Andreas Kogler

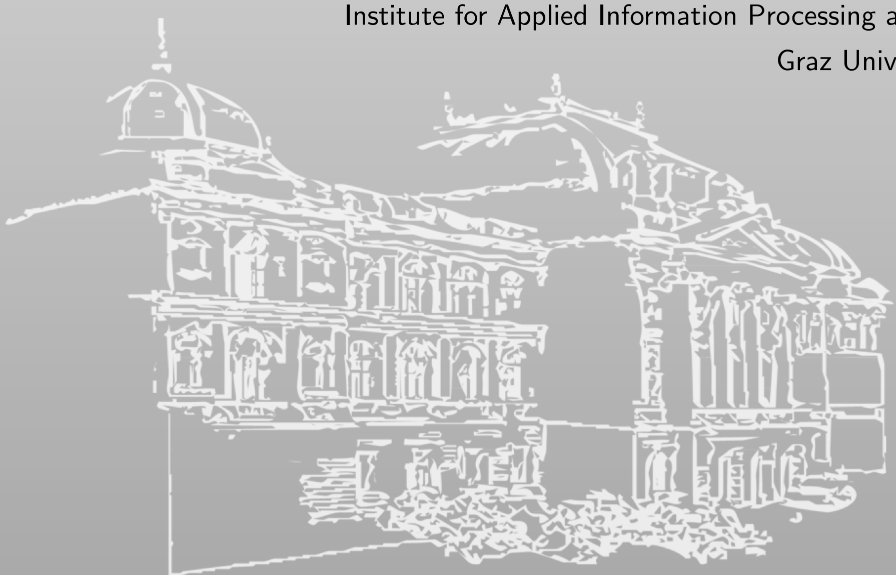
Colliding Worlds: Exploiting Physical Properties from Software

PhD Thesis

Assessors: Daniel Gruss, Thorsten Holz

August 2024

Institute for Applied Information Processing and Communications
Graz University of Technology



SCIENCE · PASSION · TECHNOLOGY

Abstract

Cloud computing is essential to increase resource efficiency. However, when utilizing cloud resources, the underlying hardware becomes implicitly shared among tenants, enabling side-channel and transient-execution attacks. Consequently, research focused on preventing such attacks, and modern hardware is no longer affected by the infamous Meltdown and MDS attacks. However, the effects of hardware sharing on physical properties remain unexplored. Furthermore, CPUs provide interfaces to control and monitor physical properties such as voltage, frequency, and power consumption. These interfaces are crucial for dynamically adjusting the hardware's operating point based on the workload. The influences of complex power management and the accessibility of system interfaces enable the shift of traditional hardware attacks from well-studied attacks on embedded devices to software-only attacks targeting systems remotely.

In this thesis, we explore the synergy of traditional hardware attacks with microarchitectural components and software interfaces. We provide an overview of the state of the art in software-based fault attacks, software-based power analysis, and transient-execution attacks. Furthermore, we extend the state of the art by demonstrating that sharing underlying hardware in the post-transient era can still enable Meltdown-style and MDS-style attack scenarios by observing physical properties. We exploit transient execution to enhance software-based fault attacks and present a novel combination of physical effects to circumvent commodity mitigations. We analyze how physical faults manifest in hardware and propose an optimized software-based fault detection mechanism. Finally, we analyze undocumented signals and configuration options, enabling novel attacks and defenses. This work highlights that *colliding* the world of hardware attacks with system interfaces and microarchitectural elements opens up a new attack landscape, reachable even purely from software.

This thesis is split into two parts. The first part outlines the contributions, provides background, and summarizes the state of the art. The second part presents my first-authored papers in their original form[1]. All of these papers were accepted at renowned international tier 1 security conferences.

[1]The two-column layout and figures of the camera-ready versions were adapted to fit the single-column format. The original content of the publications is unmodified.

Acknowledgements

First and foremost, I want to thank you, Daniel Gruss, for offering me the opportunity to join your group and pursue a PhD. Thank you for the support during my PhD, especially the chance to present my research internationally, the freedom you gave me with my research topics, and the opportunity to strengthen my international relations. Your guidance on improving my writing skills has been essential for getting the papers accepted at top-tier conferences. I wish you and the next generation of PhD students the best for the future in both social and work aspects.

Thank you, Michael Schwarz, for supporting me as a mentor, colleague, and friend over the past ten years. I still remember one of the first times we met when you explained pointers using magic wands. You supported me during my PhD and encouraged me to apply for a tenure-track position, which made me speechless when I received an offer. This thesis would not have been possible without you inspiring me to follow in your footsteps.

Thank you, Thorsten Holz, for your valuable feedback and taking the time to assess this thesis. During the conferences and my visits to CISPA, I have always admired your professionalism and calmness, which made our discussions with you and your students a social and technical highlight.

Thank you, Stefan Mangard, for the countless discussions on research ideas for software-based power analysis. Over the past years, you have provided me with valuable guidance on both social and career aspects.

