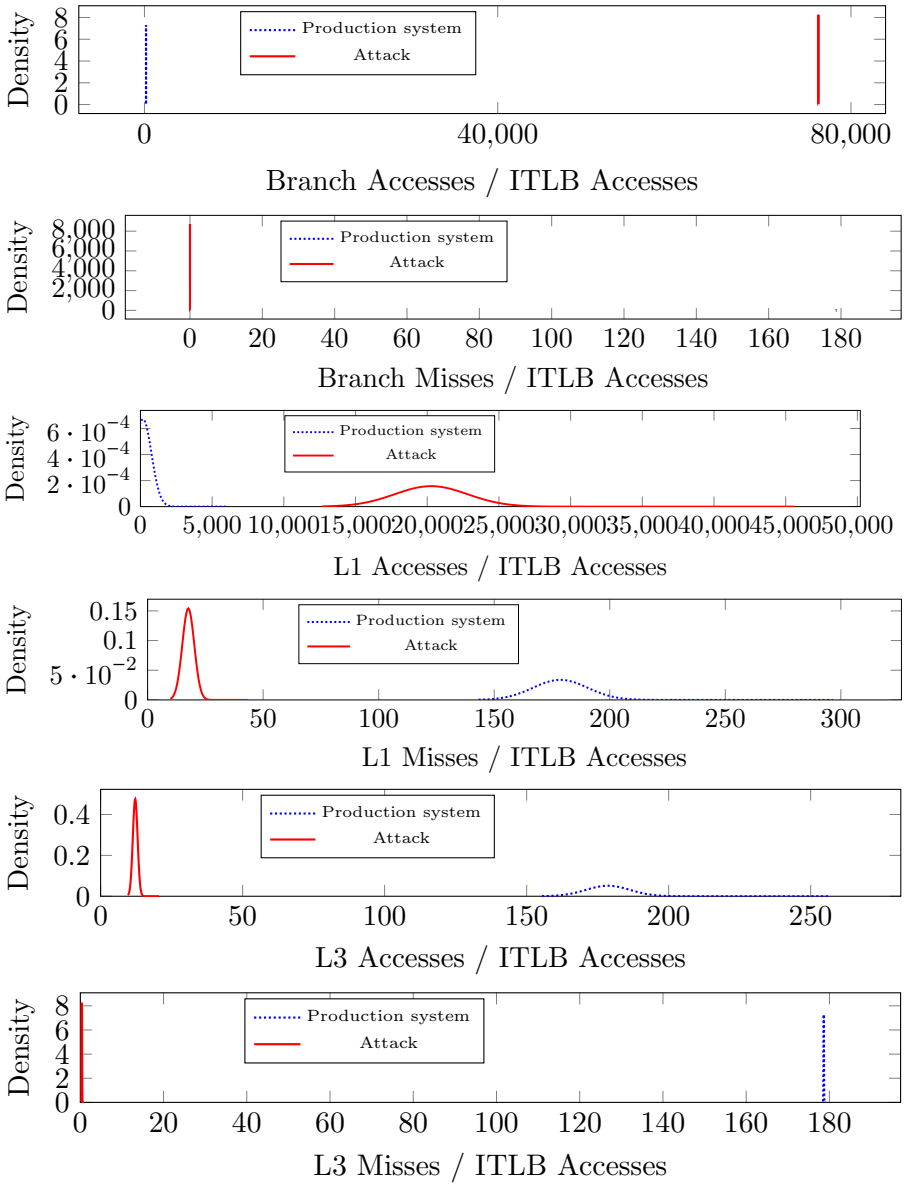


Figure 6.6.: Performance counters of average *Cloudflare Workers* and a Spectre attack on the production system.

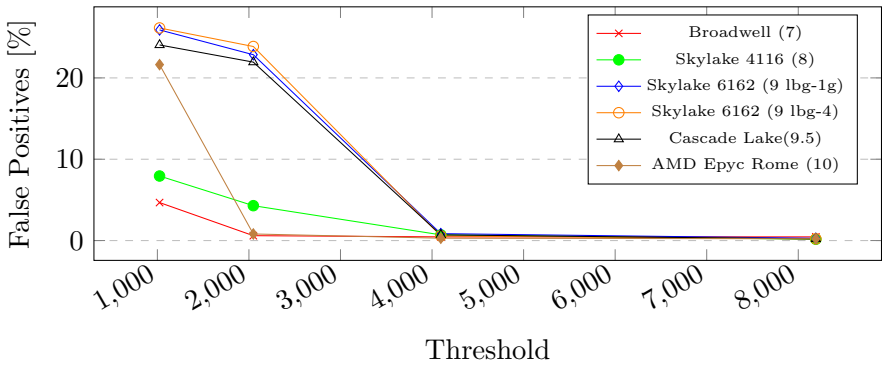


Figure 6.7.: Number of false positives depending on the normalized iTLB threshold.

Next, we look at the performance overhead of our attack when the attacker tries to get below the detection thresholds. Getting below this threshold requires the attacker to significantly slow down the amplified Spectre attack. Since the attacker cannot get rid of the cache eviction, the number of amplification iterations has to be reduced. Consequently, if the number of amplification iterations is reduced, more requests, *i.e.*, samples, are required to clearly distinguish cache hits and misses (cf. Figures 6.3a and 6.3b). We evaluate the best possible attacker in native code who only mistrains one branch. By omitting amplification or with a small factor of 10, we can reduce the number of retired branch instructions / iTLB accesses on our test devices to 604.71 and 3492.41, respectively, which is in the ranges of an average script. However, with the latter, the leakage rate is 1 bit/h. Thus, we set the threshold to 4096 and receive an average false positive rate of 0.61 % on our tested devices. Figure 6.8 illustrates the decrease in leakage if the attack degrades from an amplified Spectre attack to a sequential attack. Using a non-amplified approach, about 250 000 requests are required (Section 6.3.3). We achieve a leakage of 1 bit/h in a local-network scenario. Hence, as an additional security margin, we limit the number of subsequent requests per worker to 10 000 on the same machine. If more than 10 000 requests are issued, we redirect the request to a different machine. Thus, we can still prevent leakage from a slowed-down attack using our threshold-based approach. We assume that there are no attacks running on the production system, thus we cannot measure the number of false negatives. Our own attack is detected by the threshold, as well as the 15 Spectre samples provided by Kocher [33]. In addition, we evaluated and analyzed the new and larger Spectre-PHT

gadgets generated by FastSpec [60]. The gadgets are based on the the 15 variants, and we observe that the generated gadgets are quite similar. We evaluated 100 random gadgets from FastSpec and did not observe any false negatives with our detection. As the mistraining for those gadgets is similar, the branch accesses per iTLB access are in a similar range. We also evaluated the detection on the Spectre JavaScript PoC from Röttger and Janc [57]. Even with the low amplification factor of 4 000 in this PoC, we reliably detect the attack (n = 500, μ = 19 253.73).

`Spectre-BTB, Spectre-RSB and Spectre-STL.` In addition to Spectre-PHT we also run our performance counter analysis on other Spectre variants exploiting the branch-target buffer (BTB), return-stack buffer (RSB) and store-to-load (STL) forwarding. We create native code proof-of-concepts for these variants executing each gadget 10 000 times on a Xeon Silver 4208. We ran the PoCs 500 times and collected the number of branch and iTLB accesses. The numbers for Spectre-BTB and RSP are an order of magnitude lower than for Spectre-PHT ($\mu_{\overline{btb}}$ = 423171.54). However, they are still detected with the same metric (n = 500): Spectre-BTB ($\mu_{\overline{btb}}$ = 23401.20), Spectre-RSB ($\mu_{\overline{rsb}}$ = 38369.17), Spectre-STL ($\mu_{\overline{stl}}$ = 982.20). The metric for Spectre-STL is far below the threshold of 4096. However, the values for `memory_disambiguation.history_reset` are significantly higher on average if the store-to-load logic is exploited in Spectre-STL (n = 500, $\mu_{\overline{stl}}$ = 8993.98, $\mu_{\overline{nostl}}$ = 2644.73). Thus, we also use this counter to detect potential Spectre-STL attacks.

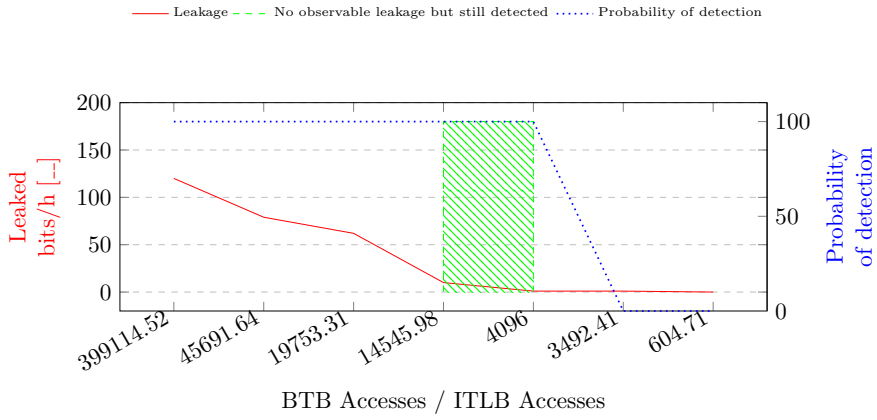


Figure 6.8.: Branch accesses / iTLB accesses and the corresponding leakage rate.

6.5.2. DyPrIs

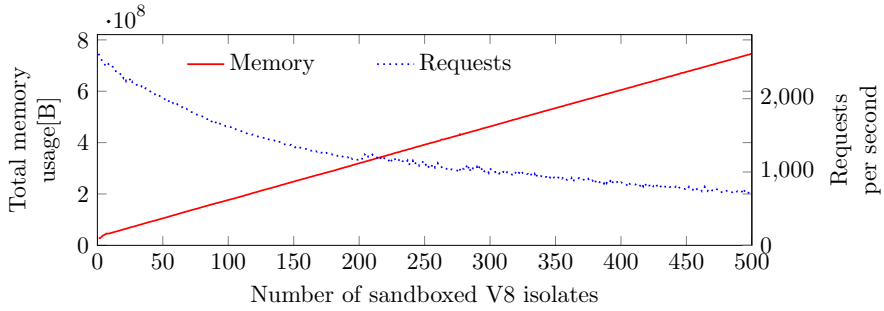


Figure 6.9.: Requests per second and memory consumption of process isolation.

We integrate DyPrIs in *Cloudflare Workers*, which requires modifications of 6459 lines of code, not including the Spectre detection mechanism. As with any isolation technology, the performance overhead varies depending on the workload [48]. *Cloudflare Workers* is an environment where typical guest workloads use very little memory and spend very little CPU time responding to any particular event. As a result, in this environment, DyPrIs's overhead is expected to be large compared to the underlying workload. In a first test, we evaluate the overhead for a test script by increasing the number of isolated processes, *i.e.*, the number of sandboxed V8 isolates, up to 500. We measure the overhead in terms of executed scripts per second, *i.e.*, the requests executed per second from the localhost and the total amount of consumed main memory. The execution is repeated 10 times per isolation level with 2000 requests (n = 20000, $\sigma_{\overline{rps}}$ = 3.87 %, $\sigma_{\overline{mem}}$ = 0.23 %). Figure 6.9 shows the requests per second and the total memory consumption based on the number of isolated V8 processes. As expected, we observe a linear decrease in the possible number of requests per second and a linear increase in the memory consumption. Further, we performed a load test of *Cloudflare Workers* runtime using a selection of sample guest workers simulating a heavy-load machine. They mostly respond to I/O in under a millisecond and allocate little memory. By forcing process isolation on the workers, the memory overhead of each guest was 2x-5x higher, and CPU time was 8x higher, compared to a worker using a single process. We performed a second test using a real-world worker known to be unusually resource hungry in both CPU and memory usage. In this case, memory overhead is 20 %-70 % worse with DyPrIs, and CPU time about 60 % worse. These numbers appear to be high,

but when only 0.61 % of workers are isolated, the overhead is negligible. As our proof of concept was not optimized, it still has big potential for optimizations. For example, it currently uses an RPC protocol [3] to communicate between processes, but does so over a Unix domain socket. This protocol is designed in such a way that it could be communicated in shared memory, reducing communication overhead. The implementation could also use OS primitives for faster context switching, such as the FUTEX_SWAP feature proposed by Google. However, while especially the CPU overhead could be reduced, there is always a significant cost incurred by context switching and marshalling to communicate between processes. The total overhead on all machines can only be estimated as it depends on the workload. The detection overhead is 2 %. In the worst case, we are slightly worse than full process isolation due to the 2 % detection overhead.

6.6. Discussion

Comparison between *Cloudflare Workers* and competing approaches. The main challenge of edge computing is to run various applications of numerous tenants efficiently. Approaches like AWS Lambda and Azure Functions rely on containers to achieve this [2, 42]. While their design strictly prevents Spectre attacks on other tenants, the performance overhead is higher for the use case of edge computing than *Cloudflare Workers* [11]. The *Cloudflare Workers* architecture is stateless in a sense that every worker in any data centre can process any request, *i.e.*, the request is processed by the worker with the lowest latency. *Cloudflare Workers* rely on a single-process architecture with language-level isolation to isolate their tenants architecturally. However, as we showed, this design leads to potential Spectre attacks. A similar design with language-level isolation of WebAssembly code from different tenants is used by Fastly [17]. Therefore, Fastly also needs to consider Spectre attacks within the same process by either applying DyPrIs or switching to full isolation via processes or containers.

Mitigation versus Detection. Especially in high-performance scenarios, such as cloud systems, Spectre mitigations [7, 31, 34] result in high power consumption. Hence, instead of paying the constant costs of mitigations, dectecing attack can reduce the costs. However, the problem of detecting side-channel and transient-execution attacks is still an open

research problem. There is no universal solution that covers all different types of attacks.

False Positives and Negatives. DyPrIs suffers from false positives and false negatives [15, 72], similar to other detection and mitigation techniques [25, 65]. False positives only impact the performance and not the security. False negatives occur when slowing down attacks to 1 bit/h (cf. Table 2 in Appendix A in the extended version [56]). Therefore, the maximum execution time is restricted to 30 s, far from 1 h. Using the machine learning approach of Gulmezoglu [26], the false positive rate could be reduced further. However, this approach would require a re-training with real-world data of *Cloudflare Workers* and a frequent re-updating of the training set. Adding additional code pages also allows getting below the thresholds. To "hide" the native attack, we access 125 additional code pages (500 kB) per bit to get the branch accesses / iTLB accesses below the threshold (cf. Figure 10 in Appendix A in the extended version [56]). While feasible in native code, the resulting code size causes V8 to abort the optimization phase, stopping the attack.

Comparison to existing detections Besides full site isolation, prior work discusses detection but not how to stop attacks once they are detected. Existing static analysis approaches [13, 27] on binaries are not applicable to the use case of *Cloudflare Workers*. Approaches that perform taint tracking and fuzzing on binaries to dynamically detect gadgets [25, 47, 52, 65] are infeasible for the high-performance requirements of *Cloudflare Workers*. The approach of Mambretti et al. [40] does not evaluate real-world workloads and cannot distinguish the different workloads of *Cloudflare Workers* from an external process.

Reliability of HPCs In DyPrIs As Zhou et al. [72] and Das et al. [15] discuss, using HPCs for detection of cache attacks can lead to flaws caused by non-determinism and overcounting. We showed that in our statistical approach both only marginally reduce the performance of DyPrIs not the security.

Alternative Spectre JS attacks Concurrent work [57] has demonstrated a Spectre exploit on V8, leaking up to 60 B/s using timers with a precision of 1 ms or worse through a L1 covert channel. Similarly to our PoC, it uses a Spectre-PHT gadget to read out-of-bound from a JavaScript TypedArray, giving an attacker access to the entire address space. The PoC uses small-sized TypedArrays for which the backing store is allocated in the isolate itself. Thus, it leaks data inside the same isolate.

In concurrent work, Agarwal et al. [1] has extended the PoC from Röttger and Janc [57] to leak data using 64-bit addresses using a local timing source. They use speculative type confusion between an ArrayBuffer and a custom object that should be properly aligned across two cache lines.

6.7. Conclusion

In this paper, we presented DyPrIs, a practical low-overhead solution to actively detect and mitigate Spectre attacks. We first presented an amplified JavaScript remote attack on *Cloudflare Workers*, which leaks 2 bit/min, *i.e.*, 1 bit per worker invocation. We proposed a practical approach for actively detecting and mitigating Spectre attacks. We show that it is still possible to efficiently detect Spectre attacks using performance counters with a false-positive rate of 0.61 % at the cost of 2 % overhead for the detection. We demonstrate that conditionally applying process isolation based on a detection mechanism has a better performance than full process isolation, under the same security guarantees.

Acknowledgments

We want to thank our anonymous reviewers and in particular our shepherds Roberto Di Pietro and Vijayalakshmi Atluri. This work was supported by generous gifts from Cloudflare. We want to especially thank Harris Hancock, Claudio Canella and Moritz Lipp for valueable feedback on this work. Any opinions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding parties.

Appendix

A. Current Worker Limits and JS Attack Numbers

The state-of-the art limits of the *Cloudflare Workers* setup for a single instance is 128 MB memory, and 30 s runtime. We provide the leakage rate depending on the amplification factor in Table 6.2.

Table 6.2.: Attack results of our JS attack.

Amplification	Required requests	Script runtime	Leaked bits/hour
1	250 000	118 ms	0 bit/h
10	25 000	123 ms	1 bit/h
100	2500	137 ms	10 bit/h
1000	250	231 ms	62 bit/h
10 000	25	1813 ms	79 bit/h
250 000	1	30 000 ms	120 bit/h

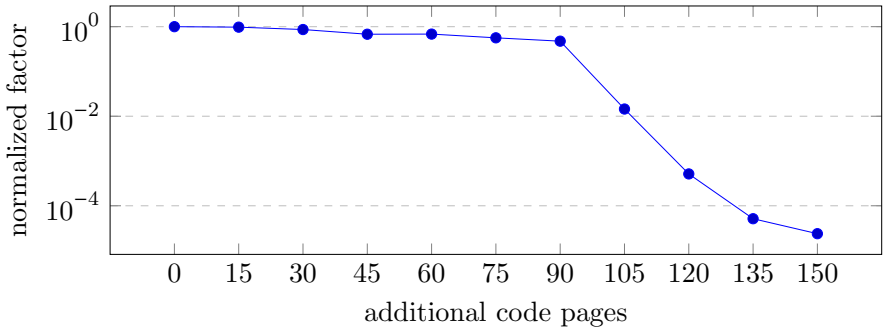


Figure 6.10.: Influence of the number of additional code pages on the branch accesses / iTLB accesses.

B. Spectre-Attack Optimizations

The function containing the Spectre gadget accesses the attacker's `ArrayBuffer` through differently-sized `TypedArrays`. This prevents the JIT compiler from making assumptions on the memory accesses on the `ArrayBuffer`. Otherwise, the JIT compiler hard-codes the size for the bounds check, which significantly decreases the success probability for the Spectre attack. As the garbage collector moves objects around, using multiple `TypedArrays` increases the probability of having correctly aligned objects, such that the backing store pointer and the size of the `ArrayBuffer` lie on different cache lines. We also increase the size of the attacker function to avoid it being inlined, which would cause de-optimization if the calling function is de-optimized. The function takes the offset to access as a parameter and is called in a loop with a branch-less code that feeds it with four in-bound offsets and an out-of-bound offset to access.

We warm-up the JIT compilation by repeatedly calling our function with an out-of-bound index higher than the one in the guard branch, but in-bound of the `TypedArray`, preventing the JIT compiler from making assumptions on the provided value. We also found that executing a different number of taken conditional branches before executing the target function affected the leakage rate considerably. By automatically tuning the number of such branches, we verified that 70 is the optimal number for our PoC.

C. Noise on Cache Attacks

We evaluated the attack PoC of Röttger and Janc [57] with respect to memory-intense system workloads. We pin the attack PoC program to a specific hyperthread and perform memory pressure on the sibling hyperthread. To simulate such a behaviour, we use the `stress` tool with `-m 10` option on the sibling hyperthread and other cores. As the PoC of Röttger relies on amplification using the L1 replacement policy, the attack is practically mitigated by the additional cache activity. For our optimal attack, we observe a decrease of the success rate to 75 % for the time the other hyperthread creates memory pressure.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution." In: *S&P*. 2022.

[2] Amazon. *AWS Lambda@Edge*. 2019. URL: `https://aws.amazon.com/lambda/edge/`.

[3] Anonymous. *Anonymized for Double Blind Submission*. 2019.

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

[5] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. "CacheShield: Detecting Cache Attacks through Self-Observation." In: *CODASPY*. 2018.

[6] David Brumley and Dan Boneh. "Remote timing attacks are practical." In: *Computer Networks* 48.5 (2005), pp. 701–716.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/. 2019.

[8] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. *Real time detection of cache-based side-channel attacks using Hardware Performance Counters*. ePrint 2015/1034. 2015.

[9] Cloudflare. *Anonymized for Double Blind Submission*. 2020.

[10] Cloudflare. *Cloudflare Workers*. 2019. URL: `https://www.cloudflare.com/products/cloudflare-workers/`.

[11] Cloudflare. *Cloudflare Workers*. 2019. URL: `https://blog.cloudflare.com/cloud-computing-without-containers/`.

[12] Cloudflare. *Limits - Cloudflare Workers*. 2021. URL: `https://developers.cloudflare.com/workers/platform/limits`.

[13] Jonathan Corbet. *Finding Spectre vulnerabilities with smatch*. 2018. URL: `https://lwn.net/Articles/752408/`.

[14] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. "Opportunities and limits of remote timing attacks." In: *ACM Transactions on Information and System Security (TISSEC)* 12.3 (2009), p. 17.

[15] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security." In: *S&P*. 2019.

[16] Deno. *A Globally Distributed JavaScript VM*. 2021. URL: `https://deno.com/deploy`.

[17] Fastly. *Serverless Compute Environment - Fastly Compute@Edge*. 2021. URL: `https://www.fastly.com/products/edge-compute/serverless`.

[18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware." In: *Journal of Cryptographic Engineering* (2016).

[19] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era." In: *CCS*. 2020.

[20] Google. *SafeSide: Understand and mitigate software-observable side-channels*. 2019. URL: `https://github.com/google/safeside`.

[21] Google Project Zero. *What is a "good" memory corruption vulnerability?* 2015. URL: `https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html`.

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

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

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

[25] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled Detection of Speculative Information Flows." In: *S&P*. 2020.

[26] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning." In: *arXiv:1907.03651* (2019).

[27] Red Hat. *Spectre And Meltdown Detector*. 2018. URL: `https://access.redhat.com/labsinfo/speculativeexecution`.

[28] Nishad Herath and Anders Fogh. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security." In: *Black Hat Briefings*. 2015.

[29] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[30] Intel. *Intel Analysis of Speculative Execution Side Channels*. Revision 4.0. 2018.

[31] Intel. *Speculative Execution Side Channel Mitigations*. Revision 3.0. 2018.

[32] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "MASCAT: Preventing microarchitectural attacks before distribution." In: *CODASPY*. 2018.

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

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

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

[36] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors." In: *AsiaCCS*. 2020.

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

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

[39] Corey Malone, Mohamed Zahran, and Ramesh Karri. "Are hardware performance counters a cost effective way for integrity checking of programs." In: *STC*. 2011.

[40] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. "Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations." In: *ACM ACSAC*. 2019.

[41] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. "Spectre is here to stay: An analysis of side-channels and speculative execution." In: *arXiv:1902.05178* (2019).

[42] Microsoft. *Azure serverless computing*. 2019. URL: `https://azure.microsoft.com/en-us/overview/serverless-computing/`.

[43] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. "WHISPER: A Tool for Run-time Detection of Side-Channel Attacks." In: *IEEE Access* (2020).

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

[45] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys." In: *arXiv:1811.07276* (2018).

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

[47] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. "SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets." In: *NDSS*. 2021.

[48] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *USENIX Security Symposium*. 2019.

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

[50] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. "Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86." In: *USENIX Security Symposium*. 2020.

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

[52] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre." In: *NDSS*. 2020.

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

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

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

[56] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. *Robust and Scalable Process Isolation against Spectre in the Cloud (Extended Version)*. 2022. URL: https://martinschwarzl.at/media/files/robust_extended.pdf.

[57] Stephen Roettger and Artur Janc. *A Spectre proof-of-concept for a Spectre-proof web*. 2021. URL: https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html.

[58] Ben Titzer. *What Spectre means for Language Implementers*. 2019. URL: https://pliss2019.github.io/ben_titzer_spectre_slides.pdf.

[59] Ben L. Titzer and Jaroslav Sevcik. *A year with Spectre: a V8 perspective*. 2019. URL: https://v8.dev/blog/spectre.

[60] M Caner Tol, Koray Yurtseven, Berk Gulmezoglu, and Berk Sunar. "FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings." In: *arXiv:2006.14147* (2020).

[61] v8 developer blog. 2020. URL: https://v8.dev/blog/v8-release-83.

[62] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. "ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys." In: *USENIX Security Symposium*. 2019.

[63] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. "Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections." In: *USENIX Security Symposium*. 2020.

[64] Pepe Vila, Boris Köpf, and Jose Morales. "Theory and Practice of Finding Eviction Sets." In: *S&P*. 2019.

[65] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. "oo7: Low-overhead Defense against Spectre attacks via Program Analysis." In: *Transactions on Software Engineering* (2019).

[66] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks." In: *ICCAD*. 2020.

[67] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, Tinoosh Mohsenin, and Houman Homayoun. "Comprehensive Evaluation of Machine Learning Countermeasures for Detecting Microarchitectural Side-Channel Attacks." In: *GLSVLSI*. 2020.

[68] X. Wang and R. Karri. "NumChecker: Detecting kernel control-flow modifying rootkits by using Hardware Performance Counters." In: *DAC*. 2013.

[69] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. "CFIMon: Detecting violation of control flow integrity using performance counters." In: *DSN*. 2012.

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

[71] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. "See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer." In: *AsiaCCS*. 2021.

[72] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. "Hardware performance counters can detect malware: Myth or fact?" In: *AsiaCCS*. 2018.

[73] Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. "HDROP: Detecting ROP Attacks Using Performance Monitoring Counters." In: *ISPEC*. 2014.

7

Specfuscator: Evaluating Branch Removal as a Spectre Mitigation

Publication Data

Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. "Specfuscator: Evaluating Branch Removal as a Spectre Mitigation." In: *FC*. 2021

Contributions

Main author.

Specfuscator: Evaluating Branch Removal as a Spectre Mitigation

Martin Schwarzl[1], Claudio Canella[1], Daniel Gruss[1],Michael Schwarz[2],

[1] Graz University of Technology, Austria [2] CISPA Helmholtz Center for Information Security, Germany

Abstract

Attacks exploiting speculative execution, known as Spectre attacks, have gained substantial attention in the scientific community and in industry with a broad range of defense techniques proposed. In particular, in-software defenses for commodity systems attempt to leave the program structure as is, but defuse every potential Spectre gadget by, e.g., stopping the speculation, or limiting value ranges. While these mitigations disrupt the program flow on every conditional branch, they still contain every single conditional branch instruction.

In this paper, we show that one dimension of Spectre mitigations has been overlooked entirely. We explore a novel principled Spectre mitigation that sits at the other end of the scale: the absence of conditional and indirect branches. Our mitigation is based on automatically linearizing the program flow through a special compiler pass, eliminating **all** conditional and indirect branches. We show that our Spectre mitigation has very clear security guarantees. We explore the feasibility of this unorthodox approach and evaluate its performance in comparison to the more conservative approaches presented so far. We observe that the performance overhead can be low, e.g., 5 %, for certain use cases, being on-par with state-of-the-art mitigations, but very high for other use cases, e.g., and overhead factor of 1000. Our results demonstrate the feasibility of Spectre defenses that eliminate branches and indicate good performance-security trade-offs for Spectre defenses can be achieved by sticking to neither of the extremes.

7.1. Introduction

Speculative execution is a significant factor in the performance of modern processors. Instead of waiting for a branch decision or branch target to be architecturally determined, the processor takes an educated guess based on behavior observed in the past. From a pipeline perspective, this linearizes the execution of instructions as the branch decision is omitted in the speculative execution flow and only subsequently validated. Spectre attacks [29] induce incorrect speculative execution flows into a victim context by manipulating the branch predictors. During this speculative execution, the attacker can make the victim access secrets and encode them into the microarchitectural state. Using a side-channel attack, e.g., Flush+Reload [54], the secrets can then be recovered.

Previous countermeasures [9, 10, 34] either attempt to thwart successful covert-channel transmission during speculation [25, 26, 53], abort the speculative execution before secrets can be accessed [1, 11, 21, 22, 37, 39, 48, 49], or ensure that secrets cannot be accessed during speculative execution [41, 42, 56]. Amit et al. [2] tried to increase the performance of indirect branches by rewriting them into two direct branches. However, from the perspective of branches in a program, all these countermeasures remain in the same range of the scale, namely all conditional and indirect branches remain in the program, in some cases even with additional branches added. This raises an important scientific question:

Can the (substantial) reduction of branches, in particular the elimination of all vulnerable branches, be a viable Spectre mitigations? Can such Spectre mitigations maintain a reasonable overhead in certain use cases?

In this paper, we answer both questions in the affirmative. To answer these scientific questions, we explore a novel Spectre mitigation at the other end of the scale: the elimination of all conditional and indirect branches. While this may sound impractical at first, it has been used for years to implement cryptographic algorithms in constant time [5]. We demonstrate the feasibility of this approach with our new mitigation, Specfuscator. Specfuscator is based on the movfuscator [12] tool that automatically linearizes the program flow through a special compiler pass. In contrast to *M/o/Vfuscator*, we do not replace all operations, but just control-flow manipulating instructions, effectively eliminating **all** conditional branches. To improve the performance of *M/o/Vfuscator*, we bring back ALU operations, the `cmp` instruction and exploit the x86 addressing mode. In

comparison to the *M/o/Vfuscator* we increase the runtime up to a factor of 50 and decrease the binary size by 30 % and compile time up to 46 %. We show that our Spectre mitigation is a principled approach with respect to security, following the simple argument that if there are no conditional or indirect branches, no branches can be mispredicted.

For our evaluation we analyzed Specfuscator in comparison with a set of other compilers: the related *M/o/Vfuscator* and LCC, a patched clang with `lfence` protections on all conditional branches, and an unpatched clang without any Spectre mitigations. We evaluate the performance of our unorthodox approach and discover that the overhead can be as low as 5 %, being on-par with state-of-the-art mitigations, but also much higher, up to factor 1000, performing clearly worse than state-of-the-art mitigations. Thus, for some use cases, the elimination of conditional and indirect branches is nearly as efficient as state-of-the-art mitigations but with a stronger security argument. This indicates that the space between the two extremes, all conditional and indirect branches and no conditional and indirect branches, should receive more attention for the design of future countermeasures.

Our key contributions are:

- We explore a previously unexplored mitigation space against Spectre: the absence of conditional and indirect branches.
- We present a solution based on a linearized control-flow with very clear and strong security guarantees.
- We evaluate our approach and observe that the performance overhead can be lower than state-of-the-art mitigations in some use cases, but also significantly higher in others.
- Our results shed light on a new direction for performance-security trade-offs for Spectre defenses.

The remainder of this paper is organized as follows. In Section 7.2, we provide background information. In Section 7.3, we discuss the landscape of existing Spectre defenses and point out blank spots. In Section 7.4, we present Specfuscator, our Spectre defense mechanism. In Section 7.5, we evaluate the performance and security of Specfuscator. In Section 7.6, we discuss the context and implications of our work. We conclude in Section 7.7.

7.2. Background

This section provides some background information about speculative execution attacks and the internals of the *M/o/Vfuscator*.

7.2.1. Speculative Execution Attacks

Modern CPUs extensively use out-of-order execution and prediction mechanisms to increase performance. Speculative execution uses branch predictions to advance the control flow speculatively. Branch prediction mechanisms are implemented via different structures, such as the Branch History Buffer (BHB) [6, 29], the Branch Target Buffer (BTB) [14, 29, 31], the Pattern History Table (PHT) [15, 29], and the Return Stack Buffer (RSB) [15, 30, 32].

Mispredicted branches are reverted on the architectural level, but not on the microarchitectural level [29]. Hence, code that should not have been executed architecturally still leaves microarchitectural traces, e.g., in various caches. By leveraging traditional side-channel attacks, these microarchitectural traces can be brought into the architectural domain, potentially recovering data that was not supposed to be accessed, *i.e.*, secrets.

Kocher et al. [29] first discussed transient-execution attacks [10] using speculative execution and demonstrated that conditional branches and indirect jumps can be exploited to leak data. Subsequent work has then shown that the idea can be extended to function returns [30, 32] and store-to-load forwarding [18]. Canella et al. [10] then systematically analyzed the field and demonstrated that the necessary mistraining can be done in the same and a different address space due to some predictors being shared across hyperthreads. Additionally, they also showed that many of the proposed countermeasures are ineffective and do not target the root cause of the problem. While the cache has been predominantly exploited for the transmission of the secret data [10, 29, 30, 32], other channels have also been shown to be effective, *i.e.*, execution port contention [7].

To mitigate all these attacks, various proposals have been made by industry and academia. Canella et al. [9] analyzed the differences between countermeasures proposed by academia and by industry, highlighting that academia proposes more radical countermeasures compared to industry. In general, the proposed mitigations either require significant changes

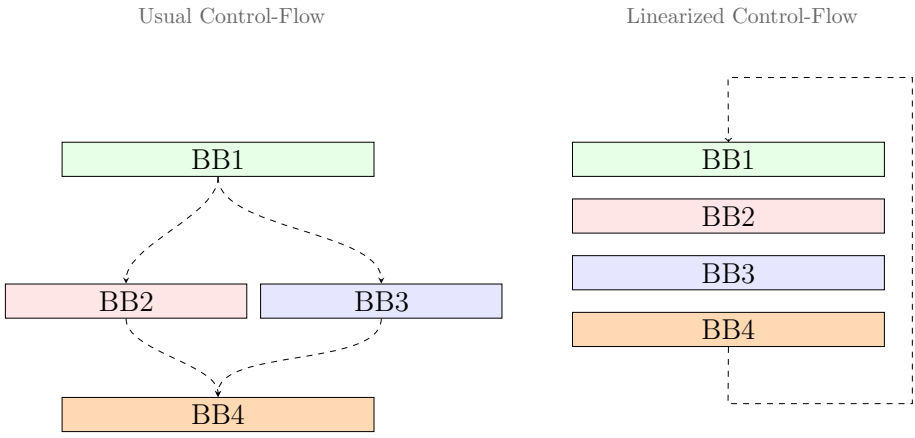


Figure 7.1.: Branch instructions typically split up the control flow. Constant-time cryptographic algorithms avoid branches (left) and instead linearize the control flow (right), e.g., square-and-always-multiply [13], turning the security-critical branches and basic blocks into one large basic block. *M/o/Vfuscator* follows the same idea of linearizing the control-flow and uses one main execution loop, turning the program into one large basic block.

to the hardware [25, 26, 53], require a developer to annotate secrets [17, 38, 42], introduce data dependencies [11, 37], or reduce the accuracy of timers [33, 40, 47, 50].

7.2.2. M/o/Vfuscator

Turing completeness is a part of computability theory that describes a set of rules or instructions that can be simulated on a single-taped Turing machine. Dolan [46] showed that the x86 `mov` instruction is Turing-complete. Based on this observation, Domas [12] invented the single-instruction compiler *M/o/Vfuscator*. The *M/o/Vfuscator* patches the Little C compiler (LCC) to use an emitter that only emits `mov` instructions. *M/o/Vfuscator* is an x86 32-bit compiler and also only supports 32-bit arithmetic operations.

The compiled program runs in a virtual machine, which basically runs like a Turing machine. The entire program is branch-free and thus executed as a single basic block, leading to a linearized control flow graph. Figure 7.1 illustrates the linearized control flow graph. Thus, the program is always executed from start to end in a loop. To ensure the correctness of the

program, a flag specifies whether an instruction should compute on the target location or a dummy *discard* location. All instructions that are not relevant for a specific iteration are discarded using this discard location. Hence, although the instruction is executed, it has no impact on the current behavior of the program. This technique is the same that is used to ensure constant-time implementations of, e.g., cryptographic algorithms [13].

Note that this is similar to constant-time cryptographic algorithms, e.g., square-and-always-multiply [13], the program executes both branches and, thus, always runs the algorithm from start to end in a loop.

Arithmetic operations, *i.e.*, additions, multiplications, divisions, bitwise-operations, are implemented using two-dimensional lookup tables. To save memory. 32-bit operations are split into two 16-bit operations, and, thus, only 16-bit lookup tables are required. By exploiting the addressing modes of x86-`mov`, the first `mov` looks up the row for the first operand, the second `mov` looks up the corresponding column for the second operand, and the value is reported as result.

M/o/Vfuscator handles internal jumps to specific parts of the code using a target register. *M/o/Vfuscator* installs two signal handlers for `SIGSEGV` and `SIGILL` to enable branching [12]. At the end of the program, an illegal instruction is emitted to trigger the `SIGILL` handler and jump back to the start of the program. To perform external library calls, *i.e.*, calling libc functions such as `printf`, segmentation faults are used [12]. To adhere to the x86 calling convention, the function's arguments are pushed onto the stack.

As the name indicates, *M/o/Vfuscator* can also be used as an obfuscation technique. However, as Kirsch et al. [27] demonstrated, it is possible to deobfuscate this technique with taint analysis.

7.3. Blank Spots in the Spectre Defense Landscape

Most Spectre countermeasures attempt to break different phases of Spectre attacks [9, 10]. These phases are described in previous work as *preparation*, *misspeculation*, *access*, *encoding*, *leakage*, and *decoding*.

Preparation. Preventing the preparation phase can often be seen as equivalent to disabling performance optimizations in the CPU. By disabling either microarchitectural states or speculation at all, an attacker is unable to prepare a Spectre attack. While disabling speculation has been suggested as a mitigation [29], modern CPUs do not support disabling speculative execution. Moreover, it can be expected that disabling speculative execution results in a considerable slowdown. Similarly, disabling the cache also has an unacceptable performance overhead as every memory access has to be served from memory. Additionally, other microarchitectural elements could be used as side channel in the absence of the cache [7, 10, 43].

Misspeculation and Preventing Access. Most focus so far was on the main cause of Spectre attacks, the misspeculation phase, or the transient access of secrets following the misspeculation. Intel, AMD, and ARM [1, 3, 23] prevent Spectre-BTB and Spectre-RSB by restricting how an attacker can influence the predictors. For Spectre-PHT, serializing instructions are recommended to stop speculation at security-critical branches [23]. However, this means that branches have to be identified and separately patched.

Furthermore, it could be that memory barrier instructions are not fully serializing [52]. To entirely protect an application, speculation barriers are required for each branch that could be followed by cache fetches. Adding memory barriers for each conditional branch can lead to runtime overheads of up to 440 % [37]. Additional to that performance overhead, Schwarz et al. [43] have shown that speculation barriers for each branch do not suffice as other channels can be used to leak data, such as the AVX unit or the TLB, as these barriers do not prevent interaction with these microarchitectural elements.

Oleksenko et al. [37] introduced data dependencies to branch conditions and the following instructions to force a stall if the branch cannot be decided yet. Similarly, Carruth [11] proposed to use branchless code to check loads, ensuring that the load is executed along a valid control-flow path. One pre-requisite for this approach is that the hardware supports branchless and unpredicted conditional updates of register values.

Schwarz et al. [42] and Fustos et al. [17] propose to annotate secrets and propagate these annotations to the CPU to ensure that secrets are inaccessible during transient execution. Speculative taint tracking (STT) [56]

uses light-weight taint tracking to taint not yet committed data and delay instructions that use it. Similarly, NDA [51] prevents the execution of potentially leaking instructions if they depend on a not yet retired operation.

All of these mitigations keep the number of branches identical but ensure that no leakage occurs by breaking the link between the `misspeculation` phase and the subsequent `access` or `encoding` phases.

Other solutions attempt to add branches that are potentially less easy to exploit [2]. Google proposed *retpoline* [48], a code sequence replacing indirect branches with return instructions, to prevent Spectre-BTB. While *retpoline* also adds more jumps to the program, these are direct jumps and, thus, likely unexploitable. Hence, the total number of branches increases, although potentially fewer are exploitable. Branco [8] proposed a probabilistic alternative to *retpoline*, called *randpoline*, which is compatible with Intel Control-flow Enforcement Technology (CET). This alternative introduces a large number of indirect branches and randomly chooses one of them, reducing the chance that an attacker can mistrain the actually executed branch.

Encoding, Leakage, and Decoding. In these phases, the secret was already accessed transiently. Preventing exploitation in these phases would require ensuring that no covert channel exists between the transient and the architectural domain. However, the way modern CPUs work, it is unrealistic to assume that covert channels can be entirely prevented. While proposals exist to limit the resolution of timers [50] or to build microarchitectural shadow structures [25, 53] to squash the results on mispredictions and leave no microarchitectural traces in the cache. However, these mitigations are typically incomplete [10].

Classification. While these defenses have different security properties, depending on the phase they target, they have in common that specific branches are either transformed into other branches, or that the flow from mispredicted branch to leakage is interrupted. We classify the existing Spectre defenses, as illustrated in Figure 7.2. From this figure, it becomes apparent that most solutions sit in the same range of keeping the number of branches identical, and some defenses increase the number of branches.

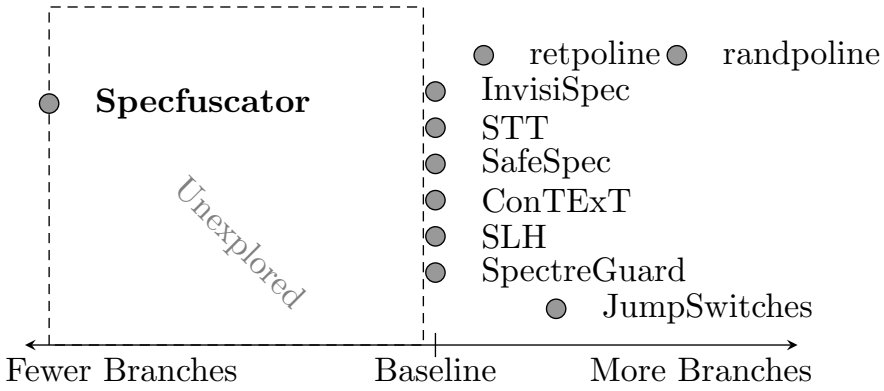


Figure 7.2.: Previous Spectre defenses were either not changing the number of conditional branches, but possibly adding more (direct) branches to a program. The space of eliminating branches is largely unexplored.

Existing software-based countermeasures try to surgically modify conditional branches or subsequent data access to prevent the exploitation of misspeculation. However, as an alternative to preventing speculative execution of conditional branches entirely [29], another possibility is to *eliminate* conditional branches. In this work, we analyze this largely unexplored mitigation technique of removing conditional branches, thus also eliminating the root cause of Spectre attacks.