Thank you, Moritz Lipp, for becoming a close friend and always being there for technical and social support. During my PhD, you were a social and humorous anchor, always offering guidance, tips, and honey.

Thank you, Jo Van Bulck, for being such an open person to talk to and adopting me during my first academic conference with your colleagues and friends. I enjoyed the technical discussions and ideas we shared.

Thank you, Lukas Giner, for being an incredible office colleague and friend. Our countless technical, social, and political discussions were a highlight. We managed to balance humor, professionalism, and the occasional "*brrt*".

Thank you, Martin Schwarzl, for being a close friend even before the PhD. I had tremendous fun participating in coding contests and discussing topics with you. Your impressions encouraged me to pursue my PhD.

Thank you, Claudio Canella, for our incredible time together in the office. You encouraged me to join the PhD and became a close friend with countless funny moments, like my frequent coffee-spilling accidents.

Thank you, Pietro Borrello, for your support in technical and career aspects. Although your visit to Graz was brief, and you never missed a chance to critique our food (often rightfully), we had tremendous fun during your stay. The Black Hat talks with you were an incredible experience.

Thank you, Jonas Juffinger, for our fantastic time in the office, at conferences, and during our talks. I want to highlight one of your funny improvisations during our Black Hat Europe talk "... because someone forgot to bring them" which is one of my favorite moments during a talk.

I want to thank my colleagues and friends. Fabian Rauscher, Stefan Gast, Catherine Easdon, Roland Czerny, Hannes Weissteiner, and Sudheendra Neela, thank you for our discussions and brainstorming sessions. Lukas Lamster, Moritz Waser, Robert Primas, and Lea Zinkanell, thank you for the balance between our technical discussions and our bouldering evenings. Anders Fogh, Antoon Purnal, Barbara Gigerl, Daniel Moghimi, David Schrammel, Fritz Alder, Lena Heimberger, Lukas Maar, Maria Eichlseder, Martin Unterguggenberger, Marton Bognar, Mathias Oberhuber, Moritz Schlögel, Robert Schilling, Rishub Nagpal, Roman Walch, and Vedad Hadzic, thank you for all the moments and conversations. Daniel Weber, Lukas Gerlach, Tobias Ebelshäuser, Leon Trampert, Sven Bugiel, Cas Cremers, Ruiyi Zhang, Lorenz Hetterich, and Fabian Thomas, thank you for welcoming me during my visits to CISPA and for all the conversations. Thank you to all the unnamed people and friends I met during this journey.

I want to thank my parents and family. Thank you, Mom, for supporting me on this path and for the time you spent to provide me with the best education and environment. Thank you, Dad, for sparking my interest in technical aspects, which encouraged me to pursue this path. Thank you, Wolfgang, for the weekends we spent together during my youth, showing me your computers and how to play games on them. I want to thank my brother, sister, aunts, uncles, and relatives for their support.

Finally, I want to thank all my close friends, the friends I made, and those whose paths are no longer shared with mine. Without you, this thesis would not have been possible. — Thank you!

Andreas Kogler

Contents

Part I.

Colliding Worlds: Exploiting Physical Properties from Software

1 Introduction

"Intelligence is the ability of a living creature to perform pointless or unnatural acts."

- Roadside Picnic

The increasing reliance on cloud computing to improve resource efficiency leads to privacy-related data being processed on machines owned by companies rather than users. These companies often share the machines across multiple tenants. Technologies that enforce integrity and confidentiality are crucial to isolate and protect privacy-related data. However, due to the sharing of the underlying hardware, the attack surface for side-channel, transient-execution, and fault attacks has substantially grown.

Side-channel attacks exploit the emission of information over an unintended side effect of the implementation. Early discussions of covertly transmitting data over such side effects date back to 1973 [137]. Moskowitz et al. [166] formally describe a timing channel already in 1990. With the evolution of our computation devices, side-channel attacks have become more related to the actual implementation and design of the hardware. Hu [87] describes a covert channel via CPU caches in 1992. Kocher [125] mounted the first *practical* timing side-channel attack on RSA in 1996, effectively exploiting the timing differences in the modular exponentiation algorithm. An important step in generalizing timing side channels was to move beyond application-specific timing leakage by transitioning the observation of timing differences from the actual implementation toward CPU caches. Timing side channels on caches are referred to as *cache side-channel attacks* and exploit the timing difference between cached and uncached data. Cache side-channel attacks are demonstrated in numerous attacks targeting cryptographic primitives [248, 247, 219, 3, 255, 195, 20, 90], inferring user behavior [145, 76], and targeting trusted execution environments [69, 24, 161, 46]. Intel provides guidelines to prevent code patterns that result in timing and cache side-channel attacks [93]. Cache attacks can be automated to some extent [77, 27], but they are fundamentally application-dependent, and arbitrary data leakage is not guaranteed.

Side channels were essential in encoding secret information in transient-execution attacks like Meltdown [147] and Spectre [126]. These transient-execution attacks abuse the side effects of wrongly executed instructions due to misspeculation or faults. Overall, the field of transient-execution attacks was systematized into Meltdown-type attacks [147, 222, 236, 33] and Spectre-type attacks [126, 33, 153]. The Microarchitectural Data Sampling (MDS) attacks [198, 201, 30, 187, 33] emerged within the class of Meltdown-type attacks. Contrary to the original Meltdown attack, MDS targets data that is in flight, i.e., short-lived data within buffers. Numerous mitigations were proposed against transient-execution attacks [80, 71], where most Meltdown-type attacks are now fixed in hardware. However, Spectre-type attacks are still fundamentally a threat depending on the interfaces and interaction of security domains within the system. Although the fixes for Meltdown-type attacks prevent transient forwarding of data from different domains, the influences of sharing the underlying hardware in combination with traditional power side channels remains unexplored.

Traditional hardware side-channel attacks observe power consumption or electromagnetic emissions instead of timing behavior [127, 25, 128, 154]. While timing side channels are extensively studied on desktop or server systems, these traditional hardware side channels mainly target embedded devices or small co-processors [154]. Lipp et al. [146] demonstrate that traditional power-analysis attacks are feasible on modern desktop and server processors, with the Platypus attack, replacing the usually required external measurement equipment with a CPU internal measurement interface. Due to the reduced signal quality, the attacks require more traces resulting in a longer attack runtime. The interface was modified to further reduce signal quality, and unprivileged access was restricted to prevent the Platypus attack. However, Wang et al. [234] and Liu et al. [148] show that there are other indirect ways to observe power-related signals, i.e., signals that correlate with the device's power consumption. Nevertheless, these instances of software-based power side-channel attacks target application-dependent leakage, similar to the first generation timing side channels, leaving the generalization of software-based power analysis to an application-independent, arbitrary data leakage primitive unexplored.

Fault attacks induce errors in computations (data-in-use) or data stored in memory (data-at-rest). In line with traditional power side-channel attacks, these attacks went through a similar transition from well-studied attacks on small form factor devices towards software-only attacks. The Rowhammer attack [123] demonstrates how to fault data-at-rest in the main memory by frequently accessing data residing near the targeted data without any physical access required. Due to the potential security risk of such an attack, refresh-based mitigations and Error Correcting Codes (ECC) were developed and integrated into all modern memory modules. However, the number of faults drastically increased with increasing memory density [122] overwhelming the correction capabilities of ECC [44]. Furthermore, the early refresh-based mitigations had limited resources to track potential attacks and were overwhelmed when combining multiple attacks at once [108, 59]. Contrary to Rowhammer, software-based undervolting attacks usually require CPU interfaces to either request a reduced target operating voltage or a higher CPU frequency (resulting in a similar effect). The CLKSCREW [212], Plundervolt [167], VoltJockey [183, 184], and V0LTpwn [120] attacks demonstrate that such interfaces enable fault attacks on data-in-use within trusted execution environments. Furthermore, the VoltPillager attack [39] shows how to inject commands directly over a bus interface to the voltage regulators given physical access to the machine. In response, the vendors restricted the interface and isolated the voltage regulators within the CPU package. However, these changes restrict potential energy and efficiency gains due to undervolting [111].

In this thesis, we further close the gap between traditional hardware and software-only attacks and advance state-of-the-art software-based power-analysis attacks beyond application-dependent leakage. Furthermore, we find new attack vectors and defenses for software-based fault attacks and explore undocumented interfaces of modern CPUs. Figure 1.1 shows the peer-reviewed papers I first- and co-authored during my PhD. The targeted field is on the x-axis and y-axis indicates if the paper is defensive- or offensive-oriented. Bold papers are the main contributions of this thesis.

First, we show an alternative probabilistic approach to prevent dynamic frequency- and voltage-scaling faults inside trusted execution environments. We analyze the fault susceptibility of x86 instructions across multiple operating points and CPUs. This analysis is the foundation for understanding how this class of attacks manifests. We discover that integer multiplication is the most susceptible instruction across all tested instructions and machines. With this insight, we design a probabilistic mitigation

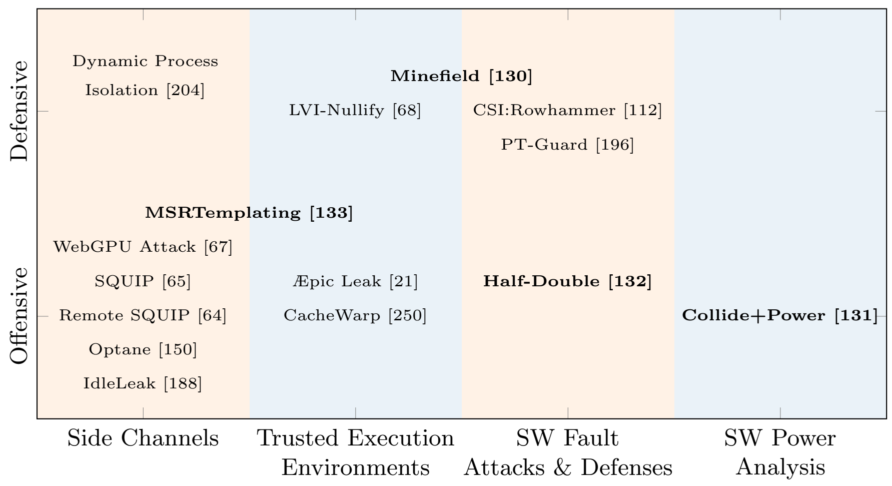


Figure 1.1.: The peer-reviewed first- and co-authored papers of this thesis. The papers are ordered by offensive and defense categories, and the positioning indicates the field. Papers in bold are the main contributions.