7.4. Specfuscator

In this section, we introduce the design of Specfuscator in the first part. Then we discuss the security guarantees of Specfuscator and outline the implementation.

7.4.1. Design of Specfuscator

Specfuscator is based on the work by Dolan [46] showing that the x86 `mov` instruction is Turing complete. Hence, it is always possible to transform a regular application into an application that consists only of `mov` instructions, and thus *no conditional branches*. This approach has been implemented by Domas [12] as *M/o/Vfuscator* with the goal of obfuscating applications and making them difficult to reverse engineer.

The main idea is always to execute both code paths of every conditional branch, similar to the constant-time square-and-always-multiply algorithm for RSA [13]. Per conditional branch, a flag decides whether the calculated results are kept and committed to the program state, or discarded by specifying a dummy location as the target. Such an approach is also considered secure for implementing side-channel resilient cryptographic algorithms [13, 36, 55]. The advantage of this approach is that it can be fully automated in the compiler.

M/o/Vfuscator leverages the code generation of the LCC compiler but replaces the emitter for single instructions by a special emitter, generating the corresponding assembly code. *M/o/Vfuscator* labels all branches and uses a software-emulated target register to decide which of the branches is currently executed. If the execution flag is set, all operations are performed as specified in the program code. Conversely, if the flag is not set, the results of the operations are discarded, similar as square-and-always-multiply [13].

Branching is emulated using branch-free comparison using subtraction and logical operations. Depending on the result of the comparison, the corresponding flags (zero flag, signed flag, carry flag, and overflow flag) are set, and the target location is selected. A flag specific to this approach is the execution flag that can be changed by compare instructions. After disabling the execution flag, the results of the subsequent instructions are stored to a scratch location. If the instruction pointer (`EIP`) reaches the target basic block, the execution is enabled again, and the results are again made architectural.

Similar to the square-and-always-multiply loop [13], the code is always executed in its entirety in a loop. Hence, the execution speed suffers while secret-dependent operations, secret-dependent branches, and secret-leaking misspeculation are eliminated. This design leads to a linearization of the program flow. Therefore, the CPU does not need to predict the outcome of branch instructions. If there are no branches in the program, there can be no mispredictions and resulting pipeline stalls [19].

While the `mov`-based approach is already secure against Spectre attacks, it introduces a considerable performance overhead. Arithmetic operations are implemented via extensive use of two-dimensional arithmetic lookup tables. For instance, a 32-bit addition requires 50 x86 mov instructions, which use 16-bit lookup tables. To increase the performance of Specfuscator, we do not solely rely on the `mov` instruction. As we only aim to prevent

Spectre attacks, we do not implement arithmetic operations using `mov`s. Instead, we rely on the native x86 arithmetic instructions, as they cannot be exploited using Spectre. In addition, we exploit the x86 addressing modes to operate directly in memory instead of moving both operands into registers. This optimization saves one additional `mov` instruction per memory operation.

Another instruction that is safe with respect to Specter is the `cmp` instruction. Thus, Specfuscator directly uses the `cmp` instruction instead of a subtraction for comparing two values. The required flag, e.g., the execution flag, is then set via arithmetic and logic instructions. Figure 7.3 illustrates how Specfuscator emits branch-free code using `mov` instructions.

The only jump instruction in Specfuscator is the jump from the end of the program to the top of the execution loop. In *M/o/Vfuscator*, this was solved using an illegal instruction and a corresponding exception handler. However, this causes a considerable performance overhead and might even lead to misspeculation in the interrupt handler [44]. Hence, as a Spectre attack cannot exploit a direct, unconditional jump, the illegal instruction can be replaced via a direct jump to the top of the execution loop.

7.4.2. Security of Specfuscator

Specfuscator is a defense against Spectre attacks that exploit control-flow misprediction, *i.e.*, Spectre-PHT [29], Spectre-BTB [29], and Spectre-RSB [30, 32], as classified by Canella et al. [10]. Straightline Spectre [4] is a special case of Spectre-BTB and Spectre-RSB, where the CPU speculatively skips a branch and continue with the instruction directly after the branch. Another Spectre variant, Spectre-STL [18], is a separate mechanism that relies on incorrect speculations for store-to-load forwarding, *i.e.*, it is a data-flow misprediction.

The idea of Specfuscator is that none of the control-flow mispredicting Spectre variants (Spectre-PHT [29], Spectre-BTB [29], and Spectre-RSB [30, 32], including Straightline Spectre [4]), work if the corresponding control-flow modifying instructions are not used at all. Specfuscator strictly avoids these instructions and only permits direct, unconditional control flow changes. As the only emitted branch is the unconditional branch at the end of the program, adding a memory fence after this jump prevents Straightline Spectre. Due to the unconditional nature of the branch, this

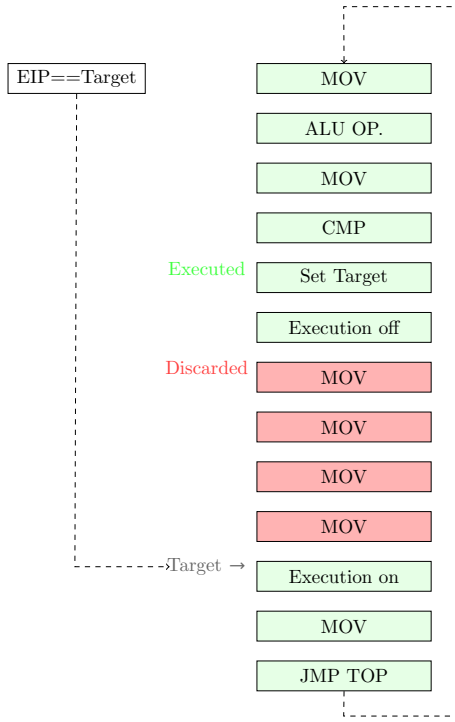


Figure 7.3.: Branching is handled via a target value for each basic block. If the target is reached, the execution flag is toggled, and the results modify the program's state. Conversely, until the target does not match, the results are written to scratch locations.

memory fence is never executed architecturally, and has therefore no performance impact. Hence, programs compiled with Specfuscator, by design, cannot be susceptible to the above Spectre variants as the corresponding instructions are not present in the binary. This is a very clear and strong security guarantee that most other defenses cannot provide [10, 29].

Specfuscator is a software-only solution and does not require hardware modifications like other proposed Spectre defense mechanisms [25, 26, 42]. Thus, it can even work in environments where other mitigations cannot be applied, e.g., because `lfence` instructions are not serializing [52], or patches are unavailable for other reasons.

7.4.3. Implementation of Specfuscator

Specfuscator is a modification to the LCC C compiler [16]. The reason we chose LCC and not gcc or clang is that we base the implementation of Specfuscator on the open-source *M/o/Vfuscator*, as this compiler already generates a branch-free binary based on the technique from Dolan [46]. *M/o/Vfuscator* itself is a patch to the current version (September 2020) of LCC. However, we require several custom changes, as outlined in Section 7.4.1. In contrast to *M/o/Vfuscator*, Specfuscator can use a broader range of native instructions without sacrificing security. By relying on arithmetic and logic operations, as well as complex addressing modes, the amount of `mov` instructions is reduced heavily, *i.e.*, for the addition, we now have 3 instructions instead of 50 `mov` instruction. For example, in a tiny AES program, the number of instructions is reduced from 222 935 to 127 631, *i.e.*, a reduction of about 43 %, when compiling with Specfuscator instead of *M/o/Vfuscator*.

As all of our changes are in the code emitter of the compiler, this could also be ported to a different compiler, such as clang. As Specfuscator is based on *M/o/Vfuscator*, we can already adopt the control-flow-linearization code from *M/o/Vfuscator* but also emit arithmetic and logic operations. Divisions and modulo operations require additional handling, as they can cause floating-point exceptions in case of a division by zero. We handle those special cases using conditional `mov` (`cmov`) instructions to ensure that we do not introduce conditional branches. The conditional `mov` instructions, e.g., `cmov`, are not affected by Spectre, as they are never predicted [20].

For comparisons, we cannot merely emit the x86 instructions instead of the `mov`-based constructs, as *M/o/Vfuscator* uses its own internal representation of CPU flags to select whether the computation results of a branch are stored or discarded. Hence, to ensure correct branching with e.g., , the `cmp` instruction, we need to update the internal flags in a branch-free way. We achieve this by transferring the CPU flags to an unused general-purpose register via the stack and using binary masks to extract the required bits.

In total, we changed (added, removed, or replaced) 437 lines of code of *M/o/Vfuscator*, which is about 10 % of the *M/o/Vfuscator* codebase.

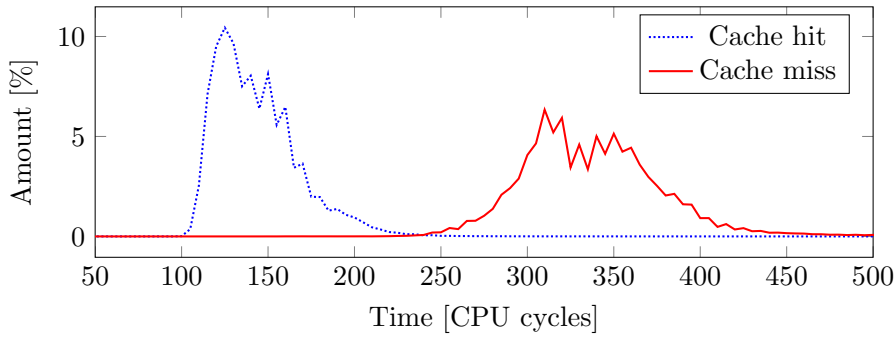


Figure 7.4.: Flush+Reload within a Specfuscator-compiled program works successfully as intended.

7.5. Evaluation

In this section, we first verify the security of Specfuscator by compiling and executing Spectre-PHT, Spectre-BTB, and Spectre-RSB gadgets. Furthermore, we evaluate the performance of Specfuscator and compare it to the original *M/o/Vfuscator*, LCC, and a modified clang version, which emits `lfences` for each conditional branch, and a basic clang compiler without Spectre mitigations activated. We compare each compiler on a set of benchmark programs and compare the averaged runtime performance, binary size, and compile time. The results of this evaluation are given in Table 7.1 and Table 7.2. Our test system was equipped with Ubuntu 20.04 (5.4.0-42-lowlatency) running on an Intel i5-8250U CPU.

7.5.1. Security Evaluation

We demonstrate that it is impossible to successfully use an existing Spectre proof-of-concept attack on Specfuscator compiled code. To verify that the *misspeculation* is indeed prevented, we separately validate all other Spectre attack steps. We add additional functionality to the compiled binaries to obtain accurate time measurements with `rdtsc` and enable flushing of a virtual address using the `clflush` instruction. This allows us to accurately verify the cache encoding of the Spectre attack with a Flush+Reload side-channel.

We verify that the cache covert channel in a compiled binary works exactly as in a regular Spectre attack by creating a histogram of cached and

uncached data. Figure 7.4 shows that it is still possible to distinguish between cached and uncached data in a program compiled with Specfuscator. Therefore, cache-based side-channel attacks are still possible in Specfuscator-compiled programs.

To validate whether Spectre is still possible, we use the 15 sample Spectre gadgets from Kocher [28]. First, we evaluate that these gadgets indeed sucessfully show Spectre attacks. We compile them using the unmodified LCC and execute each gadget 100 000 times. For all gadgets, we successfully leak data using Spectre.

For the security evaluation, we compile all sample gadgets using Specfuscator. We again execute each gadget 100 000 times on our test device, and check whether the secret is leaked. As we do not observe any leakage on our test device using any of the gadgets, we practically confirm that our mitigation that should work in theory due to the absence of misspeculation, also works in practice.

In addition, we port a Spectre-BTB and Spectre-RSB proof-of-concept to 32-bit and evaluate it on our unmitigated clang. Again, as expected, these proof-of-concepts work on an unmitigated clang. When the programs are compiled with Specfuscator, no indirect jumps, calls, or return instructions are emitted. To experimentally show that Specfuscator indeed stops the leakage for these attacks, we again compile them using our defense. We execute the proof-of-concept implementations 100 000 times and do not observe any leakage for either Spectre-BTB or Spectre-RSB.

7.5.2. Performance Evaluation

For the evaluation, we extend LLVM 10.0.1 with a new compiler pass that runs just before the final code is emitted. In this pass, we analyze every conditional branch using the `analyzeBranch` function and insert an `lfence` instruction if this instruction is not already present. To mitigate speculation on both sides of a conditional branch, we also emit an `lfence` instruction in its fall-through basic block if this code path is not already fenced. This compiler pass required changing or adding 125 lines of code across 4 files. In addition to enabling our fencing pass, we enable the *retpoline* mitigation for `clang` by adding the `-mretpoline` flag. As a result, speculative execution is stopped for all conditional branches and jumps, as e.g., , suggested by Intel [23].

For our evaluation, we compare different programs, including cryptographic implementations and real-world applications [12]. We compile each program as a 32-bit binary since our Specfuscator proof-of-concept only supports 32-bit. However, while we showcase our compiler for this architecture, our approach is generic and is equally applicable to other architectures as well.

We compile the same benchmark program in 5 different configurations. Each test case is compiled with `clang` without any Spectre mitigations, `clang` with `lfences` and *retpoline* active, the LCC, the unmodified *M/o/Vfuscator*, and Specfuscator. To get stable benchmarking results, we fixed the CPU frequency to 3.4 GHz and ran our test program on an isolated core.

Run time

We use the runtime of the `clang`-compiled programs without mitigations as a baseline to compute the runtime overhead. To measure the runtime of the programs, we use the `perf` command-line tool. We run each test case 1000 times. For the individual test cases, we observe standard deviations between 0.1 % and, for some cases, 3 %. The maximum value of 3 % was observed in the case of clang. The reason for this higher standard deviation might be speculative execution.

As shown in Table 7.1, the runtime overhead factor strongly depends on the different tasks being executed. We gained a runtime speedup in comparison to *M/o/Vfuscator* by a runtime factor of up to 50. For our benchmark programs, we observe that the LCC has a runtime overhead between 3 % and an overhead factor of 26 over `clang`. The overhead of *M/o/Vfuscator* is substantially higher, and the overhead of Specfuscator is in between. We observe the highest performance penalties in terms of runtime for a tiny program that calculates the square root of 2. Also, the modified clang reaches a maximum runtime overhead factor of 20.89. The performance of *M/o/Vfuscator* and Specfuscator deteriorates, particularly on programs where small amounts of code are executed a large number of times, as the whole program has to be completely executed for each iteration.

We leave it as future work to further optimize Specfuscator optimizing the way how branches are performed. Partial control flow linearization could be integrated as compiler optimization with a similar approach proposed

Table 7.1.: Average runtime overhead factor of our benchmarks for the different compilers compared to our baseline (clang). The baseline is given in milliseconds on the right for the unmodified `clang`

Test program	*M/o/Vfuscator*	Specfuscator	Clang (fences)	LCC	Clang (baseline)
aes	424.17	221.53	1.31	1.17	1.13 ms
arcfour	36.86	5.18	1.01	1.14	0.81 ms
base64	27.12	8.95	1.19	1.15	0.80 ms
blowfish	129.41	40.79	1.26	1.14	1.10 ms
des	1046.20	520.47	1.15	1.04	0.93 ms
md2	85.57	62.73	1.07	1.20	0.82 ms
md5	18.30	4.71	1.03	1.13	0.80 ms
rot-13	2.20	1.46	1.02	1.24	0.76 ms
arithmetic	1.25	1.05	1.05	1.03	0.96 ms
crc32	7.80	3.45	1.24	1.17	0.88 ms
hello	1.10	1.11	1.00	1.04	0.89 ms
maze	310.03	88.98	1.10	1.13	0.97 ms
mersenne	4.12	1.31	1.02	1.13	0.80 ms
sqm	1.33	1.25	1.02	1.15	0.80 ms
nqueens	319.84	234.46	1.99	4.99	1.89 ms
prime	980.27	161.59	1.93	0.96	1.65 ms
s2	46085.82	981.20	20.89	26.64	0.71 ms
sudoku	656.91	149.69	2.15	1.17	1.13 ms

by Moll et al. [35]. The partial control flow linearization improved the performance of the overall program by a factor of 146 %[35]. Furthermore, we leave it as future work to extend Specfuscator to 64-bit architecture or integrating a similar approach to LLVM. As LLVM has significantly better optimizations than LCC, as can be seen in the benchmarks, porting Specfuscator to LLVM will also improve its performance.

In addition to the runtime, we evaluate the binary size and compile time of the different compilers. For this purpose, we compile each program 1000 times for our 5 compilers and measure the compilation time using the `perf` command-line tool. Table 7.2 illustrates the averaged overhead factor in terms of binary size and compilation time.

Compile-time

Table 7.2 lists the compile-time and the binary size of our benchmark programs. In comparison to *M/o/Vfuscator*, we reduce the compile time by up to 46 %. The compile-time of *M/o/Vfuscator* and Specfuscator depends on a part in how many instructions are needed to generate the assembler. Thus, with the use of fewer instructions per operation, the compile-time is halved in most cases for Specfuscator in comparison to the original *M/o/Vfuscator*. As the results of Table 7.2 show, the compile-time is about two times higher than with the `clang` compiler. For small programs, the compile-time appears to be relatively constant for the *M/o/Vfuscator* and also Specfuscator. While this is not problematic for smaller binaries, compiling large software projects such as browsers or web servers would take substantial amounts of time with Specfuscator. We note that our approach of eliminating all conditional branches is extreme. Still, it shows that solutions that eliminate conditional branches are not infeasible, and less extreme solutions in this direction could maintain higher performance levels.

Binary size

Stripping the binary reduces the binary size by 50 %, as it removes debugging information. Hence, for a fair comparison, we strip all the binaries to only compare the actual code footprint. Compared to *M/o/Vfuscator*, Specfuscator reduces the binary size by roughly 30 % This reduction was achieved by removing most of the two-dimensional lookup tables used for arithmetic operations. The binary size could additionally be reduced by decreasing the size of the virtual stack, which is currently constant at 1.68 MB. As can be seen from Table 7.2, the binary size is about 280 times larger for Specfuscator than for binaries compiled with `clang` and for *M/o/Vfuscator* even 398 times. Again, this overhead is due to our extreme solution, but it shows that solutions eliminating conditional branches are not infeasible. Surprisingly, the programs compiled with LCC are smaller than the programs compiled with the unmodified `clang`.

Table 7.2.: Average compile time in ms and binary size in kB overhead factor for *M/o/Vfuscator*, Specfuscator, and `clang` with active mitigations compared to `clang` without active mitigations (rightmost column).

Test program	*M/o/Vfuscator*		Specfuscator		Clang (fences)		LCC		Clang (baseline)	
	time	size	time	size	time	size	time	size	time	size
hello	2.23	388.28	1.86	279.18	1.07	1.01	0.71	0.89	38.36 ms	13.62 kB
maze	3.93	394.09	2.12	274.96	1.05	1.01	0.63	0.86	46.80 ms	13.82 kB
mersenne	3.00	396.28	1.83	279.84	1.02	1.01	0.70	0.89	41.90 ms	13.63 kB
nqueens	2.39	386.75	2.05	278.22	1.17	1.01	0.75	0.88	40.19 ms	13.64 kB
prime	2.39	389.97	1.81	279.47	1.06	1.01	0.62	0.89	39.02 ms	13.64 kB
s2	2.87	395.22	1.89	279.72	1.00	1.01	0.78	0.89	39.34 ms	13.62 kB
sudoku	3.47	398.10	2.05	280.39	1.10	1.01	0.68	0.91	37.76 ms	14.00 kB
aes	4.80	218.69	2.95	151.15	1.20	1.00	0.53	1.01	101.89 ms	33.21 kB

7.6. Discussion

The goal of our paper is to clearly demonstrate the feasibility of branch reduction up to complete elimination as a Spectre mitigation. While we demonstrated the feasibility, we also identified the limitations of our extreme approach. Due to these limitations, we do not consider Specfuscator a real-world solution, but an important contribution as an explorational study that yields interesting insights. Eliminating all branches to reduce the susceptibility to Spectre has not been explored so far. Our solution inherits the performance overheads of the underlying compiler (LCC and its modification *M/o/Vfuscator*) that falls far behind the state of the art performance-wise. The fact that it can still achieve on-par performance for specific programs protected with state-of-the-art mitigations with a state-of-the-art compiler shows that the elimination or reduction of branches is a strategy to defeat Spectre that must be examined in more detail. In particular, we see potential synergies with the compiler community that explored the question of branch elimination in the past for performance reasons. For instance, Moll et al. [35] developed a technique to partially linearize the program flow by removing branches, improving performance by 146 %. Exploring related techniques, even if they incur a subtle performance overhead, may yield more efficient Spectre mitigations in future compilers. Software-based solutions are especially important as there is a lot of hardware without in-silicon fixes, and existing software-workarounds are often expensive. While Intel recommends keeping the

number of branches as low as possible to achieve the highest possible runtime performance [19], actually reducing branches is a complex task. Although branch elimination can boost the program's performance, it might also be exploited, as it has been demonstrated in the JavaScript engine V8 [24, 45]. Another direction of research is to investigate the susceptibility to control-flow hijacking attacks. Future work should evaluate whether branch-less binaries, like those compiled with Specfuscator, or branch-reduced binaries, could realistically mitigate such attacks and, thus, provide control-flow integrity.

7.7. Conclusion

Speculative execution attacks, known as Spectre attacks, have gained substantial attention both in the scientific community and in industry with a broad range of defense techniques proposed. In particular, in-software defenses for commodity systems attempt to leave the program structure as is, but defuse every potential Spectre gadget, e.g., by stopping the speculation, or limiting value ranges. While these mitigations disrupt the program flow on every conditional branch, they still contain every single conditional branch instruction. In this work, we explore a new possibility of mitigating Spectre attacks by using a branch-free compiler. Our mitigation is based on automatically linearizing the program flow through a special compiler pass, eliminating **all** conditional and indirect branches. We showed the security guarantees of this approach and evaluated the feasibility by evaluating its performance in terms of its runtime. In addition, we discussed the compile-time and the binary size of this approach. Furthermore, we verified that existing Spectre-PHT, Spectre-BTB, and Spectre-RSB proof-of-concepts compiled with Specfuscator do not leak secret data anymore. We observe that the performance overhead can be very low, e.g., 5 %, for specific use cases, being on-par with state-of-the-art mitigations. However, we also observed very high overheads of factor 1000 for other use cases. Our results indicate that the best performance-security trade-off for Spectre defenses can be achieved by sticking to neither of the extremes.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Funding was provided by generous gifts from Cloudflare, from Intel, Red Hat and from ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Advanced Micro Devices Inc. *Software Techniques for Managing Speculation on AMD Processors*. Revison 7.10.18. 2018.

[2] Nadav Amit, Fred Jacobs, and Michael Wei. "Jumpswitches: restoring the performance of indirect branches in the era of spectre." In: *USENIX ATC*. 2019.

[3] ARM. *Cache Speculation Side-channels*. Version 2.4. 2018.

[4] ARM. *Straight-line Speculation*. Version 1.0. 2020.

[5] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[6] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. "Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls." In: *Cryptology ePrint Archive, Report 2017/968* (2017).

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

[8] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. *Efficient mitigation of side-channel based attacks against speculative execution processing architectures*. US Patent App. 16/023,564. 2019.

[9] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. "Evolution of Defenses against Transient-Execution Attacks." In: *GLSVLSI*. 2020.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/. 2019.

[11] Chandler Carruth. *RFC: Speculative Load Hardening (a Spectre variant #1 mitigation)*. 2018.

[12] Christopher Domas. *M/o/Vfuscator*. 2015. URL: `https://github.com/xoreaxeaxeax/movfuscator`.

[13] Jean-Sébastien Coron. "Resistance against differential power analysis for elliptic curve cryptosystems." In: *CHES*. 1999.

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

[15] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2016.

[16] Christopher W Fraser and David R Hanson. *A retargetable C compiler: design and implementation*. 1995.

[17] Jacob Fustos, Farzad Farshchi, and Heechul Yun. "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks." In: *DAC*. 2019.

[18] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[19] Intel. *Avoiding the Cost of Branch Misprediction*. 2020. URL: `https://software.intel.com/content/www/us/en/develop/articles/avoiding-the-cost-of-branch-misprediction.html`.

[20] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[21] Intel. *Intel Analysis of Speculative Execution Side Channels*. Revision 4.0. 2018.

[22] Intel. *Retpoline: A Branch Target Injection Mitigation*. Revision 003. 2018.

[23] Intel. *Speculative Execution Side Channel Mitigations*. Revision 3.0. 2018.

[24] Jeremy Fetiveau. *Circumventing Chrome Typer Bugs*. 2020. URL: https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/.

[25] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation." In: *DAC*. 2019.

[26] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors." In: *MICRO*. 2018.

[27] Julian Kirsch, Clemens Jonischkeit, Thomas Kittel, Apostolis Zarras, and Claudia Eckert. "Combating Control Flow Linearization." In: *32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*. 2017. URL: https://www.sec.in.tum.de/i20/publications/combating-control-flow-linearization/@@download/file/CFL.pdf.

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

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

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

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

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

[33] Microsoft. *Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer*. 2018.

[34] Matt Miller. *Mitigating speculative execution side channel hardware vulnerabilities*. 2018.

[35] Simon Moll and Sebastian Hack. "Partial control-flow linearization." In: *Proceedings of the 39th Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 543–556. DOI: 10.1145/3192366.3192413.

[36] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. "The program counter security model: Automatic detection and removal of control-flow side channel attacks." In: *International Conference on Information Security and Cryptology*. 2005.

[37] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. "You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass." In: *arXiv:1805.08506* (2018).

[38] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. "Mitigating data leakage by protecting memory-resident sensitive data." In: *ACSAC*. 2019.

[39] Andrew Pardoe. *Spectre mitigations in MSVC*. 2018.

[40] Filip Pizlo. *What Spectre and Meltdown Mean For WebKit*. 2018.

[41] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *USENIX Security Symposium*. 2019.

[42] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre." In: *NDSS*. 2020.

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

[44] Martin Schwarzl, Michael Schwarz, Thomas Schuster, and Daniel Gruss. "It's not Prefetch: Speculative Dereferencing of Registers." In: *(in submission)* (2020).

[45] Sense Post. *v8 - Documentation*. 2020. URL: https://sensepost.com/blog/2020/intro-to-chromes-v8-from-an-exploit-development-angle/.

[46] Stephen Dolan. *mov is Turing-complete*. 2013. URL: https://drwho.virtadpt.net/files/mov.pdf.

[47] The Chromium Projects. *Actions required to mitigate Speculative Side-Channel Attack techniques*. 2018.

[48] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: https://support.google.com/faqs/answer/7625886.

[49] *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*. ARM, 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[50] Luke Wagner. *Mitigations landing for new class of timing attack*. 2018.

[51] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. "Nda: Preventing speculative execution attacks at their source." In: *MICRO*. 2019.

[52] *x86/cpu/AMD: Make LFENCE a serializing instruction*. LKML, 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e4d0e84e490790798691aaa0f2e598637f1867ec.

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

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

[55] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing." In: *NDSS*. 2019.

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

8

Remote Memory-Deduplication Attacks

Publication Data

Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. "Remote Memory-Deduplication Attacks." In: *NDSS*. 2022

Contributions

Main author.

Remote Memory-Deduplication Attacks

Martin Schwarzl[1], Erik Kraft[1], Moritz Lipp[1], Daniel Gruss[1]

[1] Graz University of Technology, Austria

Abstract

Memory utilization can be reduced by merging identical memory blocks into copy-on-write mappings. Previous work showed that this so-called *memory deduplication* can be exploited in local attacks to break ASLR, spy on other programs, and determine the presence of data, *i.e.*, website images. All these attacks exploit memory deduplication across security domains, which in turn was disabled. However, within a security domain or on an isolated system with no untrusted local access, memory deduplication is still not considered a security risk and was recently re-enabled on Windows by default.

In this paper, we present the first fully remote memory-deduplication attacks. Unlike previous attacks, our attacks require no local code execution. Consequently, we can disclose memory contents from a remote server merely by sending and timing HTTP/1 and HTTP/2 network requests. We demonstrate our attacks on deduplication both on Windows and Linux and attack widely used server software such as Memcached and InnoDB. Our side channel leaks up to 34.41 B/h over the internet, making it faster than comparable remote memory-disclosure channels. We showcase our remote memory-deduplication attack in three case studies: First, we show that an attacker can disclose the presence of data in memory on a server running Memcached. We show that this information disclosure channel can also be used for fingerprinting and detect the correct libc version over the internet in 166.51 s. Second, in combination with InnoDB, we present an information disclosure attack to leak MariaDB database records. Third, we demonstrate a fully remote KASLR break in less than 4 minutes allowing to derandomize the kernel image of a virtual machine over the Internet, *i.e.*, 14 network hops away. We conclude that memory deduplication must also be considered a security risk if only applied within a single security domain.

8.1. Introduction

Memory deduplication is a widely used technique to reduce memory utilization by detecting physical pages with the same content and merging them. Merged pages are marked as read-only and copy-on-write. If one of the merged pages is modified, a copy-on-write page fault is triggered, and the page is again copied to a new physical location. With the introduction of Windows 8.1, memory deduplication had become a default feature [58]. On Linux, kernel-same-page merging is used by kernel-virtual machines or if the `madvise` syscall is used with a flag indicating that the page is mergeable.

Previous work demonstrated memory-deduplication attacks performed by a local attacker in both local environments (*i.e.*, local native code execution) and the cloud (*i.e.*, local code execution in a virtual machine) [4, 50] exploiting page combining on Windows and kernel-same-page merging on Linux. Memory-deduplication attacks can detect co-location in the cloud [50], hide communication in virtualized environments [56, 57], fingerprint operating systems [39], fingerprint websites via JavaScript [15] and break ASLR on Linux as well as on Windows by exploiting pages with almost fixed content [4]. Bosman et al. [6] leveraged memory deduplication in combination with Rowhammer to escape from a browser sandbox. Razavi et al. [41] used memory deduplication to facilitate Rowhammer attacks on co-located virtual machines. Palfinger et al. [40] demonstrated that memory deduplication can also be exploited in file systems like ZFS. Lindemann et al. [26] demonstrated efficient fingerprinting via memory deduplication in co-located virtual machines. In concurrent work, Kim et al. [21] showed a KASLR break on virtual machines on VMWare ESXi. Following the recommendation of all these attack papers, memory deduplication was disabled on Linux and Windows by default.

More recently, vendors switched to more fine-grained security policies. Windows 10, for instance, again enables page combining by default but restricts it to only deduplicate within a security domain but not across security domains, stopping existing attacks. We also observe that the popular Ubuntu 20.04 Linux distribution enables kernel-same-page merging by default for KVM-based virtual machines. Memory-deduplication attacks with local code execution are considered out of scope in their threat model. Systems without local code execution (native or in a virtual machine) for the attacker can still be considered secure with these mitigation strategies.

However, it remains unclear whether remote attacks *without local code execution* are possible.

Our work faces three challenges which have to be solved to perform remote memory-deduplication attacks:

- *C1: Remotely amplify latencies for non-repeatable events.* Remote timing attacks require high latencies in the side channel to deal with noisy networks. Page-fault-type interrupts cannot be arbitrarily repeated (e.g., for copy-on-write page faults, the page is copied and writeable after the page fault). Hence, existing amplification techniques are not directly applicable.
  All previous memory deduplication attacks focused on cross-domain deduplication. Deduplication within one domain is considered secure (Windows re-enabled it for that reason). Intra-domain deduplication is visible outside of the domain if the timing latency is exposed over a web server or public API to the attacker domain.
- *C2: Trigger and observe copy-on-write pagefaults in a victim domain that shares no memory with any attacker domain.* All previous memory deduplication attacks require local code execution (in native or sandboxed code). Remote requests are usually not held in memory for a long time. To speed up, the access of frequent data, in-memory caching mechanisms like Memcached are used in websites.
- *C3: Find remote request paths that do not only keep attacker-controlled data in memory but also provide the attacker with control over alignment and in-memory representation.* To enable byte-by-byte leakage, a target is required that allows alignment changes as described by Bosman et al. [6].

In this paper, we solve the mentioned challenges and demonstrate the **first fully remote memory-deduplication attacks**, just using requests to an HTTP web server. Our attacks infer timing differences caused by copy-on-write page faults on the server from the latency of network requests and responses. We demonstrate attacks on default-configured and fully updated Windows (native) and Linux (virtual machines) installations using default-configured standard server software such as Memcached. We measure the capacity of our side channel in a remote covert channel scenario and achieve a transmission rate of 302.16 B/h in a local area network and 34.41 B/h over the internet, which is faster than comparable remote memory-disclosure channels (e.g., NetSpectre [48] achieved 7.5 B/h in a local area network).

We demonstrate three different remote memory-deduplication attacks, illustrating the potential of our technique. In the first attack, we disclose the presence of data on a remote server running Memcached. The information disclosure works by uploading data blobs into the key-value store, freeing the deduplicated item, getting the same item reassigned, and triggering a copy-on-write page fault by modifying the page's content. We also exploit this information disclosure channel for fingerprinting, *i.e.*, which shared libraries are used on the remote system. Our attack detects the correct libc version over the internet in 166.51 s.

In the second attack, we present a fully remote KASLR break on a virtual machine running on a remote cloud machine. By targeting kernel pages that contain kernel addresses but have all remaining bytes of the page fixed, we can successfully derandomize the kernel offset of a Linux virtual machine. We show that we can not only mount this attack in a local area network setting using HTTP/1 but, moreover, leverage HTTP/2 to successfully break KASLR on a server that is 14 network hops away within 4 minutes. We emphasize that vendor responses to local KASLR breaks are often that KASLR is only meant as a mitigation for remote attacks.

In a third attack, we disclose database records byte-by-byte from a MariaDB database server with an InnoDB storage engine. Our attack works by crafting requests that create byte misalignments within target pages, allowing byte-wise content guessing. This attack is particularly dangerous as it leaks attacker-unknown memory contents from a remote server, similar as in powerful Spectre attacks [23, 48]. We can leak 1.5 B/h in a local area network.

We conclude that memory deduplication must also be considered a security flaw if only applied within a security domain and even if local attackers are excluded from the threat model. As our attacks are full remote attacks, we emphasize that the remote attack vector has to be mitigated as well. Consequently, we responsibly disclosed all of our attacks to the corresponding vendors and work with them on finding mitigations before the public release of this paper. We will open-source our tools on GitHub with the conclusion of the responsible disclosure [1].

Responsible Disclosure. We responsibly disclosed our findings to Microsoft, Red Hat, Canonical, and AWS, on February 8th, 2021. The issues are tracked under CVE-2021-3714.

[1] `https://github.com/IAIK/Remote-Page-Deduplication-Attacks`

Contributions. The main contributions of this work are:

1. We present the first fully remote memory-deduplication attacks and show that these must be considered a security flaw even if only applied within a security domain.
2. We show that we can remotely fingerprint shared libraries to infer the exact versions via Memcached in-memory databases.
3. We present a fully remote KASLR break on a Linux virtual machine running in the cloud within only 4 minutes.
4. We demonstrate a fully remote byte-by-byte memory disclosure attack on a MariaDB database server with an InnoDB storage engine, leaking 1.5 B/h.

Outline. The remainder of the paper is organized as follows. In Section 8.2, we provide the required background about memory deduplication and remote timing attacks. In Section 8.3, we state a threat model and provide an attack overview. In Section 8.4, we present the attack primitives that we use for remote memory-deduplication attacks. In Section 8.6, we evaluate the performance of our remote memory-deduplication attacks in three case studies on Windows and Linux, targeting Memcached, MariaDB (with InnoDB), and the Linux kernel. In Section 8.7, we discuss the results and state-of-the-art mitigations for remote memory-deduplication attacks. We conclude in Section 8.8.

8.2. Background

In this section, we provide background on memory deduplication, memory-deduplication attacks, and remote timing attacks, as well as Address Space Layout Randomization.

8.2.1. Memory Deduplication

Sharing memory is not only crucial for inter-process communication but also to reduce memory utilization and cache pressure. Modern operating systems use different techniques to use shared memory whenever possible. For instance, when creating a new process with `fork()`, the memory is marked as *copy-on-write*, meaning that it is first shared between parent and child process and only copied (*i.e.*, duplicated) when one of the processes attempts to write to it. This is implemented by marking the

memory read-only and raising a page fault upon a write access. Another example is the loading of any type of file (including, e.g., a program or library binary files). The operating system keeps files in the page cache and maps them into all processes that request access.

Neither of these approaches leads to the deduplication of identical but dynamically generated memory pages. Hence, operating systems have introduced content-based memory deduplication, which regularly scans the entire physical memory for pages with identical content. All but one of the identical pages are released, while the remaining one is marked as copy-on-write. Content-based memory deduplication has traditionally been applied across all security domains on all major operating systems. On Windows, the mechanism is called page combining [58] and kernel same-page merging on Linux [3]. However, security research has revealed that this enables a range of attacks, as we discuss in the next sub-section.

8.2.2. Memory-Deduplication Attacks

In a memory-deduplication attack, the attacker first generates candidate pages for deduplication. If the attacker guesses the content of a page in memory fully correctly, it is deduplicated. Until the deduplication took place, the attacker repeatedly writes to the candidate pages (without changing the content). As soon as the deduplication took place, this triggers a copy-on-write page fault, increasing the access latency drastically. Hence, the access latency reveals whether a victim process had a page with the exact same content, *i.e.*, memory deduplication forms a content-probing oracle.

The first memory-deduplication attack, demonstrated by Suzaki et al. [50], was used to detect applications running in other virtual machines. Owens et al. [39] also exploited memory deduplication to fingerprint the operating system version via unique pages per operating system in virtual machines. Gruss et al. [15] showed that memory-deduplication attacks are possible from JavaScript running on a website opened in a browser. Barresi et al. [4] demonstrated that it is possible to break address space layout randomization (ASLR) on both Windows and Linux using memory deduplication. Razavi et al. [41] exploited memory deduplication to perform Rowhammer attacks on applications in virtualized environments. Bosman et al. [6] used memory-deduplication attacks to create more sophisticated exploits and used the ASLR break via memory deduplication

to create an end-to-end JavaScript exploit which leverages Rowhammer to achieve arbitrary memory read and write. Oliverio et al. [38] proposed a mitigation against active memory-deduplication attack called VUsion, which enforces same behavior when accessing shared and non-shared pages, a write-xor-fetch policy, and random memory allocation. Lindemann et al. [26] showed another fingerprinting attack to detect co-location in virtual machines.

Palfinger et al. [40] showed that memory deduplication can also be leveraged in file systems like ZFS to fingerprint the operating system in the cloud of co-located machines. In concurrent work, Kim et al. [21] demonstrated a KASLR break on VMWare ESXi.

8.2.3. Remote Timing Attacks

Timing attacks were heavily researched in the last two decades. Since network connections are getting more and more stable, at higher transmission rates, as well as lower and more consistent latencies, remote timing attacks have become increasingly interesting for attack research. Brumley and Boney et al. [7] demonstrated that it is possible to extract SSL private keys over a local area network. Acıiçmez et al. [1] attacked AES via a remote cache based attack. In 2009, Crosby et al. [10] showed the possibilities of remote timing attacks and how to reliably determine the number of requests required to distinguish certain timing differences over the network. There were several remote timing attacks on AES [2, 20, 47, 59] following Bernstein's idea of attacking AES [5]. Van Goethem et al. [51] exploited timing side channels in browsers. Irazoqui et al. [18] showed that it is possible to exploit cache timing differences in TLS in a local area network. Van Hoef et al. [53] leveraged TCP windows to observe the exact size of a cross-origin resource. Van Goethem et al. [52] showed that remote timing attacks can be performed over the Internet by exploiting concurrency in HTTP/2 and observing the order the packets return, which depends on the server-side timing, instead of the client-side timing. Kurt et al. [25] showed that Data Direct I/O can be used in combination with Remote Direct Memory Access to spy on keystrokes during SSH sessions.

More closely related to our work is Schwarz et al. [48], who showed that Spectre attacks are possible over the network if certain gadgets exist on the target system. Similar to the most powerful attacks we present, they can leak arbitrary data from an execution context. They achieve a leakage

rate of up to 7.5 B/h, which can be sufficient to leak a cryptographic key over the time frame of multiple hours.

8.2.4. Address Space Layout Randomization

To exploit memory corruption bugs, the knowledge of addresses of specific data is often required since address randomization is applied in both user space and kernel space. Over the past years, different side-channel attacks allowed to reduce the entropy of the randomization or to break it entirely. Hund et al. [17] measured the execution time of page-fault handling to observe which kernel addresses are mapped and thus cached in the TLB. Jang et al. [19] used hardware transactional memory to observe the same effect. Other software-based side channel attacks exploited predictors [12, 27], side channels introduced by mitigations against other attacks [9], the power consumption of the processor [28], and other microarchitectural properties [13, 16, 24], even from JavaScript [8, 14]. As a consequence of these local attacks on KASLR, operating system vendors but also parts of the academic community considered KASLR only as a defense against remote attackers. In remote attacks, KASLR indeed is still considered a valuable line of defense since the attacker cannot as easily probe the address space as with local attacks.

8.3. Threat Model & Attack Overview

In our threat model, the attacker has no ability to execute code on the target machine: not natively, not in a virtualized environment, and also not via JavaScript [6, 15] or another scripting language. However, the attacker can provide attacker-controlled content to the remote target, e.g., a network request the attacker sends to the host with content the attacker controls.

We assume that the victim keeps the attacker-controlled content in RAM. This occurs, for instance, if the attacker sends network requests that are cached in request pools, or binary large objects provided to a web application and later on stored in a database or cached in a buffer.

We assume that memory deduplication techniques are active on the victim's machine. We emphasize that this is the case under default settings on

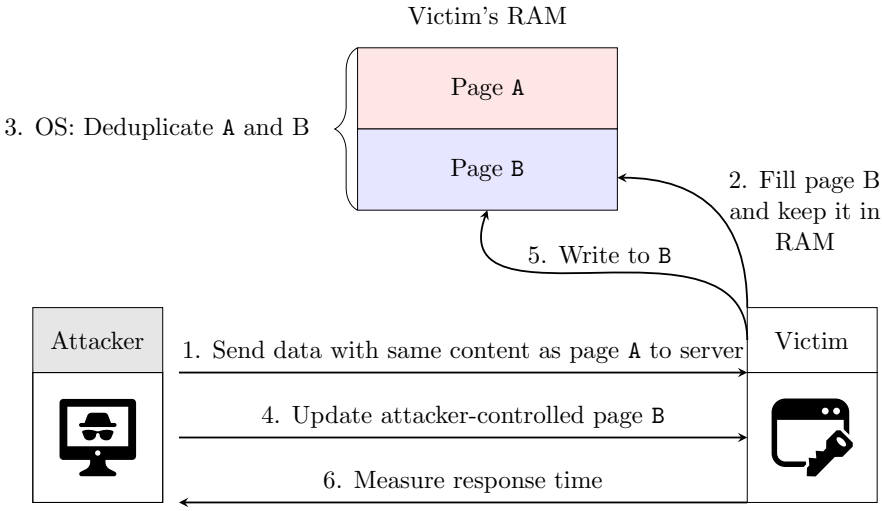


Figure 8.1.: Overview of a remote memory-deduplication attack.

current Ubuntu Linux installations (kernel-same-page merging for virtual machines) and on current Windows installations (page combining).