that places so-called trap instructions within the program, which are constantly validated to ensure correct computation results. We design the compiler infrastructure to automatically harden SGX enclaves with our defense and evaluate the effectiveness and performance of our mitigation. Finally, the insights of our analysis are the foundation of follow-up work to improve energy efficiency [111]. Second, we present Half-Double, a new Rowhammer variant that breaks locality assumptions of existing real-world mitigations. We carefully evaluate the Half-Double effect and show that Half-Double exploits the combined disturbance errors of neighboring rows beyond distance-1 neighbors. To analyze the real-world impact of this new variant, we test a large variety of devices and showcase an end-to-end exploit to gain root privileges on a Chromebook tablet. Third, CPU interfaces are an essential building block to mount traditional hardware attacks purely from software, as seen in the Platypus [146] and Plundervolt [167] attacks. Unfortunately, not all CPU interfaces are documented, and the search space for undocumented features is too large for exhaustive analysis. We design a framework that scans this configuration space and can identify the influences of configuration bits on instruction groups. Furthermore, we correlate registers that expose continuous signals with existing known signal sources to find potential alternatives if a given signal is restricted. Our case studies show multiple configuration registers and bits that enable new attacks and defenses. Finally, newer CPU generations deploy hardware mitigations for Meltdown-type attacks. However, we show that although these mitigations prevent accidental transient forwarding, sharing the underlying hardware between security domains still exposes leakage in the power domain. We generalize power side channels beyond application-dependent leakage by targeting the combined power leakage of data from different domains within CPU caches instead of application-specific leakage, mimicking a similar evolution from timing side channels towards cache attacks. We present Meltdown-Power and MDS-Power as attack variants with threat models similar to Meltdown and MDS.

1.1. Main Contributions

This section introduces the first-authored papers of my PhD. Overall, I first-authored 4 tier 1 papers covering a probabilistic mitigation against dynamic frequency- and voltage-scaling attacks, a novel Rowhammer variant circumventing state-of-the-art mitigations, the analysis of undocumented CPU registers and interfaces to identify their security implications, and software-based power analysis beyond application-dependent leakage.

CLKSCREW [212], Plundervolt [167], Voltjockey [184, 183], V0LTpwn [120], and VoltPillager [39] are attacks demonstrating that increasing the operating frequency or reducing the operating voltage of the CPU results in faulty computations. Furthermore, these attacks target trusted execution environments purely from software. Vendors prevent software-based attacks by disallowing CPUs to be undervolted from software, drastically reducing potential performance increases and power savings [111]. In Minefield [130], we propose a probabilistic software defense to protect against such attacks. We built a framework to analyze instructions to identify those most affected by undervolting and use these instructions as guards within a program to periodically check if their results are still valid. If a faulted computation within Intel's Software Guard Extension (SGX) is detected, the *secure* execution of the enclave can no longer be guaranteed, and the application halts. We evaluated the detection rate of the defense and found that 1 trap after 1-2 instructions mitigates 99 % of the known attacks. Furthermore, the performance overhead of 148.4 % for the added instructions outperforms fault redundancy checks in specific configurations. The paper was published at the USENIX Security Symposium 2022 [130] in collaboration with Daniel Gruss and Michael Schwarz.

The Rowhammer attack [123] exploits disturbance errors within DRAM cells when neighboring cells are frequently accessed. This fundamental problem resulted in vendors deploying active mitigations for DRAM modules that considered the direct neighboring cells [59]. The refresh-based defenses considered rows beyond the direct neighboring rows as irrelevant for Rowhammer attacks as the number of accesses and, therefore, the resulting access duration is outside the internal refresh window of the cells. With the Half-Double attack [132], we demonstrated that only considering direct neighbors is insufficient and showed that modern DRAM modules with mitigations against Rowhammer in place can still be attacked. Our experiments indicate that the combination of accesses to the direct and further away neighbors accumulate electrical disturbance sufficient to flip bits within a victim cell. We evaluated Half-Double on 10 commodity systems and discovered that overall 5 out of 7 mobile devices are affected by Half-Double. We demonstrate the applicability of Half-Double on state-of-the-art hardware with the Targeted Row Refresh (TRR) and Error Correcting Code (ECC) mitigations. Our end-to-end exploit targets a Chromebook and obtains root on the devices within 45 min on average. This work was published at USENIX Security Symposium 2022 [132] and was joint work with Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss.

The number of configuration registers of modern CPUs is too large for exhaustive analysis, and vendors only document 55.4 % of the registers on average [133] in their publicly available documentation. The documentation covers the necessary options to support hardware features used by open-source projects like the Linux kernel. However, the remaining registers remain undocumented or are only documented in proprietary documents available to BIOS and hardware vendors. Even though there is no public documentation, the registers are still accessible through privileged software. To explore the security implications, Domas [55, 53, 54] grouped registers with similar behavior and tested whether registers expose new instructions to the instruction set architecture and found one security-critical instruction. To analyze the influences of these configuration registers on the microarchitectural level, we designed MSRevelio [133], a framework that allows scanning of Model Specific Registers (MSRs) to find undocumented interfaces for side channels or even hidden configuration options that change the behavior of instructions for both offensive and defensive aspects. We show six case studies related to undocumented or partially documented configuration registers. For instance, we found a configuration bit that turns off the side channel-resistant AES hardware instructions.

Therefore, forcing an existing library to fall back to an alternative AES implementation, e.g., an insecure T-table implementation. Furthermore, we found configuration bits that disable the prefetch instructions, mitigating prefetch-based Kernel Address Space Layout Randomization (KASLR) attacks [144]. The paper was published at the IEEE Symposium on Security & Privacy 2022 [133] in collaboration with Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz.

Software-based power-analysis attacks [146, 234, 148] usually target cryptographic algorithms like AES or edge cases of these algorithms with distinct energy signatures to extract secret information. With Collide+Power [131], we close the gap between these specialized attacks and enable attacks on general-purpose data similar to the threat models of Meltdown [147] and Microarchitectural Data Sampling (MDS) [201, 198]. Collide+Power is a technique agnostic to the power-related signal and deploys a measurement method that significantly improves the signal quality. The paper introduces the notion of *colliding* data, i.e., if a CPU component is shared between security domains, an attacker can force data collisions between attacker-controlled and victim data. These collisions do not influence the correctness of the computation, appear only within the actual hardware, and are neither transiently nor architecturally observable. Instead of targeting structured data during computation like cryptographic keys, we target the CPU's memory hierarchy where data collisions occur frequently. However, these primitives do not rely on CPU vulnerabilities that accidentally forward the data to the attacker domain. We show Collide+Power and leak 4.82 bit/h with an MDS-style attack and 0.84 bit/h with a Meltdown-style attack when using direct energy readings via a CPU interface. The work was published at the USENIX Security Symposium 2023 [131] and was joined work with Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard.

1.2. Other Contributions

This section introduces the peer-reviewed co-authored papers of my PhD. Overall, I co-authored 11 publications, 7 of which are accepted at tier 1 conferences. These papers cover Rowhammer defenses, offensive and defensive research on trusted execution environments, mitigations against Spectre, and novel side channels on various hardware components.

With the insights from the Half-Double attack [132], we concluded that mitigations against Rowhammer and DRAM fault attacks in general should not rely on characteristics of the induced faults. Therefore, we designed CSI:Rowhammer [112], a hardware-software co-design that does not rely on faulting characteristics like the locality of bit flips. CSI:Rowhammer replaces the memory's Error Correcting Codes (ECC) with a cryptographic secure Message Authentication Code (MAC). The MAC allows the detection of arbitrary bit flips (within cryptographic margins) but requires an adapted correction algorithm. CSI:Rowhammer features a software component for a highly flexible error correction mechanism that, for instance, can reload read-only data from the disk and potentially correct an unlimited amount of bit flips. This paper was published at the IEEE Symposium on Security & Privacy 2023 [112] in collaboration with Jonas Juffinger, Lukas Lamster, Moritz Lipp, Maria Eichlseder, and Daniel Gruss.

We followed a similar approach with PT-Guard [196] to protect page-table entries, a common target for Rowhammer attacks. In this design, we partition a MAC into the free bits of page-table entries and verify the integrity of the entries when loaded from memory. The paper was published at the IEEE/IFIP International Conference on Dependable Systems and Networks 2023 [196] in collaboration with Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, Daniel Gruss, and Moinuddin Qureshi.

During our research on trusted execution environments, we discovered that existing classifications for software vulnerabilities are similar to the root causes of transient execution attacks. Therefore, we proposed in ÆPIC Leak [21] to apply this methodology to existing attacks and explore unobserved software classes concerning the hardware. Our analysis found an architectural attack targeting SGX that does not require a side channel to encode information. Instead, we directly read from undefined areas of the memory-mapped region of the Advanced Programmable Interrupt Controller (APIC) to leak stale data from an internal buffer. We demonstrate attacks with this primitive leaking AES-NI, RSA, and even the SGX attestation keys. This work was published at the USENIX Security Symposium 2022 [21] in collaboration with Pietro Borrello, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz.