We make no assumptions about software bugs, for instance, memory safety violations in the applications we analyze.

Attack Overview. Six steps are required to perform a remote memory-deduplication attack as illustrated in Figure 8.1. First, the attacker sends a request to the victim with a page of data (page B) containing the same content as a page already present in memory (page A). Afterwards, the attacker waits for some time until the two pages are merged by the operating system and point to the same physical address. Next, the attacker updates the attacker-controlled data and triggers a page-fault on the victim application. Depending on the response time of the victim, the attacker observes whether the page was deduplicated or not.

Difference to already presented attacks. All of the previous presented attacks [4, 6, 15, 39, 50] require local code execution via a native binary or JavaScript and co-location to the victim's machine. Remote memory-deduplication attacks extend the scope by enabling attacks on remote web servers by exploiting an API that allows uploading of attacker-controlled data and place it into the main memory such that it might be deduplicated.

Comparison of state-of-the-art memory deduplication attacks to our work is listed in Table 8.1. While some of the techniques shown by previous work are similar, we solved those challenges for memory deduplication attacks in the context of a remote attacker. As evidenced by other fully remote attacks [48, 52], specific timing requirements and the applicability to many-hop internet connections, remain a challenge that is only solved for specific cases.

8.4. Attack Primitives

In this section, we describe our basic attack primitives and define the requirements for a remote attacker to perform a fully remote memory-deduplication attack without execution of any attacker-controlled code on the victim system.

The main primitives for our attack are memory deduplication being enabled, a web service/API that lets a remote attacker read/modify data stored in RAM and an accurate remote timer that allows distinguishing the round-trip time of the network packets.

8.4.1. Memory Deduplication

Page combining. Page combining was introduced in Windows 8.1. On Windows, a special kernel thread scans over the whole memory to detect pages that have identical content [58]. This scan is triggered about every 15 minutes on Windows 10 [58]. If pages with identical content are found, the pages are combined to a single page to save memory. The page-table entries of the pages then point to one of the two pages, which is then shared across processes and marked as read-only and copy-on-write. When writing to this shared page, a copy-on-write fault occurs, and a new copy of the page is created for the writing process [58].

Page combining can easily be disabled via the Windows registry or using Powershell, e.g., using the `Disable-MMAgent` command. Page combining was temporarily disabled for security reasons after several memory-deduplication attacks were discovered [4, 6, 15]. However, page combining was re-enabled on Windows more recently and is active on desktop machines by default, as well as on server machines if the `full` terminal server

Attacks	Location	Environment	Local	Type	Attack Type	Perf
Suzaki et al. [50]	Co-located	Cross-VM (Cloud)	Yes	Native binary	Fingerprinting	-
Owens et al. [39]	Co-located	Cross-VM (Cloud)	Yes	Native binary	Fingerprinting	-
Gruss et al. [15]	Remote	Browser/Cross-VM (Cloud)	Yes	JS	Fingerprinting	-
Barresi et al. [4]	Remote	Cross-VM (Cloud)	Yes	Native binary	ASLR break	8.7 days
Bosman et al. [6]	Remote	Browser (Same-machine)	Yes	JS	Bytewise leakage, ASLR break, Rowhammer	2.75 h
Lindemann et al. [26]	Co-located	Cross-VM (Cloud)	Yes	Native binary	Fingerprinting	1.8 h
Kim et al. [21]	Co-located	Cross-VM (Cloud)	Yes	Native binary	KASLR break	12 min
Our work	Remote	**Inet/Local-NW**	**No**	**None**	Bytewise leakage, KASLR break, Fingerprinting	1.5 B/h (Local-NW) / 4 min / 166.51 s

Location: Attacker's location Local: local code execution
Type: Type of local code execution Perf: Reported Attack Performance
JS: JavaScript

Table 8.1.: Comparison of state-of-the-art memory deduplication attacks.

role is enabled. In addition, a Windows 10 process has the possibility to disable page combining [35].

We observed this effect by checking all terminal server options in Windows 2016 (Version 1607, Build 14393.693) and Windows Server 2019 (Version 1809, Build 17763.737). We also empirically validated that for Windows 10 Professional 20H2 19042.746 and Windows 10 Home 19041.746 page combing was active by default. On Microsoft's Azure Cloud [36] it is also possible to acquire such Windows Server VMs with this configuration. We created a Windows 2019 Server (Version 1809, Build 17763.1697) and can also confirm that page combining is enabled after setting the full terminal server role. On Windows, it is also possible to force page combining using the `RtlAdjustPrivilege` and `NtSetSystemInformation` functions.

Linux Kernel Same-Page Merging Kernel-Same-Page Merging (KSM) is the counterpart of page combining on Linux [3, 42]. KSM is enabled and mainly used for Kernel Virtual Machine (KVM) virtualized machines, for instance, on Red Hat Linux [42]. On Ubuntu 20.04, we observed that when `qemu-system-common` with KVM support is installed on a host machine, `KSM_ENABLED` is set to `AUTO` in `/etc/default/qemu-kvm`, enabling KSM per default for non-virtualized instances. We also set up an Ubuntu 20.04 server image and observed the same behavior after installing QEMU. Like on Windows, a kernel thread scans over the memory and merges pages with identical content to a single page, which is then marked as copy-on-write [42].

On Linux, only pages are merged that are marked as mergeable, *i.e.*, using the `madvise` syscall and setting the `MADV_MERGEABLE` flag [3]. This is the default for pages of KVM virtual machines. The user can configure how many pages should be scanned per invocation (`pages_to_scan`). The default value on a Ubuntu 20.04 is 100 `pages_to_scan` in a time interval of 200 ms. Therefore, in the optimal case, up to 500 4 kB pages can be deduplicated per second. Figure 8.2 illustrates the required time for a single page being deduplicated, with a different number of pages_to_scan set, and the default value of 200 ms for sleep_millisec. We evaluate the deduplication time for a single page depending on the scanned pages on a remote server equipped with an Intel Xeon E3-1240 running Ubuntu 20.04. However, it is recommended to increase the number of `pages_to_scan` to increase the deduplication performance [49]. The tool KSMtuned sets the time interval per default to 10 ms and increases `pages_to_scan` to 1250 [42].

This would lead to a maximum 512 MB being deduplicated per second. We asked a **cloud provider**, which hosts multiple hundred thousand websites, for the KSM config used in **production**. The cloud provider uses a configuration of `sleep_millisecs=30,pages_to_scan=500`, leading to at maximum 65.84 MB (16500 pages) being deduplicated. The average time after a single page is deduplicated with that configuration is 34.57 s ($n = 10, \sigma = 6.3\%$).

8.4.2. Service/Web API.

We assume that the victim machine provides network-accessible services, e.g., a REST API, enabling users to store and modify data blobs. There are no restrictions in the way these data blobs are controlled, *i.e.*, the user could either upload and replace files or send strings to the server, as long as the memory location of the data blob does not change.

8.4.3. Remote Timer.

To get the best possible low-latency timing information, we use the hardware timestamps from the network interface card. We measure the timing difference between the last packet sent and the first response byte received from the server (tcp_flags=PUSH,ACK).

The victim side (which the attacker cannot control) runs under default configuration. However, on the attacker side (that is under full control of the attacker), we disable the following optimizations in the Linux network parsing `sudo ethtool -K enp3s0 tso off gso off gro off`. These options disable offloading of TCP packets to the network interface card. Offloading might influence the timestamps on the attacker (receiver) side. We observed for some network interface cards that due to receiver side packet coalescing, the TCP receive timestamp of the first received packet might be overwritten. To ensure that no coalescing happens, we developed a kernel module which disables packet coalescing on the receiver side, for network cards which have this problem.

Network Timestamps. We found that one of the bottlenecks of remote attacks is the limited number of HTTP requests which can be sent using a simple HTTP requests library like `pyrequests`. Therefore, we use asynchronous IO mechanisms to increase the number of requests per

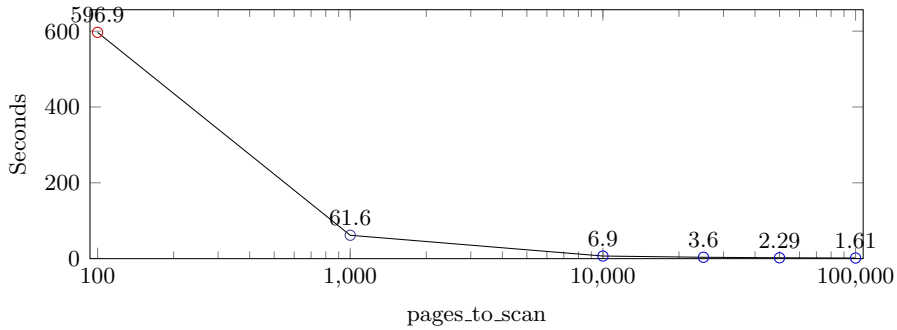


Figure 8.2.: The deduplication time of a single 4 kB-page strongly depends on the number of pages_to_scan (sleep_millisecs=200).

second. Furthermore, we observed that remotepagededup:Wireshark's TCP-field `tcp.time_delta` reflects the timing difference between copy-on-write pages and non copy-on-write pages best. This field calculates the timing difference between two captured packets. Compared to the network timestamp read from the NIC, we require only 20 requests instead of 40 to distinguish 16 overwritten copy-on-write pages over 14 hops in the internet to build a histogram.

8.4.4. Attack Setup.

For all our case studies, we use the following setup for our local and remote scenario.

Local Scenario. The local victim machine uses an i7-6700K processor with Ubuntu 20.04 (kernel 5.4.0) and runs QEMU 4.2.1 with KVM support enabled and virtualization extensions enabled. Co-located in the same local area network, we have our attacker machine, which also uses an i7-6700K processor and Ubuntu 20.04 (kernel 5.4.0). For the Linux setup, we host a virtual machine with KVM running Ubuntu Server 20.04 LTS (kernel 5.4.0-53-generic).

Remote Scenario. In addition, we used a remote Linux server by remotepagededup:Equinix [44], running on an Intel Xeon E3-1240 CPU. We installed the same virtual machine on the Linux server. For our Linux machine, which was located in Amsterdam, we observed `14` network hops.

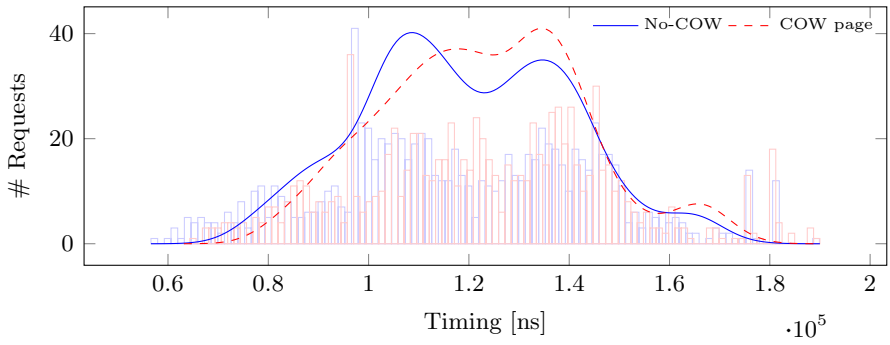


Figure 8.3.: Timing distribution of a single deduplicated page of a virtual machine in a local area network scenario on Linux KVM (n = 1000).

We created a virtual machine on Microsoft Azure of size Standard D4s v3 [36] and set up a Windows Server 2019 (Version 1809, Build 17763.1697) with page combining enabled. We observed `28` hops, using the nmap traceroute command, between our network and the Windows 2019 server virtual machine, which was located in Amsterdam. We use the same attacker machine from our local setup to perform the internet attacks.

Settings. To estimate the highest possible capacity of our remote covert channel, we try to reduce the noise as far as possible, *i.e.*, by fixing the CPU frequency of the KVM virtual machine. To enable full scans on a moderate CPU utilization, we set the value of `/sys/kernel/mm/ksm/pages_to_scan` to 100 000. The `/sys/kernel/mm/ksm/sleep_milliseconds` remains at the default value of 200 ms. Furthermore, we set the CPU performance governor to performance using the `cpupower` tool to avoid noise from wake-up delays. We later on use the default configuration of Ubuntu, Windows, and the cloud provider to calculate the leakage rates for the cases studies.

Evaluation. For a single page, we measure a local timing difference directly in the virtual machine (KVM) and observe that the average local timing difference between a regular write memory access and a memory access causing a copy-on-write page fault is 7209.3 ns (n = 100, σ_{COW} = 26.23%, σ_{NOCOW} = 29%) using a local timer. We evaluate the timing difference in our local area network and on the internet using a simple HTTP server with a key-value store. Figure 8.3 illustrates the timing difference for a single page accessed with a copy-on-write page fault and

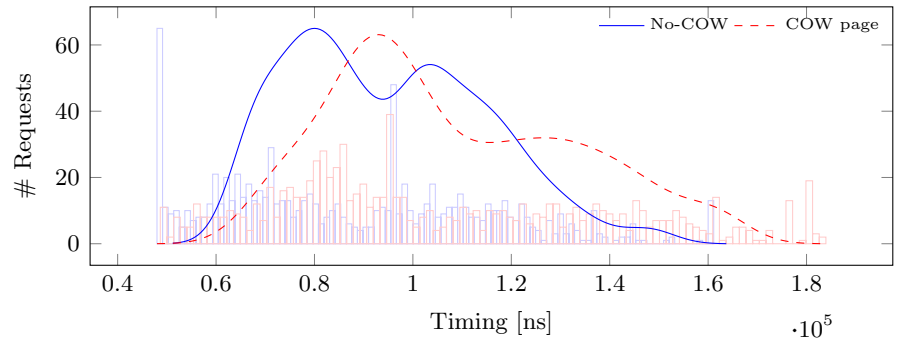


Figure 8.4.: Timing distribution of a single deduplicated page of a virtual machine in the internet(14 hops) (n = 1000).

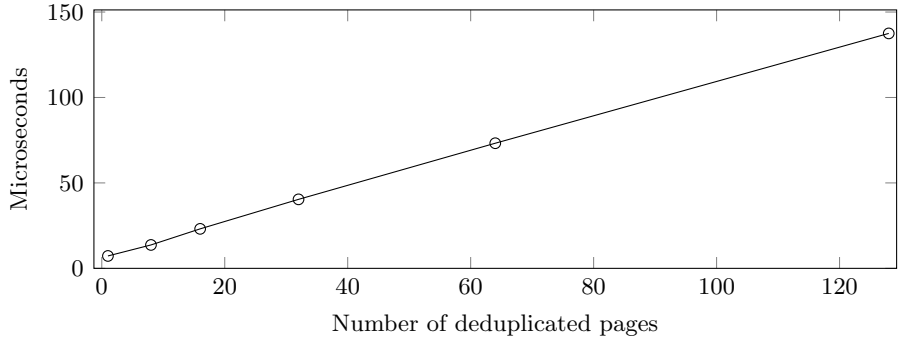


Figure 8.5.: Timing difference between amplified pages.

a normal write access in a local area network. In the local area network, we observe a mean timing difference of 4353.91 ns (n = 1000). Figure 8.4 shows the timing difference for a single page accessed with a copy-on-write page fault and a normal write access from our Linux server on the internet (14 hops). While those two distributions overlap, they can be clearly distinguished in the mean respectively median values if enough samples are taken. In addition, the timing difference can be amplified by overwriting multiple copy-on-write pages in a single request.

Amplification. In the following paragraph we solve *C1: (Remotely amplify latencies for non-repeatable events.)*. A copy-on-write page fault can be amplified if multiple pages belonging to the same semantic entity (i.e., pages of an image file) get duplicated at the same time [15]. Therefore, we can amplify the timing difference between multiple deduplicated pages by

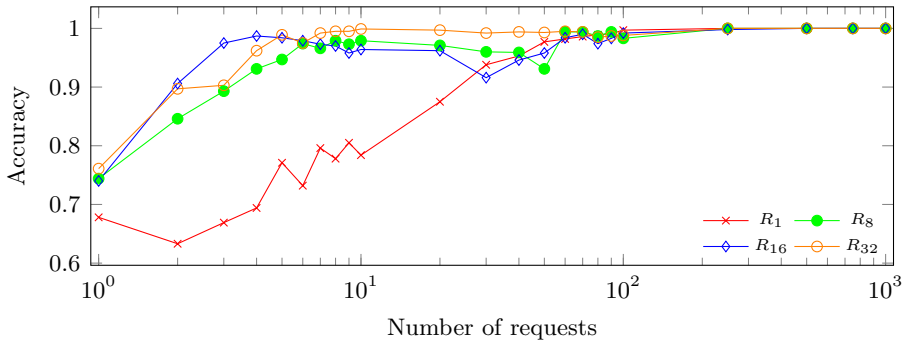


Figure 8.6.: Success rate of the classifier using the box test with a different number of deduplicated pages (R_x).

sending a single request, writing to those which trigger the copy-on-write, and responding back. To evaluate the timing differences of multiple copy-on-write page faults, we evaluate a different set of pages, which triggers the page fault. We define a test set in our local KVM machine with a test set of 1, 8, 16, 32, 64, and 128 deduplicated pages and measure the average timing difference between triggering a copy-on-write page fault and a regular write access. We repeat the experiment 100 times and calculate the difference between the average times, which is plotted in Figure 8.5. We can see that there is a linear increase in terms of the timing difference with the increase of the number of deduplicated pages. For instance, with 8 pages, we get an average timing difference of 13 610.82 ns and with 16 pages, it is on average 22 946.14 ns.

Next, we evaluate the effect of amplification in our local area network setup with KVM. We use the term amplification factor to indicate the number of additional pages used to amplify the signal. We sample 1000 times and fit a CDF(cumulative distribution function) for each of the amplification factors $(1, 8, 16, 32)$ and randomly sample from the CDF. To discover the number of requests required to achieve an accuracy higher than 95% percent, we perform the box test by Crosby et al. [10]. Figure 8.6 illustrates the number of network requests required to achieve a certain success rate for a different number of pages deduplicated by the server. In this idealized setup, we observe that 10 requests with an amplification factor of 8 are enough to achieve a 95% confidence of distinguishing write accesses on a deduplicated page (incurring a page fault) and a non-deduplicated page in a local area network if amplification is used. The number of requests

required is in a similar range for the local area network observed by Van Goethem et al. [52].

C1

Remotely amplify latencies for non-repeatable events.

We showed the applicability of memory-deduplication attacks within the same security domain. We can amplify the timing differences for the copy-on-write page faults arbitrarily by leveraging the deduplication of multiple pages belonging to the same semantic entity. If the attacker is in control of overwriting the data, multiple copy-on-write page faults increase the latency.

R/W bit stays cleared. On Windows and Linux with page combining respectively kernel-same-page merging, we observe that when a page is deduplicated, and a write access occurs to one of the corresponding virtual pages, the remaining mappings of the same physical page remain marked as **copy-on-write**. We empirically validate this in an experiment, where we first map two pages A and B with identical content and wait for deduplication. We then write to page A and thus trigger a copy-on-write page fault. Subsequently, we analyze the R/W bit of the page-table entries for both pages and see that the R/W bit remained cleared for page B. This observation is especially useful when the attacker can align data, as was shown by Bosman et al. [6]. In Section 8.6.3, we exploit this behavior to amplify a single copy-on-write request via Memcached.

8.5. Remote Covert Channel

For our evaluation on both Windows and Linux, we first create a covert channel using our remote memory-deduplication channel. For this purpose, we implement a small HTTP/1.1 server in C++ to maximize the performance. We evaluate this attack on a local-area network with a hardware switch between the attacker and the victim.

Capacity. We build a covert channel to measure the performance of our remote memory-deduplication attack in a local area network scenario. The victim system for our transmission hosts a website that allows storing and

updating files. The website keeps the files in in-memory storage, *i.e.*, in RAM.

The sender and receiver upload an identical large file to the website hosted on the victim system. Both use a 4 kB page in this large file to encode a '1'-bit. To transmit a '1'-bit, the sender puts the same page into RAM by updating the file via the website. The page is deduplicated with the page in the receiver's file. Conversely, to transmit a '0'-bit, the sender modifies the page in its file such that it is not deduplicated. The receiver sends a network request that either triggers a copy-on-write page fault or not. With measured round-trip time, the receiver distinguishes between a '1' and a '0'. The transmission can be parallelized in our setup by storing multiple bits at once and evaluating them in parallel.

Local Area Network. We transmit a random secret that is 8 bytes long, and repeat the experiment 100 times. On each repetition, we re-randomize a new 8 B secret. We observed that the Python capturing library has problems correctly parsing the packets when performing too many requests asynchronously on our webserver, we always leak 2 bytes (16 bit) in parallel for stable results. Between the send and receive process, a delay of 3 s was used to wait for deduplication.

In our Linux setup using amplification of 16, we achieve an overall performance of 302.16 B/h ($n = 100, \sigma = 5.81\%$), with an error rate of 0.6 %.

Internet. We run the same experiment as for the local area network. On Linux, we used 20 requests per bit and used an amplification factor of 16 pages. On Windows, we used 20 requests per bit and an amplification factor of 32 pages.

On the Linux server, we achieve an overall performance of 34.41 B/h ($n = 100, \sigma = 5.87\%$) with an error rate of 0.83 %. On the Windows server, we use constant triggering of memory deduplication and a delay of 50 ms and achieve an overall performance of 26.64 B/h ($n = 100, \sigma = 0.69\%$) with an error rate of 0.18 %. We use this number to calculate the timing for the actual wait time on Windows of 15 minutes until the deduplication succeeded, which is 0.4 B/h.

Using the same methodology as state-of-the-art work [4, 6], we simulate the covert's channel performance for the default configuration of 100 pages_to_scan on Linux. As it takes 596.9 s on the remotepagededup:Equinix

server to perform a full scan, the covert channel's performance shrinks down to 0.59 B/h. For the provided numbers of the cloud provider, the covert channel would achieve 20.62 B/h. These numbers are in a higher range as previous work, with the additional overhead of TCP, compared to the UDP sockets used in a similar attack scenario [48]. The other remote timing attacks did not provide concrete numbers on their covert channel [1, 2, 20, 47, 52, 59].

8.6. Case Studies

In this section, we evaluate three case studies and demonstrate what types of attacks are possible with remote memory-deduplication. First, we demonstrate that we can exploit remote memory-deduplication in Memcached to fingerprint the system, including the operating system. We successfully detect the correct libc library over the internet in 166.51 s. Second, we demonstrate a fully remote KASLR break by exploiting remote memory-deduplication within 4 minutes. Third and finally, we demonstrate how to leak database records byte-by-byte from InnoDB used in MySQL and MariaDB. In the following subsections we show how to solve C2, and C3.

8.6.1. Memcached

Memcached is a fast in-memory database offering a key-value store for applications [34]. The memory is managed using a slab allocator. A slab consists of a single or multiple memory pages, which are contiguous in physical memory. Memcached always allocates a 1 MB region and splits it into smaller chunks of equal size [34]. Chunks or objects with a similar object size get assigned to a certain slab class. For instance, if a slab class is 64 B, the 1 MB page is split into 16 384 chunks. Newly inserted data is assigned to the smallest slab class that the data fits in [34]. This means a certain slab class contains objects of a certain size and assigns the objects to a chunk. A key-value pair is managed by the `item` structure, a linked list that contains the size of the key, the value of the object, and some more metadata [34]. Each slab class has a free list, which is a linked list [34]. If an item gets freed, its former location is moved to the head of the free list.

Memory Management. The key-value pairs are stored contiguously in memory, which is ideal for triggering memory deduplication. We analyzed and profiled the source code of Memcached to check which functions are used and how the memory allocation works internally. In contrast to our expectations, Memcached does not perform an in-place replacement of the value to update. Even with the same key used in `memcached_set` and `memcached_replace` operations, a new location is assigned to the updated value. This new location is either an available free slab item from the head of the free list (do_slabs_alloc) or a new slab item.

After all input data from the new item is read, the old item is unlinked, and the new location linked for the item. The old location is freed and inserted to the head of the slab's free list (code path is from complete_nread → do_item_link). If a fixed memory size is reached, a least-recently-used (LRU) eviction policy is applied on a slab-base [11] This means that "old" items are replaced by more frequent items in a certain slab class.

Attack.

Our basic remote memory-deduplication attack on Memcached works as follows on Linux and Windows: First, the attacker places the targeted pages into the key-value store with a specific identifier. Then, the attacker waits some amount of time (delay) such that the pages are deduplicated. The deduplicated content can be for instance a static unique binary page of a specific version of the C standard library or other static binary pages in the system. After the delay, the attacker creates a new dummy item with the same key, which puts the deduplicated target page on the free list of Memcached. Then, the attacker updates the same item, which causes a copy-on-write page fault on the deduplicated page which is now overwritten.

Alignment. In general, it is not guaranteed that allocating memory with `malloc` internally uses `mmap` for a specific allocation size (this may depend on the libc variant, *i.e.*, glibc MMAP_THRESHOLD is 128 kB,system configuration, and operating system versions of the victim system). Thus, it is also not guaranteed that the allocated 1 MB region is aligned to any specific offset. Using `mmap` would ensure a page alignment, meaning the page offset would always be 0. However, in our experiments, we observed

that malloc always used `mmap` internally for the 1 MB allocations on a default configured Ubuntu Linux installation.

It is also not guaranteed that the attacker inserts the first item in the slab class, which also causes an unknown alignment as also other chunks might be inserted on the 1 MB page. To overcome this limitation we propose a method to generate chunks of all different sizes possible for a slab class. We calculate all possible offsets the chunk could have on the page for a certain slab class. These possible offsets can be computed for each possible chunk per page i as `offset`:

$$(malloc\_offset + i \cdot chunk\_size + item\_header\_size + key\_size) \mod 4096$$

where the `key_size` is attacker-controlled, the `chunk_size` depends on the slab class, `malloc_offset` = 16 and the size of the item header is defined as `item_header_size` = 56. Hence, to overcome the alignment issue, we use the same page with the different offsets to cover all possible alignments, which is guaranteed to include the correct alignment required for deduplication.

LRU. In a real-world application, Memcached can be expected to be heavily used by other users as well. However, we still need to keep the data inside the data store. We achieve this by frequently accessing the data using GET requests on the service to avoid being evicted by the LRU eviction strategy. Note that this does not trigger copy-on-write page faults as we only read the data but do not modify it. We discuss the eviction in more details in an attacker scenario in Section D.

Evaluation. We evaluate our attack on Memcached 1.6.8 and connect to the Memcached service using UNIX sockets. We evaluate this scenario using a PHP site (version 7.4.3), which is hosted on an Nginx server (version 1.18). Our evaluation uses the local area network setup, and we also run on a Linux server on the internet 14 hops away. Our victim server and attacker setup are the same as described in Section 8.5.

We alternate between pages that do not trigger a copy-on-write page fault and pages that trigger a copy-on-write page fault due to deduplication. In addition, we alternate the order to avoid a potential bias, which could

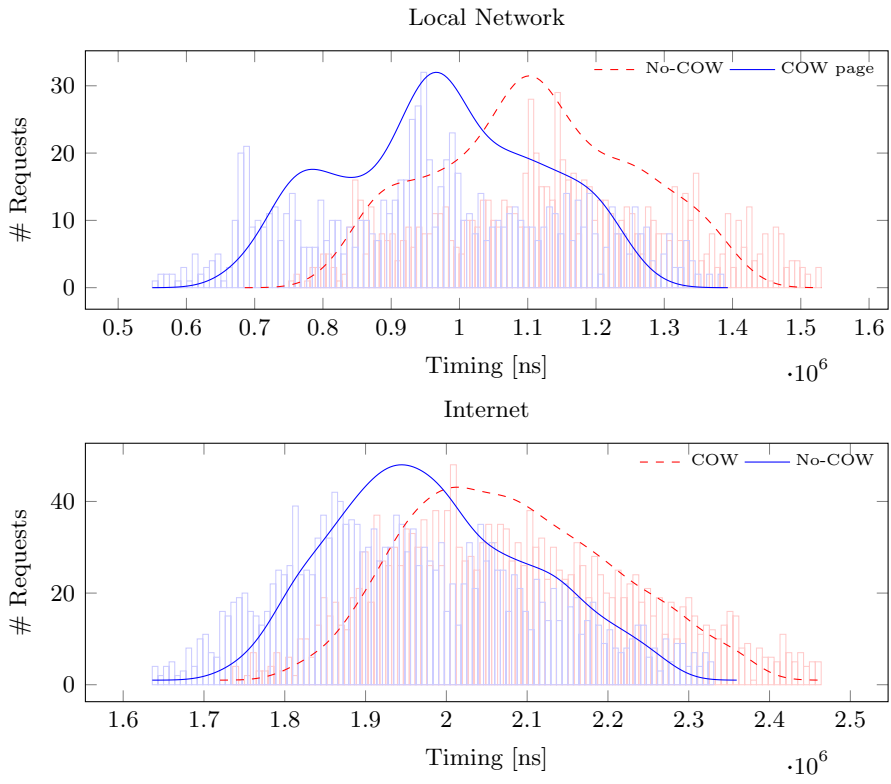


Figure 8.7.: Histogram of the network requests in a local area network and in the internet setup using 16 pages to amplify the results in Memcached.

be introduced by a fixed request order. In total, we perform 1000 HTTP requests. Figure 8.7 shows the timing differences we observe in this setup. We can see that it is easy to distinguish between deduplicated pages and non-deduplicated pages.

Libc Fingerprinting. Operating system and library fingerprinting is a good starting point for penetration testing to determine potential vulnerabilities on the identified operating system or the running applications. Those results observed from Memcached can be used to perform fingerprinting of operating systems by looking at fixed memory pages as was proposed by Owens et al. [39]. We use the same setup as before and try to fingerprint the exact standard C lib (libc) version. In our experiment, we probe 3 different versions of the libc. We perform 20 subrequests for each version we probe on Memcached. Our information disclosure

attack detects the correct version in one sample within 44.28 seconds (n = 100,σ = 0.19%) and an accuracy of 90%, depending on which library is mapped on the victim. Memcached can be used as an additional possibility to force deduplication and evaluate the response time, which we show in Section 8.6.3. The timing differences for the correct library guesses in PHP via Memcached can be seen in Figure 8.14 (Section A).

Internet. We run the same experiment with the same setup (Nginx, PHP, Memcached) over the internet targeting the remotepagededup:Equinix Linux VM `14` hops away. We detect the correct version in one sample within 166.51 seconds (n = 100,σ = 9.67%) and an accuracy of 90%. Using the default settings for KSM, the attack would take 3.36 h. With the settings provided by the cloud provider, the attack would take 0.22 h.

C2

Trigger and observe copy-on-write page faults in a victim domain that shares no memory with any attacker domain.

With our attack on PHP-Memcached hosted on an Nginx server, we demonstrated that it is possible to trigger copy-on-write page faults within the same security domain without relying on shared memory with the attacker domain. This can be used to perform operating system fingerprinting like was shown via Memcached over the internet.

8.6.2. Breaking KASLR Remotely

By randomizing the location of kernel code, data, and drivers at every boot, KASLR makes the exploitation of memory corruption bugs in the kernel much harder (Section 8.2.4) as an adversary needs to guess the addresses for the attack correctly. In the past, different side-channel attacks allowed to reduce the entropy of the randomization or to break it entirely [8, 9, 12–14, 16, 17, 19, 24, 27, 28].

While Klein and Pinkas [22] used an information leak in IP headers to break KASLR, so far, no remote side-channel attack has been demonstrated against KASLR. In this section, we exploit memory deduplication to break KASLR of a virtual machine remotely. Concurrent work by Kim et al.

[21] demonstrated a KASLR break on co-located machines on VMWare ESXi break via memory deduplication within 12 minutes.

We describe the necessary building blocks and threat model to mount the attack targeting one virtual machine over the network.

Attack Scenario & Attacker Model

We assume that the version of the operating system running on the victim machine is known to the attacker. That memory deduplication is active and enabled by the operating system (or hypervisor). This information can be obtained by an information leak or a fingerprinting attack, similar to the one described on Memcached in Section 8.6.1.

Attack & Building Blocks

Finding the content of memory pages that are identical to the ones used by the victim operating system forms the basis of our KASLR break. If the content of the attacker-controlled page is identical, the hypervisor deduplicates it. Thus, a subsequent write to the page yields a higher execution time forming the side-channel we exploit throughout this paper. While a page with the same content as a kernel page allows fingerprinting the operating system, data and pointers stored on the page either change during runtime or are randomized on every boot and are, thus, less predictable.

However, on Linux, the text segment is mapped between the 1 GB region of `0xffff ffff 8000 0000` and `0xffff ffff c000 0000`. As the kernel is 2 MB aligned, there are only 512 possible offsets in this region where the kernel can be placed. If we find kernel pages that only contain kernel addresses and static values, *i.e.*, data that is not modified during runtime, we can generate 512 different versions of the page. Each version corresponds to one possible offset and contains the kernel addresses if the kernel would be mapped to said offset.

A page on the victim machine is now filled with a possible content candidate. The remote attacker uses the API provided by the victim machine to set the content of a page. Depending on the configuration of the hypervisor on the target machine, the adversary waits until pages should be deduplicated. Now the adversary writes to the same page using the

API. The adversary measures the time it takes to write to the page, *i.e.*, the time it takes for the network request to be handled. If the content set by the adversary matches the targeted kernel page, the hypervisor has deduplicated the pages, and to handle the write. They have to be duplicated again. Thus, if the content matched, the adversary observes a higher timing. For all of the 512 different possibilities, the adversary performs these measurements, yielding a single candidate that corresponds to the currently used randomization offset. To deal with measurement noise, the adversary has to repeat these measurements.

In addition, it is possible to amplify the side-channel leakage. Instead of a single kernel page, multiple different kernel pages can be generated based on the assumed kernel offset and set at the same time. Thus, instead of a single deduplication, the adversary observes multiple ones within a single measurement.

Finding Suitable Kernel Pages. To send the content of possible kernel pages, the adversary first needs to scan possible page candidates. This can be done upfront in an offline phase and used for kernels of the same version, thus, one assumption is that the adversary knows the version used by the victim.

To find possible page candidates, we walk the page table levels of the Linux kernel and inspect the content of each mapped 4 kB page. We know in which region the text segment can be mapped and check each possible position of a pointer, *i.e.*, each 64 bit, if it lies in this region. If so, we dump the contents of the page as well as all the offsets representing a pointer within the page. We also extend this approach to kernel pages belonging to kernel modules, as they are also randomized in a certain memory region and could be used to break the randomization of the modules. On a machine running Linux 5.4.92, we find 15 737 pages where 4070 contain values matching pointers within these memory regions.

In a second step, we filter the dumped pages for possible candidates that we can use for the attack. We try to find corresponding symbol names to the detected addresses by matching them to `/proc/kallsyms`, yielding 15 pages that only contain resolvable kernel text addresses. 3973 pages contained module addresses, 39 resolvable and unresolvable addresses, and 43 no symbols at all. These pages now need to be checked if their content is static and, thus, does not change over time. This can be achieved by dumping the content periodically and checking it for modifications.

Further, we want the pages to not contain any data initialized during boot time and, thus, we need to check if the content of those pages changes (excluding the kernel addresses) while rebooting the system multiple times. In order to rule out hardware-specific data, this should be done on different physical machines.

Remote Attack

For our remote KASLR break, we implemented the victim server in two ways. First, as a RESTful API listening for HTTP/1 requests implemented in C++ using the pistache framework [46]. For simplicity reasons, the API allows the adversary to set and modify the content of pages directly. However, as we have shown in Section 8.6.1, the data could be stored in an in-memory database as well. Second, we elevate the service for HTTP/2 to support multiplexing, allowing us to mount timeless timing attacks described by Van Goethem et al. [52]. In both scenarios, an Nginx [45] web server running on the target machine forwards the request to the victim service. The attacker sends the crafted pages for the offset to test to the victim using the API. After 2 s, *i.e.*, the time the page would be deduplicated on our system with a high chance, the attacker sends a network request modifying and, thus, causing the probable duplication, and measures its response time.

HTTP/1. In the first scenario, we use HTTP/1 to communicate with the network service and measure the response time of the network requests. Figure 8.9 illustrates the distribution of response times for the correct offset and an incorrect offset. Figure 8.8 shows the mean response time of a network request for a specific offset in a remote-attack scenario. After sending 100 requests, we can clearly see the increased response time for the currently used randomized kernel offset.

In the local setting, we were able to recover the correct randomization offset with a success rate of 100 % and an average runtime of 21.3 s (n = 100). In the remote setting, we were able to recover the correct randomization offset with a success rate of 73 % and an average runtime of 5 min 57.9 s (n = 100). With the default configuration of the cloud provider (cf. Section 8.4.1) this would yield an average simulated runtime of 34 min 28.69 s. With the default Linux settings, it would take 9 hours and 2 minutes.

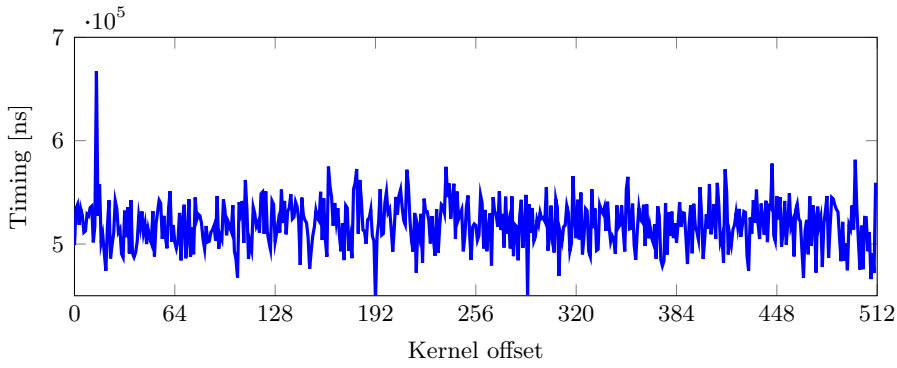


Figure 8.8.: Execution time to a page containing the content adjusted to one of the possible kernel offsets. The high peak at offset 14 yielded the same content as the kernel page of the VM and, thus, has been deduplicated by the hypervisor.

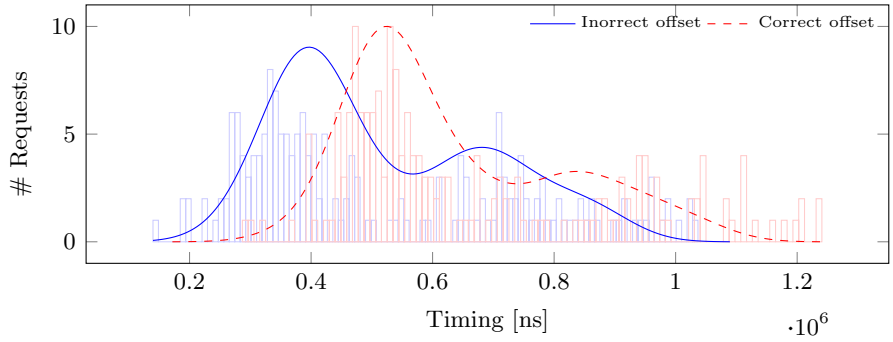


Figure 8.9.: Histogram of the measured access times for an incorrect and the correct offset for the KASLR break. A correct guess can be clearly distinguished from an incorrect guess.

HTTP/2 Multiplexing. To improve on the measurement noise introduced by the connection between the victim and the adversary and the necessity of accurate time stamps, we utilize Timeless Timing attacks [52] to overcome this issue. HTTP/2 allows to pack multiple requests within a single packet and, thus, the requests reach the server at the same time. However, the response of the request that reaches the sender faster has likely been processed quicker.

In contrast to the sequential HTTP/1 attack, we pick pairs of kernel offsets that we send to the server using multiplexed HTTP/2 requests. For every attempt, we send each pair to the server and record for which

request we receive the response first. Note that we do not need to rely on measured access times but just on the response order of the requests. For pairs of both incorrect kernel offsets, we should observe a uniform distribution between the offsets. However, if one of the offsets is the correct one, we should observe an unequal distribution. To optimize the approach, we reduce the number of candidates and filter out pairs with a uniform distribution early. With each filter step, we re-combine the candidates to new pairs.

To amplify the signal, we crafted 7 kernel pages for each possible kernel offset. In the local-network setting, we achieved a success rate of 88.89 % with an average runtime of 1 minute and 38 seconds (n = 100). The raw timing differences observed in the HTTP/1 setting enable a faster attack than HTTP/2. We were able to successfully find the correct offset in the remote setting with a success rate of 92 % with an average runtime of 3 minutes and 15 seconds (n = 100). With a prolonged waiting time using the default configurations of the cloud provider of how many pages are scanned by the operating system per minute, this would yield an average simulated attack time of 18 minutes and 25 seconds. With the default Linux settings, it would take 4 hours and 48 minutes.

8.6.3. InnoDB Record Data Leakage

InnoDB is a storage engine used by default in the database management systems MySQL and MariaDB. The storage engine efficiently buffers record data and index caches in the memory and is used instead of using the operating system's page cache directly. InnoDB has the advantage of providing faster access to frequently used data.

Database systems use indices to allow quick access to records, *i.e.*, normally, an index is placed automatically on columns marked as primary key. InnoDB implements indices using a B+ tree, which allows fast record lookups. The nodes of the tree are represented by index pages, which are the basic storage unit of InnoDB and have a size of 16 kB by default. The leaf index pages contain the actual user data. The non-leaf ones link to other leaf or non-leaf pages. Index pages on the same tree level are linked together to allow scanning operations. User records in an index page are logically linked in ascending order by their key but may be placed anywhere in the page's physical memory.

High-Level Overview of the Attack. To achieve byte-by-byte leakage, the attacker needs to control the content and size of data that is stored before the target data to bytewise shift the target data onto the attacker-controlled page. Bosman et al. [6] showed that byte-by-byte leakage is possible.

InnoDB performs a data reorganization of data if an insert or update query fails as an optimization. This optimization enables byte-by-byte leakage if the attacker controls most of the InnoDB record. Using this primitive to perform memory massaging, an attacker can shift the secret.