AMD SEV aims to provide a trusted execution environment on a virtual machine basis. With CacheWarp [250], we introduced a software-based fault attack that exploits a misconfiguration in the cache invalidation instruction. We combine this primitive with cache eviction techniques to selectively erase the state of a program, circumventing sudo permissions checks, breaking cryptographic primitives, and bypassing the SSH login of the secure virtual machine. CacheWarp was published at the USENIX Security Symposium 2024 [250] in collaboration with Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, and Michael Schwarz.

With LVI-Nullify [68], we designed a compiler extension that hardens Intel Software Guard Extension (SGX) enclaves against LVI-Null [223]. LVI-Null injects zeros into faulting load instructions arbitrarily. The compiler extension modifies each load instruction and adds a specific offset within the enclave, effectively hindering the effects of injected zeros. This work was published at the USENIX Security Symposium 2022 [68] in collaboration with Lukas Giner, Claudio Canella, Michael Schwarz, and Daniel Gruss.

In-process isolation is a technique where instead of individual processes a single process contains the workloads of multiple tenants [41, 42]. Therefore, this type of isolation is potentially affected by Spectre attacks [126]. With Dynamic Process Isolation [203, 204], we developed a probabilistic mechanism that detects Spectre attacks and moves the attacker into an isolated process to prevent further exploitation. Our defense is now actively deployed by the Cloudflare Workers infrastructure. The paper was published at the European Symposium on Research in Computer Security 2022 [203] in collaboration with Martin Schwarzl, Pietro Borrello, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz.

Modern AMD processors have distinct scheduler queues per execution unit compared to the unified scheduler queue that Intel CPUs deploy. We reverse-engineered these distinct queues in SQUIP [65] and found a novel contention-based side channel that enables new covert channels and we were able to exfiltrate RSA keys. The paper was published at the IEEE Symposium on Security & Privacy 2023 [65] in collaboration with Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Simone Franza, Markus Köstl, and Daniel Gruss.

We extended upon the SQUIP attack and analyzed if these types of attacks are possible in more restricted environments like JavaScript in our paper [64]. This work was published at the Financial Cryptography and Data Security 2024 [64] conference in collaboration with Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, and Daniel Gruss.

Intel Optane memory is an extension for persistent data storage [97]. Optane memory uses multiple buffers to improve performance and reduce access latencies. However, we showed that these buffers introduce timing side channels, which allow us to infer contention and inter-keystroke timings or build persistent covert channels. The paper was published at the USENIX Security Symposium 2023 [97] in collaboration with Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Daniel Gruss, and Samira Khan.

With IdleLeak [188], we found undocumented functionality in an optimized waiting instruction that allows us to observe interrupts even across logical cores and inside virtual machines that allow highly accurate fingerprinting of inter-keystroke timings, websites, and videos. This work was published at the Network and Distributed System Security Symposium 2024 [188] in collaboration with Fabian Rauscher, Jonas Juffinger, and Daniel Gruss.

The WebGPU standard enables websites to utilize the parallel computation power of GPUs. We present cache attacks via the WebGPU standard to leak inter-keystroke timings and AES keys [67]. We utilize the GPU to parallelize eviction set creation. The paper was published at the ACM ASIA Conference on Computer and Communications Security 2024 [67] in collaboration with Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Daniel De Almeida Braga, and Daniel Gruss. Furthermore, we received a *Best Paper* award for the publication.

1.3. Outline

Chapter 2 provides background on memory organization, the microarchitecture of a processor, side-channel analysis, and fault attacks. Chapter 3 summarizes the state of the art of transient-execution attacks, Rowhammer, dynamic voltage- and frequency-scaling attacks, and software-based power analysis. Chapter 4 concludes Part I. Part II provides a complete list of the first- and co-authored papers and the camera-ready versions of the main contributions of this thesis in Chapters 5 to 8.

2 Background

This chapter provides background for this thesis. Section 2.1 introduces memory organization, including virtual memory and the memory subsystem. Section 2.2 discusses the microarchitecture of modern superscalar pipelines with speculative and out-of-order execution. Section 2.3 introduces side-channel and power analysis where software-based interfaces replace traditional measurement equipment. Section 2.4 discusses traditional fault attacks that require physical access to the device and introduces software-based fault attacks that do not require physical access.

2.1. Memory Organization

Programs interact with memory through an abstraction layer that provides a homogeneous view, hiding the complex memory hierarchy of various types and sizes designed for maximum performance. We introduce the concept of virtual memory in Section 2.1.1. Virtual memory allows for efficient memory management and isolation for modern operating systems. Furthermore, we discuss the memory subsystem, including the cache hierarchy in Section 2.1.2 and the main memory in Section 2.1.3.

2.1.1. Virtual Memory

Virtual memory was introduced to run multiple isolated processes on a single system. Each process is assigned a unique virtual address space to isolate the processes. The virtual address space cannot be accessed from another process except if explicitly allowed. The operating system is responsible for creating the virtual address spaces for each process and switching between them when scheduling a process. The virtual address spaces are partitioned into virtual pages. Virtual pages map to physical

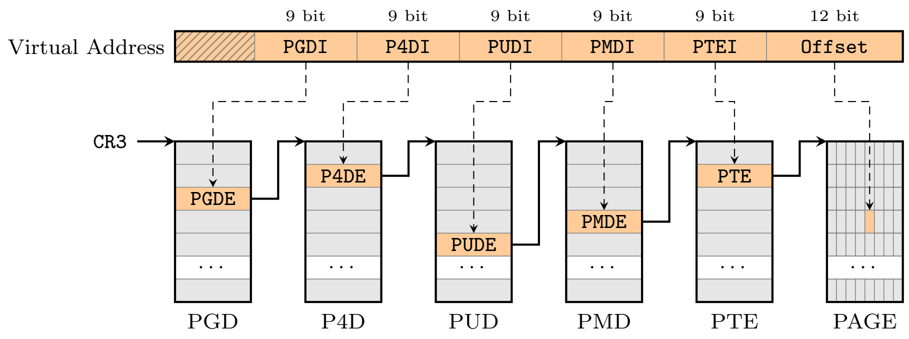


Figure 2.1.: Virtual-to-physical address translation using 5-level paging, assuming a 4 KiB page size on the x86-64 architecture. We use the Linux terminology for the page-table levels and entries [91, 142].

pages in the main memory. From the operating system's perspective, a page is the smallest manageable unit of memory on modern hardware.

Virtual addresses consist of up to 57 bit [91], representing up to 128 petabytes of virtual memory per process. However, commodity devices currently support only up to 48 bit of physical memory. Since each virtual address space is process-specific, virtual addresses of distinct processes can overlap but map to different physical addresses. The CPU uses page tables to resolve this aliasing problem and make virtual-to-physical mappings possible. Additionally, page tables support non-present virtual pages. The page tables translate a given virtual page to a physical page, specifically, a physical page frame number (PPFN), and allow the operating system to associate metadata with the virtual address, such as whether the page has a physical counterpart in the main memory. The page size is hardware-dependent, and the usual sizes are 4 KiB or 16 KiB [83]. Although page tables allow marking pages as not present, the amount of memory to store the actual page tables in memory depends on the number of page-table levels. Most recent Intel CPU support five levels of page tables [91].

The physical address of the highest page-table level is stored in the `CR3` register on x86 and the `TTBR` register on ARMv8. We refer to the names used in the Linux kernel [142] for the page-table levels. The different levels of a five-level paging structure are from highest to lowest: Page Global Directory (PGD), Page Level 4 Directory (P4D), Page Upper Directory (PUD), Page Middle Directory (PMD), Page-Table Entry (PTE). Figure 2.1 shows a virtual-to-physical address translation. The translation

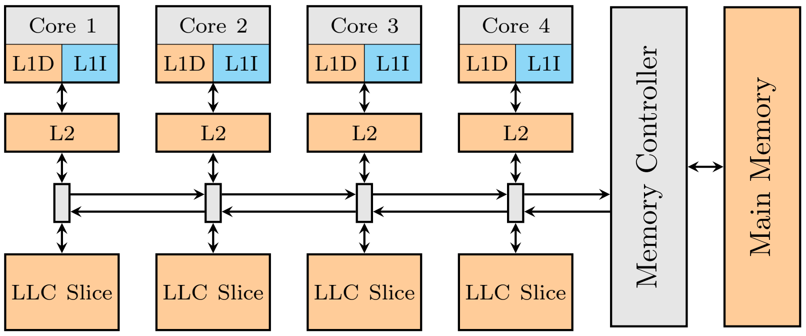


Figure 2.2.: The memory subsystem of a multicore processor including the private caches, the shared last-level cache, and the main memory. The last-level cache is split into cache slices. The ring bus connects the cores, the last-level cache slices, and the memory controller with each other.

is a process where the virtual address is partitioned into indices that index the current page-table level to retrieve the PPFN of the next page-table level. The page offset is added to the PPFN of the last page-table entry to compute the translated physical address. The page-table walkers perform the actual page-table translations in hardware. Partial results of translations are cached in the page-table caches [73], and the final virtual-to-physical address mapping is cached in the Translation Lookaside Buffer (TLB) [57]. Furthermore, most architectures support huge pages, i.e., physically contiguous chunks of pages, to save on the number of TLB entries required to cache the physical contiguous range.

Besides the metadata indicating whether a page is mapped to physical memory, the page tables contain information about which privilege level is allowed to access the page, if the page can be written to, or if code on the page is prevented from being executed. Although the hardware enforces these bits, delayed checking enabled the Meltdown attack and its variants (cf. Section 3.1.1). Finally, these bits and the PPFNs stored in the page tables are common targets for Rowhammer attacks (cf. Section 3.2).

2.1.2. Caches

Caches temporarily store data to speed up subsequent accesses or modifications to the data without reloading the data from the *slow* main memory. The size of caches is smaller than the main memory and is usually built using a different memory technology, e.g., SRAM cells [164]. Modern caches usually store data with a data granularity of 64 B or 128 B [57]. Memory accesses that are served from the cache are referred to as *cache hits* and those that are not cached as *cache misses*.