Assumptions. We assume that Memcached can be used in addition as a leakage primitive to leak the secret data co-located to the attacker-controlled data bytewise. As we will later analyze, the Linux page cache caches Note that this can be any primitive used for triggering deduplication and copy-on-write page faults, *i.e.*, remotepagededup:nginx, as was shown by Bosman et al. [6]. We assume a database application with a user table which is defined in Figure 8.15 and that the InnoDB index page has a certain layout, which is explained in more detail in Section 8.6.3. We assume that the attacker can perform multiple tries in parallel until such a layout is given. If the layout is given, the attacker can verify whether the requirements are fulfilled.

Attack steps. Figure 8.10 illustrates the five steps of the InnoDB reorganization attack. In the first round, the attacker triggers the reorganization and shifts the first byte of the secret value (**"SECRET"**) onto the controlled 4 kB-page. Next, the attacker stores multiple guesses into Memcached. The attacker waits until the deduplication happened. After the deduplication happened, the attacker updates the Memcached guess pages and measures the round-trip time of the network packets. The right guess should lead to a significantly higher timing than the other guesses with enough samples taken. Afterwards, the attacker repeats the procedure to shift the second byte into the attacker-controlled InnoDB page, updates the guesses in Memcached, including the first recovered byte, and leaks the second byte. This procedure can be repeated up to a certain leakage size. The limits are discussed in Section B.

Why an Additional Leakage Primitive is Required. InnoDB tries to circumvent the page cache of the Linux kernel by using the `O_DIRECT`

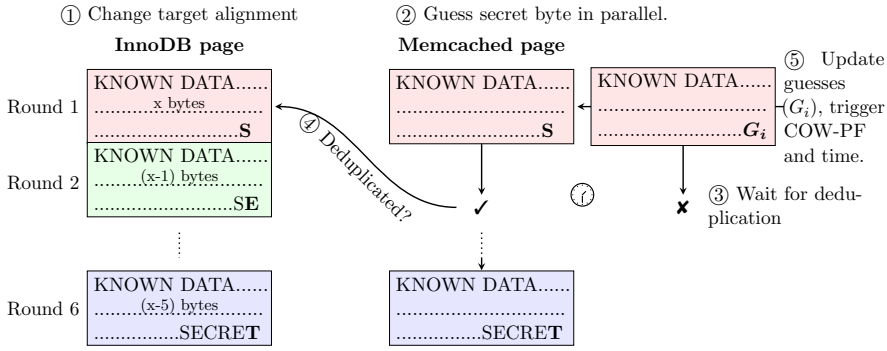


Figure 8.10.: High-level idea of the InnoDB Reorganization attack.

flag in `mmap` [37]. However, the data is still in the page cache and gets deduplicated. The page-cached data cannot be overwritten directly via InnoDB. Therefore, we cannot use a second InnoDB record to trigger a copy-on-write page fault since the data would also get deduplicated. We found no convenient and reliable way to get external blobs consistently in the memory and replace them to trigger copy-on-write page faults in InnoDB. For external blobs, we have a similar race as in Memcached, since updates are not performed in-place. Instead, resource releasing is performed in a similar way compared to Memcached. Consequently, for our attack, we use a memory-resident second channel (Memcached) to trigger the copy-on-write page fault. However, this could, in general, be any web application/resource providing such a leakage primitive that is running on the same machine.

Determining InnoDB Attack Requirements.

We analyze the memory ordering of InnoDB by performing different SQL statements:

Insert. Upon inserting a new record, InnoDB first tries to place the record in its corresponding index page. If no such page exists, a new one is created.

In case of an existing index page with sufficient consecutive free space, e.g., unused space at the end of the page or a gap from previous deletes, the record is placed on this index page. Should this not be possible, e.g., the page is full, or the free space is too fragmented, either the current

index page is defragmented (*reorganized*), a new page is allocated, or a page *split* is performed. Inserting into a new index page is only possible if it does not break the existing relations in the index tree. Otherwise, a page split has to be performed. As the space of previously deleted records is reused, the physical order of records in an index page does not always reflect their logical order, e.g., a record with key 5 might be inserted in memory before a record with key 2.

Delete. When a record is deleted, it is added to the index pages free record list. Should the free space resulting from deletions reach a certain merge threshold, InnoDB tries to perform a `merge` operation to save space. A merge operation is possible if the utilization of the next or previous linked index page is low enough to combine it with the current page [29].

Update. Update queries in InnoDB update a record in-place, as long as the updated record fits in the same size as the old one (`new_record_size` ≤ `old_record_size`) [30]. Otherwise, the update operation is realized as a delete with a subsequent insert operation, inserting the updated record.

Reorganization. An insert or update query can fail even if enough space is available on the index page because the free space is fragmented. In such a case, InnoDB performs an optimization called `reorganization` [31]. During reorganization, the page is rebuilt by clearing its contents and inserting existing records in their logical order. Afterwards, the pending insert or update operation is completed using the freed space at the end of the page.

Reorganization Attack.

As shown by Bosman et al. [6], an attacker can use memory-deduplication attacks to leak data byte-by-byte if the attacker can change the memory layout on a byte granularity. We demonstrate that this approach can also be applied in a fully remote attack scenario.

We leak data byte-by-byte by exploiting the `reorganization` of database records in InnoDB index pages. The reorganization is triggered if data is updated or inserted, and reorganization keeps the data on the same index page. Figure 8.15 in Section C shows the user table and its fields in the

database, including an `id`, `username`, `password`, and an `image` field. We assume that the attacker can register an arbitrary number of users and modify their content.

Alignment Changing. To leak attacker-unknown record data, we need a large record r_{AT} to `shift` bytes from a target record r_T into an attacker-controlled region. To trigger the `reorganization`, we require an additional record r_{AX} in the user table. The reorganization orders the records in RAM in their logical order. With targeted size modifications of the attacker-controlled records r_{AT} and r_{AX}, we can trigger the reorganization and bytewise shift record data from r_T into an attacker-controlled 4 kB region. To leak the targeted byte, we use Memcached used in a simple HTTP web server as a second channel, same as in Section 8.6.1.

Amplification. To use amplification, we fill multiple pages on both Memcached and InnoBD with different fill bytes but with a constant leaked offset, as shown in Figure 8.11. As already mentioned, the copy-on-write bit stays set for the correct guess in Memcached. Therefore we can amplify by updating both Memcached and InnoDB fill bytes and waiting again for the deduplication. This procedure can be repeated up to a certain amplification factor. To trigger copy-on-write pagefaults, we send an HTTP request to the server, which overwrites the content of the Memcached pages. To reset to an index page layout, which allows leaking a different byte offset, it is required to trigger another reorganization by modifying the sizes of the records r_{AX} and r_{AT}. All requirements are explained in full detail in Section 8.6.3.

In Detail Analysis of Attack Requirements.

To perform the reorganization attack, the attacker and victim have to be placed on the **same** InnoDB index page. While this is a question on the workload of the system, we assume that an attacker can perform sufficient repetitions to generate a layout leading to data leakage.

A `reorganization` can be caused by updating the size of a record so that it does initially not fit in any available consecutive free space on the page but does fit after defragmentation. By choosing record sizes in the right way, it can be guaranteed that such reorganizations are always possible.

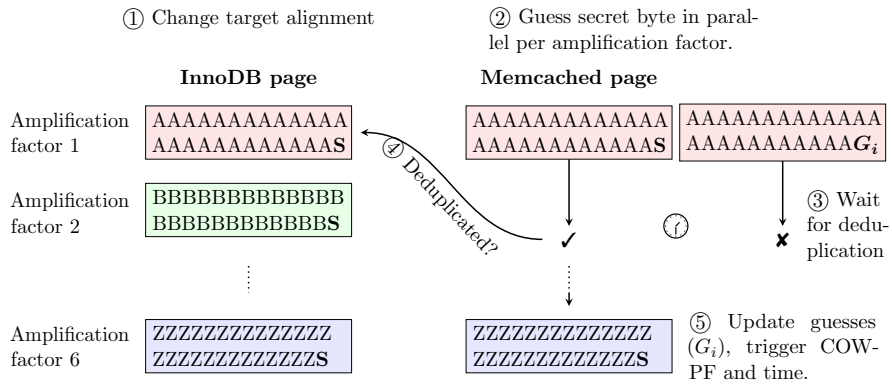


Figure 8.11.: Leakage of a secret byte (**S**) from an InnoDB record using Memcached with amplification.

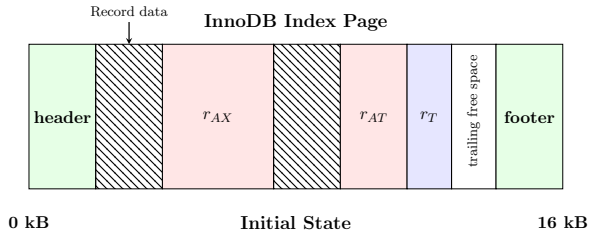
Initial Page Layout. Figure 8.12a describes the initial page layout required by InnoDB to leak record data. We exploit InnoDB's reorganization feature as a primitive for the attacker to align the secret on a byte granularity. At the beginning of an index page, there are a couple of headers and system records, summing up to 120 B [32]. Then the data of the user records follows. At the end of the page, there is a so-called page directory and further meta-data. The user records are also preceded by a dynamic-sized header, which depends on the table layout and contains information necessary for using and organizing the records. Figure 8.12a shows the assumed initial physical and logical layout for our InnoDB attack. The hatched areas represent unknown records. r_{AT} and r_{AX} are attacker-controlled records and r_T is the target record to leak.

Analysis of Required Sizes for Exploiting Reorganization. During the rebuilding of index pages, records are inserted consecutively in memory by their logical order, except for the record that triggered the reorganization, which is inserted last, regardless of its key. In total, it is possible to insert 16 252 B (`max_free_space`) of record data into an index page. The layout requires that the record r_{AT} is logically located before r_T. r_{AX} is required to be physically before the two records, somewhere in between the two hatched regions in Figure 8.12a. The record r_{AT} is used to change and control the target record's alignment r_T. The attacker wants to make r_{AT} as large as possible to change the target record's r_T alignment. In the default setting of InnoDB with a default index page size of 16 kB, the maximum size for a record is 8125 B [32]. Thus, to leak as much data as

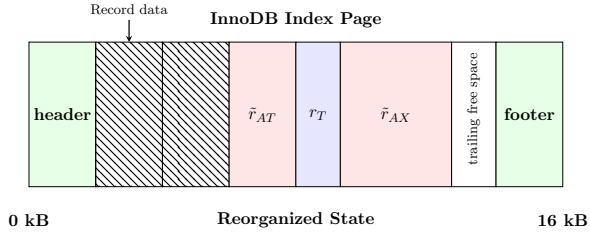
possible, we choose r_{AT} to be 8125. The validation of all requirements and the potential leakage rate is described in Section B. Next, we discuss the attack steps in more detail.

Preparing the Alignment and Triggering the Reorganization. To trigger a reorganization, the attacker increases the size of record r_{AX} using an update query. The reorganization only happens if the updated size still fits into the total free size of the index page. The new reorganization moves r_{AX} to the `trailing free space` within the index page, which is large enough to contain at least r_{AX} + 1.

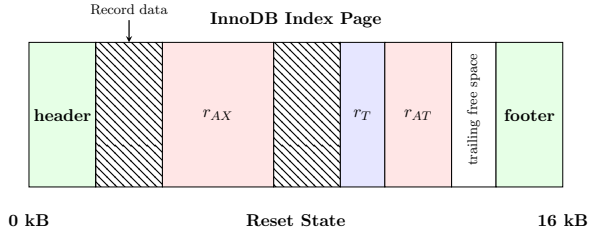
If the attacker wants to shift a byte of the target record by δ bytes such that the byte moves closer to r_{AT}, the attacker can update the size of r_{AT} and decrease it by δ and increase the size of r_{AX} by δ. The reorganization takes out the record r_{AX} and moves it to the newly created free location. It causes the record $\tilde{r}_{AX}$ to be moved after $\tilde{r}_{AT}$ and r_T. r_{AX} can only be modified up to the maximum record size. While the header of the user record has a dynamic size, we assume that the record does not change during the attack. If there is an additional record after r_T, the header stays the same. If there is no record after r_T, the record header points to the Supremum [33], which is at the beginning of the page. With each alignment change, the next offset field in the records header needs to be incremented by the byte offset to leak.



(a) InnoDB page layout, which is susceptible to a reorganization attack. A simple model of an index page consists of a fixed-size header, user records, the unused space at the page end, and a footer. In this scenario the attacker controls the records r_{AX} and r_{AT}. r_{AT} is used to control the alignment of target record to leak r_T. r_{AX} is used to trigger the reorganization within an InnoDB index page.



(b) If the reorganization was triggered $\tilde{r}_{AX}$ is moved to the beginning of the trailing free space.



(c) Reset and reorganized InnoDB record.

Figure 8.12.: InnoDB reorganization steps.

Leaking the Secret Byte. Since we cannot leak the records in MariaDB alone on Linux, a second way to trigger the memory deduplication is required. Furthermore, we apply amplification for our bytewise leakage, as discussed in Section 8.5. Figure 8.11 illustrates the amplified version of the InnoDB record attack. We use different pages with the same content but only a different last byte in the page, which is our probe byte in Memcached (in Figure 8.11 the secret byte is x). In MariaDB, we update our record r_{AT} with the content of the first page (`AAA....S`). Afterwards, we wait for a certain delay until the memory deduplication is triggered. If

the secret byte is correct, the page gets deduplicated. After the delay, we modify r_{AT} again. The page in Memcached still has the R/W bit cleared. If all amplification pages are deduplicated, we use the web application to write on each of our amplification pages and measure the response time.

Reset. After the reorganization, the alignment is changed, and we get a record layout, as illustrated in Figure 8.12b. Unfortunately, we cannot modify the base alignment since we do not know the size of the other records on the index page. However, we can either try to repeat the attack until we start at the beginning of a 4 kB page or leak the base alignment. To reset the state back to the initial one, we again exploit reorganization, changing to the previous sizes. The requirement to trigger another reorganization via r_{AT} is that the trailing free space is smaller than the reset size of r_{AT}. The reorganization causes that the updated record r_{AT} is now moved after r_T, leading to the memory layout illustrated in Figure 8.12c. However, this is no problem since we can force another reorganization, bringing back our reorganized state, as illustrated in Figure 8.12b. After each alignment change, we switch between the reset and reorganized state and never return to the initial state.

Evaluation. We implement and evaluate our attack on MariaDB version 10.5.8, using UNIX sockets and a simple HTTP server to connect to it and to Memcached 1.6.8. The database is setup as shown in Figure 8.12a. We choose a random secret of 4 B and repeat our data leakage experiment 20 times. We apply the amplification technique shown to leak a single byte via 8 pages. To be on the safe side, we send 40 requests for all 256 possibilities. Afterwards, we probe all 256 possibilities for the secret byte at once via Memcached. We look at the timing difference between the means of the received distribution of writing to copy-on-write and non-copy-on-write pages. Our attack automatically detects if a byte was accidentally classified as copy-on-write in case we do not get clear results for the following byte to leak. In this case, we can backtrack to the last byte that was correctly guessed. Therefore, our approach is self-correcting in case we accidentally received a wrong byte, and, thus, our approach is nearly complete error-free, despite the last byte where an error might occur.

On Linux, we observed that the time to wait for the deduplication on InnoDB is, in many cases, more than twice as big as in the previous cases.

To be on the safe side, we increased the wait time to 4 seconds. As the amplification needs to be triggered sequentially, this leads to a wait time of 32 seconds per guess round. This longer delay is required since the target page is constantly changed, and KSM does not immediately deduplicate pages which are often modified [42]. The runtime of the attack to leak four random bytes is on average 5644.20 seconds ($n = 100, \sigma = 0.54\%$). Thus, the attack leaks on average a single byte in 39.07 minutes or about 1.5 B/h from a virtual machine running on a remote server in the local area network. We simulate the attack's performanc using the default configuration to 0.018 B/h. With the provided configuration of the cloud provider, we got a simulated time of 0.07 B/h. Note that the large bottleneck of this attack is the amplification technique *i.e.*, for one iteration 32 s have to be waited.

Limitations. For the initial setup, cf. Figure 8.12a, the uncontrolled record data before r_{AT} can be modified in-place as long as the overall size is not changed. Every in-place update of other records does not influence the attack. However, a memory split, merge, or reorganization would interfere with the attack and potentially destroy the needed layout.

C3

Find remote request paths that do not only keep attacker-controlled data in memory but also provide the attacker with control over alignment and in-memory representation.

We demonstrated a scenario for InnoDB used in MariaDB and MySQL, which allows changing the alignment of database records remotely. By changing the sizes of two attacker-controlled records, an attacker can load bytewise parts of victim's data to an attacker-controlled 4 kB page. Amplification can be achieved by leveraging the fact that deduplication can be triggered multiple times by modifying the attacker-controlled record and adding certain amplification pages to Memcached (like shown in Figure 8.11.)

8.7. Mitigations and Further Attack Targets.

8.7.1. Mitigations

Our attack showed that memory deduplication is still a threat and even exploitable over the network. Even isolation into security domains like performed on Windows is not enough to mitigate information disclosure via memory deduplication.

Deactivation. While the simplest solution would be to altogether disable memory deduplication on Windows and Linux (Ubuntu), it is probably the most costly in terms of performance overhead. Especially on Windows server, where multiple users would use the same application, this could lead to immense memory overhead. Windows allows disabling of memory deduplication per process [35].

Only Deduplicate Zero Pages. Another mitigation by Bosman et al. [6] would be only to deduplicate zero pages. According to their evaluation, between 84% and 94% of the deduplication in Microsoft Edge are only zero pages [6]. However, the covert channel is still possible with this solution since we can still trigger copy-on-write page faults on deduplicated zero pages.

TPS. VMWare TPS [54] uses additional salts to enable memory deduplication. The salt value and the content of page have to be identical to be shared. If VMs want to deduplicate shared content, the salted value is unknown to an attacker. While this approach protects against cross-VM attacks, TPS does not protect against remote memory-deduplication attacks in the same domain.

CovertInspector. Wang et al. [55] demonstrated an approach to detect memory-deduplication attacks by modifying KVM by 300 lines of code. Their approach has a particular focus on intercepting the `rdtsc` instruction triggered by the VM and also the number of pagefaults. Remote timers are not considered by CovertInspector.

VUsion. VUsion [38] mitigates all kinds of memory-deduplication attacks by applying a share-XOR-fetch policy and fake merging. All pages that are considered for deduplication behave the same in terms of access times and copy-on-write pagefaults. Fake merging guarantees that every access on a page, both shared or non-shared, behaves the same in terms of access time. This mechanism prevents attacks on the detection of pages being actually deduplicated [38]. While fake merging would mitigate all of our attacks based on the copy-on-write page fault, it is not implemented nor intended to be merged in the Linux kernel.

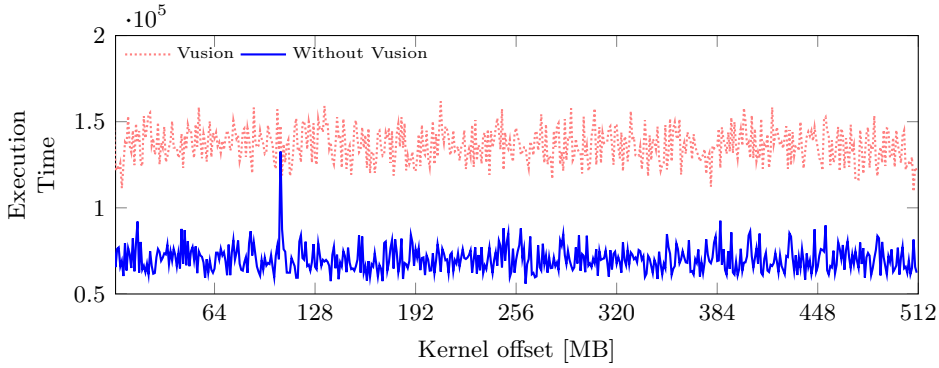


Figure 8.13.: Mean response time for all possible kernel offsets. While an adversary can easily observe the correct offset 106 on an unprotected system (blue), the VUsion-protected system (red) prevents the leakage.

We experimentally verified the effectiveness of VUsion against our remote memory-deduplication attacks in a local area network setting. Figure 8.13 illustrates the KASLR break (cf. Section 8.6.2) on a protected (red) and an unprotected Linux kernel 4.10 (blue) running Ubuntu 17.04 LTS. We measured the response time for every possible offset 100 times and reported the mean value. One can clearly see that our attack successfully recovers the correct offset 106 while the attack against the VUsion-protected kernel only observes higher timings.

Network-layer Countermeasures. On the network layer, we can mitigate remote memory-deduplication attacks via network packet inspection tools and DDoS monitoring. Another possibility to mitigate remote memory-deduplication attacks is by adding additional noise to the network, *i.e.*, by performing load balancing or adding discrete time delays. This would

require more samples for the attacker, and at a certain point, it could make the attack infeasible.

8.7.2. Alternative Attack Targets

We want to emphasize that fixing Memcached does not mitigate the problem of remote memory-deduplication attacks as our techniques are generic and can be applied to other applications as well. In addition to Memcached and InnoDB, we analyzed further applications which could be susceptible to remote memory-deduplication attacks. Many web applications offer the possibility to use Memcached, such as PHPBB, WordPress, Moodle, and PrestaShop. Moodle allows image caching, which might be already used to perform the fingerprinting attack. We analyzed the in-memory DB Redis and found that 4 kB pages can be also placed into the memory. There is again meta-data stored about the stored item, and it is again a question of the correct alignment for the attacker to perform remote memory-deduplication attacks. If the attacker's guess about the alignment is correct, copy-on-write pagefaults can be triggered in a similar manner to Memcached by freeing an item and again inserting a new one with the equal size. This leads to an overwrite of the deduplicated memory. Furthermore, we analyzed the other popular alternative for in-memory databases SQLite. However, we found that we could not fully place a single 4 kB page into memory. We also checked Aerospike and observed that memory is in DRAM as key-value pair and that the `aerospike_key_put` function directly replaces the content and could be used to trigger copy-on-write pagefaults. As already shown by Bosman et al. [6] also request pools like used in remotepagededup:nginx are susceptible to memory deduplication attacks.

8.8. Conclusion

In this work, we presented how memory deduplication can be exploited from a remote perspective. This attack does neither require local code execution nor JavaScript execution in the browser, as demonstrated in previous work. With targeted web requests, we can observe timing differences between duplicated pages over the network. We first evaluated the speed of our remote covert channel based on an HTTP web server achieving a performance of up to 302.16 B/h in a LAN setting and 34.41 B/h over

the internet. Further, we fingerprinted libraries used on the system by exploiting the Memcached database. It is possible to fingerprint libraries within 166.51 s over the internet. Within only 4 minutes, we successfully broke KASLR from a virtual machine running on a server 14 network hops away. Even though there are potential mitigations against memory deduplication within the same security domain, they are not applied in Linux systems. Finally, we leaked the database records' content from InnoDB with 1.5 B/h.

Acknowledgments

We would like to thank our anonymous reviewers for valueable feedback and comments on the paper. Furthermore, we want to thank Tom Van Goethem for feedback on the draft and support on the HTTP/2 experiments. We want to thank remotepagededup:Equinix Metal for providing us bare metal servers. This work was supported by generous funding and gifts from the EU project SOPHIA, Red Hat and AWS. Any opinions or recommendationsw expressed in this work are those of the authors and do not necessarily reflect the views of the funding parties.

Appendix

A. Timing Difference of Library Fingerprinting in PHP

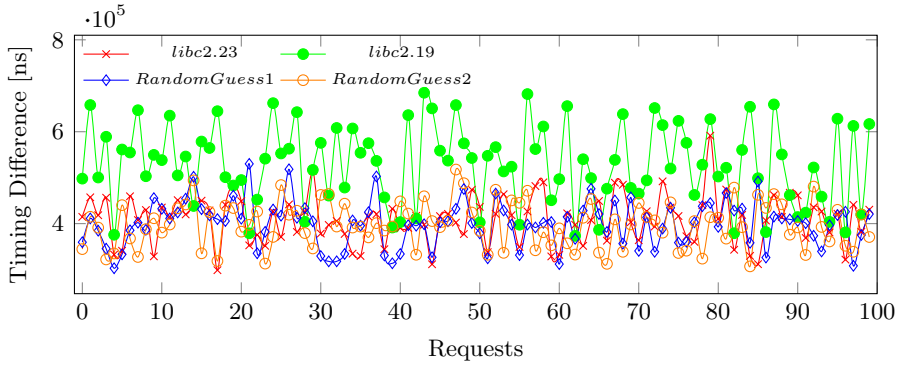


Figure 8.14.: Timing difference for mapped libc version (2.19) vs. other guesses.

The copy-on-write page faults can be observed in PHP when triggering the deduplication via Memcached Figure 8.14.

B. Validation of Requirements for Reorganization.

For mounting a successful oracle attack against an InnoDB record, it has to be guaranteed that a reorganization can be triggered reliably. Reorganizing is needed to switch between the different states introduced in Section 8.6.3.

The condition for the first reorganize from the initial state (Figure 8.12a) to the reorganized state (Figure 8.12b) is already guaranteed by the calculation of the initial size of $|r_{AX}|$ in Section 8.6.3.

For the switch from the reorganized state in Figure 8.12b to the reset state in Figure 8.12c it must be guaranteed that the restoring of $|r_{AT}|$ always triggers reorganization. Therefore the following inequality must hold:

$$|r_{AT}| > max\_free\_space - |\tilde{r}_{AT}| - |r_T| - |r_{AX}| - footer\_sz$$

We can claim that $|\tilde{r}_{AT}| + |r_T| \geq 4096$ must hold as otherwise the attacker does not even control one full page which is needed for the deduplication side channel. Using this and neglecting the `footer_sz` we get:

$$\begin{aligned} |r_{AT}| = 8125 &> max\_free\_space - (|\tilde{r}_{AT}| + |r_T|) - |r_{AX}| \\ &> 16252 - 4096 - 4064 = 8092 \end{aligned}$$

For the last state switch from the reset state to a new reorganized state there are two possibilities: $|r_{AX}|$ is either increased by δ as long as the resulting size is smaller than the maximum record size or it is set to the maximum record size. In both cases a reorganize should be triggered. Therefore for case 1 the following inequality must hold:

$$\begin{aligned} |r_{AX}| + \delta &> max\_free\_space - (|r_{AT}| - \delta) - |r_T| - |r_{AX}| \\ &\quad - footer\_sz \\ 2 * |r_{AX}| &> max\_free\_space - |r_{AT}| \\ 2 * 4064 = 8128 &> 16252 - 8125 = 8127 \end{aligned}$$

For case two we again use that $|\tilde{r}_{AT}| + |r_T| \geq 4096$ must hold:

$$\begin{aligned} |r_{AX,max}| &> max\_free\_space - |\tilde{r}_{AT}| - |r_T| - |r_{AX}| - footer\_sz \\ |r_{AX,max}| &> max\_free\_space - (|\tilde{r}_{AT}| + |r_T|) - |r_{AX}| \\ 8125 &> 16252 - 4096 - 4064 = 8092 \end{aligned}$$

Required Sizes of Records and Potential Leakage Rate. The record r_{AX} is required to trigger the reorganization of records. Therefore, it initially has to be large enough so that we can trigger a reorganization. We determine the worst case size of the left free space for records within an index page as follows after the first reorganization:

$$
\begin{aligned}
&|r_{AX}| + 1 > \text{trailing free space, which is always the case if}\\
&|r_{AX}| + 1 > \texttt{max\_free\_space} - |r_{AX}| - |r_{AT}| - |r_T| (-\texttt{footer\_sz}).\\
&\qquad \texttt{footer\_sz},\ |r_T| \text{ can be neglected in worst case inspection}\\
&\qquad |r_{AX}| > \frac{\texttt{max\_free\_space} - |r_{AT}| - 1}{2} = 4064\,\text{B}\\
&\text{therefore:}\\
&\qquad \texttt{left\_free\_space} = 16\,252\,\text{B} - 8125\,\text{B} - 4064\,\text{B} = 4063\,\text{B}.
\end{aligned}
$$

Next we want to determine the boundaries for the shift into our attacker-controlled 4 kB-page and the requirements. We define the maximum alignment change *max_alignment_change* as $r_{AT} - r_{AT_{header}}$. To leak data from r_T, one page of our attacker-controlled $\tilde{r}_{AT}$ record needs to be page aligned. As we chose the size of r_{AT} to be 8125 B, we do not fully control 2 pages. We use a certain part of r_{AT} to bring the last 4096 B into a page alignment. A certain page misalignment is even required to enable a successful attack, since with a very low misalignment (e.g., 42), we cannot control a full 4 kB page. For instance with a misalignment of 42 bytes we only control 4071 (8125 − 4096 + 42) bytes of the page to leak the record data (leak page). Therefore, the misalignment needs to be large enough to control a full leak page. Furthermore, the misalignment is unknown and we need to leak the misalignment of the page. This can be done via a remote memory-deduplication attack by trying different offsets until the correct one is found. The misalignment is the start of the record r_{AT} (cf. Figure 8.12a). We determine the minimal necessary misalignment $\mathit{offset}_{r_{AT}}$ after the initial reorganization:

$$
\begin{aligned}
\text{offset}_{r_{AT}} mod 4096 + |r_{AT}| - 1 &\geq 4096 * 2\\
\text{offset}_{r_{AT}} mod 4096 &\geq 4096 * 2 + 1 - |r_{AT}|\\
\text{offset}_{r_{AT}} &\geq 68.
\end{aligned}
$$

Hence, we need a misalignment of r_{AT} of at least 68 B. Furthermore, in case the record header moves to the leak page, we would have another unknown value to leak. Therefore, the misalignment needs to be smaller or equal to $4096 - |r_{AT_{header}}|$. Each record is preceded by a record header ($r_{AT_{header}}$), which is maximum 27 bytes in our scenario. We have the

probability of 0.66 % that the $r_{AT_{header}}$ moves to the leak page and 1.66 % that the misalignment is less than 68. We derive the maximum leakage rate for a InnoDB index page as:

$$\begin{aligned} &max\_leakage\_possible = \\ &\min(|r_{AT}| - 1 - |r_{AT_{header}}| - \\ &\text{offset}_{r_T} \bmod 4096 + |r_{AT}| - 2 \cdot 4096 + 1), |r_T|, 4096) \end{aligned}$$

Hence, we can leak up to a full size of $r_T \leq 4096$ if we leak the misalignment and the requirements for $\text{offset}_{r_{AT}}$ hold. This limits the leakage potential of the InnoDB attack.

C. MariaDB User Table.

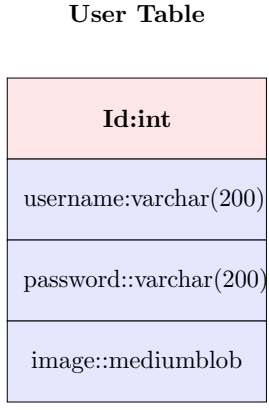


Figure 8.15.: MariaDB user table, which is susceptible to a remote memory-deduplication attack.

D. Memcached Eviction.

User table used in attack on InnoDB used in MariaDB Figure 8.15.

The attacks on Memcached rely on the assumption that an attacker can reliably trigger copy-on-write pagefaults by updating the same item. However, one problem that can occur in the Memcached attack is that another user gets the free deduplicated page assigned instead of the attacker. Therefore, it is a race between the attacker and potential other users to get the page and then trigger the copy-on-write page fault on the deduplicated page. Another issue is whether the pages stays cached in Memcached for a longer period until the deduplication by the operating

system happens, *i.e.*, 15 minutes on Windows. The eviction totally depends on the size of the Memcached instance itself.

In this experiment, we validate how long an entry is cached in an Memcached instance with different memory limits. First, we launch a new memcached instance and load as many entries into the instance until the memory limit is exhausted and the instance has to evict existing entries.

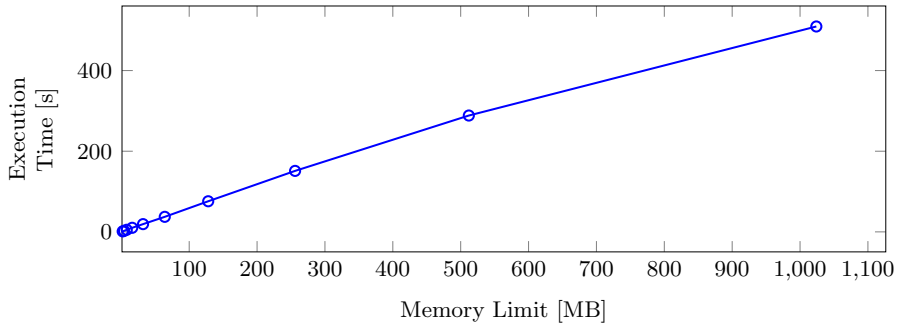


Figure 8.16.: Average execution time in seconds (n = 10) until a newly added entry is evicted from memcached depending on its given memory limit.

Then, we add a new entry and probe how many seconds this entry remains cached while we simultaneously apply a realistic workload on the instance. We utilize memtier_benchmark [43], spawning 4 threads that simultaneously write and read entries from the memcached instance using a gaussian access pattern with an average of 671 121 ops/sec and an average bandwith of 268.26 MB/s. Figure 8.16 illustrates after how many seconds on average the added entry is evicted from the memcached instance (n = 10). Figure 8.16 illustrates the eviction time for different Memcached node sizes. As can be seen, the larger the node is, the longer it takes to evict a certain item.

References

[1] Acıiçmez, Onur and Schindler, Werner and Koc, Cetin K. "Cache Based Remote Timing Attack on the AES." In: *CT-RSA*. 2006.

[2] Hassan Aly and Mohammed ElGayyar. "Attacking aes using bernstein's attack on modern processors." In: *International Conference on Cryptology in Africa*. 2013.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM." In: *Proceedings of the linux symposium*. Citeseer. 2009, pp. 19–28.

[4] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. "CAIN: Silently Breaking ASLR in the Cloud." In: *WOOT*. 2015.

[5] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

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

[7] David Brumley and Dan Boneh. "Remote Timing Attacks Are Practical." In: *USENIX Security*. 2003.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs." In: *CCS*. 2019.

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. "KASLR: Break It, Fix It, Repeat." In: *AsiaCCS*. 2020.

[10] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. "Opportunities and limits of remote timing attacks." In: *ACM Transactions on Information and System Security (TISSEC)* 12.3 (2009), p. 17.

[11] Skyscanner Engineering. *Journey to the centre of memcached*. 2020. URL: `{https://medium.com/@SkyscannerEng/journey-to-the-centre-of-memcached-b239076e678a}` (visited on 01/21/2020).

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

[13] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era." In: *CCS*. 2020.

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

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

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

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

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

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

[20] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. "Remote cache timing attack on advanced encryption standard and countermeasures." In: *ICIAFs*. 2010.

[21] Taehun Kim, Taehyun Kim, and Youngjoo Shin. "Breaking KASLR Using Memory Deduplication in Virtualized Environments." In: *Electronics* (2021). URL: https://www.mdpi.com/2079-9292/10/17/2174.

[22] Amit Klein and Benny Pinkas. "From IP ID to Device ID and KASLR Bypass." In: *USENIX Security*. 2019.

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

[24] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs." In: *EuroS&P*. 2020.

[25] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "NetCAT: Practical Cache Attacks from the Network." In: *S&P*. 2020.

[26] Jens Lindemann and Mathias Fischer. "A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines." In: 2018.

[27] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors." In: *AsiaCCS*. 2020.

[28] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. "PLATYPUS: Software-based Power Side-Channel Attacks on x86." In: *S&P*. 2021.

[29] Percona Marco Tusa. *InnoDB Page Merging and Page Splitting*. 2017. URL: https://www.percona.com/blog/2017/04/10/innodb-page-merging-and-page-splitting/.

[30] MariaDB. 2020. URL: https://github.com/MariaDB/server/blob/09a1f0075a8d5752dd7b2940a20d86a040af1741/storage/innobase/row/row0upd.cc.

[31] MariaDB. 2020. URL: https://github.com/MariaDB/server/blob/09a1f0075a8d5752dd7b2940a20d86a040af1741/storage/innobase/btr/btr0cur.cc.

[32] MariaDB. 2020. URL: https://github.com/MariaDB/server/blob/09a1f0075a8d5752dd7b2940a20d86a040af1741/storage/innobase/include/page0page.ic.

[33] MariaDB. 2020. URL: https://dev.mysql.com/doc/internals/en/innodb-infimum-and-supremum-records.html.

[34] Memcached. 2020. URL: https://memcached.org/.

[35] Microsoft. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_side_channel_isolation_policy.

[36] Microsoft. *Azure serverless computing*. 2019. URL: https://azure.microsoft.com/en-us/overview/serverless-computing/.

[37] MySQL. 2017. URL: https://blog.toadworld.com/2017/10/19/data-flushing-mechanisms-in-innodb.

[38] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Secure Page Fusion with VUsion." In: *SOSP*. 2017.

[39] Rodney Owens and Weichao Wang. “Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines.” In: *International Performance Computing and Communications Conference*. 2011.

[40] Gerald Palfinger, Bernd Prünster, and Dominik Ziegler. “Prying CoW: Inferring Secrets across Virtual Machine Boundaries.” In: *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*. 2019.

[41] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. “Flip Feng Shui: Hammering a Needle in the Software Stack.” In: *USENIX Security Symposium*. 2016.

[42] Red Hat. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*. 2017.

[43] Redis. 2013. URL: `https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached`.

[44] remotepagededup:Equinix. 2021. URL: `https://www.equinix.com`.

[45] remotepagededup:nginx. 2021. URL: `https://www.remotepagededup:nginx.com/`.

[46] remotepagededup:Pistache. 2020. URL: `http://pistache.io/`.

[47] Vishal Saraswat, Daniel Feldman, Denis Foo Kune, and Satyajit Das. “Remote Cache-timing Attacks Against AES.” In: *Workshop on Cryptography and Security in Computing Systems*. 2014.

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

[49] SUSE. 2021. URL: `https://documentation.suse.com/sles/12-SP4/pdf/article-vt-best-practices_color_en.pdf`.

[50] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. “Memory Deduplication as a Threat to the Guest OS.” In: *EuroSys*. 2011.

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

[52] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. "Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections." In: *USENIX Security Symposium*. 2020.

[53] Mathy Vanhoef and Tom Van Goethem. "HEIST: HTTP Encrypted Information can be Stolen through TCP-windows." In: *Black Hat US Briefings, Location: Las Vegas, USA*. 2016.

[54] Vmware. 2021. URL: `https://kb.vmware.com/s/article/2097593`.

[55] S. Wang, Weizhong Qiang, H. Jin, and Jinfeng Yuan. "CovertInspector: Identification of Shared Memory Covert Timing Channel in Multi-tenanted Cloud." In: *International Journal of Parallel Programming* (2015).

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

[57] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. "Security implications of memory deduplication in a virtualized environment." In: *International Conference on Dependable Systems and Networks (DSN)*. 2013.

[58] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals Part 1*. 7th ed. Microsoft Press, 2017.

[59] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. "Cache Timing Attacks on Camellia Block Cipher." In: *Cryptology ePrint Archive, Report 2009/354* (2009).

9

Practical Timing Side Channel Attacks on Memory Compression

Publication Data

Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. "Practical Timing Side Channel Attacks on Memory Compression." In: *S&P*. 2023

Contributions

Main author.

Practical Timing Side-Channel Attacks on Memory Compression

Martin Schwarzl[1], Pietro Borrello[2], Gururaj Saileshwar[1], Hanna Müller[1] Michael Schwarz[3], Daniel Gruss[1],

[1] Graz University of Technology, Austria [2] Sapienza University of Rome, Italy [3] CISPA Helmholtz Center for Information Security, Germany

Abstract

Compression algorithms have side channels due to their data-dependent operations. So far, only the compression-ratio side channel was exploited, e.g., the compressed data size.

In this paper, we present Decomp+Time, the first memory-compression attack exploiting a timing side channel in compression algorithms. While Decomp+Time affects a much broader set of applications than prior work, a key challenge is precisely crafting attacker-controlled compression payloads to enable the attack with sufficient resolution. We develop an evolutionary fuzzer, Comprezzor, to find effective Decomp+Time payloads that optimize latency differences and find payloads that are so effective that decompression timing can even be exploited in remote Decomp+Time attacks across the Internet. Decomp+Time has a capacity of 9.73 kB/s locally, and 10.72 bit/min across the internet (14 hops, **> 700** miles). Using Comprezzor, we develop attacks that leak data byte-by-byte in four different case studies: First, we leak 1.50 bit/min from Memcached on a remote server running a PHP application. Second, we leak database records with 2.69 bit/min from PostgreSQL, managed by a Python-Flask application, over the internet. Third, we leak secrets with 49.14 bit/min locally from ZRAM-compressed pages on Linux. Fourth, we leak internal heap pointers from the V8 engine within the Google Chrome browser on a system using ZRAM. This highlights the importance of re-evaluating the use of compression on sensitive data even if the application is only reachable via a remote interface.

9.1. Introduction

Data compression plays a vital role for reducing the memory and storage utilization and in file formats such as PDF, image, and video files. Similarly, operating systems (OSs) rely on memory compression [6, 86] to reduce system memory utilization. Memory compression is also used in databases [63] and key-value stores [62]. Compression can even increase performance and efficiency when storing or transferring data to slow storage devices or across networks. Hence, data compression is also widely used for HTTP traffic [20, 54] and file-system compression [12]. Recent trends include columnar (column-oriented) compression to reduce the disk utilization for databases [5, 15, 50, 57]. If the data to compress is secret, the compression ratio depends on the secret, introducing a *compression-ratio side channel*, often exploited in the context of TLS-encrypted traffic [8, 25, 39, 55, 66, 77, 78]. All these attacks focused on web traffic and only exploited differences in the compressed size of data when compressed together with attacker-controlled data. The size of the compressed data is either accessed directly [66] in a man-in-the-middle scenario or indirectly by observing the transmission time that linearly depends on the size of the compressed data [8] and, thus, the compression ratio.

Compression trades data size for computation time. However, so far, only the *result* of the compression, *i.e.*, the compressed size, has been exploited to leak data but not the time consumed by the *process* of compression or decompression itself. First described by Kelsey et al. [40], most attacks focus on compressed web traffic. Surprisingly, security implications of compression in other settings, such as virtual memory or databases, have not been studied much. This raises two questions:

***Q1**: Can timing differences in compression and decompression time be exploited if the compression ratio is not observable?*
***Q2**: Can these timing differences be significant enough to exploit them in a fully remote setting?*

In this paper, we present Decomp+Time, the first memory-compression attack exploiting a *timing side channel in memory decompression*. We show that the *decompression time* directly leaks information about the compressed data. Our timing side channel exploits large timing differences for edge cases when decompressing nearly incompressible data. Since these edge cases require surgically crafted attacker-controlled payloads, we

developed Comprezzor, an evolutionary fuzzer to generate memory layouts to trigger and amplify the edge cases. The techniques we present are generic and can be applied to various compression algorithms implementing sequence compression. We show that the Comprezzor-based payloads influence the decompression time so significantly that they can be observed remotely when the compressed data never leaves the victim system, *i.e.*, the compression-ratio side channel is not exploitable.

We compare the latency differences induced by Comprezzor-generated algorithm-specific attacker payloads and manually crafted ones and find that Comprezzor-generated attacker payloads have latency differences up to three orders of magnitude above manually crafted layouts. We evaluate four realistic secret-leakage scenarios by generating these precise high-latency-inducing payloads. We even demonstrate remote attacks on an in-memory database system without executing code on the victim machine and without observing the victim's network traffic. Hence, our case studies show that compressing sensitive data poses a security risk in any scenario using compression and not just for web traffic.

We systematically analyze six compression algorithms, including widely-used algorithms such as DEFLATE (in zlib), PGLZ (in PostgreSQL), and zstd (by Facebook). Comprezzor is easy to extend to new compression algorithms, and it already fully supports all of these compression algorithms. Our findings show that the decompression time not only correlates with the entropy of the uncompressed data but also with various other aspects, such as the relative position of secret data or alignment of compressible data. In general, these timing differences arise due to the design of the compression algorithm and, *importantly, also its implementation.* Our results show that all analyzed compression algorithms are susceptible to timing side channels when observing data-compression and -decompression times.

We evaluate Decomp+Time in scenarios where secret data is compressed alongside attacker-controlled data. This is a common scenario in virtual memory and also in databases where victim data and attacker-controlled data may be placed in a single cell, e.g., when storing structured data like JSON documents.[1] The attacker guesses the secret byte by byte while measuring the decompression time e.g., via a web request that on

[1]Importantly, there is no indication or recommendation to not place victim data and attacker-controlled data in a single cell. Thus, there is no reason why users should not do this as of today. Our work and related documentation updates inform users about the risks they are exposing their data to.

the server-side performs a simple read access to the data in compressed memory. We evaluate the capacity of Decomp+Time in a covert channel abusing the memory compression of Memcached, an in-memory object caching system. We can, on average, transmit 9.73 kB/s locally and 10.72 bit/min across the internet (14 hops, > 700 miles).

We present four case studies leaking sensitive compressed data byte by byte: First, we attack an internet-facing PHP application using Memcached internally to leak a secret in 5.32 min per byte over the internet, *i.e.*, 1.50 bit/min. Second, we leak database records from a remote web app using PostgreSQL internally, with transparent database compression, at 2.97 min per byte, *i.e.*, 2.69 bit/min. Third, we exploit ZRAM, the memory compression module in Linux, transparently introducing timing side channels regardless of the security needs of the application.[1] In this setting, we leak a secret locally in 0.16 min per byte, *i.e.*, 49.14 bit/min. Fourth, we demonstrate an end-to-end exploit leaking internal heap pointers from sandboxed JavaScript inside the Google Chrome browser.

Our work highlights the importance of re-evaluating the use of compression on sensitive data on any layer, even if the application is only reachable via a remote interface.

Contributions. The main contributions of this work are:

1. We present a systematic analysis of timing leakage for several lossless data-compression algorithms.
2. We develop an evolutional fuzzer to find surgically precise attacker payloads to trigger extremely slow edge cases in memory decompression algorithms.
3. We demonstrate the possible leakage rate with a remote covert channel leaking 9.73 kB/s locally, and 10.72 bit/min across the internet (14 hops, > 700 miles).
4. We leak secrets byte-by-byte using Memcached, PostgreSQL, and ZRAM, with leakage rates between 1.5 bit/min to 2.69 bit/min in the remote setting and 1.5 kB/s to 9.73 kB/s in the local setting.

Disclosure. We responsibly disclosed our findings to the developers, and the issues were assigned `CVE-2022-0925`. We will open-source Comprezzor

[1] This is particularly dangerous as users may be unaware and uninformed about this behavior of the OS, that introduces leakage in their applications.

and attacks on Github[1]. A video of the PGLZ attack can be found on Streamable[2].

9.2. Background and Related Work

9.2.1. Data Compression Algorithms

Lossless compression reduces the size of data without losing information. One of the most popular algorithms is the DEFLATE compression algorithm [18], which is used in gzip (zlib). The DEFLATE compression algorithm consists of two main parts, LZ77 followed by Huffman encoding. The `Lempel-Ziv` (LZ77) part scans for the longest repeating sequence within a sliding window and replaces repeated sequences with a reference to the first occurrence [14]. This reference stores distance and length of the occurrence. The `Huffman`-coding part tries to reduce the redundancy of symbols. When compressing data, DEFLATE first performs LZ77 encoding and Huffman encoding [14]. When decompressing data (inflate), they are performed in reverse order. The algorithm provides different compression levels to optimize for compression speed or compression ratio. The smallest possible sequence has a length of 3 B [18]. Other algorithms provide different design points for compressibility and speed. Zstd, designed by Facebook [16] for modern CPUs, improves both compression ratio and speed, and is used for compression in file systems (e.g., btrfs, squashfs) and databases (e.g., AWS Redshift, RocksDB). LZ4 and LZO are optimized for compression and decompression speed. Especially LZ4 gains its performance by using a sequence compression stage (LZ77) without the symbol encoding stage (Huffman) like in DEFLATE. FastLZ, similar to LZ4, is a fast compression algorithm implementing LZ77. PGLZ is a fast LZ-family compression algorithm used in PostgreSQL for varying-length data in the database [63].

9.2.2. Prior Data Compression Attacks

In 2002, Kelsey [40] first showed that any compression algorithm is susceptible to information leakage based on the compression-ratio side channel. Duong and Rizzo [66] applied this idea to steal web cookies with the

[1] `https://github.com/anonymized-for-submission`
[2] `https://streamable.com/qxr9h4`

CRIME attack by exploiting TLS compression. In the CRIME attack, the attacker adds additional sequences in the HTTP request, which act as guesses for possible cookies values, and observes the request packet length, *i.e.*, the compression ratio of the HTTP header injected by the browser. If the guess is correct, the LZ77-part in gzip compresses the sequence, making the compression ratio higher, thus allowing the secret to be discovered. For CRIME, the attacker needs to spy on the packet length, and the secret needs a known prefix such as `cookie=`. To mitigate CRIME, TLS-level compression was disabled for requests [8, 25]. The BREACH attack [25] revived the CRIME attack by attacking HTTP responses instead of requests and leaking secrets in the HTTP responses such as cross-site-request-forgery tokens. The TIME attack [8] uses the time of a response as a proxy for the compression ratio, as it can be measured even via JavaScript. To reliably amplify the signal, the attacker chooses the size of the payload such that additional bytes, due to changes in compressibility, cross a boundary and cause significantly higher delays in the round-trip time (RTT). TIME exploits the compression ratio to amplify timing differences via TCP windows and does not exploit timing differences in the underlying compression algorithm itself. Vanhoef and Van Goethem [78] showed with HEIST that HTTP/2 features can also be used to determine the size of cross-origin responses and to exploit BREACH using the information. Van Goethem et al. [77] similarly showed that compression can be exploited to determine the size of any resource in browsers. Karaskostas and Zindros [39] presented Rupture, extending BREACH attacks to web apps using block ciphers. Voracle [55] exploits compression in VPNs using similar techniques as CRIME. Tsai et al. [75] demonstrated cache timing attacks on compressed caches, leaking a secret key in under 10 ms. **Common Theme.** Prior attacks primarily exploit the compression-ratio side channel. However, the time taken by the underlying compression algorithm is not analyzed or exploited as side channels. Additionally, these attacks largely target the HTTP traffic and website content, and do not focus on the broader applications of compression such as memory compression, databases, and others, that we target in this paper.

9.2.3. Fuzzing to Discover Side Channels

Historically, fuzzing has been used to discover memory corruption bugs in applications [21, 46, 59, 87]. Typically, it involves feedback based

on novelty search, executing inputs, and collecting ones that cover new program paths in the hope of triggering bugs. Other fuzzing proposals use genetic algorithms to direct input generation towards interesting paths [65, 73]. Directed fuzzing guides the exploration towards specific program points that are identified as interesting [9, 24, 26]. Recently, fuzzing has also been used to discover side channels both in software and in the microarchitecture [23, 27, 51, 81]. `ct-fuzz` [34] used fuzzing to discover timing side channels in cryptographic implementations. Nilizadeh et al. [56] used differential fuzzing to detect compression-ratio side channels that enable the CRIME attack. Bang et al. [7] used symbolic execution to discover side-channel leakage for compression-ratio attacks.

9.3. High-level Overview

In this section, we discuss the high-level overview of memory compression attacks and the attack model.

9.3.1. Attack Model & Attack Overview

Attack Model. Most prior attacks discussed in Section 9.2.2 focus on the compression ratio side channel. However, observing the compression-ratio over the network requires a strong attacker who can monitor the network traffic. Additionally, this information must be exposed by the system, which is typically not the case if the compressed data is only handled on the remote system and not transferred to the attacker. Even the TIME attack and its variants by Vanhoef and Van Goethem et al. [77, 78] only exploit timing differences due to the TCP protocol. None of these exploited or analyzed timing differences due to the compression algorithm itself, which is the focus of our attack. Once the attacker data is compressed with the secret, the attacker only needs to measure the latency of a subsequent read access to the attacker-controlled data. As we expect system noise, when performing the experiment, we assume that the attacker can repeat the measurement multiple times. Furthermore, we assume no software vulnerabilities in the application itself. A public API provides an interface to upload, read and modify data, which is compressed and stored either in main memory or on the disk. The threat model is similar to fully remote attacks, as, presented by Schwarzl et al. [71].

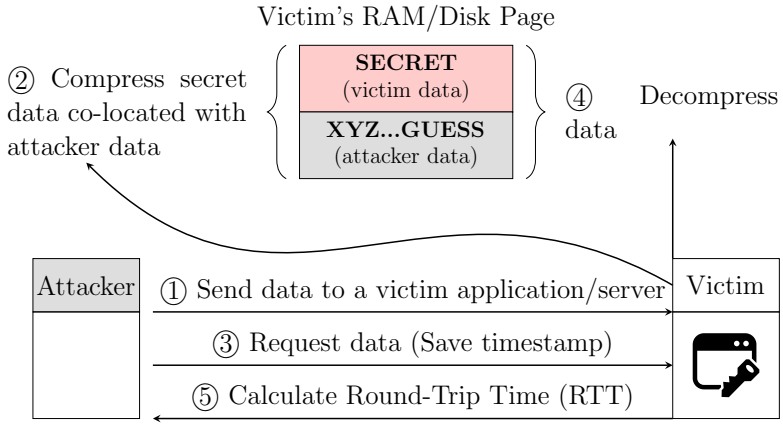


Figure 9.1.: Overview of a memory compression attack exploiting a timing side channel.

Data Co-location. We assume that the attacker can co-locate data with secret data. This assumption is in line with all previous memory compression attacks [8, 25, 39, 66, 77, 78]. For HTTP requests/responses, the attacker was able to arbitrarily co-locate guesses of the cookie value, e.g., `Known Data (Prefix) || Secret Data || Attacker-controlled data`. Moreover, co-location can also occur in the other direction with a known suffix or direct co-location of attacker-controlled and secret data e.g., `Secret Data || Known Data (Suffix) || Attacker-controlled data`.

In applications, co-location is possible not only in HTTP requests, but also via a memory storage API like Memcached, with a shared database between attacker and victim that compresses multiple rows or columns. For cellular compression, co-location might occur in JSON fields storing data from different origins. Moreover, co-location can occur directly in virtual memory. For instance, pointers can be co-located with other attacker-controlled data structures (on the heap) and compressed by the operating system. In such a setting, potential targets are internal malloc pointers to libc functions for breaking ASLR or internal pointers to metadata in JavaScript engines. We demonstrate four case studies, where co-location leads to data leakage in commonly used software in Section 9.6.

Attack Overview. Figure 9.1 illustrates an overview of a memory compression attack in five steps. The victim application can be a web server with a database or software cache, or a filesystem that compresses

stored files. **First**, the attacker sends its data to be stored to the victim's application. **Second**, the victim application compresses the attacker-controlled data, together with some co-located secret data, and stores the compressed data. The attacker-controlled data contains a partial guess of the co-located victim's data `SECRET` or, in the case where a prefix or suffix is known, `prefix=SECRET`. The guess can be performed bytewise to reduce the guessing entropy. If the partial guess (e.g., SECR) is correct, the compressed data not only has a higher compression ratio, but it also influences the decompression time. **Third**, after the compression happened, the attacker requests the content of the stored data again and takes a timestamp. **Fourth**, the victim application decompresses the attacker-controlled input together with the secret data and acknowledges the request. **Fifth**, the attacker takes another timestamp when the application responds and computes the RTT as the difference between the two timestamps. Based on the RTT, which depends on the decompression latency of the algorithm, the attacker infers whether the guess was correct or not and leaks the secret data. Thus, the attack relies on the *timing differences* of the compression algorithm itself, which we characterize next.

9.4. Systematic Study: Compression Algorithms

In this section, we provide a systematic analysis of timing leakage in compression algorithms. We choose six popular compression algorithms (zlib, zstd, LZ4, LZO, PGLZ, and FastLZ), and evaluate the compression and decompression times based on the input data's entropy. Zlib, implementing the DEFLATE algorithm, is commonly used for compressing files and is used in gzip. Zstd is Facebook's alternative to Zlib. PGLZ is used in the popular relational database management system PostgreSQL. LZ4, FastLZ, and LZO were built to increase compression speeds. For each algorithm, we observe timing differences in the range of hundreds to thousands of nanoseconds based on the content of the input data (4 kB-page).

9.4.1. Experimental Setup

We conducted the experiments on an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) with a fixed frequency of 4 GHz. We evaluate the latency of each compression algorithm with three different input values, each 4 kB in size. The first input is the same byte repeated 4096 times, which should be

Table 9.1.: Different compression algorithms yield distinguishable timing differences when decompressing 4 kB content with a different entropy (n = 100000).

Algorithm	Fully Compressible (ns)	Partially Compressible (ns)	Incompressible (ns)
FastLZ	7257.88 (±0.23%)	4264.56 (±2.27%)	1155.57 (±0.92%)
LZ4	605.79 (±1.02%)	218.68 (±1.76%)	107.90 (±2.49%)
LZO	2115.65 (±2.05%)	1220.07 (±3.64%)	309.44 (±6.27%)
PGLZ	813.75 (±0.71%)	5340.47 (±0.38%)	-
zlib	7016.02 (±0.33%)	13 212.53 (±0.35%)	1640.09 (±1.51%)
zstd	941.05 (±0.94%)	772.55 (±0.77%)	370.59 (±2.87%)

fully compressible. The second input is *partly compressible* and a hybrid of two other inputs: half random bytes and half compressible repeated bytes. The third input consists of random bytes which are theoretically *incompressible*. With these, we show that compression algorithms have different timings depending on the compressibility of the input.

9.4.2. Timing Differences for Different Inputs

For each algorithm and input, we measure the decompression and compression time of a 4 kB data blob over 100 000 repetitions and compute the mean values and standard deviations.

Decompression. Table 9.1 lists the decompression latencies for all evaluated compression algorithms. Depending on the entropy of the input data, there is considerable variation in the decompression time. All algorithms incur a higher latency for decompressing a fully compressible page compared to an incompressible page, leading to a timing difference of few hundred to few thousand nanoseconds for different algorithms. This is because, for incompressible data, algorithms can augment the raw data with additional metadata to identify such cases and perform simple memory copy operations to "decompress" the data, as is the case for zlib where the decompression for an incompressible page is 5375.93 ns faster than for a fully-compressible page. For decompression of partially-compressible pages, some algorithms (FastLZ, LZ4, LZO, zstd) lead to lower latencies compared to fully-compressible pages. Zlib and PGLZ lead to a higher

decompression latency for partially-compressible pages compared to fully-compressible pages. This shows the existence of even algorithm-specific variations in timings. PGLZ does not create compressible memory in the case of an incompressible input, and hence we do not measure its latency for this input.

Compression. For compression, we observed a trend in the other direction (Table 9.4 in Section B lists compression latencies for different algorithms). For different levels of compressibility, there are also latencies between the three different inputs, which are clearly distinguishable in the order of multiple hundreds to thousands of nanoseconds. Thus, timing side channels from compression might also be used to exploit compression of attacker-controlled memory co-located with secret memory. However, attacks using the compression side channel might be harder to perform in practice as the compression of data might be performed in a separate task (in the background), and the latency is, therefore, not easily observable for an attacker. Hence, our work focuses on attacks exploiting the decompression timing side channel.

Handling of Corner Cases. For incompressible pages, the "compressed" data can be larger than the original size with the additional compression metadata. Additionally, it is slower to access after compression than raw uncompressed data. Hence, this corner-case with incompressible data may be handled in an implementation-specific manner, which can itself lead to additional side channels. For example, a threshold for the compression ratio can decide when a page is stored in a raw format or in a compressed state, like in Memcached-PHP [62]. PGLZ, the algorithm used in PostgreSQL database, which computes the maximum acceptable output size for input by checking the input size and the strategy compression rate, could fail to compress inputs in such corner cases.

In Section 9.6, we show how real-world applications like Memcached, PostgreSQL, and ZRAM deal with such corner cases and demonstrate attacks on each of them.

9.4.3. Leaking Secrets via Timing Side Channels

Thus far, we analyzed timing differences for decompressing different inputs, which in itself is not a security issue. In this section, we demonstrate Decomp+Time to leak secrets from compressed pages using these timing differences. We focus on sequence compression, *i.e.*, LZ77 in DEFLATE.

Building Blocks for Decomp+Time

Decomp+Time consists of 3 building blocks: *sequence compression* to modulate the compressibility of an input, *co-location* of attacker data and secrets, and *timing variation for decompression* depending on the change in compressibility of the input.

Sequence compression: Sequence compression *i.e.*, LZ77 tries to reduce the redundancy of repeated sequences in an input by replacing each occurrence with a pointer to the first occurrence. This results in a higher compression ratio if redundant sequences are present in the input and a lower ratio if no such sequences are present. This compressibility side channel can leak information about the compressed data.

Co-location of attacker data and secrets: If the attacker can control a part of data that is compressed with a secret, as described in Figure 9.1, then the attacker can place a *guess* about the secret and place it co-located with the secret to exploit sequence compression. If the compression ratio increases, the attacker can infer if the guess matches the secret or not. While the CRIME attack [66] previously used a similar set up and observed the compressed size of HTTP requests to steal secrets like HTTP cookies, we introduce a more general attack that does not require observability of compressed sizes.

Timing Variation in Decompression: We infer the change in compressibility via its influence on the decompression timing. We observe that even sequence compression can cause variation in the decompression timing based on compressibility of inputs (for all algorithms in Section 9.4.2). If the sequence compression reduces redundant symbols in the input and increases the compression ratio, we observe faster decompression due to fewer symbols. Otherwise, with a lower compression ratio and more symbols, decompression is slower. Hence, the attacker can infer the compressibility changes for different guesses by observing differences in decompression time. For a correct guess, the guess and the secret are compressed together and the decompression is faster due to fewer symbols. For incorrect guesses with more symbols it is slower.

Launching Decomp+Time

Using the building blocks described above, we set up the attack with an artificial victim program that has a 6 B secret string (`SECRET`) embedded

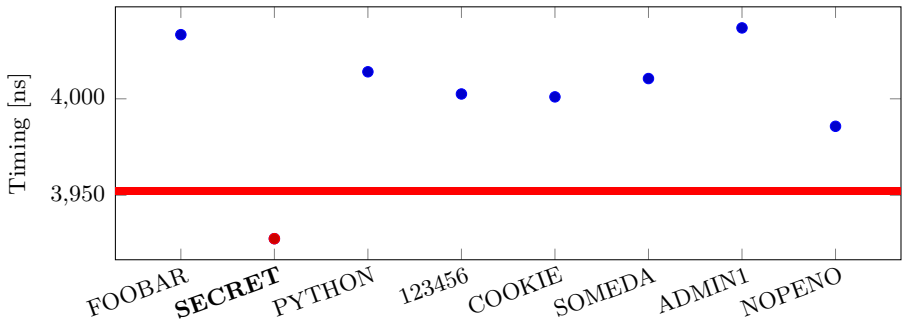


Figure 9.2.: Decompression time with Decomp+Time for different guesses of the secret value. A threshold (line) separating the correct from wrong secrets.

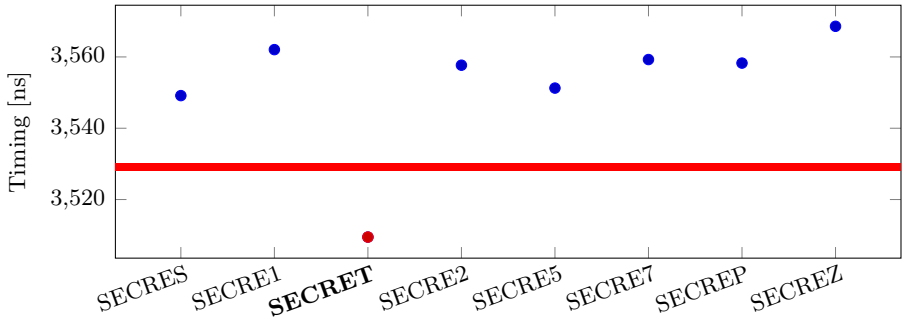


Figure 9.3.: Bytewise-leakage of the secret's last byte. A threshold (line) separates the correct from the wrong guess.

into a 4 kB page. The page also contains attacker-controlled data that is compressed together with the secret, like the scenario shown in Figure 9.1. The attacker can update its own data in place to make multiple guesses. The attacker can also read this data, which triggers a decompression of the page and allows the attacker to measure the decompression time. A correct guess that matches the secret results in faster decompression.

We perform the attack on the zlib library (1.2.11) and use 8 different guesses, including the correct guess. For each guess, a single string is placed 512 B away from the secret value; Note that this offset is arbitrarily chosen, and other offsets also work. Other data in the page is initialized with dummy values (repeated number sequence from 0 to 16). To measure the execution time, we use the `rdtsc` instruction.

Evaluation. Our evaluation was performed on an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) with a fixed frequency of 4 GHz. To get stable results, we repeat the decompression step with each guess 10 000 times and repeat the entire attack 100 times. For each guess, we take the minimum timing difference per guess and choose the global minimum timing difference to determine the correct guess. Figure 9.2 illustrates the minimum decompression times. With zlib, we observe that the correct guess is faster on average by 71.5 ns ($n = 100, \sigma_{\bar{\mu}} = 199.55\%$) compared to the second-fastest guess. Our attack correctly guessed the secret in all 100 repetitions of the attack. While we used a 6 B secret, our experiment also works for smaller secrets down to a length of 4 B.

Bytewise Leakage. If the attacker manages to guess or know the first three bytes of the secret, the subsequent bytes can even be leaked bytewise using our attack. Both CRIME and BREACH assume a known prefix such as `cookie=`. Similar to CRIME and BREACH [25, 39, 66], we try to perform a bytewise attack by modifying our simple layout. We use the first 5 characters of $SECRET$ as a prefix (`"SECRE"`) and guess the last byte with 7 different guesses. On average, the latency is 28.37 ns ($n = 100, \sigma_{\bar{\mu}} = 186.61\%$), between the secret and second fastest guess. Figure 9.3 illustrates the minimum decompression times for the different guesses. However, we observe an error rate of 8 % for this experiment, which might be caused by the Huffmann-decoding part in DEFLATE.

While techniques like the Two-Tries method [25, 39, 66] have been proposed to overcome the effects of Huffman-coding in DEFLATE to improve the fidelity of byte-wise attacks exploiting compression ratio, we seek to explore whether bytewise leakage can be reliably performed via the timing only by amplifying the timing differences.

Challenge of Amplifying Timing

While the decompression timing side channels can be used in attacks, the timing differences are quite small for practical exploits on real-world applications. For example, the timing differences we observe for the correct guess are in tens of nanoseconds, while most practical use cases of compression, like a Memcached server accessed over the network or PostgreSQL database accessed from a disk, could have access latencies of milliseconds.

Amplification. To enable memory compression attacks even via the network, we need to amplify the timing difference between correct and incorrect guesses. However, it is impractical to manually identify inputs that could amplify the timing differences, as each compression algorithm has a different implementation that is often highly optimized. Moreover, various input parameters could influence the timing of decompression, such as frequency of sequences, alignments of the secret and attack-controlled data, size of the input, entropy of the input, and different compression levels provided by algorithms. We develop an evolutionary fuzzer, Comprezzor, to automatically find inputs that amplify the timing difference between correct and incorrect guesses for compression algorithms.

9.5. Evolutionary Compression-Time Fuzzer

Compression algorithms are highly optimized and complex. Hence, we introduce Comprezzor, an evolutionary fuzzer to discover attacker-controlled inputs for compression algorithms that maximize differences in decompression times for certain guesses enabling bytewise leakage.The motivation for this automated tool is that there are too many possibilities for crafting efficient payloads manually. Our manual attempts only result in minimal timing differences that are difficult to exploit.

Comprezzor empowers genetic algorithms to amplify decompression side channels. It treats the decompression process of a compression algorithm as an opaque box and mutates inputs to the compression while trying to maximize the output, *i.e.*, timing differences for decompression with different guesses. The mutation process in Comprezzor focuses on the entropy of the data and the memory layout and alignment that end up triggering optimizations and slow paths. Figure 9.4 illustrates a high-level overview of the steps Comprezzor performs.

While previous approaches used fuzzing to detect timing side channels [34, 56], Comprezzor can dramatically amplify timing differences by being specialized for compression algorithms by varying parameters like the input size, layout, and entropy that affect the decompression time. The inputs discovered by Comprezzor can amplify timing differences to such an extent that they are even observable remotely.

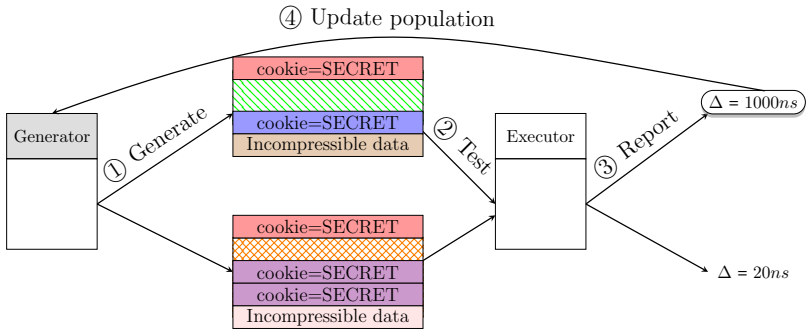


Figure 9.4.: Comprezzor generates memory layouts with different entropy, input size, and secret alignment, leading to high-latency decompression times. Every iteration, samples with the highest latency difference are used as input to generate newer layouts.

9.5.1. Design of Comprezzor

In this section, we describe the key parts of our fuzzer: Input Generation, Fitness Function, Input Mutation and Evolution.

Input Generation. Comprezzor generates memory layouts for Decomp+Time by maximizing the timing differences on decompression of the correct guess compared to incorrect ones. Comprezzor creates layouts with sizes in the range of 1 kB to 64 kB. It uses a helper program that takes the memory layout configuration as input, builds the requested memory layout for each guess, compresses them using the target compression algorithm, and reports the observed timing differences in the decompression times among the guesses. A memory layout configuration is composed of a file to start from, the offset of the secret in the file, the offset of guesses, how often the guesses are repeated in the layout, the compression level (*i.e.*, between 1 and 9 for zlib), and a modulus for entropy reduction that reduces the range of the random values. The fuzzer can be used in cases where a prefix or suffix is known and unknown.

Fitness Function. The evolutionary algorithm of Comprezzor starts from a random population of candidate layouts (samples) and takes as feedback the difference in time between decompression of the generated memory containing the correct guess and the incorrect ones. Comprezzor uses the timing difference between the correct guess and the second-fastest guess as the fitness score for a candidate. The fitness function is evaluated using a helper program performing an attack on the same setup

as in Section 9.4.1. The program performs 100 iterations per guess and reports the minimum decompression time per guess to reduce the impact of noise. This minimum decompression time is the output of the fitness function for Comprezzor.

Input Mutation. Comprezzor is able to amplify timing differences thanks to its set of mutations over the samples space specifically designed for data compression algorithms. Data compression algorithms leverage input patterns and entropy to shrink the input into a compressed form. For performance reasons, their ability to search for patterns in the input is limited by different internal parameters, like lookback windows, look-ahead buffers, and history table sizes [14, 63]. We designed the mutations that affect the sample generation process to focus on input characteristics that directly impact compression algorithm strategies and limitations towards corner cases.

Comprezzor mutations randomize the entropy and size of the samples that are generated. This has an effect on the overall compressibility of sequences and literals in the sample [14]. Moreover, the mutator varies the number of repeated guesses and their position in the resulting sample, stressing the capability of the compression algorithm to find redundant sequences over different parts of the input. This affects the sequence compression and triggers corner cases, e.g., subsequent blocks to be compressed are directly marked as incompressible (cf. Section 9.5.2). All these factors contribute to Comprezzor's ability to amplify timing differences.

Input Evolution. Comprezzor follows an evolutionary approach to generate inputs that maximize timing differences. It generates and mutates candidate layout configurations for the attack. Each configuration is forwarded to the helper program that builds the requested layout, inserts the candidate guess, compresses the memory, and returns the decompression time.

Comprezzor iterates through different generations, with each sample having a probability of survival to the new generation that depends on its fitness score. The fitness score is the time difference between the correct guess and the nearest incorrect one. Comprezzor discards all the samples where the correct guess is not the fastest or slowest. A retention factor decides the percentage of samples selected to survive among the best ones in the old generation (5 % by default). The population for each new generation is initialized with the samples that survived the selection process and enhanced by random mutations of such samples. By default, 70 % of

the new population is generated by mutating the best samples from the previous generation. To avoid locally optimal solutions, a percentage of completely random new samples is injected in each new generation. Comprezzor runs until the maximum number of generations is evaluated, and returns the best candidate layouts.

9.5.2. Results: Fuzzing Compression Algorithms

Evaluation. Our test system has an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) with a fixed frequency of 4 GHz. We run Comprezzor on four compression algorithms: zlib (1.2.11), Facebook's Zstd (1.5.0), LZ4 (v1.9.3), and PGLZ in PostgreSQL (v12.7). Comprezzor can support new algorithms by just adding compression and decompression functions.

We run Comprezzor with 50 epochs and a population of 1000 samples per epoch. We set the retention factor to 5 %, selecting the best 50 samples in each generation of 1000 samples. We randomly mutate the selected samples to generate 70 % of the children and add 25 % of randomly generated layouts to the new generation. The overall runtime of Comprezzor was 2.46 h for zlib, 1.73 h for zstd, 1.64 h for LZ4, and 2.09 h for PGLZ. Table 9.3 (Section A) lists the maximum timing differences found for the four compression algorithms. Particularly, for zlib and PGLZ, the fuzzer discovers cases with timing differences of multiple microseconds between correct and incorrect guesses. Since zlib is a popular algorithms, we analyze it in more detail.

Zlib. Comprezzor discovers a corner case in zlib where all incorrect guesses lead to a slow code path, and the correct guess leads to a significantly faster execution time. Using Comprezzor with a known prefix, we observe a high timing difference of 71 514.75 ns, which is **3** orders of magnitude larger than the manually-discovered latency difference (cf. Section 9.4.3). This memory layout also leads to similarly high timing differences across all compression levels of zlib. To rule out microarchitectural effects, we confirm the experiment on different systems with an Intel i5-8250U, AMD Ryzen Threadripper 1920X, and Intel Xeon Silver 4208.

On further analysis, we observe that the corner case identified by the fuzzer is due to incompressible data. The initial data in the page, from a uniform distribution, is primarily incompressible. For such incompressible blocks, DEFLATE can store them as raw data blocks, called *stored blocks* [18]. Such blocks have fast decompression times as only a single

`memcpy` operation is needed on decompression instead of the actual DEFLATE process. In this particular corner case, the correct guess results in such an incompressible *stored block* which is faster, while an incorrect guess results in a partly-compressible input which is slower.

Correct Guess. In the case where the guess matches the secret, the entire guess string, *i.e.*, `cookie=SECRET`, is compressed with the secret string. All subsequent data in the input is incompressible and treated as a stored block and decompressed with a single `memcpy` operation, which is significantly faster than Huffman and LZ77 decoding.

Incorrect Guess. In the compression case where the guess does not match the secret, only the prefix of the guess, *i.e.*, `cookie=`, is compressed with the prefix of the secret, while another longer sequence, *i.e.*, `cookie=FOOBAR` leads to forming a new block. Therefore, when decompressing, this block must now undergo the Huffman decoding (and LZ77), which results in several table lookups, memory accesses, and higher latency. Thus, the timing differences for the correct and incorrect guesses are amplified by the layout that Comprezzor discovered. We provide more details about this layout in Figure 9.10 in the Section D and also provide listings of the debug trace from zlib for the decompression with the correct and incorrect guesses, to illustrate the root-cause of the amplified timing differences with this layout.

Larger Secret Sizes. Evaluating a larger secret size (1 kB random string) with Comprezzor on zlib, results in similar high timing differences in the range of tens of microseconds for the correct guess using a byte-by-byte attack.

Takeaway We showed that it is possible to amplify timing differences for decompression timing attacks (answers **Q1**). With Comprezzor, we presented an approach to automatically find high timing differences in compression algorithms.

9.6. Case Studies

In this section, we present case studies showing the security impact of the timing side channel. We present a local covert channel that allows hidden communication by leveraging the high-latency scenarios found by Comprezzor. Furthermore, we present a remote covert channel that exploits the decompression of memory objects in Memcached. We demonstrate

Decomp+Time on a PHP application that compresses secret data together with attacker data to leak the secret bytewise. We leak inaccessible values from a database, exploiting the internal compression of PostgreSQL. We show that OS-based memory compression (ZRAM) also has timing side channels that can leak secrets. In these case studies we do not artificially restrict the possible layouts of the pages, since these are possible ways in which those systems may be used, as confirmed by their developers during responsible disclosure. In our fourth study on Google Chrome, we show how to create co-location between attacker-controlled data and internal heap pointers and exploit ZRAM compression to leak the internal pointer.

9.6.1. Covert channel

To evaluate the transmission capacity of memory-compression attacks, we evaluate the transmission rate for a covert channel, where the attacker controls the sending and receiving end. Similar to previous works [30–32, 43, 85], we evaluate a cross-core covert channel using shared memory. The maximum capacity poses a leakage rate limit for our other attacks. Our local covert channel achieves a capacity of 9.73 kB/s (n = 100, $\sigma_{\bar{\mu}}$ = 0.00097%).

Setup. We create a simple key-value store that communicates via UNIX sockets. The store takes input from a client and stores it on a 4 kB-aligned page. The sender inserts a key and value into the first page to communicate with the server. The receiver inserts a small key and value as well, which should be placed on the same 4 kB page. If the 4 kB-page is fully written, the key-value store compresses the whole page. Compressing full 4 kB-page separately also occurs on filesystems like BTRFS [12].

Sender and receiver agree on a time frame to send and read content. The basic idea is to communicate via the observation on zlib that memory with low entropy, e.g., 4096 times the same value, requires more time when decompressing compared to pages with a higher entropy, e.g., repeating sequence number from 0 to 255. Note that the content of the page controlled by the receiver is co-located to the senders controlled part. Figure 9.5 shows the histogram of latency when decompression is triggered for both cases for the key-value on an Intel i7-6700K running at 4 GHz. On average, we observe a timing difference of 3566.22 ns (14 264.88 cycles, n = 100000).

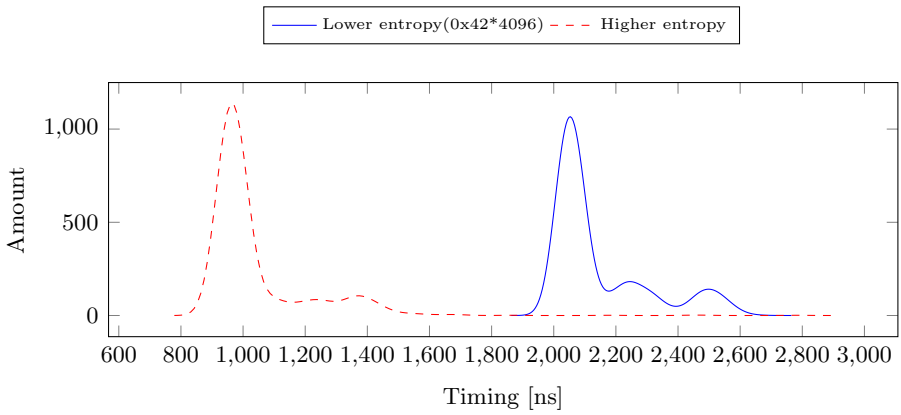


Figure 9.5.: Timing when decompressing a zlib-compressed 4 kB page with high entropy compared to a page with low entropy.

Transmission. We evaluate our cross-core covert channel by generating and sending random content from `/dev/urandom` through the memory compression timing side channel. The sender controls 4095 B of a 4 kB page. The sender transmits a '1'-bit by performing a store with high-entropy data. Conversely, to transmit a '0'-bit, the sender stores a low-entropy data. To trigger the compression, the receiver also stores data in the store which fills a full 4 kB page. The key-value store compresses the entire 4 kB page, as it is fully used. The receiver performs a fetch request from the key-value store, which triggers a decompression of the full 4 kB page. To distinguish bits, the receiver measures the mean RTT of the fetch request.

Evaluation. Our test machine uses an Intel Core i7-6700K (Ubuntu 20.04, kernel 5.4.0), and all cores are running on a fixed CPU frequency of 4 GHz. We repeat the transmission 50 times and send 640 B per run. To reduce the error rate, the receiver fetches the receiver-controlled data 50 times and compares the average response time against the threshold. Our cross-core covert channel achieves an average transmission rate of 9.73 kB/s (n = 100, $\sigma_{\bar{\mu}}$ = 0.0068%) with an error rate of 0.082 % (n = 100, $\sigma_{\bar{\mu}}$ = 0.023%). The capacity of the unoptimized covert channel is in line with other state-of-the-art microarchitectural cross-core covert channels that do not rely on shared memory [19, 45, 48, 61, 64, 72, 81, 83].

9.6.2. Remote Covert Channel

We extend the scope of our covert channel to a remote covert channel. In the remote scenario, we rely on Memcached on a web server for memory compression and decompression.

Memcached. Memcached is a widely used caching system for web sites [49]. Memcached is a simple key-value store that internally uses a slab allocator. A slab is a fixed unit of contiguous physical memory, which is assigned to a certain slab class which is typically a 1 MB region [36]. PHP offers the possibility to use Memcached for caching, and memory compression is enabled by default if Memcached is used [62]. PHP-Memcached has a threshold that decides at which size data is compressed, with the default value being 2000 B. Furthermore, PHP-Memcached compares the compression ratio to a compression factor and decides whether it stores the data compressed or uncompressed in Memcached. The default value for the compression factor is 1.3, *i.e.*, 23 % of the space needs to be saved from the original data size to store it compressed [62].

Bypassing the Compression Factor. While the compression factor already introduces a timing side channel, we focus on scenarios where data is always compressed. This is used in Section 9.6.3 useful for leaking co-located data. Intuitively, it should suffice to prepend highly-compressible data to enforce compression. However, we found that only prepending and adopting the offsets for secret repetitions, as for zlib, also influenced the corner case we found and the large timing difference. We integrate prepending of compressible pages to Comprezzor and also add the compression factor constraint to automatically discover inputs that fulfills the constraint and leads to large latencies between a correct and incorrect guess.

Transmission. We use the page found by Comprezzor that triggers a significantly lower decompression time to encode a '1'-bit. For a '0'-bit, we choose content that triggers a significantly higher decompression time. The sender places a key-value pair for each bit index at once into PHP-Memcached. The receiver sends GET requests to the resource, causing decompression of the data containing the sender content. The timing difference of the decompression is reflected in the RTT of the HTTP request. Hence, we measure the timing difference between the sent HTTP request and the first received response.

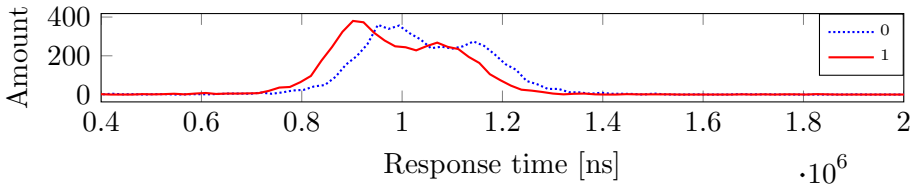


Figure 9.6.: Distribution of HTTP response times for zlib-decompressed pages stored in Memcached on memory compression encoding a '1' and a '0'-bit.

Evaluation. Our sender and receiver use an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0) and connect to the internet with a 10 Gbit/s connection. For the web server, we use a dedicated server in the compression:Equinix[1] cloud, 14 hops away from our network (over 700 miles physical distance) with a 10 Gbit/s connection. The victim server uses an Intel Xeon E3-1240 v5 (Ubuntu 20.04, kernel 5.4.0). Our server runs Nginx 1.18.0, with a PHP (version 7.4, FPM enabled) website that allows storing and retrieving data, backed by Memcached 1.5.22, the default version on Ubuntu 20.04. We perform a simple test where we perform 5000 HTTP requests to a PHP site that stores zlib-compressed memory in Memcached. Figure 9.6 illustrates the timing difference between a '0'-bit and a '1'-bit. The timing difference between the mean values for a '0'- and '1'-bit is 61 622.042 ns. We transmit a series of random messages of 8 B over the internet. Our simple remote covert channel achieves an average transmission rate of 10.72 bit/min (n = 20, $\sigma_{\bar{\mu}}$ = 15.96%) at an average error rate of 0.93 %. We achieve a similar transmission rate as Schwarzl et al. [71] with remote memory-deduplication attacks. Our covert channel outperforms the one by Schwarz et al. [70] and Gruss et al. [29], even though our attack works with HTTP instead of the more lightweight UDP sockets. Other remote timing attacks usually do not evaluate their capacity with a remote covert channel [1, 4, 37, 68, 76, 88]. Note that our numbers to mount a successful covert channel over such a distance is way below the numbers reported by Van Goethem et al. [76, Table 1].

9.6.3. Remote Attack on PHP-Memcached

Using our building blocks to perform Decomp+Time and the remote covert channel, we perform a remote attack on PHP-Memcached to leak

[1] https://equinix.com

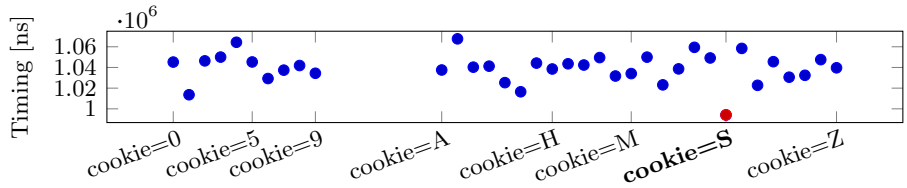


Figure 9.7.: Distribution of the response times for the correct byte guess (S) and the incorrect guesses (0-9, A-R, T-Z) leaking the secret from PHP-Memcached. Subsequent bytes are similar (cf. Section E). Standard error margins are below 1 % of the value for all guesses and, thus, not visible in the plot.

secret data from a server over the internet. We assume a memory layout where secret memory is co-located to attacker-controlled memory, and the overall memory region is compressed. As mentioned in Figure 9.1, we assume that the attacks can arbitrarily co-locate arbitrary data to secret data. This is reasonable as the developer might store additional metadata, e.g., API keys co-located to the attacker-controlled data, or might make some modifications. Also, structured document data might be cached within the same memory slab as the API of Memcached does not prevent an user from merging data of multiple users together. The compressed data is never visible to the user, and the compression ratio is not exposed, relying on the compression ratio is not possible.

Attack Setup. We use the same setup as in Section 9.6.2 and run the attack using the same server setup as used for the remote covert channel. We define a 6 B long secret (`SECRET`) with a 7 B long prefix (`cookie=`) and prepend it to the stored data of users. PHP-Memcached compresses the data before storing it in Memcached and decompress it when accessing it again. For each guess, the PHP application stores the uploaded data to a certain location in Memcached. On each data fetch, the PHP application decompresses the secret data together with the co-located attacker-controlled data and then responds only the attacker-controlled data. The attacker measures the RTTs and discerns the timing differences between the guesses.

Evaluation. For the byte-wise attack, we assume each byte of the secret is uppercase alphanumeric (36 different options). For each of the bytes to be leaked, we generate separate memory layouts using Comprezzor that maximize the latency between guesses. We repeat the experiment 20 times. On average, our attack leaks the entire secret string in 31.95 min

($n = 20$, $\sigma_{\bar{\mu}} = 60.58\%$) *i.e.*, 5.32 minutes per byte or 1.5 bit/min. Since the latencies between a correct and incorrect guess are in the microseconds range, we do not observe false positives with our approach. Figure 9.7 shows the median response time for each guess in the first iteration as a representative example. It can be seen that the response time for the correct guess is significantly faster than the incorrect guesses.

Takeaway: We show that a PHP application using Memcached to cache blobs hosted on Nginx enables covert communication with a transmission rate of 10.72 bit/min (answers **Q2**). Moreover, we demonstrate a remote memory-compression attack on Memcached leaking 1.5 bit/min.

9.6.4. Leaking Data from Compressed Databases

In this section, we show that an attacker can exploit compression in databases to leak inaccessible information from the internal database compression of PostgreSQL. In this setting, the compression ratio is not visible to the attacker, only the timing can be observed. A potential attack scenario for structured text in a cell is where JSON documents are stored and compressed within a single cell, and the attacker controls a specific field within the document. While our focus is restricted to cell-level compression, compressed columnar storage [5, 15, 50, 57] or columnar databases, may also be vulnerable to decompression timing attacks. The attacker controls data in the same cell or the content of a cell in the same column as the target data.

PostgreSQL Data Compression. PostgreSQL is a widespread open-source relational database system using the SQL standard. PostgreSQL maintains tuples saved on disk using a fixed page size of commonly 8 kB, storing larger fields compressed and possibly split into multiple pages. By default, variable-length fields that may produce large values, e.g., `TEXT` fields, are stored compressed. PostgreSQL's transparent compression is known as `TOAST` (The Oversized-Attribute Storage Technique) and uses a fast LZ-family compression, `PGLZ` [63]. Data in a cell is stored compressed if such a form saves at least 25 % of the uncompressed size to avoid wasting decompression time. Data stored uncompressed is accessed faster than data stored compressed as the decompression algorithm is not executed.

Attack Setup. To assess the feasibility of an attack, we use a local database server with the database stored on an SSD and access two differently compressed rows with a Python wrapper using the `psycopg2`

library. The first row contains 8192 characters of highly compressible data, while the second one 8192 characters of random incompressible data. Both rows are stored in a table as `TEXT` data and accessed 1000 times. The median for the number of clock cycles required to access the compressible row is 249 031, while for the uncompressed one is 221 000, which makes the two accesses distinguishable. On our 4 GHz CPUs, this is a timing difference of 7007.75 ns. We use Comprezzor to amplify these timing differences and demonstrate bytewise leakage.

Leaking First Byte. For the bytewise leakage of the secret, we first create a memory layout to leak the first byte using Comprezzor against a standalone version of PostgreSQL's compression library, using a similar setup as the previous Memcached attack. A key difference in the use of Comprezzor with PostgreSQL is that the helper program measuring the decompression time returns a time of 0 when the input is not compressed, *i.e.*, the data compressed with PGLZ does not save at least 25 % of the original size. Comprezzor found a layout that sits exactly at the corner case where a correct guess in the secret results in a compressed size that saves 25 % of the original size. Hence, a correct guess is saved compressed, while for any wrong guess, the data is saved uncompressed.

Leaking Subsequent Bytes with Secret Shifting. We observed that one good layout can be reused for bytewise leakage in PGLZ. The prefix can be shifted by one character to the left by a single character, *i.e.*, from "cookie=S" to "ookie=SE", to accommodate an additional byte for the guess. Shifting allows bytewise leakage with the same memory layout. Note that we could not mount this shifting approach on DEFLATE.

Evaluation. We perform a remote decompression timing attack against a Flask [22] web server that uses a PostgreSQL database to store user-provided data. We used the same compression:Equinix cloud-server setup as used for the Memcached remote attack (cf. Section 9.6.3). The server runs Python 3.8.5 with Flask 2.0.1 and PostgreSQL 12.7. We use a similar setup as in Section 9.6.3 with the difference that attacker-controlled data is co-located to a secret in a database cell. The secret is never shown to the user. Using the layout found by Comprezzor, the entry in the database is stored compressed only when the secret matches the provided data. A second endpoint in the server accesses the database to read the data without returning the secret to the attacker. The attack leaks bytewise over the internet by guessing again uppercase alphanumeric characters (36 possibilities per character), including the correct one. We repeat the attack 20 times. The average time for the attack, *i.e.*, the

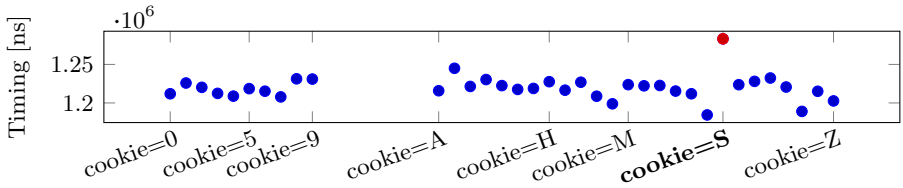


Figure 9.8.: Distribution of response times for the bytewise leakage in the remote PostgreSQL attack, with the correct guess (S) and incorrect guesses (0-9, A-R, T-Z). This is similar for subsequent bytes leaked (cf. Section E). Standard error margins are below 1 % for all guesses and, thus, not visible in the plot.

time required to determine the guess with the highest latency that the server had to decompress before returning, is 17.84 min (2.97 min/B) (n = 20, $\sigma_{\bar{\mu}}$ = 0.33%) *i.e.*, 2.69 bit/min. Figure 9.8 illustrates the median response times showing how the correct guess results in a slower response. Without fixing the CPU frequency, twice as many requests are required to clearly determine the correct secret. However, keeping the server busy automatically leads to an almost constant frequency. We guess over the set of all printable characters and observe one secret byte in 7.83 min.

Takeaway: Secrets can be leaked from databases due to timing differences caused by PostgreSQL's transparent compression, if applications store untrusted data with secrets in the same cell. Our decompression timing attack on PostgreSQL leaks a byte across the internet with 2.69 bit/min.

9.6.5. Attacking OS Memory Compression

In this section, we show how memory compression in modern OSs can introduce exploitable timing differences. We demonstrate bytewise leakage of secrets from compressed pages in ZRAM, the Linux implementation of memory compression. Co-location can be achieved here if virtual memory is compressed together with mostly attacker-controlled data. As we show in Section 9.6.6, co-location can occur for internal pointers in the V8 engine, together with attacker-controlled data. The compression ratio is not observable for the attacker since the attacker cannot read the compressed memory from ZRAM.

Background. Memory compression is a technique used in many modern OSs, e.g., Linux [42], Windows [35], or MacOS [17]. Similar to traditional swapping, memory compression increases the effective memory capacity of a system. When processes require more memory than available, the OS can transparently compress unused pages in DRAM to ensure they occupy a smaller footprint in DRAM rather than swapping them to disk. This frees up memory while still allowing the compressed pages to be accessed from DRAM. Compared to disk I/O, DRAM access is an order of magnitude faster, and even with the additional decompression overhead, memory compression is significantly faster than swapping. Hence, memory compression can improve the performance despite the additional CPU cycles required for compression and decompression. The Linux kernel implements ZRAM [33], enabled by default on Fedora [42] and Chrome OS [17].

Characterizing Timing Differences in ZRAM

To understand how memory compression can be exploited, we characterize its behavior in ZRAM. On Linux systems, ZRAM appears as a DRAM-backed block device. When pages need to be swapped to free up memory, they are instead compressed and moved to ZRAM. Subsequent accesses to data in ZRAM result in a page fault, and the page is decompressed from ZRAM and copied to a regular DRAM page for use again. We show that the time to access data from a ZRAM page depends on its compressibility and thus the data values. According to the previous experiments, we characterize the latency of accessing data from ZRAM pages with different entropy levels: pages that are *incompressible* (with random bytes), *partially-compressible* (random values for 2048 bytes and a fixed value repeated for the remaining 2048 bytes), and *fully-compressible* (a fixed value in each of the 4096 bytes). We ensure a page is moved to ZRAM by accessing more memory than the memory limit allows. To ensure fast run times for the proof of concept, we allocate the process to a *cgroup* with a memory limit of a few megabytes. We measure the latency for accessing a 8-byte word from the page in ZRAM, and repeat this process 500 times. Table 9.2 shows the mean latency of ZRAM accesses for different ZRAM compression algorithms on an Intel i7-6700K (Ubuntu 20.04, kernel 5.4.0). The latency for accesses to ZRAM is much higher for partially-compressible pages (with lower entropy) compared to incompressible pages (with higher entropy) for all compression algorithms.

Table 9.2.: Mean latency of accesses to ZRAM. Distinguishable timing differences exist based on data compressibility in the pages (n = 500 and 6 %of samples removed as outliers with more than an order of magnitude higher latency).

Algorithm	Incompressible (ns)	Partly Compressible (ns)	Fully Compressible (ns)
deflate	1763 (±12%)	12 208 (±2%)	1551 (±12%)
842	1789 (±11%)	8785 (±2%)	1556 (±10%)
lzo	1684 (± 9%)	4866 (±4%)	1479 (±12%)
lzo-rle	1647 (± 9%)	4751 (±4%)	1453 (±12%)
zstd	1857 (±10%)	2612 (±9%)	1674 (±11%)
lz4	1710 (±11%)	1990 (±7%)	1470 (±10%)
lz4hc	1746 (± 9%)	2091 (±9%)	1504 (±11%)

This is because the process of moving compressed ZRAM pages to regular memory on an access requires additional calls to functions that decompress the page. ZRAM pages that are stored uncompressed do not require these function calls (cf. Section C). We observe the largest timing difference for the *deflate* algorithm (close to 10 000 ns) and `842` algorithm (close to 7000 ns); we observe moderate timing differences for `lzo` and `lzo-rle` (close to 1000 ns), and `zstd` (close to 750 ns); the smallest timing difference are for `lz4` and `lz4hc` (close to 250 ns). These timing differences largely correspond with the algorithm's raw decompression latency (cf. Table 9.1). Accesses to a fully-compressible page in ZRAM, *i.e.*, a page containing the same byte repeatedly, are faster (by 200 ns) than accesses to an incompressible page for all the compression algorithms. This is because ZRAM stores such pages with a special encoding as a single-byte (independent of the compression algorithm) that only requires reading a single byte from ZRAM on an access to such a page.

Leaking Secrets via ZRAM Decompression Timings

In this section, we exploit timing differences between accesses to a partially-compressible and an incompressible page in ZRAM (using *deflate* algorithm) and use Comprezzor to leak secrets byte-by-byte.

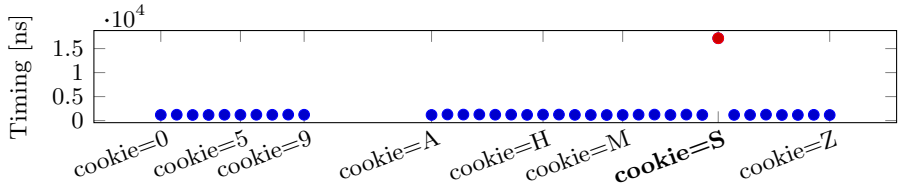


Figure 9.9.: Times for guesses (0-9, A-Z) for the first byte (S) of the secret leaked byte-wise from ZRAM. The highest times correspond to the secret-byte value (shown in red). Standard error margins are below 1 % of the value for all guesses and, thus, not visible in the plot.

Attack Setup. We demonstrate byte-wise leakage attack on a program with a 4 kB page stored in ZRAM containing both a secret value and attacker-controlled data, as is common in many applications like databases. To determine optimal data layouts an attacker might use, we combine this program with Comprezzor. With a known secret value, Comprezzor runs the program with the attacker guessing each byte position successively. For each byte position, Comprezzor generates the optimal memory layouts. In such an optimal layout, when the attacker's guess matches the secret-byte, the page entropy reduces (page is partially compressible), and ZRAM decompression takes longer; and for all other guesses, the entropy is high, and ZRAM decompression is fast. We repeat this process to generate optimal data layouts for each byte position. Note that this optimal data layout only relies on the number of repetitions of the guess and the relative position of the guessed data and the secret (a property of the compression algorithm), and is applicable with any data values. Using these attacker data layouts, we perform byte-wise leakage of an unknown secret. At each step, the attacker guesses one byte (0-9, A-Z) and denotes the guess with the highest latency as correct. We repeat this attack for 100 random secrets.

Evaluation. Figure 9.9 shows the bytewise leakage for a secret value (`cookie=SECRET`), with the decompression times for guesses of the first four bytes depicted in each of the graphs. For each byte, among guesses of (0-9, A-Z), the highest decompression time successfully leaks the secret byte value (shown in red). For example, for byte 0, the highest time is for `cookie=S`. Similar trends are observed for the remaining bytes, as shown in the Figure 9.14 in Section E for byte 1, the highest latency is observed for `cookie=SE`. For byte 2, we observe a false positive, `cookie=SE8`, which also has a high latency, along with the correct guess `cookie=SEC`. But in

the subsequent byte 3, when both these strings are used as prefixes for the guesses, the false positives are eliminated, and `cookie=SECR` is obtained as the correct guess. The next two bytes are also successfully leaked to fully obtain `cookie=SECRET`, as shown in the Figure 9.14 in Section E. Repeating the experiment with 100 randomly generated secrets, we observe that in 90 out of 100 cases, the secret is leaked successfully. Our attack successfully completes in 58.6 s ($n = 90, \sigma_{\bar{\mu}} = 642.22\%$) on average, *i.e.*, 49.14 bit/min. In 9 out of the 10 remaining cases, we narrow down the secret to within four candidates (due to false positives for the last-byte guess), and in the last case, we recover 4 bytes out of the 6-byte secret (the false positives grow for the 5th byte and beyond in this case). The false positives in our ZRAM PoC are caused by the Comprezzor-generated data layouts that are not as robust as in previous PoCs. Comprezzor with ZRAM is a few orders of magnitude slower (almost 0.03x the speed) compared to iterations with raw algorithms studied in Section 9.5.2. Moving a page to ZRAM and compressing it requires accessing sufficient memory to swap the page out, which is much slower than executing just the compression algorithm. Consequently, the explored search space is smaller. Such false positives can be addressed by using multiple strong data layouts, or by fuzzing for a longer duration to generate more robust data layouts.

9.6.6. Leaking Heap Pointers from Google Chrome

For exploits in Javascript environments like V8 in Chrome, breaking memory randomization is often the first step. Such ASLR breaks usually rely on information leaks to disclose pointers from V8 isolates. However, vulnerabilities like out-of-bound reads that allow information leaks are promptly patched when discovered. In this section, we use our side channel to disclose a heap pointer and thus break heap-memory randomization. We assume a Chrome browser running on a device with ZRAM enabled. We run our experiments on a notebook equipped with an Intel i5-8250U CPU and 16 GB DDR4 RAM running Ubuntu 20.04 (kernel 5.4.0-124-lowlatency) and Google Chrome 90.0.4430.72. We setup a 4 GB ZRAM device as swap partition with the deflate algorithm.

Co-Location. The first major requirement to leak a pointer is co-location between attacker-controlled data and a heap pointer (secret) on a 4 kB page. In JavaScript, elements of `TypedArrays` (e.g., `Uint8Array`) are memory-backed by an `ArrayBuffer` object. A backing heap pointer points to the location where the `ArrayBuffer` stores the data in memory [11]. All

such 64-bit values in JavaScript (including pointers) are encoded using the IEE754 floating-point representation. Hence, to store a pointer in memory, non-typed arrays of numbers encoding 64-bit pointers are used. Thus attacker-controlled numbers and secret heap-pointers (to `TypedArray`) can be co-located in a non-`TypedArray`, leading to the desired co-location of attacker data and the target pointer. We massage the allocations such that the target pointer is at the beginning of a 4 kB page. Listing 9.8.2 illustrates a snippet that co-locates the backing pointer of a `TypedArray` with a mostly attacker-controlled 4 kB region. Listing 9.8.3 is a memory dump showing the resultant co-location within the memory of a Chrome process. The resultant layout is indeed dependant on Chrome's allocator and may not always be the same. So, we measure which offsets we get with repeated runs of the same code snippet. For the same pointer, we observe that offsets of 0x0 or 0xc0 within the page occur in 84 % of the allocations (cf. Section F). Using Comprezzor, we can generate memory layouts for different byte offsets of the pointers with our setup. Note that the 32-bit compressed-pointers V8 uses to refer to objects in the same isolate are not randomized for each execution and do not affect our attack.

Timer. Similar to previous work [28, 41, 69, 79], we use a WebWorker counting thread via a `SharedArrayBuffer`. As the server is attacker-controlled, re-enabling `SharedArrayBuffer` is possible via HTTP headers [53, 80]. This timer is sufficient to measure timing differences between correct and incorrect byte guesses.

Trigger Swapping from JavaScript. The default `swapiness` value for Ubuntu 20.04 is 60. This means that if 80 % of memory is used, the kernel starts to perform swapping.Therefore, to swap the target data from RAM to the ZRAM swap device area, the attacker has to create high memory pressure. As the heap size per process is limited to 4 GB, this can be challenging. One approach an attacker can adopt is to spawn multiple processes to achieve the desired memory pressure. We observe that every iframe gets a separate renderer process, therefore, a higher memory pressure can be achieved. However, iframes from same domains (including subdomains) might get merged back into a similar process [2]. This could lead to the main tab crashing as the memory limit per tab is exceeded. Therefore, the attacker requires to use multiple iframes embedding content from different web servers to trigger swapping reliably. As `Cross-Origin-Embedder-Policy` is set to `cross-domain`, the server under the attackers control requires to set the `Cross-Origin-Resource-Policy` to `cross-origin` [80]. Each of the domains can allocate about 4 GB. Therefore,

to trigger swapping frequently, 4 additional remote servers from different domains are required. To ensure that the target is always evicted, we delete the iframes and repeat the allocation a second time. Note that this is the worst case scenario, assuming an idle system. Other system activity can only increase the probability of swapping out target pages. We run an experiment which constantly loads an iframe and tries to evict a certain target page. To successfully evict the target page from memory, the attacker requires, on average, 14.91 s (n = 100, $\sigma_{\bar{x}}$ = 7.59%).

Total Attack Runtime. The attacker has to guess all 256 possibilities per byte to leak the correct pointer. We use Comprezzor to generate layouts for the 6 byte offsets. To get stable results, 20 measurements per guess are required. Therefore, leaking a single byte requires about 20 s for the swapping part in JavaScript and about one second for evaluating the memory layout, leading to 21 s per guess. This leads to a total runtime of 29.8 h (21 s * 20 (tries per guess) * 256 = 107520/3600 = 29.8h) per byte. An attacker can invest additional engineering effort to perform multiple guesses in one iteration. Then, the theoretical runtime is only 7 min (21 s * 20 / 60 = 7m) per byte.

Takeaway: We show that even if an application does not explicitly use compression, its data may still get compressed by the OS due to memory compression. We demonstrate a local attack leaking 49.14 bit/min and port the attack to JavaScript leaking heap pointers in Google Chrome.

9.7. Mitigations

Taint tracking. The best strategy to mitigate compression side channels is to avoid sensitive data being with potential attacker-controlled data. Mutexion [52] enforces mutually exclusive compression between attacker-controlled data and secret data in HTTP. This approach uses automated annotations of secret and attacker-controlled data. However, finding all the sources and sinks can be a complex problem for software developers, especially in large and complex software projects. Taint tracking tries to trace the data flow and mark input sources and their sinks. Paulsen et al. [58] use taint analysis to track the flow of secret data before feeding data into the compression algorithm. Their tool, Debreach, is about 2-5 times faster than SafeDeflate [58]. However, it is only compatible with PHP, and the developer needs to flag the sensitive input which is being tracked.

Disabling LZ77. A naive solution is to disable compression or at least disable the LZ77 part. Karakostas et al. [38] showed for web pages that, this adds an overhead between 91 % and 500 %. Furthermore, attacks on symbol compression have not been studied well enough to provide security guarantees.

Mitigating the Timing Side Channel. Constant time implementations might remove the timing side channel [3, 47, 60]. There are several solutions to automatically transform code into a constant-time version to mitigate side channels [10, 13, 44, 74, 82]. However, such solutions usually rely on code linearization, executing both the taken and not-taken path of branches. While this is feasible for cryptographic implementations with a limited number of branches, compression algorithms have too many input-dependent branches. Moreover, the overhead of decompression might get considerably worse without the optimization of copying incompressible data during decompression. Thus, the mitigation can perform worse than fully disabling sequence compression.

Masking. Karakostas et al. [38] presented a generic defense technique called Context Transformation Extension (CTX). The general idea is to use context-hiding to protect secrets from being compressed with attacker-controlled data. Data is permuted on the server side using a mask, and on the client side, an inverse permutation is performed (JavaScript library). The overhead compared to the original algorithms decrease with the number of compressed data [38].

Duplicating secrets. As Decomp+Time uses the generated layouts by Comprezzor, the guess is placed multiple times to trigger edge cases. Placing the secret multiple times might already be effective enough to mitigate Decomp+Time. We leave it as future work to evaluate the effectiveness.

Randomization. Yang et al. [84] showed an approach with randomized input to mitigate compression side-channel attacks. The service would require adding an additional amount of random data to hide the size of the compressed memory. However, as the authors also show, randomization-based approaches can be defeated at the expense of a higher execution time. Also, Karaskostas et al. [38] showed that size randomization is ineffective against memory compression attacks. It is also unclear if size randomization mitigates the timing-based side channel of the memory decompression.

Keyword protection. Zieliński demonstrated an implementation of DEFLATE called SafeDeflate [89]. SafeDeflate mitigates memory compression attacks by splitting the set of keywords into subsets of sensitive and non-sensitive keywords. Depending on the completeness of the sensitive keyword list, this approach can be considered as secure. As Paulsen et al. [58] mention, it is easy to overlook a corner case. Furthermore, this approach leads to a loss of compression ratio of about 200 % to 400 % [38].

The aforementioned mitigations focus on mitigating compression-ratio side channels. As the compression and decompression timings are not constant, a timing side channel is harder to mitigate. Since the latency for a correct guess is in the region of microseconds, not many requests (≤ 200) are required per guess to distinguish the latency. Therefore, in a remote setting, a simple DDoS detection might detect an attack but only after a certain amount of data being leaked.

9.8. Conclusion

In this paper, we presented Decomp+Time, a timing side-channel attack on several memory-compression algorithms. We developed Comprezzor, an evolutionary fuzzer to amplify timing latencies when performing attacks on different compression algorithms. Our remote covert channel achieves a transmission rate of 9.73 kB/s locally and 10.72 bit/min over the internet (14 hops, > 700 miles). We showed bytewise leakage with a leakage rate of 1.50 bit/min across the internet from a server using Memcached hosting, a PHP application. We leaked database records from PostgreSQL with 2.69 bit/min. We showed that we can locally attack ZRAM on Linux and leak heap pointers from Google Chrome. Our results show that compression of sensitive data can be dangerous even if the compressed data is not directly observable.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback and comments on the paper. Furthermore, we want to thank Moritz Lipp, Jonas Juffinger and Claudio Canella for feedback on this work. This work was supported by generous funding and gifts from Red Hat. Any opinions

or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding parties.

Appendix

A. Comprezzor-discovered Timing Differences

Table 9.3 shows the timing differences discovered by Comprezzor for different compression algorithms. These results are a result of running Comprezzor with 50 epochs and a population of 1000 samples per epoch. We set the retention factor to 5 %, selecting the best 50 samples in each generation of 1000 samples. We randomly mutate the selected samples to generate 70 % of the children and add 25 % of completely randomly generated layouts in the new generation.

B. Timing Difference for Compression

Table 9.4 shows the timing differences when compressing incompressible, partially incompressible, and fully-compressible memory. The execution time and, thus, latency depends on the level of compressibility. For compressible memory, the timing is lower, which may appear counter-intuitive, but it means higher redundancy in the data and, thus, e.g., smaller Huffman trees and fewer sequences. For incompressible memory, the opposite case occurs, with more sequences and a larger tree, consuming more computation time. Additionally, when the compression ratio for a block is then too low, the compression is discarded, and additional `memcpy` operations are performed instead. All of this consumes computation time, leading to the timing differences we see in Table 9.4. For the intermediate case of partially incompressible data, both cases occur for part of the blocks leading to an intermediate timing. However, observing the compression time can be difficult, as no request or operation latency observable by the attacker depends on the compression time. Hence, we we focused on attacks exploiting the decompression timing side channel.

Table 9.3.: Timing differences between correct and incorrect guesses found by Comprezzor and the corresponding runtime.

Algorithm	Max difference for correct guess (ns)	Runtime (h)
PGLZ	109233.25	2.09
zlib	71514.75	2.46
zstd	4239.25	1.73
LZ4	2530.50	1.64

Table 9.4.: Different compression algorithms yield distinguishable timing differences when compressing content with a different entropy. (n = 100000

Algorithm	Fully Compressible (ns)	Partially Compressible (ns)	Incompressible (ns)
FastLZ	38 619.07 (±0.74%)	58 887.40 (±0.50%)	79 384.89 (±0.40%)
LZ4	44 748.02 (±0.15%)	47 731.08 (±0.16%)	47 316.56 (±0.16%)
LZO	5645.86 (±2.18%)	5915.28 (±2.78%)	7928.21 (±3.91%)
PGLZ	44 275.84 (±0.13%)	65 752.55 (±0.12%)	-
zlib	38 479.53 (±0.22%)	80 284.72 (±0.23%)	76 973.82 (±0.20%)
zstd	3596.41 (±0.42%)	22 288.14 (±0.52%)	29 284.77 (±0.34%)

C. Kernel Trace for ZRAM Decompression

To highlight the root cause of the timing differences in ZRAM accesses, we trace the kernel functions called on accesses to ZRAM pages using the `ftrace` [67] utility. Listing 9.8.1 shows the trace of kernel functions called on an access to an incompressible and compressible page in ZRAM. The incompressible page contains random bytes, while the compressible page contains 2048 bytes of the same value and 2048 random bytes, similar to the partially-compressible setting in Section 9.6.5. We only list the functions which are called when the ZRAM page is swapped in to regular memory.

The main difference between the two cases (colored in red) is that the functions performing decompression of the ZRAM page are only called when a compressible page is swapped-in, while these functions are skipped for the page stored uncompressed. As the operating system knows from the

stored meta-data whether the page is stored compressed or uncompressed and can skip the corresponding functions for uncompressed pages. Of these additional function calls, the main driver of the timing difference is the `__deflate_decompress` function in Listing 9.8.1 which consumes 12555 ns. This ties in with the characterization study in Section 9.6.5 which showed the average timing difference between accesses to compressible and incompressible pages to be close to 10000 ns for ZRAM with *deflate* algorithm. These timings are for the *deflate* implementation in the Linux kernel, a modified version of `zlib v1.1.3`; hence these timings differ from the `zlib` timings in Table 9.1 for the more recent `zlib v1.2.11`.

```
1  Incomp.(ns) Comp.(ns) Function
2  Incomp.(ns)  Comp.(ns)  Function
3        0          0      swap_readpage
4       62         61      page_swap_info
5      126        123      __frontswap_load
6      195        188      __page_file_index
7      254        248      bdev_read_page
8      326        310      blk_queue_enter
9      395        379      zram_rw_page
10     460        442      zram_bvec_rw.isra.0
11     527        505      generic_start_io_acct
12     590        575      update_io_ticks
13     661        634      part_inc_in_flight
14     729        755      __zram_bvec_read.constprop.
15     813        838      zs_map_object
16     967       1040      _raw_read_lock
17       -       1229      zcomp_stream_get
18       -       1306      zcomp_decompress
19       -       1373      crypto_decompress
20       -       1433      deflate_decompress
21       -       1499      __deflate_decompress
22       -      14053      zcomp_stream_put
```

Listing 9.8.1.: Kernel function trace for ZRAM access to an incompressible and compressible page.

D. Layout Discovered by Comprezzor

Figure 9.10 shows the layout discovered by the Comprezzor that amplifies the timing difference for decompression of the correct and incorrect guesses in Zlib dictionary attack. Note that for correct guesses, the entire guess string, *i.e.*, `cookie=SECRET`, is compressed with the secret string. And as discussed in Section 9.5.2, the subsequent data is incompressible and invokes only a single `memcpy` operation, which faster than Huffman or LZ77 decoding. For wrong guesses, only the prefix is compressed, introducing the timing difference we exploit. Listing 9.11a and Listing 9.11b show the debug trace from the Zlib code for decompression with the correct and incorrect guesses to illustrate the root cause of the timing differences.

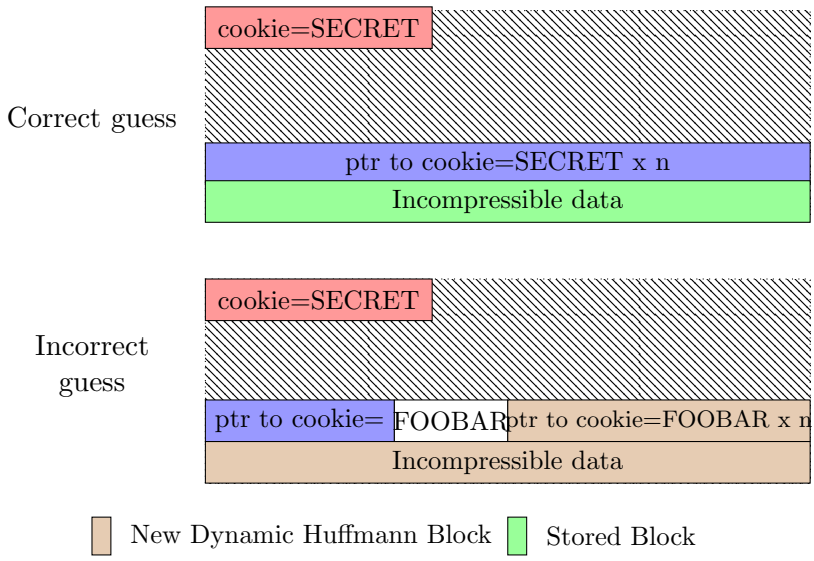


Figure 9.10.: Incorrect guesses with the corner case discovered by Comprezzor lead to a dynamic Huffman block creation for the partially-compressible data that is slow to decompress.

On a decompression, this block must now undergo the Huffman decoding (and LZ77), which results in several table lookups, memory accesses, and higher latency.

E. Bytewise Leakage

Figure 9.12 illustrates the bytewise leakage of the secret (SECRET) for a PHP application using PHP-Memcached. Figure 9.13 shows the bytewise of the secret string for a Flask application that stores secret data together with attacker-controlled data into PostgreSQL. The prefix value can be shifted bytewise, which allows reusing the same memory layouts found by Comprezzor. Figure 9.14 shows the last two bytes leaked from a secret (SECRET) in a ZRAM page. This is a continuation of Figure 9.9 which showed the leakage of the first four bytes. All three cases expose extremely high timing differences with an orders-of-magnitude gap between the correct and wrong guesses. The standard error margins are below 1 % for all guesses.

```
      length 12
      distance 16484
      literal 0x17
      length 13
      distance 14
      literal 0xb3
      length 13
      distance 14
      literal 'x'
      length 13
      distance 14
      literal 0x05
      length 13
      distance 14
      literal 0xa9
      length 13
      distance 14
      literal 0x81
      length 13
      distance 14
      literal '['
  stored block (last)
    stored length 16186
    stored end
 check matches trailer
end
```

(a) Trace for correct guess in zlib. Here the entire guess string is compressed, and the remainder is incompressible (decompressed fast as a stored block).

```
      length 6
      distance 16484
      literal 'F'
      literal 'O'
      literal 'O'
      literal 'B'
      literal 'A'
      literal 'R'
      literal 0x17
      length 13
      distance 14
      literal 0xb3
      length 13
      distance 14
      literal 'x'
      length 13
      distance 14
      literal 0x05
 dynamic codes block (last)
   table sizes ok
   code lengths ok
   codes ok
```

(b) Trace for incorrect guess in zlib. Here only part of the guess string (`cookie=`) is compressed, and the remainder `cookie=FOOBAR` is separately compressed (decompression requires a slower code block for Huffman tree decoding).

Figure 9.11.: Zlib traces for compression

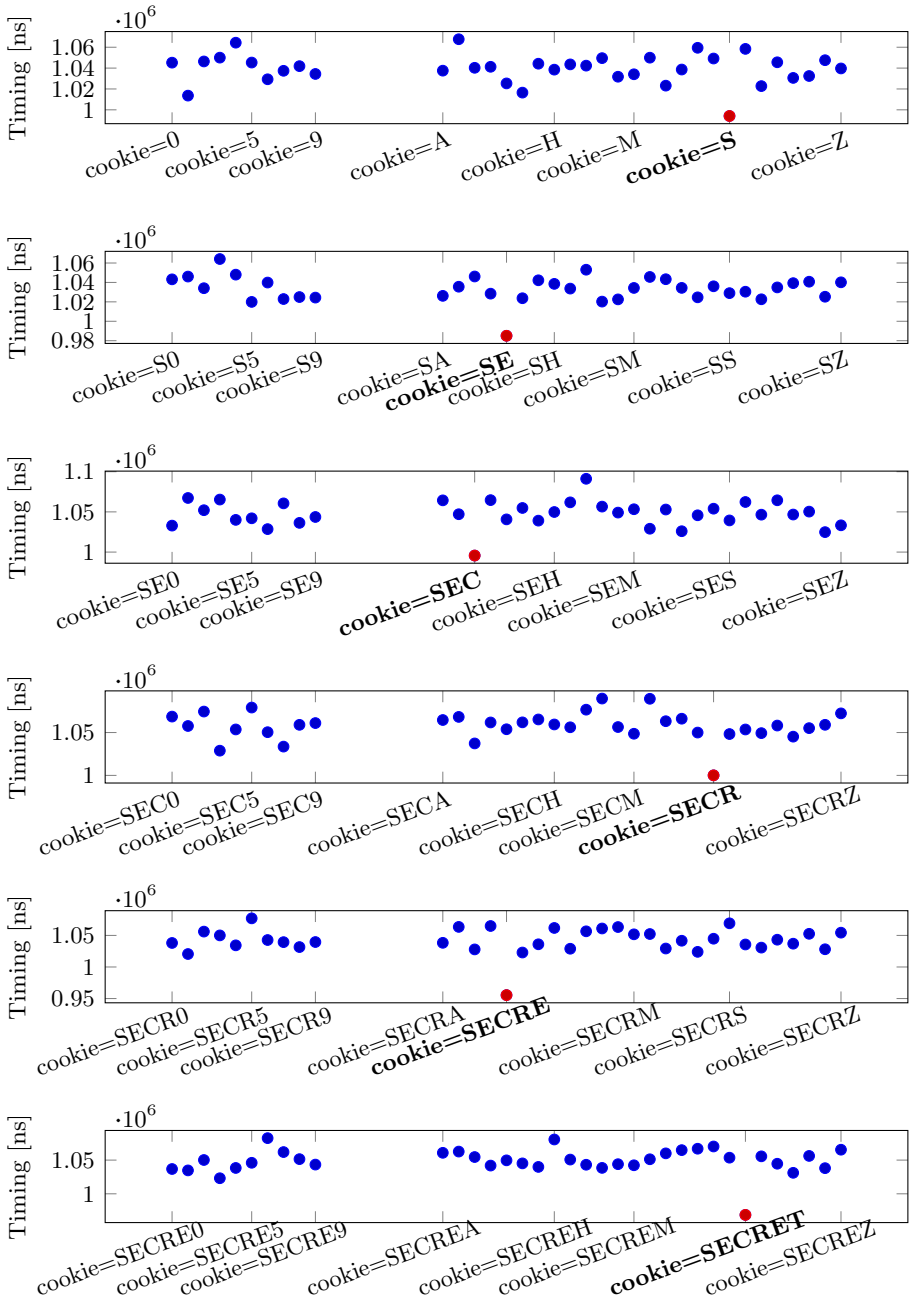


Figure 9.12.: Bytewise leakage of the secret (S,E,C,R,E,T) from PHP-Memcached. In each plot, the lowest timing (shown in red) indicates the correct guess. Standard error margins are below 1 % of the values and, thus, not visible in the plot.

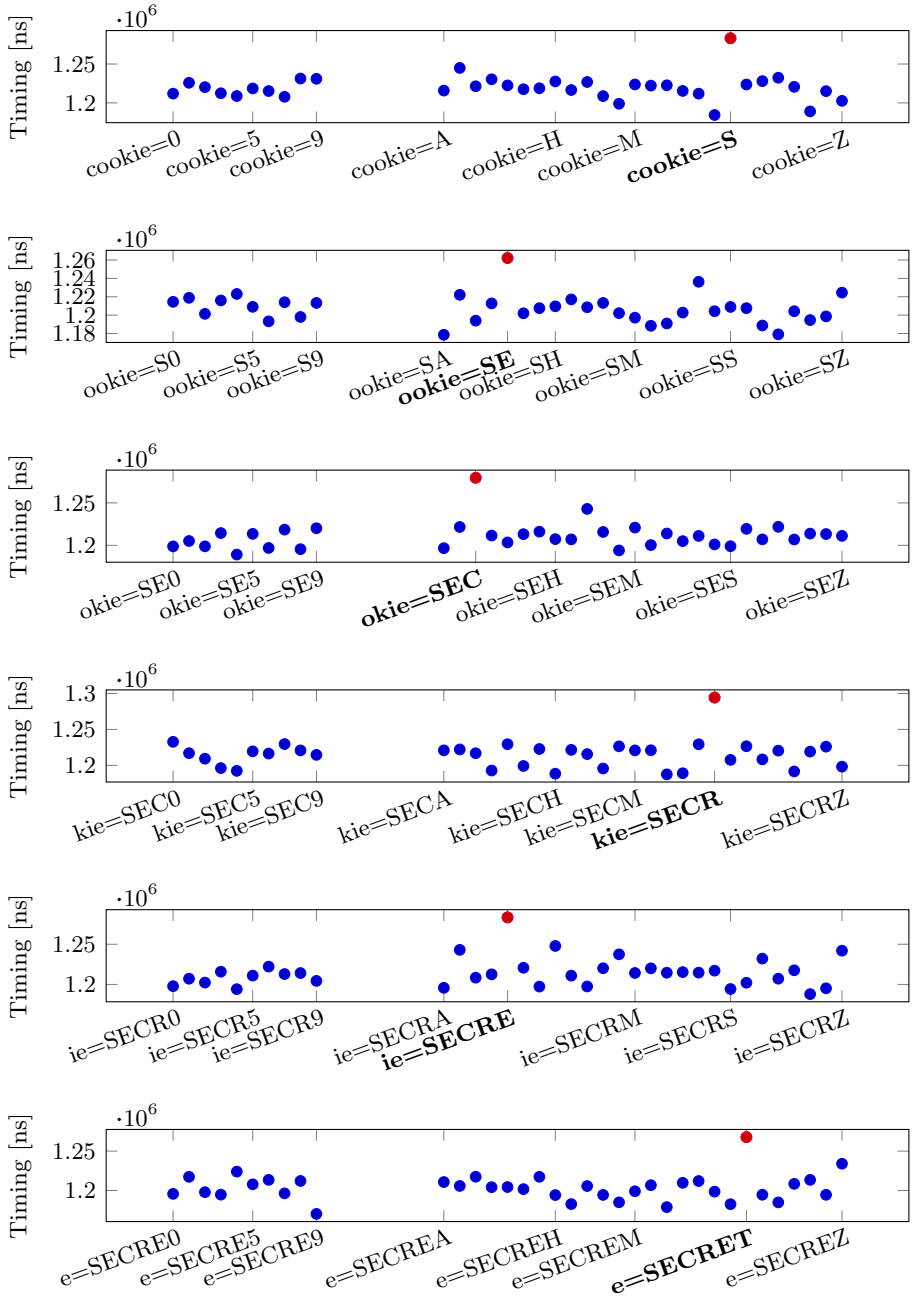


Figure 9.13.: Bytewise leakage of the secret (S,E,C,R,E,T) from PostgreSQL. The known prefix (cookie=) is shifted left by 1 character in each step, allowing the same memory layout to be reused. In each plot, the highest timing (shown in red) indicates the correct guess. Standard error margins are below 1 % of the values and, thus, not visible in the plot.

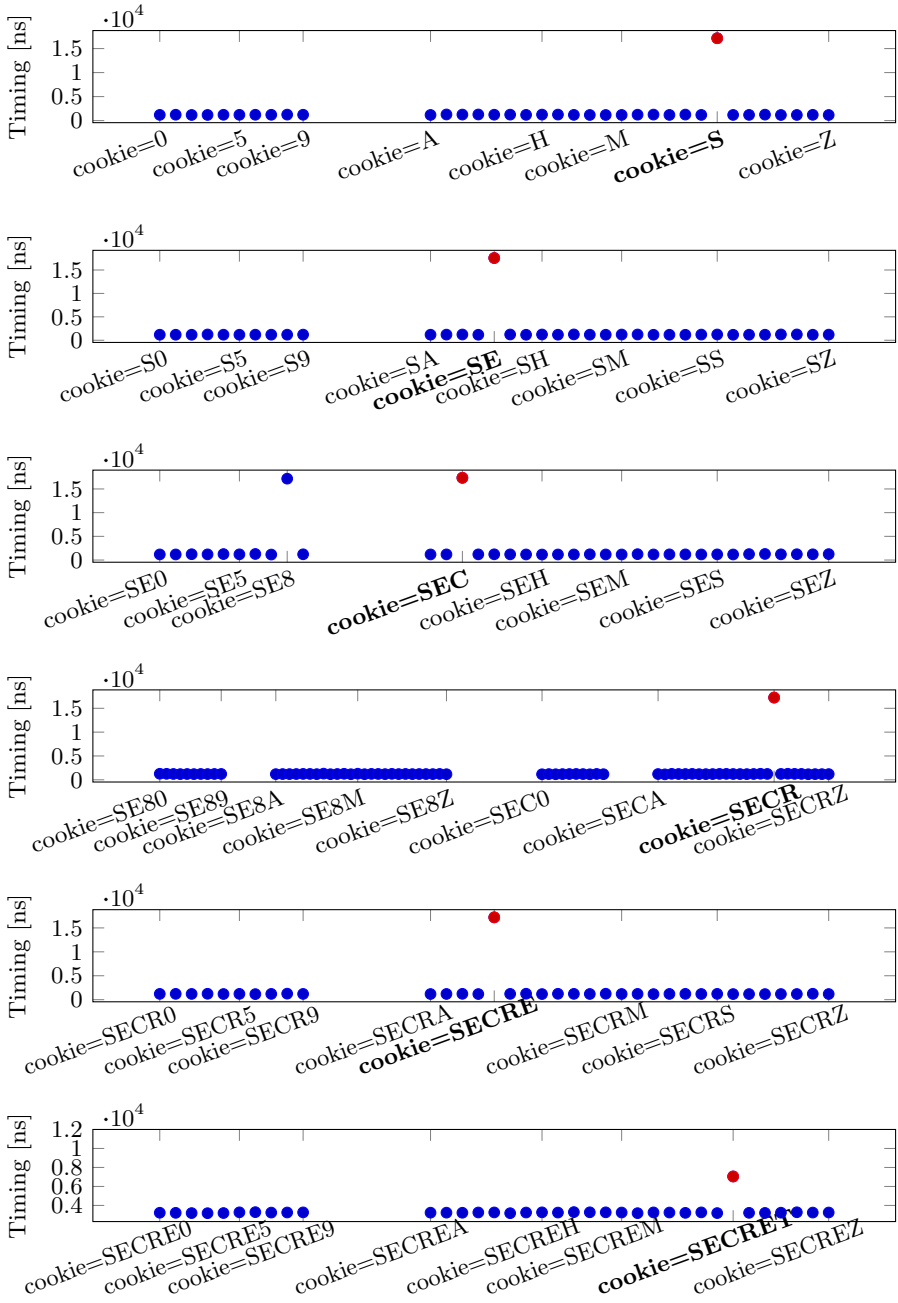


Figure 9.14.: Bytewise leakage of the secret (S,E,C,R,E,T) from ZRAM. Times for guesses (0-9, A-Z) for each of the bytes are shown. The highest value in each plot (shown in red) indicates the correct secret value for the byte. Standard error margins are below 1 % of the values and, thus, not visible in the plot.

F. JavaScript Memory Layout

```
const NUM_VALS = 203;
var non_typed_arrays = new Array(NUM_VALS);
non_typed_arrays.fill(Object);
// inserts target TypedArray including the 64-bit pointer
non_typed_arrays[0] = allocTypedArray(4096,0x31);
for(var k = 1; k < NUM_VALS; k++) {
 let colocate_data = [];
 for (let i = 0; i < 4096; i++) {
  // itof converts a BigInt to IEE 754
  // floating-point representation
  // the suffix n is used to represent a BigInt
  colocate_data[i] = itof(0xcafebabecafebaben);
 }
 non_typed_arrays[k] = colocate_data;
 triggerGC(); // trigger garbage collection
}

```

Listing 9.8.2.: Co-locate V8 heap pointers with attacker-controlled data.

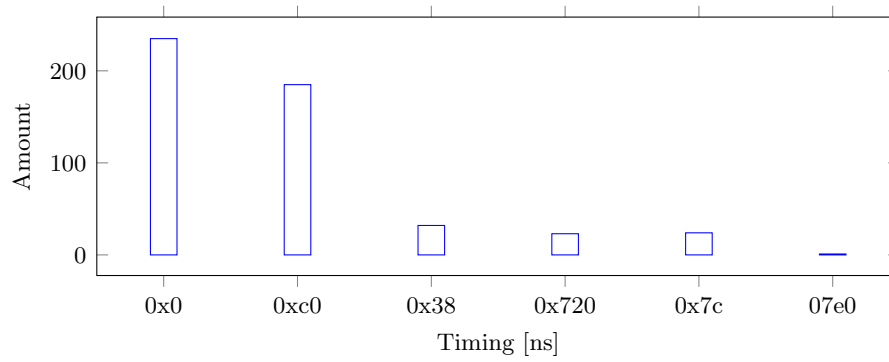


Figure 9.15.: Pointer offset distribution for multiple allocations in Google Chrome.

```
00: 00 00 01 01 58 3c 00 00 30 72 9c 00 58 3c 00 00
10: 02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
20: 00 00 00 00 05 22 04 08  10 00 00 00 b5 23 04 08
30: b5 23 04 08 b5 23 04 08  b5 23 04 08 b5 23 04 08
40: b5 23 04 08 b5 23 04 08  b5 23 04 08 05 22 04 08
50: 10 00 00 00 c1 46 24 08  5d 44 24 08 c1 e7 24 08
60: dd 30 25 08 71 c7 24 08  25 f3 24 08 59 d7 24 08
70: b1 2a 25 08 99 2a 04 08  a2 22 00 00 ca fe ba be
80: ca fe ba be ca fe ba be ca fe ba be ca fe ba be
*
1000
Legend: 00-0f:|64-bit pointers at offset|
10-7b:|Static data|, 7c-fff:|Attacker-controlled data
```

Listing 9.8.3.: Memory dump of the target page from Google Chrome after allocating non-typed and typed arrays and adding them to a list. The attacker can co-locate two V8 64-bit heap pointers and attacker-controlled data.

Our allocation script cf. Listing 9.8.2 creates a memory layout such that a 64-bit heap pointer pointing to the backing store is stored into a regular JavaScript array (non_typed_arrays). This can be achieved by first inserting a TypedArray with the full length of a page in the first slot of the list. To co-locate attacker-controlled data, the script inserts 4096 64-bit numbers into a regular JavaScript Array (`colocate_data`). This array will then be inserted into the array containing the 64-bit pointers (non_typed_arrays) and the garbage collection will be triggered. As 64-bit values in JavaScript are represented using the IEE754 floating-point representation, we use a conversion function `itof` to encode a 64-bit hexadecimal pointer to IEE 754 floating-point number. This function takes the BigInt and stores it into an `Float64Array`. By dumping the V8 memory we found that a number of 203 elements in the list to a memory layout, where the attacker controls most of the page and the only data that varies are the two heap pointers at offset 0. Listing 9.8.3 illustrates the generated memory layout after running the script from Listing 9.8.2. The first line (00) shows the 64-bit heap pointers aligned to page offset 0. We observe that the data between offset (10:) and offset (7c:) is constant and contains some compressed pointers to JavaScript objects. The data from offset `7c` to `0xfff` is fully attacker-controlled indicated in Listing 9.8.3 by the value (0xcafebabe). Using a fixed suffix, the attacker can use Decomp+Time to leak the correct byte values of the pointer. However, we do not observe that the heap pointer is always place at page offset 0. We repeat our experiment 500 times to see the distribution between the offsets. Moreover, we observe that if only the attacker-controlled data is modified, the garbage collection does not reorganize the heap. Figure 9.15

shows the distributions in 47 % of the cases the pointer is positioned at offset 0. In 37 % of the cases the pointer is located at offset 0xc0. The remaining 16 % of the positions the pointer was at 4 other locations. The attacker can train and probe for all of these 4 offsets and spawn multiple tabs to increase the probabilty of receiving offset 0x0 or 0xc0.

References

[1] Onur Acıiçmez, Werner Schindler, and Cetin K. Koc. "Cache Based Remote Timing Attack on the AES." In: *CT-RSA*. 2006.

[2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution." In: *S&P*. 2022.

[3] Johan Agat. "Transforming out timing leaks." In: 2000.

[4] Hassan Aly and Mohammed ElGayyar. "Attacking aes using bernstein's attack on modern processors." In: *International Conference on Cryptology in Africa*. 2013.

[5] Amazon. 2012. URL: `https://docs.aws.amazon.com/redshift/latest/dg/t_Compressing_data_on_disk.html`.

[6] Apple Insider. 2013. URL: `https://appleinsider.com/articles/13/06/13/compressed-memory-in-os-x-109-mavericks-aims-to-free-ram-extend-battery-life`.

[7] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. "String analysis for side channels with segmented oracles." In: *SIGSOFT*. 2016, pp. 193–204.

[8] Tal Be'ery and Amichai Shulman. "A Perfect CRIME? Only TIME Will Tell." In: *Black Hat Europe* (2013).

[9] Marcel Bohme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing." In: *CCS*. 2017.

[10] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization." In: *CCS*. 2021.

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

[12] BTRFS. *Compression*. 2021. URL: https://btrfs.wiki.kernel.org/index.php/Compression#Why_does_not_du_report_the_compressed_size.3F.

[13] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. "FaCT: A DSL for Timing-Sensitive Computation." In: *PLDI*. 2019.

[14] Euccas Chen. *Understanding zlib*. 2021. URL: https://www.euccas.me/zlib/#deflate.

[15] Citus. *Columnar: Distributed PostgreSQL as an extension*. 2021. URL: https://github.com/citusdata/citus/blob/master/src/backend/columnar/README.md.

[16] Yann Collett. *Smaller and faster data compression with Zstandard*. 2016. URL: https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/.

[17] DennyL. *Enabling ZRAM in Chrome OS*. 2020. URL: https://support.google.com/chromebook/thread/75256850/enabeling-zram-doesn-t-work-in-crosh?hl=en&msgid=75453860.

[18] P. Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification Version 1.3*. USA, 1996.

[19] Dmitry Evtyushkin and Dmitry Ponomarev. "Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations." In: *CCS*. 2016.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: http://www.rfc.net/rfc2616.html.

[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research." In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2020.

[22] Flask. *Flask*. 2021. URL: https://flask.palletsprojects.com/en/2.0.x/.

[23] Anders Fogh. *Covert Shotgun: automatically finding SMT covert channels*. 2016. URL: https://cyber.wtf/2016/09/27/covert-shotgun/.

[24] Vijay Ganesh, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing." In: *31st International Conference on Software Engineering*. 2009.

[25] Yoel Gluck, Neal Harris, and Angelo Prado. "BREACH: reviving the CRIME attack." In: *Unpublished manuscript* (2013).

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *ACM Sigplan Notices*. Vol. 40. 6. 2005, pp. 213–223.

[27] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures." In: *NDSS*. 2020.

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

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

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

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

[32] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES." In: *COSADE*. 2015.

[33] Nitin Gupta. *zram: Compressed RAM-based block devices*. 2021. URL: `https://www.kernel.org/doc/html/latest/admin-guide/blockdev/zram.html`.

[34] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. "ct-fuzz: Fuzzing for Timing Leaks." In: *International Conference on Software Testing, Validation and Verification (ICST)*. 2020.

[35] Chris Hoffman. *What Is Memory Compression in Windows 10?* 2017. URL: `https://www.howtogeek.com/319933/what-is-memory-compression-in-windows-10/`.

[36] holmeshe.me. *Understanding The Memcached Source Code - Slab II*. 2020. URL: {https://holmeshe.me/understanding-memcached-source-code-II/} (visited on 01/21/2020).

[37] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. "Remote cache timing attack on advanced encryption standard and countermeasures." In: *ICIAFs*. 2010.

[38] Dimitris Karakostas, Aggelos Kiayias, Eva Sarafianou, and Dionysis Zindros. "CTX: Eliminating BREACH with context hiding." In: *Black Hat EU* (2016).

[39] Dimitris Karakostas and Dionysis Zindros. "Practical new developments on BREACH." In: *Black Hat Asia* (2016).

[40] John Kelsey. "Compression and Information Leakage of Plaintext." In: *Fast Software Encryption*. 2002.

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

[42] Michael Larabel. *What Is Memory Compression in Windows 10?* 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=Fedora-33-Swap-On-zRAM-Proposal.

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

[44] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation." In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015.

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

[46] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing.* 2018. URL: https://llvm.org/docs/LibFuzzer.html.

[47] Heiko Mantel and Artem Starostin. "Transforming out timing leaks, more or less." In: 2015.

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

[49] Memcached. *memcached - a distributed memory object caching system*. 2020. URL: https://memcached.org/.

[50] Microsoft. *concepts-hyperscale-columnar*. 2021. URL: https://docs.microsoft.com/en-us/azure/postgresql/concepts-hyperscale-columnar.

[51] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis." In: *USENIX Security Symposium*. 2020.

[52] Taegeun Moon, Hyoungshick Kim, and Sangwon Hyun. "Mutexion: Mutually Exclusive Compression System for Mitigating Compression Side-Channel Attacks." In: *ACM Transactions on the Web (TWEB)* (2022).

[53] Mozilla. 2012. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer.

[54] Mozilla. *Compression in HTTP*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Compression.

[55] Ahamed Nafeez. "Compression Oracle attacks on VPN networks." In: *Blackhat, USA* (2018).

[56] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. "DifFuzz: differential fuzzing for side-channel analysis." In: *International Conference on Software Engineering (ICSE)*. 2019.

[57] Oracle. 2022. URL: https://www.oracle.com/a/ocom/docs/database/hybrid-columnar-compression-brief.pdf.

[58] Brandon Paulsen, Chungha Sung, Peter AH Peterson, and Chao Wang. "Debreach: mitigating compression side channels via static analysis and transformation." In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019.

[59] Mathias Payer. "The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes." In: *IEEE Security and Privacy* (2019).

[60] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make Sure DSA Signing Exponentiations Really Are Constant-Time." In: *CCS*. 2016.

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

[62] php.net. *memcached.constants.php*. 2021. URL: https://www.php.net/manual/en/memcached.constants.php.

[63] PostgreSQL. *TOAST Compression*. 2021. URL: https://www.postgresql.org/docs/current/storage-toast.html.

[64] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real." In: *S&P*. 2021.

[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "Vuzzer: Application-aware evolutionary fuzzing." In: *NDSS*. 2017.

[66] Juliano Rizzo and Thai Duong. "The CRIME attack." In: *ekoparty security conference*. Vol. 2012. 2012.

[67] Steven Rostedt. *ftrace - Function Tracer*. 2017. URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[68] Vishal Saraswat, Daniel Feldman, Denis Foo Kune, and Satyajit Das. "Remote Cache-timing Attacks Against AES." In: *Workshop on Cryptography and Security in Computing Systems*. 2014.

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

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

[71] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. "Remote Page Deduplication Attacks." In: *NDSS*. 2022.

[72] Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. "One Covert Channel to Rule Them All: A Practical Approach to Data Exfiltration in the Cloud." In: *TrustCom*. 2020.

[73] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. "NEUZZ: Efficient Fuzzing with Neural Program Smoothing." In: *IEEE Symposium on Security and Privacy*. 2019. DOI: 10.1109/SP.2019.00052.

[74] Luigi Soares and Fernando Magno Quintao Pereira. “Memory-Safe Elimination of Side Channels.” In: *CGO*. 2021.

[75] Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. “Safecracker: Leaking Secrets through Compressed Caches.” In: *ASPLOS*. 2020.

[76] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. “Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections.” In: *USENIX Security Symposium*. 2020.

[77] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. “Request and conquer: Exposing cross-origin resource size.” In: *USENIX Security Symposium*. 2016.

[78] Mathy Vanhoef and Tom Van Goethem. “HEIST: HTTP Encrypted Information can be Stolen through TCP-windows.” In: *Black Hat US Briefings, Location: Las Vegas, USA*. 2016.

[79] Pepe Vila, Boris Köpf, and Jose Morales. “Theory and Practice of Finding Eviction Sets.” In: *S&P*. 2019.

[80] WebDev. 2022. URL: `https://web.dev/coop-coep/`.

[81] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. “Osiris: Automated Discovery Of Microarchitectural Side Channels.” In: *USENIX Security Symposium*. 2021.

[82] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. “Eliminating Timing Side-Channel Leaks Using Program Repair.” In: *ISSTA*. 2018.

[83] Zhenyu Wu, Zhang Xu, and Haining Wang. “Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud.” In: *USENIX Security Symposium*. 2012.

[84] Meng Yang and Guang Gong. “Lempel-Ziv Compression with Randomized Input-Output for Anti-compression Side-Channel Attacks Under HTTPS/TLS.” In: *International Symposium on Foundations and Practice of Security*. Springer. 2019.

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

[86] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals Part 1*. 7th ed. Microsoft Press, 2017.

[87] Michał Zalewski. *American Fuzzy Lop*. 2021. URL: https://github.com/Google/AFL.

[88] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. "Cache Timing Attacks on Camellia Block Cipher." In: *Cryptology ePrint Archive, Report 2009/354* (2009).

[89] Michał Zielinski. *SafeDeflate: compression without leaking secrets*. Tech. rep. Cryptology ePrint Archive, Report 2016/958. 2016, 2016.

10

Layered Binary Templating

Publication Data

Martin Schwarzl, Erik Kraft, and Daniel Gruss. "Layered Binary Templating." In: *ACNS*. 2023

Contributions

Main author.

Layered Binary Templating

Martin Schwarzl, Erik Kraft, Daniel Gruss

Abstract

We present a new generic cache template attack technique, *LBTA*, layered binary templating attacks. *LBTA* uses multiple coarser-grained side channels to speed up cache-line granularity templating, ranging from 64 B to 2 MB in practice and in theory beyond. We discover first-come-first-serve data placement and data deduplication during compilation and linking as novel security issues that introduce side-channel-friendly binary layouts. We exploit this in inter-keystroke timing attacks and, depending on the target, even full keylogging attacks[1], e.g., on Chrome, Signal, Threema, Discord, and the passky password manager, indicating that all Chromium-based apps are affected.

10.1. Introduction

Techniques like Flush+Reload [77] advanced cache attacks from cryptographic [4] to non-cryptographic applications operating on secret data have been the research focus, e.g., breaking ASLR (address-space layout randomization) [28, 35], attacking secure enclaves [6, 19, 27, 44], spying on websites and user input [41, 69], and covert channels [42, 53, 76, 77]. In particular, user input, especially keystrokes, has become a popular attack target for inter-keystroke timing attacks [50, 56, 61]. Gruss et al. [32] showed that libraries leak more information than just inter-keystroke timings, e.g., distinguishing groups of keys.

Compilers and linkers [36] can facilitate or even introduce side-channel leakage [47, 60], invisible on the source level, through optimizations targeting runtime, memory footprint, and binary size. Similarly, JIT compilation can also introduce timing side channels [7]. These side channels are typically invisible in the source code and often remain undetected. Numerous

[1] Demo: The user first announces via Signal messenger to send money to a friend, then switches to Chrome to visit a banking website and enters the credentials there. All keystrokes are correctly leaked. `https://streamable.com/dgnuwk`.

works explored the automatic identification of cache side-channel leakage, albeit with a focus on cryptography and the goal of making code constant-time [9, 13]. However, for general-purpose applications, e.g., browsers, it is not feasible to linearize the entire instruction stream to constant-time code, especially for different user inputs that trigger vastly different program behavior. Cache templating takes a practical approach by scanning for leakage on real systems, providing a leakage template either to a defender (to close the side channel) or an unprivileged attacker who maps binaries as shared memory and infers events from side-channel activity. The templating itself runs on an attacker-controlled system with full privileges, a binary is mapped into the address space of the templating process to profile which memory locations show side-channel activity upon specific events. Since cache templating works with binary offsets it is entirely unaffected by mechanisms such as ASLR. While the fine cache-line granularity is beneficial in the attack phase, it also leads to extremely high templating runtimes. For instance, templating the binary, shared libraries, and memory-mapped files used by the Chrome browser (about 210 MB) with the published cache template attack tool [32] on our test system, would take 113.17 days. Unfortunately, this prohibits integration of cache leakage analysis into development workflows. Hence, we need to ask the following questions:

Which role does spatial granularity play for template attacks? Does a coarser granularity bear benefits in the templating phase?

In this paper, we answer both questions with *LBTA*, *Layered Binary Templating Attack*. *LBTA* introduces the previously unexplored dimension of *spatial granularity* into software-based templating attacks. *LBTA* combines the information of multiple side channels that provide information at different spatial granularity to accelerate the search for secret-dependent activity substantially. Our templating starts with the channel with the most coarse spatial granularity and, based on the activity, uses more fine-grained spatial granularity to detect the exact location (cache-line granularity 64 B).

Our evaluation of *LBTA* on state-of-the-art systems shows that a variety of hardware and software channels with different granularity are available. We focus in particular on a combination of a software channel, the page-cache side channel, with 4 kB granularity, and the cache side channel, with 64 B granularity. Page cache attacks are hardware-agnostic [30], resulting in cross-platform applicability, *i.e.*, our templater supports both Windows and Linux with the 4 kB page-cache side channel. We show that this

two-layered approach already speeds up cache templating [32] by three orders of magnitude (*i.e.*, 1848x).

We evaluate *LBTA* on different software projects, including Chrome, Firefox, and LibreOffice Writer. The most significant finding is that **first-come-first-serve data placement** and **data deduplication during compilation and linking** during compilation and linking introduce side-channel-friendly binary layouts, with spatial distances of multiple 4 kB pages between key-dependent data accessed during a keystroke. Using *LBTA* [62], we find distinct leakage for all alphanumeric keys, allowing us to build a full unprivileged cache-based keylogger using Flush+Reload that leaks all keystrokes from Chromium-based applications involving password input fields, e.g., Chrome on banking websites, popular messengers including Signal, Threema, Discord, and password manager apps like passky. Based on our findings, we conclude that any app using the Chromium framework should be considered affected [23]. In addition, we demonstrate that where full keylogging is not possible, *LBTA* still finds enough leakage for inter-keystroke timing attacks [20, 32, 50, 61, 79], e.g., on Firefox and LibreOffice Writer.

We confirm that the Linux `preadv2` syscall can be used instead of the now mitigated `mincore` syscall [30] for page cache attacks [37] albeit with a lower temporal resolution of about 2 seconds. Since system-level defenses like ASLR have no effect on our attack, we provide a systematic discussion of the possible mitigation vectors specific to *LBTA*.

Contributions. The main contributions of this work are:

1. We introduce a new dimension, side-channel granularity, into cache template attacks and use it to speed up the templating by three orders of magnitude.
2. We show that the leakage discovered by *LBTA* can be exploited in hardware (*i.e.*, Flush+Reload) and software attacks (*i.e.*, via the page cache).
3. We discover first-come-first-serve data placement and data deduplication generate amplify and introduce side-channel leakage, invisible on the source level.
4. We present inter-keystroke timing and, depending on the target, full keylogging attacks, e.g., Chrome, Signal, and the passky password manager.

Responsible Disclosure. We responsibly disclosed our findings to the Chromium team. The underlying issue is tracked under CVE-2022-2612

(6.5, medium severity) and was patched in the M104 release in August 2022.

Outline. In Section 10.2, we provide the background. In Section 10.3, we explain the *LBTA* building blocks. In Section 10.4, we describe first-come-first-serve data placement and data deduplication. In Section 10.5, we evaluate *LBTA* on different targets. In Section 10.6, we discuss mitigations, and we conclude in Section 10.7.

10.2. Background

In this section, we provide background on hard- and software cache attacks, side-channel discovery, and compiler- and linker-introduced side channels.

Shared Memory Operating systems (OSs) apply various optimizations to reduce the system's general memory footprint. One such optimization is shared memory, where the OS actively tries to remove duplicate data mappings. An example would be shared libraries, such as the `glibc`, used in many programs, and thus, can be shared between processes. Moreover, with the `mmap` respectively `LoadLibrary` functions, a user program can request shared memory from the OS by mapping the library as `read-only` memory. Another optimization to reduce the memory footprint commonly used for virtual machines is memory deduplication on a page-wise level. The OS deduplicates pages with identical content and maps the deduplicated page in a copy-on-write semantic.

Deduplication The concept of deduplication is generic and can be applied in the context of various memory systems to save memory. For storage systems, one example is cloud storage systems that deduplicate files to minimize the amount of storage required [34, 38]. For main memory, there are multiple mechanisms: Copy-on-write avoids duplicating memory during process creation, the OS's page cache [30] avoids duplicating memory pages from the disk, and the OS also avoid duplicating the zero page when zeroed memory is requested. However, the most prominent example is data-based page deduplication [63]. With data-based page deduplication, the OS or hypervisor scans the main memory page-wise and identifies identical pages, e.g., using hashes or byte-wise data comparison, deduplicating them. In all above types of deduplication, attempting to modify the

deduplicated memory triggers a 'copy-on-write' operation which is known to introduce side-channel leakage, e.g., for file deduplication [2, 34], page deduplication [63], from JavaScript [17, 29] and even remotely [58].

While all above types of deduplication target memory, there is also deduplication in other contexts. In this paper, we focus on a different type of deduplication that has little to do with the above or memory systems in general. We instead focus on deduplication during compilation and linking. The goal of deduplication here is similar though, *i.e.*, reducing memory usage, and improving runtime performance due to reduced memory or cache utilization. However, the security implications of deduplication during compilation and linking are unknown.

Cache Attacks Caches introduce exploitable timing differences between cached and uncached data. While the first cache attacks targeted cryptographic primitives [4, 39], more recent ones target secure enclaves [6, 19, 27, 44], monitor user interaction and keystrokes [41, 56, 69], and build stealthy and fast covert channels [42, 53, 76]. The Flush+Reload attack technique requires shared memory with the victim, e.g., shared libraries [77]. However, as Flush+Reload works on the attacker's own addresses pointing to the same physical shared memory, there is no need to know the victim's ASLR offsets, as file offsets are used instead.

Cache attacks were also demonstrated from JavaScript to spy on keystrokes and break memory randomization [28, 41, 46]. Most cache attacks focus on hardware caches with a 64 B cache line granularity. Cache attacks on the TLB instead have a spatial granularity of 4 kB, 2 MB, 1 GB, or 512 GB [31, 66].

In particular, for SGX, so-called controlled channels have been demonstrated as powerful attack primitives [44, 64, 75] with high spatial and temporal resolution, as well as a very high accuracy. Controlled channels are side channels running with elevated privileges, e.g., kernel privileges, with a typical attack target being secure enclaves that are protected against regular kernel access.

There are also software caches, e.g., the page cache in Linux and Windows. Both Linux and Windows also offer functions to verify whether a specific virtual address is resident in memory or not, namely `mincore` and `QueryWorkingSetEx` respectively. Gruss et al. [30] demonstrated cache attacks on the page cache by either using these functions or by measuring timing

differences. Despite hardening attempts on Linux and Windows, the Linux `preadv2` system call can still be used to mount cache attacks [37] in the same way as with the `mincore` syscall: Using the `RWF_NOWAIT` flag, an attacker can observe whether a page is resident in the page cache or not, yielding the same side-channel information as `mincore`. The results of the `preadv2` templating attacks can be found in Section 10.5.

Automated Discovery of Side Channel Attacks Templating attacks have been first shown and mentioned on cryptographic primitives running on physical devices [13, 43, 49]. Brumley and Haka [9] first described templating attacks on caches. Doychev et al. [22] presented a static analyzer that detects cache side-channel leakage in applications. Gruss et al. [32] showed that the usage of certain cache lines can be observed to mount powerful non-cryptographic attacks, namely on keystrokes. Lipp et al. [41] showed cache attacks and cache template attacks on ARM. Van Cleemput et al. [65] proposed using information gathered in the templating phase to detect and mitigate side channels. Wang used symbolic execution and constraint solvers to speed up cache templating of cryptographic software [70]. Schwarz et al. [54] demonstrated template attacks on JavaScript to enable host fingerprinting in browsers. Weiser et al. [72] and Wichelmann et al. [73] showed that Intel PIN tools can be used to automatically detect secret-dependent behavior in applications, especially in a cryptographic context. Wang et al. [69] presented a similar automated approach to detect keystrokes in graphics libraries. Carre et al. [10] mounted an automated approach for cache attacks driven by machine learning. With that approach, they were able to attack the `secp256k11` OpenSSL ECDSA implementation and extract 256 bits of the secret key. Brotzmann et al. [8] presented a symbolic execution framework to detect secret-dependent operations in cryptographic algorithms and database queries. Li et al. [40] demonstrated a neural network to perform power analysis attacks automatically. Yuan et al. [78] demonstrated that manifold learning can be used to detect and locate side-channel leakage in media software.

Compiler-introduced Side Channels While developers typically focus on the source code level and care is taken to not introduce side channels there, the compiler translates the source code to a binary, essentially a different language. However, this step can introduce program behavior that is not visible on the source level and introduces or amplifies side-channel leakage. Page [47] demonstrated that dynamic compilation in

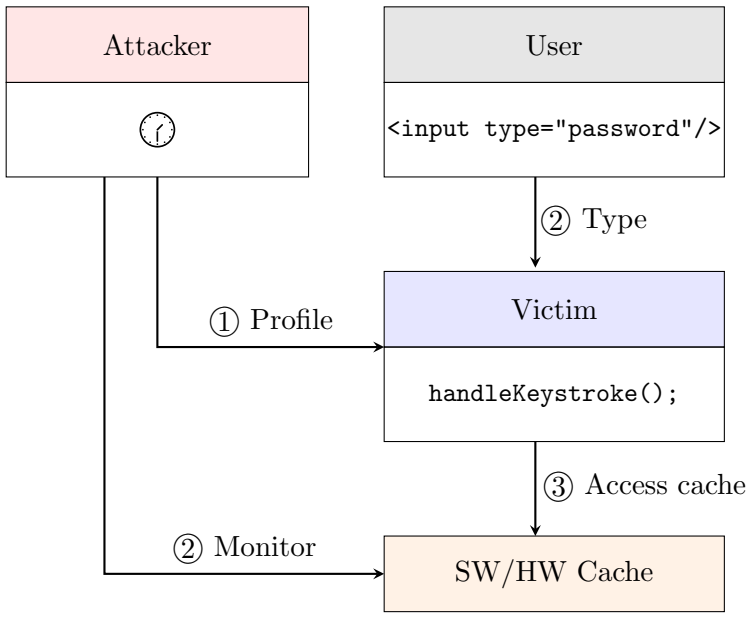


Figure 10.1.: Overview of the *LBTA*

Java leads to power side-channel leakage in a side-channel-secured library. Simon et al. [60] showed that mainstream C compilers optimizations can break cryptographically secure code by introducing timing side channels. Brennan et al. [7] showed that timing side channels can be introduced by exploiting JIT compilation.

Due to this significant influence of compilers on side-channel leakage in binaries, they are also frequently used for new mitigation proposals against side-channel leakage [5, 11, 16, 18, 26, 48].

10.3. Layered Binary Templating Attack

For large binaries, like the Chrome browser with multiple shared libraries (220 MB), templating with fine granularity like prior work [32, 41, 69], e.g., a cache line, becomes impractical. *LBTA* takes advantage of coarser granularity channels, which usually are considered a disadvantage for the attacker. In this section, we present the high-level view on *LBTA* and show how *LBTA* reduces the templating runtime by three orders of magnitude (*i.e.*, 1848x).

10.3.1. Threat Model

The **templating** (or profiling) runs on a fully attacker-controlled system with any privileges the attacker wants to use to facilitate the templating. This system is assumed to have the same side channels as the victim system, such as the page cache and CPU cache, and the same software versions as on the victim system were deployed, e.g., from package repositories. For this reason, the typical template attack threat model only restricts the attacker in the exploitation phase [32], which we follow in this work.

In the **exploitation phase**, the attacker runs an **unprivileged** attack program on the victim's system, possibly under a separate user account. Hence, we assume the victim application is started independently by the victim user, and cannot be started, stopped, or debugged by the attacker. This also excludes "preloading", which, e.g., on Wayland (the default Ubuntu display server), would allow monitoring all inputs to the application [3]. For non-Wayland systems, we assume that the attacker cannot use other keylogging techniques (e.g., on X11 [1]), or Windows (e.g., using the `getasynckeystate` API call [68]), e.g., due to system hardening or enforced security policies. Some of the applications we attack provide auto-fill features or are password managers. However, since we focus on the **keylogging scenario**, we assume that the victim user enters the password in these applications manually, *i.e.*, the user does not use an auto-filler or another password manager to unlock the password manager. Furthermore, many websites set the `autocomplete="off"` option for sensitive input fields, suppressing the in-built auto-fill and password management features.

10.3.2. High-Level Overview of the Templating Phase

Figure 10.1 illustrates the steps of *LBTA*. First, the attacker templates the library and creates templates of the cache usage for different keystrokes. After the templating phase, the attacker monitors the cache usage to infer inter-keystroke timings and, depending on the target, even distinguish key values.

10.3.3. Templating with Different Spatial Granularity

A novel aspect of *LBTA* is to utilize the spatial granularity of different side channels, forming a practical and generic multi-layered approach.

64 B Granularity. Previous cache template attacks [32] used cache-line granularity (64 B). One disadvantage of this approach is the runtime of the templating phase. When templating a single cache line with Flush+Reload, we observe an average runtime of 490 cycles (n = 1000000, $\sigma_{\overline{x}}$ = 20.35%). On a 4.0 GHz CPU, this would take 122.5 ns. The Chrome binary has a file size of about 210 MB leading to 2 949 120 addresses to template with Flush+Reload. This leads to a runtime of 0.36 s for templating every cache line once. However, Gruss et al. [32] describe that multiple rounds of Flush+Reload are required to get reliable cache templating results. Running an Intel 6700k CPU at 4.0 GHz with a Ubuntu 20.04, templating 1 MB of the Chrome browser (version 100.0.4896.60) with the provided implementation of Gruss et al. [32], we observe a runtime of 817.652 s for 1 MB and a total runtime of 1.98 days for the full binary, including shared libraries, of 210 MB. Moreover, this templating tool only reports whether a certain address was cached or not and does not match the cache hits with the entered keystrokes. To template, for instance, the 57 different common keys sequentially with the method by Gruss et al. [32], we would need an **impractical** total runtime of 113.17 days to obtain useful templates. We conclude that such an approach is not feasible for browser developers as the code base changes frequently, and releases sometimes occur on a monthly basis [14].

4 kB Granularity. Page cache attacks exploit the OS page cache, which works at a coarser granularity of 4 kB [30]. Page cache attacks have the advantage of working independent of the underlying hardware. To identify the exact memory locations causing leakage, they also resorted to templating. However, they did not combine this information with timing differences from hardware caches.

Our intuitive idea here is to combine the 4 kB-granularity side channel with the more fine-grained side channel into a two-layered approach. Hence, we **do not** template all cache lines on a 64 B granularity but instead, filter memory locations on a 4 kB granularity. Instead of 2 949 120 memory locations, we then only monitor 46 080 memory locations for the Chrome example, *i.e.*, a templating runtime speedup of at least 64. In addition, the templating phase on the 4 kB granularity level implicitly identifies locations exploitable via the page cache.

The templating phase runs on the attacker's own machine (cf. Section 10.3.1). Hence, we can use the page cache side channel or privileged channels, e.g., controlled-channel attacks [75] via page-table bits [64] during the templating, *i.e.*, we use the kernel's idle bit for tracking. For the

full Chrome binary (cf. Section 10.3.5), this results in a runtime of only 1.47 hours for all 57 keystrokes.

2 MB Granularity. Each page-table layer provides `referenced` bits that are set by the hardware when a location in this region is accessed. The 2 MB-granularity side channel is also exposed via various side channels [31, 66]. We observe the activity on 2 MB pages via the PMD paging structure and the `referenced` bit. We use PTEditor [55] to check and clear the referenced bit of the PMD, *i.e.*, a 2 MB page, with a runtime of 661.965 ns ($n = 1000000, \sigma_{\overline{x}} = 0.049\%$) per check. Hence, to template the 57 different common keys in Chrome with 20 repetitions per key, we estimate the total runtime of templating to be about 0.15 seconds.

10.3.4. Beyond Huge Pages

LBTA extends to arbitrarily coarser granularity channels.

1 GB, 512 GB and 256 TB Granularity. For the 1 GB granularity level and beyond, we experimentally validated that we can again use controlled-channel attacks [64, 75], using the corresponding higher page-table layers. Following a similar approach as for the previous levels, we use PTEditor to template and clear the referenced bit for the single offset in a 1 GB (respectively 512 GB or 256 TB) range. The runtime for checking and clearing the referenced bit on these layers is the same as for the PMD (661.965 ns ($n = 1000000, \sigma_{\overline{x}} = 0.049\%$)). We emphasize that scanning layers that exceed the binary size, e.g., the 1 GB layer for a 180 MB binary, provides no additional information and does not reduce the search space, as the search will always proceed to the next smaller layer for the entire memory range then. Therefore, in the evaluation, we skip all layers that exceed the binary size. Still, these layers of *LBTA* may become relevant in the future with constantly growing binaries and libraries.[1]

10.3.5. Templating Phase Implementation

The high-level idea is that the templater tracks page usage and actively filters pages not related to keystrokes to reduce the search space of pages to template and, as a result, reduce the overall runtime of the templating. We implement our templater in Python and provide the code in our Github

[1] The Chrome binary had 100 MB in 2017 and 180 MB in 2022, an increase of 80 %.

repository[1]. The templater takes as input the set of different keys, the PID or process names that should be monitored, and the number of samples per key.

Algorithm 1 summarizes the steps of the templating. First, the templater runs a warmup phase, where all keystrokes to template are entered once to load all related memory locations into RAM. Then the templater collects all the memory mapping information from all files from the target processes where activity has been found. These memory mappings include all shared libraries. The templater generates random key sequences based on the set of keys to template. For each key in the sequence, the templater iterates over all the memory locations on the current granularity level, and resets the access information, *i.e.*, resets the `referenced` or `idle` bit, or flushes the cache line depending on the side channel used. Based on the number of samples, the templater computes the hit ratio for each location. Subsequently, the templater repeats this step for all memory locations above a specific hit ratio with the next lower spatial granularity. With this search strategy, the templater continues down to the lowest level, where only regions are templated that showed activity on coarser granularity. On the lowest level, the templater determines a hit ratio for each single cache line.

Algorithm 1: *LBTA* Templating Algorithm

Input: Set of keys K, target PIDs P_n, number of samples N
Output: hit ratio matrix of all memory mappings H

1: Enter all keys in K once // Warmup
2: Collect all valid memory mappings of P_n
(possibly from previous layer)
3: **for** $i = 0; i < N; i++$ **do**
4: **for** each $k \in K$ **do**
5: Reset memory mappings (reset referenced/idle bits or flush)
6: Enter key k
7: Check state for all present memory mappings (via interface or timing)
8: Compute hit ratios for k and update H_k
9: **end for**
10: **end for**
11: **return** H, and repeat algorithm for next layer

[1] `https://github.com/IAIK/LayeredBinaryTemplating`

Linux. On the upper layers, we start by obtaining the memory mappings for the target process. On Linux, we read these mappings from procfs (with root privileges in line with the threat model). We group the memory locations then according to the most coarse granularity we use in our templating. By using the `referenced`-bit side channel according to Algorithm 1, we narrow down the set of memory locations for the next layer.

Windows. On Windows, we obtain a list of memory mappings using the `EnumProcessModules` PSAPI call, which lists all loaded libraries and executables, and `GetModuleInformation` for their actual sizes. Subsequently, we again use the `referenced`-bit channel to narrow down the set of memory locations using Algorithm 1. Subsequently, we continue with the next layer.

4 kB Page Granularity

While for the upper layers, we read referenced bits using PTEditor [55], we use a more optimized approach for the page granularity.

Linux. Our page usage tracker iterates over active mappings, reads the idle bit for the corresponding physical page from `/sys/kernel/mm/page_idle/bitmap` and checks if the page was accessed. We start by resetting the bit so that the page usage tracker is ready. We use the Python3 `keyboard` library to inject keystrokes into an input field. After the templater performs the sequence of keystrokes, we check all pages that are still in the candidate list for activity. A 1 at the page offset in the bitmap means the page was not accessed. Conversely, if we observe a 0 at the page offset, we reset the page offset and add it to the set of correlated pages to track on the next layer. This approach is fully hardware-agnostic, implemented in software in the Linux kernel. After each iteration, we reset the state again by marking the pages as idle again and repeat the measurements.

In case of a sequential read access pattern, the Linux kernel speculatively prefetches further pages of the same file after a new page was added to page cache. This optimization is called `readahead` [33]. On Ubuntu 20.04 (kernel 5.4.0), the default read-ahead size is 128 kB and can be found in the sysfs (`/sys/block/<block_device>/bdi/read_ahead_kb`). For file-mappings the kernel performs a different optimization called `read-around`.[1] There, the kernel prefetches pages surrounded by the

[1] https://elixir.bootlin.com/linux/v5.4/source/mm/filemap.c#L2437

page causing the pagefault e.g., 16 pages before the page causing the pagefault and 15 pages after. To reduce triggering read-ahead prefetching for sequential reads, we use the madvise system call with the `MADV_RANDOM` flag to indicate a random read order.

Overlapping event (*i.e.*, keystroke) groups for the current candidate page and pages that might trigger the read-ahead of the current candidate page could cause false positives in the Linux case. In addition, if the number of read-ahead suppress pages is too small, false positives can occur. Our classifier tries to reduce the number of false positives by checking out the read-ahead/read-around windows and systematically rule out other keystrokes. Based on the results of the templater, the classifier actively accesses surrounding pages from the target page to suppress the read-ahead/read-around optimization. Note that the read-ahead and read-around windows might overlap for some keys. If the keys to template are not on the same 4 kB-page, we can still distinguish two keys by checking the first and last surrounded pages being accessed. The templater actively creates warnings in the templating phase in case the keys are still indistinguishable.

Windows. Windows uses a different page replacement strategy with working sets [52]. For the page usage tracker on Windows, we use the PSAPI call `QueryWorkingSetEx` and monitor the `Shared`, `ShareCount` and `Valid` flags. If the page is marked as valid and shared and the share count is larger than 1, we mark the page as used. For the reset, `EmptyWorkingSet` is used to remove the pages from all workings sets. This PSAPI call is only available for unprotected processes, which is no issue during the templating phase (cf. Section 10.3.1).

On Windows, we observed no prefetching optimization within working sets, *i.e.*, read-ahead does not affect hit ratio or spatial accuracy. Alternatively, the templating could also be performed via controlled side channels [25, 59, 75], tracing tools such as Intel PIN, machine learning [10, 69] or architecturally monitoring the accesses of pages using PTEditor [55].

Classifier. On the 4 kB level, we collect the page-hit ratios for all events (*i.e.*, keys) and pages, showing the link between event and observable page hit. To distinguish 'no activity' from 'activity', we also template a dummy idle event [32] to measure which hit ratios are observed as a baseline. This idle event will not be linked to any page hit but rather should represent unrelated system activity the templater might pick up while profiling events. Our classifier links events or groups of events with single page hits

to keep the number of observed pages as low as possible. This is a trade-off between search time and completeness of the search that can be chosen differently for any *LBTA* on any target application. Furthermore, a more sophisticated attack could increase detection accuracy from monitoring multiple pages or cache lines for each event. However, we decided to use the search-time-optimized path, as side-channel attacks typically cannot observe an arbitrary amount of memory addresses anyway, *i.e.*, we focus on a more practical set of leaking addresses.

The algorithm to find a suitable page hit for an event e works as follows:

1. We normalize page-hit ratio vector for event e by average page-hit ratio from other events (baseline activity). The resulting vector is the correlation strength between each page and the event e.
2. We select the page with the highest correlation strength as a candidate.
3. If the candidate is not above the **location-specific** baseline activity, our algorithm merges events (e.g., going from single keys to key groups) until it is. We continue with the resulting event group $E = \{e_1, \ldots, e_M\}$ in step 1.
4. Once a candidate is found that is above the **location-specific** baseline activity, the algorithm returns this page to subsequent templating layers.

While running, the classifier collects information on potential read-around prefetching pages to filter them out. After successful classification, the attacker has a mapping of pages to events (*i.e.*, key) and groups of events (*i.e.*, groups of keys).

10.4. Compiler- and Linker-introduced Spatial Distance

Before we evaluate *LBTA*, we present one significant leakage-facilitating effect that we discovered while applying *LBTA* on a variety of targets. This effect is particularly critical as it originates in compiler optimizations in LLVM/clang that are enabled by default and the available compiler flags that can control this behavior come with serious limitations. Compiler optimizations aim for a minimal program runtime, small memory footprint, and small binary size. Moreover, linker optimizations try to further

optimize the binary in the linking stage. We primarily found two effects to facilitate cache side-channel leakage: One is the other is first-come-first-serve data placement in readonly sections, the other one is **data deduplication during compilation and linking**. While memory deduplication at runtime has been explored as a security risk already (cf. Section 10.2), data deduplication (e.g., of strings) during compilation is not widely known and its security implications are entirely unexplored. The security of constant-time implementations has been analyzed for side channels being introduced by compilers [7, 47, 60, 65]. In this section, we show that deduplication in combination with first-come-first-serve population during compilation (cf. Section 10.4.2) and linking (cf. Section 10.4.3) can amplify this effect by increasing the chance that secret-dependently accessed victim data is placed in an attacker-facilitating way. Deduplication can also be performed at the linking stage. The spatial distance between secret-dependent accesses can be introduce by both compiler and linker optimizations. We present two scenarios that we also found in widely used real-world applications, where the placement of read-only data, especially strings, amplifies side-channel leakage dramatically.

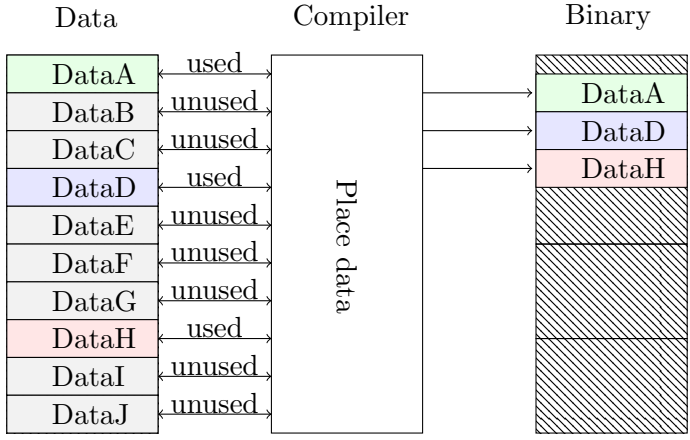


Figure 10.2.: First-come-first-serve population of the `.rodata` section in the binary.

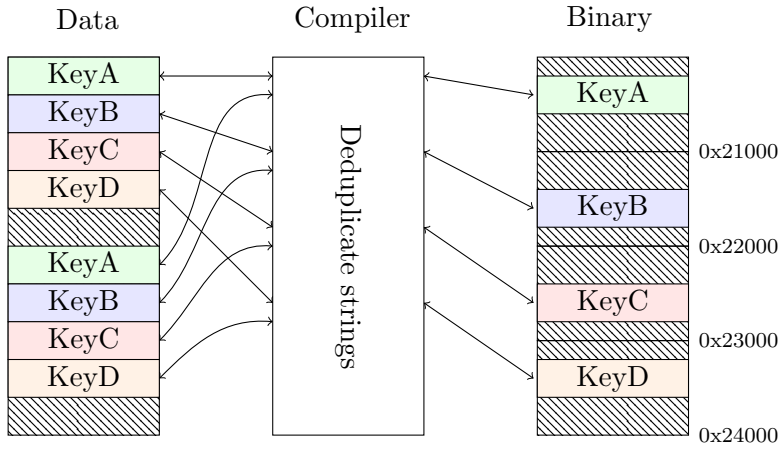
```
1  struct MapEntry { const char* key, value; };
2  #define LANGUAGE_CODE(key,value) { key, value }
3  #define MAP_DECL constexpr MapEntry mappings[] = MAP_DECL {
4    LANGUAGE_CODE("KeyA","DataA"), LANGUAGE_CODE("KeyB","DataB")
5  };
6  #undef MAP_DECL
7  void string_funcA(vector<string>& v) {
8    string local_ro_string = "DataB";
9    v.push_back(local_ro_string);
10   string padding_string = "<64-byte-string>";
11   v.push_back(padding_string);
12 }
13 void string_funcB(vector<string>& v) {
14   MapEntry k1 = mappings[0];   //KeyA
15   v.push_back(k1.value);
16   MapEntry k2 = mappings[1]; //KeyB
17   v.push_back(k1.value);
18 }
```

Listing 10.1: Strings are deduplicated in the binary and could lead to spatial distance between readonly-strings in the same array in combination with first-come-first-serve data placement.

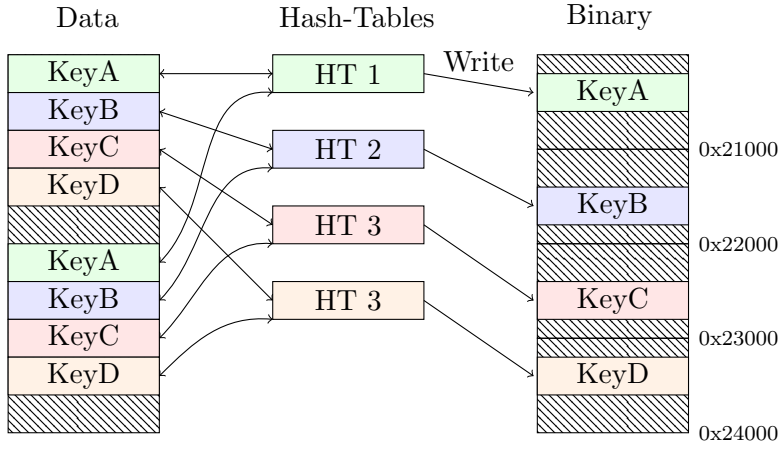
10.4.1. First-Come-First-Serve Data Placement

Lookup tables are frequently used to speed-up memory accesses and store constant data like locality strings. For the developer, it is not transparent how constants are stored in the compiled binary. Thus, even if the code seems to be placed in a cache line, *i.e.*, 64 B granularity, the compiler might reorder strings and add more spatial granularity between data. One optimization to reduce the binary size is to only populate the read-only data section if the compiler observes that only certain indices of a lookup table are accessed. If the developer uses a macro to dynamically populate a lookup table, e.g., with key mappings or similar, compilers do not insert all elements into the read-only section of the binary to reduce the binary size. Instead, the compilers use a first-come-first-serve data placement strategy to place the data in the read-only section. Figure 10.2 illustrates how data can be placed in `.rodata` section caused by this strategy.

10.4.2. Data Deduplication during Compilation



(a) Compiler



(b) Linker

Figure 10.3.: String deduplication in the compiler and linker causing spatial distance in `.rodata` section of the binary.

Another optimization facilitating cache attacks, also in combination with the first-come-first-serve data placement we just discussed, is data deduplication during compilation. Deduplicating strings can reduce the binary size significantly but also the memory resident size when running the program, as strings do not have to be kept in memory multiple times. Figure 10.3a demonstrates how string deduplication can introduce spa-

tial distance in sections of the binary, for instance, the `.rodata` section. C/C++ compilers deduplicate strings that occur more than once in the source code. Listing 10.1 illustrates a situation where string deduplication can be performed. Both the lookup table `mappings` and the function `string_funcA` contain the string `DataA`. The compiler traverses over the functions, and `DataB` is first inserted into the `.rodata` section. Again, data processed by the compiler (padding) could cause spatial distance between `DataA` and `DataB`. Before the compiler inserts `DataB` (`mappings[1]` in `string_funcB`), the compiler checks for duplicates and only points to the existing occurrence of `DataB` in the `.rodata` for all future usages. We evaluate Listing 10.1 for GCC and Clang. For Clang, we observe again for all optimizations levels the ordering `DataA.<64-byte-string>.DataB` in the `.rodata` section. For GCC, we observe the same result that for optimization levels O0/O1, both values are populated next to each other in the `.rodata` (`DataA.DataB`). For the other levels, the small strings are encoded as immediate values.

10.4.3. Deduplication in the linking step.

As we showed, string deduplication can cause spatial distance between strings and enable side-channel attacks in the compile step. For large software projects such as the Chromium project, it is important to merge strings also across object files. Since 2017, lld uses multiple hash tables to compensate some of the overhead caused by this link-time optimization by increasing concurrency using hash tables that can be accessed in parallel [51]. However, as there are multiple tables, inserting merged strings can cause a different layout for strings in the `.rodata` section than in the final linked binary. Figure 10.3b illustrates how the concurrent merging can lead to spatial distance in the final binary. With the highest optimization level of `lld` linker, *i.e.*, `-O2` [45], the linker merges duplicate substrings contained in larger strings. The smaller substring will be removed, and the tail of the larger string is used to index the substring. The security implications of string deduplication need to be considered in software projects since large spatial distance between secret dependent values, such as different key inputs, can lead to leakage of all user input, as we show in Section 10.5.

10.5. Evaluation and Exploitation Phase

In this section, we evaluate our templater on large binaries, such as browsers, that have not been targeted with templating attacks so far. We evaluate how well the templates work in the exploitation phase in terms of the attack F-Score. For the exploitation phase, we, the attacker, runs without privileges on a default configured system, with background activity (running e.g., browser, mail client, chat clients, music and video streaming, virus scanning, system updates running, etc.) leading to a realistic amount of system activity and noise. Overall we found that Flush+Reload is extremely noise-resilient, in line with previous works [32, 77]. We also focus on widespread Chromium-based products and demonstrate that they are susceptible to *LBTA*. We analyze the root cause for the leakage and show that it is caused by a compiler optimization. Table 10.1 lists all the evaluated applications, including the Chromium-based browsers and applications, Firefox, and LibreOffice Writer.

Templating of HTML form input fields Chrome. We first run our templating tool while generating keystrokes. We run our templater on an Intel i7-6700K with a fixed frequency of 4 GHz running Ubuntu 20.04 (kernel 5.4.0-40) on Chrome version 100.0.4896.60. To get more accurate results during the templating phase, we recommend dropping the active caches before executing the templater via procfs (`/proc/sys/vm/drop_caches`). Moreover, we blacklist file mappings from the `/usr/share/fonts/` as they lead to inconsistent results during the evaluation phase. Our templater traces 57 different key codes of a common `US_EN` keyboard in HTML password fields over the total size of memory mappings in Chrome of 209.81 MB (including the main binary and shared libraries). For each key code, we sample 20 times. On average, we observe a runtime of 1.47 hours ($n = 10, \sigma_{\overline{x}} = 0.33\%$) for 57 key codes, including the time for key classification. For a single key code, the runtime is 92 seconds. For comparison, the cache template attack implementation by Gruss et al. [32] takes 113.17 days to template the same files. Thus, with 1.47 hours *LBTA* speeds up the templating by a factor of 1848.

Leakage Source in Chrome.

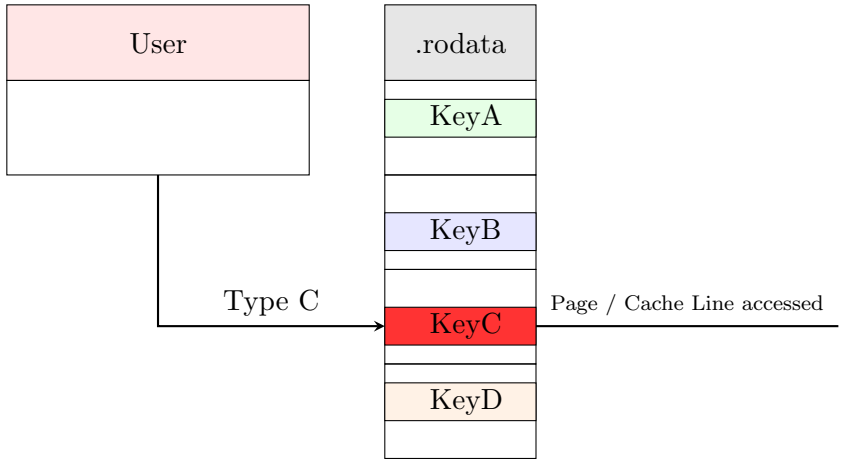


Figure 10.4.: Key code strings in the `.rodata` section introduce cache leakage.

As we discovered the page offsets related to the different keystrokes, we want to find the exact cache line causing the cache leakage. We extend our monitor with Flush+Reload to determine the cache line within the page. To speed up the templating time and obtain precise information on which cache line has the highest correlation, we disable most of the Intel prefetchers by writing the value `0xf` to MSR `0x1a4` [67], as otherwise multiple cache lines would have the highest correlation. We map the Chrome binary as shared memory and perform Flush+Reload on all mapped cache lines to determine the corresponding cache lines for each key. We analyze the Chrome binary and lookup the offsets causing the leakage for a specific keystroke. Each cache line causing the leakage of a certain character contains a string for the key event, e.g., "KeyA". We observe that all offsets lie in the read-only data (`.rodata` section of the binary. The leakage source are key-dependent accesses to the key code strings in the dom code table,[1] e.g., `DOM_CODE(0x070004, 0x001e, 0x0026, 0x001e, 0x0000, "KeyA", US_A);`. Figure 10.4 illustrates the leakage source for a user typing in a certain character and the corresponding DOM_CODE for the UI event. To verify if the leakage is related to string deduplication, we download the Chromium source, disable the string deduplication `-fno-merge-all-constants` and rebuild the Chromium browser. We still observe, that the single keystrokes are spread over multiple pages in the `.rodata` section, which can still be exploited by

[1] `https://source.chromium.org/chromium/chromium/src/+/main:ui/events/keycodes/dom/dom_code_data.inc`

the attacker despite the overheads in binary size and execution runtime caused by disabling the optimization. Hence, the compiler flag to disable string deduplication *does not fully close the side channel*. As a next step, we analyze the compiled object files after the build process. We observe that the created object file `keycode_converter.o` still contains all the key event strings adjacent to each other in the binary. This indicates that the linker introduces the spatial distance between key event strings. We perform a binary search on older Chrome binaries from a public Github repository containing archived Chrome Debian packages [71] to see when the spatial distance for key event strings was introduced. As a result, we observe that between version 63 and 64 of Chrome (year 2017), the single key event string was placed in the `.rodata` at different 4 kB pages. According to [51], the linker optimizations have been constantly improved since 2017. As discussed in Section 10.4, the parallelism in string deduplication can also cause spatial distance between key events. Disabling the string merging optimization is currently only possible by *disabling all optimizations* using optimization level `O0` for the linking with `-Wl,-O0`. This removes the spatial distance between the key event strings but comes with a substantial overhead as optimizations are disabled. At any higher optimization level, e.g., `-Wl,-O1`, the spatial distance reappears as strings are again deduplicated. This confirms that one of the effects we exploit is introduced by the linker. In comparison to state-of-the-art keyloggers on Linux like xkbcat [1], our keylogger does not rely on running as the same user within the same X-session. We verify this by running our keylogger as a different user and can still recover the keys from Chrome.

Keylogging in Chrome with Flush+Reload.

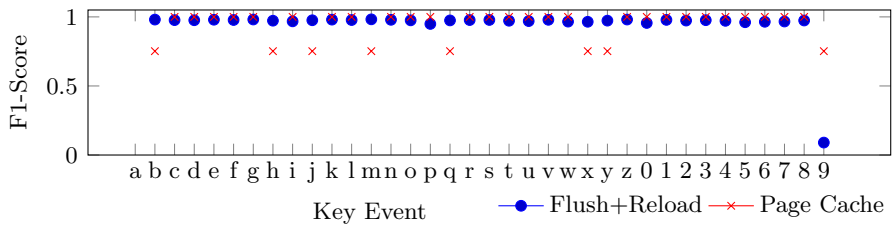


Figure 10.5.: F-Score per key using Flush+Reload and page cache attacks for all alphanumeric characters in Chrome.

	0	1	2	3	4	5	6	7	8	9
0x1521203	115	0	0	0	0	0	1	0	0	43
0x151b9bd	5	185	0	0	0	0	0	0	0	47
0x1513d05	0	0	165	0	0	0	0	0	0	53
0x1510118	2	0	0	178	0	0	0	0	0	45
0x150d59a	0	0	0	0	171	0	0	0	0	49
0x150b526	0	0	0	0	0	173	0	0	0	56
0x1509a35	0	0	0	0	0	0	156	0	0	51
0x1508991	0	0	0	0	0	0	0	169	0	58
0x1506eb8	0	0	0	0	0	0	0	0	165	52
0x1505e5b	0	0	0	0	0	0	0	0	0	140

Figure 10.6.: Cache-hit ratio using Flush+Reload for all digits letters in Chrome.

We run our monitor in three experiments for 180 seconds with all lowercase alphanumeric characters and observe cache activity for every single keystroke. The first experiment runs with fast user input with 1 ms between each keystroke. We count cache hits following a keystroke as true positives if they occur on the cache line that is correct according to our template and as false positive otherwise. To obtain the number of false positives, we run the monitor in a second experiment without performing any keystrokes in the input field, *i.e.*, idling. To complete our data on false negatives and true positives, we run the monitor in a third experiment while performing user input with 1 s between each keystroke. Over the total 540 second measurement time frame, we observed no false negatives. Figure 10.7 (Appendix) shows the cache-hit ratio for the cache lines detecting lowercase letters in Chrome. Figure 10.6 shows the cache-hit ratio for the cache lines detecting numeric digits in Chrome. As shown from Figure 10.6, the different digits can be highly-accurately classified. As can

be seen, the cache line accessed for digit 9 also contains other data that is constantly accessed by code handling other events in Chrome. Therefore, the cache line is constantly accessed also in an idle state and, in practice, cannot be used to spy on digit 9. From all the 36 alphanumeric keys, this is the only character where code or data is co-located with other (unrelated) frequently accessed code or data. The F-Score is the harmonic mean of precision and recall. Section 10.5 illustrates the F-Score for all alphanumeric characters. We also observed that a single keystroke causes up to three cache hits. These cache hits could be related to the window events `key_up`, `key_pressed` and `key_down`. To avoid printing the same character multiple times, a cache miss counter between the keystrokes can be used [32]. Note that multiple cache lines can be considered to further increase the accuracy of the keylogger [10, 41, 69].

Keylogging with the page cache. To demonstrate that the Chrome leakage is not specific to a certain CPU, we run our keylogger on Chrome version 99.0.4844.84. Our test device runs Ubuntu 20.04 (kernel 5.18.0-051800-generic), equipped with an AMD Ryzen 5 2600X CPU, 16 GB of RAM, and a Samsung 970 EVO NVME SSD. We circumvent the read-around and read-ahead optimization, as explained in Section 10.3.5. The keylogger uses the keystroke template for the main Chrome binary and monitors the page cache utilization for the corresponding pages using the `preadv2` syscall. It then reports the detected activity as keystrokes and subsequently evicts the page cache. While the page cache attack using the `mincore` syscall was able to observe keystrokes on a fine temporal granularity, we observe that using `preadv2` comes with practical limitations. In particular, with large eviction set sizes, guessed by the attacker, we conclude that only very slow keyboard interaction with gaps of 2 s and more can be observed. However, our evaluation of the page-cache side channel is generic and would also apply to scenario where the `mincore` syscall is available, which allowed fast and non-destructive continuous probing.

Based on the page cache accesses, we compute the page-hit ratio for Chrome over the page cache. Figure 10.7 (Appendix) shows the page-hit ratio for the page cache detecting alphanumeric letters in Chrome. For the Chrome version, we observe that the characters `b,m,9,h,y,x` are grouped and cannot be uniquely distinguished. We again perform the experiment in three phases to determine true positive, false positive, and false negative rate by simulating fast, slow, and no user input. Section 10.5 shows the F-Score for all alphanumeric characters running the page cache

attack. While most characters have very high F-Scores, the character group `b,m,9,h,y,x` has a lower F-Score due to false positives when other keys are pressed. Also, same as in the Flush+Reload attack, the character `9` suffers from a high number of false positives, negatively impacting the F-Score.

Electron. As we observed the leakage of keystrokes within the Chrome binary, we further analyzed Chromium-based applications like the Electron framework. As Chromium-based applications largely use the same keystroke handling code, we can directly scan the `.rodata` section for the keystroke offsets. We evaluate the templates on Chromium, Threema, Passky, VS-Code, Mattermost, Discord and observe similar leakage rates to Chrome with F-Scores of at least 85 %. Table 10.1 contains the F-Scores for the different applications. Based on these clear results, we deduce that, in principle, all Electron applications are susceptible to *LBTA* and cache-based keylogging attacks.

Chromium Embedded Framework. The Chromium Embedded Framework (CEF) is widely used and another interesting target for *LBTA*. While Electron directly uses the Chromium API, CEF tries to hide the details of the Chromium API [24]. CEF is actively run on more than 100 million devices [12]. We target Spotify, and the Brackets editor application, which are both based on CEF. To attack a CEF application, an attacker needs to read out the `.rodata` section from the shared library `libcef.so`. We run our monitor again with Flush+Reload and observe an F-Score of 96 % over the lowercase alphanumeric characters. For Brackets (1.5.0), we observe, that the `libcef.so` was built with an older linker version as the different key-event related strings for the lowercase alphanumeric characters are co-located in three different cache lines. Therefore, we consider all CEF applications to be susceptible to cache templating in principle. We observe an F-Score of 94 % for detecting key events. However, we also observe that hardware prefetching practically thwarts the distinction of different blocks in this scenario more than in the other attack scenarios, leaving only inter-keystroke timing attacks as an option for the attack phase.

Firefox. Firefox uses a different build system where optimizations such as data deduplication may still apply but with slightly different behavior than with LLVM/clang. Therefore, we templated Firefox and found cache activity for each keystroke in the `libxul.so` library (offset: 0x332d000). However, we did not find leakage to distinguish keys. However, an attacker can still determine whether a user is typing and perform an inter-keystroke

timing attack [20, 32, 50, 61, 79] to recover the keystrokes. The accuracy we observed for such an attack is 96 %.

LibreOffice Writer. We profile the LibreOffice Writer version 6.4.2 on our Linux setup. Our profiler shows that the library `libQt5XcbQpa-.so.5.12.8` (offset: 0x51000) offset reveals cache activity on all letters but no digits. The library `libswlo.so` (0x53e000) leaks keystrokes reliably with an F-Score of 1.

Chrome on Windows. Chrome on Windows is built with a different compiler and linker. Therefore, we tested Chrome versions 103.0.5060.53 and 114 on an Intel i5-4300U notebook running Windows 10 (1803, 17134.1726). We use the `LoadLibrary` function create read-only shared mappings with victim applications. We observe that in the `chrome.dll` (offset: 0xa4ee000) the different key values are co-located instead of having a spatial distance of multiple 4 kB pages. With our Flush+Reload cache monitor we are able to observe all key presses and distinguish presses in the key groups A-F, G-S, T-Z and 0-4, and 5-9, with an F-Score of 99 %. However, due to prefetching we can only monitor a single key group at a time. We also found user input leakage on many other locations, e.g., `msctf.dll` (0x45000), and `imm32.dll` (0x3000).

Search bar. Templating user queries in the browser would tremendously reduce the privacy of browsers. Running the templater on the search bar of Chrome 103.0.5060.53 revealed that the search bar uses a different method to load the keys and there is only a single page (offset: 0x91d4000) in Chrome with cache activity upon keystrokes. Based on our results, we conclude that the search bar does not use the same internal structures for key events as HTML input data. Still, the leakage we discovered enables inter-keystroke timing attacks on keystrokes. Running the profiling experiment with all alphanumeric, we achieve an F-Score of 99 % for detecting key presses.

10.6. Mitigation and Discussion

Different mitigation vectors could prevent either *LBTA* or the underlying leakage utilized in the exploitation phase, albeit at a significant performance and usability cost. We identified five conditions for an attack to succeed:

Table 10.1.: Evaluated applications. Page cache (PC) and cache line (CL) indicate whether precise keystroke attacks are possible on that granularity. Inter-Keystroke Timing (IK) indicates that key events can be detected on the application via Flush+Reload or the page cache.

Name	Category	CL	PC	IK (key groups)	Avg. F-Score (Flush+Reload)
Chrome (99.0.4844.84)	Browser	✓	✓	✓	94 %
Signal-Desktop (5.46.0)	Private Messenger	✓	✓	✓	98 %
Threema (2.4.1)	Private Messenger	✓	✓	✓	84 %
Passky (7.0.0)	Password Manager	✓	✓	✓	99 %
VS-Code (1.69.1)	Editor	✓	✓	✓	85 %
Chromium Browser (103.0.5060.114)	Browser	✓	✓	✓	99 %
Mattermost-Desktop (5.1.1)	Collaboration Platform	✓	✓	✓	94 %
Discord (0.0.18)	Text and Voice Chat	✓	✓	✓	98 %
Spotify (1.1.84.716)	Audio Streaming	✓	✓	✓	96 %
Brackets (1.2.1)	Editor	✘	✘	✓	94 %
Chrome 103.0.5060.134(Windows)	Browser	✘	✘	✓	99 %
Chrome 103.0.5060.53 (Search Bar)	Browser	✘	✘	✓	99 %
libxul.so (Firefox 102)	Browser	✘	✘	✓	99 %
LibreOffice Writer (6.4.2)	Office Software	✘	✘	✓	99 %

Golden device availability Templating attacks consist of two phases. In the templating phase, the attacker uses a setup that is similar to the victim system [9, 13, 32]. This is trivial for cache attacks on most desktop and laptop processors, as they are virtually identical in terms of attacks like Flush+Reload (*i.e.*, the processor has cache lines and eviction or flushing of these is possible). Software diversity [18], in principle, could break the link between templating and exploitation, but is not widely used. Thus, in practice, the vast majority of users runs binaries obtained from the official repositories or websites, making it trivial to create templates for them. Furthermore, even with software diversity, once the attacker knows what the target byte sequences (e.g., strings) in the binary are, the attacker can simply search for these on the victim system (without the need for templating again) and attack the victim binary in the same way again. Hence, we also consider software diversity no mitigation to *LBTA*.

Disable Compiler and Linker Optimizations For the Chromium example, disabling the linker optimizations (deduplication and spatial distancing) would reduce the accurate keylogging to inter-keystroke timings for key groups in 4 different cache lines. However, this may still enable inferring user input accurately [61]. On the negative side, removing these optimizations typically increases binary sizes and cache utilization due to runtime use of duplicated data. Note that this type of deduplication and spatial distancing is introduced on the compiler and linker level, which is completely transparent to the OS. While the OS could dynamically rewrite

binary pages at runtime to counteract this behavior, this would introduce huge amounts of complexity, overhead, and the potential for unhandled corner cases. Instead, the Chromium team opted for a compiler- and linker workaround, which triggers the string placement explicitly by placing and initializing dummy data structures such that the current compiler and linker versions do not spatially separate the secret data. However, this approach is fragile as it depends on the specific behavior of the compiler.

Secret-dependent execution For cryptographic code, the state of the art against side channels is the linearization to so-called constant-time code, *i.e.*, constant code and data accesses, regardless of the secrets, albeit with a considerable performance cost [15]. For general purpose code, always running all the code and accessing all the data is infeasible. Different works linearized the control flow of general purpose code [5, 21, 57] and observed a prohibitively high runtime overhead for realistic workloads. Hence, the problem of secret dependency on user input in large applications remains an open problem.

Side-channel observability Tools like CacheAudit [22] or CaSym [8] follow the cryptography-focused notion of constant time to consider an application leakage-free. However, in practice, distinguishing keys may be infeasible for an unprivileged attacker when key-dependent execution exists but does not cross, e.g., page or cache-line boundaries, depending on the side channel. In particular, within a page, the hardware prefetcher is a substantial obstacle introducing spurious cache activity on the target cache lines, foiling exploitation in practice [32]. The compiler could utilize this effect by grouping potentially secret-dependent accesses, minimizing the number of cache lines data structures are spread across, and placing strings interleaved with frequently used code or data.

Noise resilience Since user input cannot be triggered and repeated by the attacker millions of times, noise resilience is also one condition. Hence, inducing noise, unsuitable to secure cryptographic operations, can provide strong security guarantees for user input [56]. A low number of memory accesses could substantially limit the presented attacks, especially if user annotations of potentially secret data tell the compiler where to add these accesses.

LBTA is also interesting as a defensive technique revealing leakage as part of a continuous integration pipeline [74], revealing leakage that is not or not to the actual extent visible to developers on the source level, but only in the binary due to compiler and linker optimizations introducing

these spatial distances. Moreover, languages like JavaScript, Java, PHP, and Python also perform string deduplication (under the term 'string interning') to reduce memory utilization, potentially leading to similar effects.

We demonstrated that keystrokes in form input fields in Chrome can be detected using cache attacks on hardware and software caches. While Chrome is a valuable target, the dependency of many frameworks on the Chromium project, such as CEF and Electron, leads to a significantly higher impact as browser-based desktop applications, e.g., using the popular Electron framework [23], are susceptible to accurate keylogging with our attack.

10.7. Conclusion

First-come-first-serve data placement and data deduplication during compilation and linking facilitate side-channel leakage in compiled binaries. We show that this effect can even induce side-channel leakage where, without these optimizations, no secret-dependent accesses cross a 64-byte boundary. The foundation to discover this attack was our extension to cache template attacks, called Layered Binary Templating Attacks, *LBTA*. *LBTA* is a scalable approach to templating that combines spatial information from multiple side channels. Using *LBTA* we scan binaries compiled with LLVM/clang, which applies first-come-first-serve data placement and deduplication by default. Our end-to-end attack is an unprivileged cache-based keylogger for all Chrome-based / Electron-based applications, including many security-critical apps, e.g., the popular Signal messenger app. While mitigation strategies exist, they come at a cost, and further research is necessary to overcome the open problem of side-channel attacks on user input.

Acknowledgments

We want to thank our anonymous reviewers for valueable feedback on the draft. This work was supported by a generous gift from Red Hat Research. We want to thank Hanna Müller, Claudio Canella, Michael Schwarz and Moritz Lipp for valuable feedback. Any opinions or recommendations

expressed are those of the authors and do not necessarily reflect the views of the funding parties.

Appendix

A. Cache-hit ratios (extended)

The cache hit ratio for all lowercase characters with Flush+Reload can be seen with Figure 10.8 and all alphanumeric characters for the page cache attack Figure 10.7.

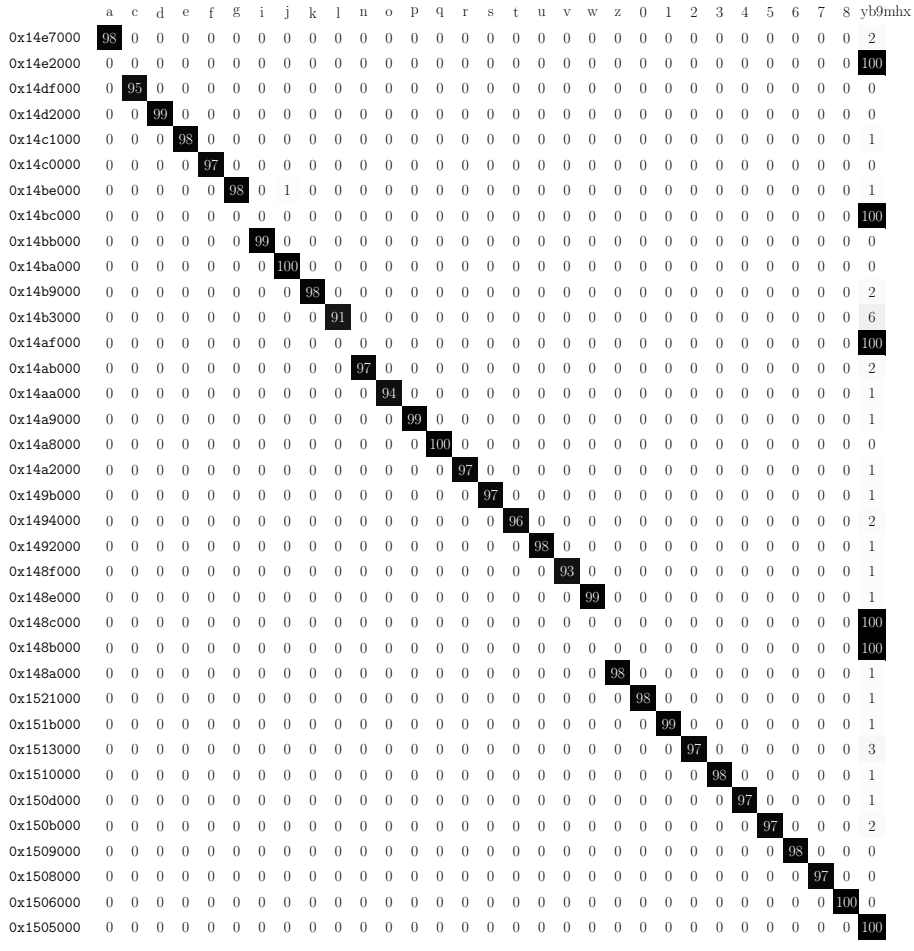


Figure 10.7.: Cache-hit ratio using a page cache attack for alphanumeric characters in Chrome.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0x14e7e5b	99	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	2	0	0	2	0	0	8	7	0	0
0x14e24a8	2	178	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14df043	0	0	151	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14d2ab9	0	0	0	173	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14c1a5e	0	1	0	0	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14c0a76	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14be05f	0	0	0	0	0	0	96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0x14bc5ad	0	0	0	0	0	0	0	169	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14bb06c	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14bae3b	0	0	0	0	0	0	0	0	0	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14b981f	0	1	0	0	0	0	0	0	0	0	171	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14b3350	0	0	0	0	0	0	0	0	0	0	0	146	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14af573	0	1	0	0	0	0	0	0	0	0	0	0	125	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14ab755	0	0	0	0	0	0	0	0	0	0	0	0	0	148	0	0	0	0	0	0	0	1	0	0	0	0
0x14aa938	0	0	0	0	0	0	0	0	0	0	0	0	0	0	165	0	0	0	0	0	0	0	0	0	0	0
0x14a9172	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	87	0	0	0	0	0	0	0	0	0	0
0x14a8e1c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	156	0	0	0	0	0	0	0	0	0
0x14a2f87	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	141	0	0	0	0	0	0	0	0
0x149b36d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	108	0	0	0	0	0	0	0
0x149411c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	157	0	0	0	0	0	0
0x1492c1d	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	135	0	0	0	0	0
0x148f151	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107	0	0	0	0
0x148e546	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	124	0	0	0
0x148cb27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	138	0	0
0x148b08b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	137	0
0x148ac19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	141

Figure 10.8.: Cache-hit ratio using Flush+Reload for lowercase letters in Chrome.

References

[1] Antti Korpi. *xkbcat*. 2021. URL: `https://github.com/anko/xkbcat`.

[2] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. "DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels." In: *FAST*. 2022.

[3] Maarten Baert. *wayland-keylogger*. 2022. URL: `https://github.com/Aishou/wayland-keylogger`.

[4] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[5] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization." In: *CCS*. 2021.

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

[7] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. "JIT Leaks: inducing timing side channels through just-in-time compilation." In: *S&P*. 2020.

[8] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. "CaSym: Cache aware symbolic execution for side channel detection and mitigation." In: *S&P*. 2019.

[9] Billy Brumley and Risto Hakala. "Cache-Timing Template Attacks." In: *AsiaCrypt*. 2009.

[10] Sebastien Carre, Victor Dyseryn, Adrien Facon, Sylvain Guilley, and Thomas Perianin. "End-to-end automated cache-timing attack driven by Machine Learning." In: *Journal of Cryptology* (2019).

[11] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. "FaCT: A flexible, constant-time programming language." In: *SecDev*. 2017.

[12] CEF. *Chrome Embedded Framework*. 2022. URL: `https://github.com/chromiumembedded/cef`.

[13] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. "Template attacks." In: *CHES*. 2002.

[14] Chromium. *Speeding up Chrome's release cycle*. 2022. URL: `https://blog.chromium.org/2021/03/speeding-up-release-cycle.html`.

[15] Szu-Chi Chung, Jen-Wei Lee, Hsie-Chia Chang, and Chen-Yi Lee. "A high-performance elliptic curve cryptographic processor over GF(p) with SPA resistance." In: *International Symposium on Circuits and Systems (ISCAS)*. 2012.

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

[17] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. "On the effectiveness of same-domain memory deduplication." In: *European Workshop on Systems Security*. 2022, pp. 29–35.

[18] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity." In: *NDSS*. 2015.

[19] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks." In: *CHES*. 2018.

[20] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis." In: *S&P*. 2016.

[21] Christopher Domas. *M/o/Vfuscator*. 2015. URL: `https://github.com/xoreaxeaxeax/movfuscator`.

[22] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. "CacheAudit: A Tool for the Static Analysis of Cache Side Channels." In: *USENIX Security Symposium*. 2013.

[23] Electron. *Electron Apps*. 2022. URL: `https://www.electronjs.org/apps`.

[24] Electron JS. *Electron Internals: Building Chromium as a Library*. 2022. URL: `https://www.electronjs.org/blog/electron-internals-building-chromium-as-a-library`.

[25] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. "SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults." In: *RAID*. 2017.

[26] Cesar Pereida García and Billy Bob Brumley. "Constant-Time Callees with Variable-Time Callers." In: *USENIX Security Symposium*. 2017.

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

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

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

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

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

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

[33] halolinux. *Page Cache Readahead*. 2022. URL: `https://www.halolinux.us/kernel-architecture/page-cache-readahead.html`.

[34] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. "Side channels in cloud services, the case of deduplication in cloud storage." In: *IEEE Security & Privacy* 6 (2010).

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

[36] John Richard Moser. *Optimizing Linker Load Times*. 2006. URL: `https://lwn.net/Articles/192624/`.

[37] Jonathan Corbet. *Fixing page-cache side channels, second attempt*. 2019. URL: `https://lwn.net/Articles/778437/`.

[38] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. "DupLESS: Server-Aided Encryption for Deduplicated Storage." In: *USENIX Security Symposium*. 2013.

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

[40] Guanlin Li, Chang Liu, Han Yu, Yanhong Fan, Libang Zhang, Zongyue Wang, and Meiqin Wang. "SCNet: A Neural Network for Automated Side-Channel Attack." In: *arXiv:2008.00476* (2020).

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

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

[43] Marcel Medwed and Elisabeth Oswald. "Template attacks on ECDSA." In: *WISA*. Springer. 2008.

[44] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX amplifies the power of cache attacks." In: *CHES*. 2017.

[45] nxmnpg.lemoda. *Manual Pages - LD.LLD*. 2022. URL: `https://nxmnpg.lemoda.net/1/ld.lld`.

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

[47] Dan Page. "A note on side-channels resulting from dynamic compilation." In: *Cryptology ePrint archive, Report 2006/349* (2006).

[48] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution." In: *USENIX Security Symposium*. 2015.

[49] Christian Rechberger and Elisabeth Oswald. "Practical template attacks." In: *WISA*. 2004.

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

[51] Rui Ueyama. *lld: A Fast, Simple and Portable Linker*. 2017. URL: `https://llvm.org/devmtg/2017-10/slides/Ueyama-lld.pdf`.

[52] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows internals*. Pearson Education, 2012.

[53] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion." In: *ASPLOS*. 2021.

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

[55] Michael Schwarz, Moritz Lipp, and Claudio Canella. *misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8*. 2018. URL: https://github.com/misc0110/PTEditor.

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

[57] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. "Specfuscator: Evaluating Branch Removal as a Spectre Mitigation." In: *FC*. 2021.

[58] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. "Remote Page Deduplication Attacks." In: *NDSS*. 2022.

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

[60] Laurent Simon, David Chisnall, and Ross Anderson. "What you get is what you C: Controlling side effects in mainstream C compilers." In: *EuroS&P*. 2018.

[61] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH." In: *USENIX Security Symposium*. 2001.

[62] statcounter Global Stats. *Browser Market Share Worldwide*. 2022. URL: https://gs.statcounter.com/.

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

[64] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution." In: *USENIX Security Symposium*. 2017.

[65] Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere. "Adaptive compiler strategies for mitigating timing side channel attacks." In: *TDSC* (2017).

[66] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think." In: *USENIX Security Symposium*. 2018.

[67] Vish Viswanathan. *Disclosure of Hardware Prefetcher Control on Some Intel Processors*. 2014. URL: https://web.archive.org/web/20160304031330/https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[68] Ahsan Wajahat, Azhar Imran, Jahanzaib Latif, Ahsan Nazir, and Anas Bilal. "A Novel Approach of Unprivileged Keylogger Detection." In: *iCoMET*. 2019.

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

[70] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. "CacheD: Identifying Cache-Based Timing Channels in Production Software." In: *USENIX*. 2017.

[71] Webnicer Ltd. *chrome-downloads*. 2022. URL: https://github.com/webnicer/chrome-downloads/.

[72] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. "Single Trace Attack Against RSA Key Generation in Intel SGX SSL." In: *AsiaCCS*. 2018.

[73] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries." In: *ACSAC*. 2018.

[74] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. "Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications." In: *arXiv preprint arXiv:2208.14942* (2022).

[75] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *S&P*. 2015.

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

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

[78] Yuanyuan Yuan, Qi Pang, and Shuai Wang. "Automated Side Channel Analysis of Media Software with Manifold Learning." In: *arXiv preprint arXiv:2112.04947* (2021).

[79] Kehuan Zhang and XiaoFeng Wang. "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems." In: *USENIX Security Symposium*. 2009.