Cache Coherency. Mutlicore architectures duplicate CPU *cores* to execute multiple processes in parallel instead of concurrently. Each core has unique private caches that must be coherent with all other caches. The cache coherence protocol defines how coherency is enforced within the CPU. The cache lines store additional metadata about the cache line's state and its coherence state. For instance, the MESI coherence protocol [169] uses four states to describe the coherency state of a cache line: modified, exclusive, shared, and invalid. To synchronize state changes to the different caches, different approaches can be used [169]: First, snooping-based approaches communicate directly with all other caches to update the states of the cache lines. However, this *one-to-many* communication scales with the number of cores on modern CPUs. Second, directory-based approaches forward requests to a shared cache directory, reducing the number of messages over the internal buses.

Memory Subsystem. Modern CPUs employ a hierarchy of caches to hide access latencies of the main memory and close the gap between the slow response times of the memory and the fast execution time of the CPU. Figure 2.2 shows the memory subsystem, including the main memory and the cache hierarchy. On Intel CPUs, the cache hierarchy consists of the first-level (L1), second-level (L2), and last-level (LLC) caches. The L1 and L2 caches are core private caches and are only shared between sibling cores where the L1 cache is split into a data cache (L1D) and an instruction cache (L1I). The LLC is the largest cache inside the CPU package and is shared between all cores. AMD CPUs use a similar hierarchy, but the LLC is only shared across a so-called core complex [210], and an additional component handles coherency between the core complexes. Smaller form factor CPUs may skip the L2 cache. The CPU registers are the fastest available memory inside the core as they are required to execute instructions. Registers are stored in a register file that offers more physical than architectural registers to implement optimizations like out-of-order execution (cf. Section 2.2). The L1 cache has an access latency of 4 cycles, followed by the L2 with 14 cycles, and the LLC with 50 cycles on the Intel Skylake server microarchitecture [94] outperforming the rather slow access latency of the main memory with 60 ns [140].

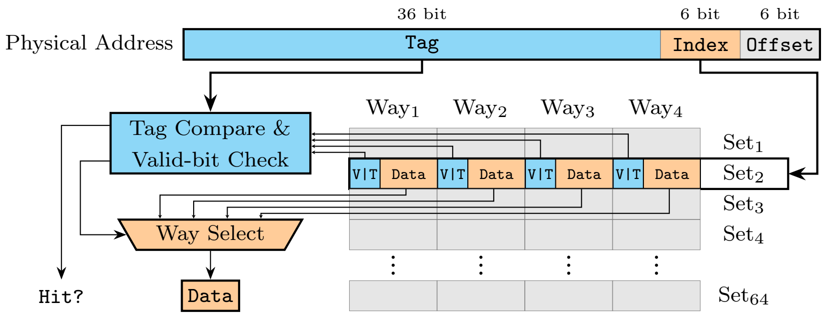


Figure 2.3.: A set-associative cache using 64 sets with 4 ways and a cache line size of 64 B. We use 48 bit physical addresses in a physically indexed and physically tagged design resulting in a tag size of 36 bit.

Set-Associative Caches. Figure 2.3 shows a set associate cache design, currently the design used in Intel, AMD, and Apple CPUs. A set-associative cache maps a physical address to one of multiple *ways* in a *set* of the cache [94]. The number of ways usually ranges between 4 and 16 [94]. The physical address of the memory request is partitioned into cache line offset (e.g., bits 0 to 6), set index (e.g., bits 6 to 12), and tag (e.g., starting from bit 12) to identify the exact location in the cache. The set index specifies the set where the data resides in one of the ways. The cache compares the address tag against the stored tags in each way of the set to find the cache line containing the data. Furthermore, the valid bit of the selected way is checked to ensure no stale data is accessed. Finally, the data is extracted from the cache line based on the cache line offset and the requested memory access size. A cache miss occurs when the tag is not found in the set, or the valid bit of a matching cache line is zero.

Programs use virtual addresses when interacting with memory and the cache must wait until the physical address from the TLB or a page-table walk is available (cf. Section 2.1.1). To hide this delay, modern CPUs are designed to use a specific page size. This ensures that the cache line offset and the set index of the L1 cache are always identical between physical and virtual addresses. Although this limits the number of sets in the L1 cache, it allows for a Virtually Indexed and Physically Tagged (VIPT) cache design. VIPT caches directly extract the set index from the virtual address while the physical address for the tag is resolved in parallel. Hence, the L1 cache can already perform the set selection without waiting for the physical address [164]. Higher cache levels use PIPT designs as the physical address is available after checking the L1 cache.

Cache Inclusion Policies. The cache hierarchy within a CPU adheres to specific inclusion policies, with the most common being *inclusive* and *non-inclusive* [169, 94, 101, 70, 145]. An inclusive cache design requires that all the data in other caches is present in the inclusive cache. For example, an inclusive LLC, with respect to the lower cache levels, must contain the data from the private caches. Such a design makes it easy to directly store metadata required for directory-based coherency protocols within the cache lines, as all cached addresses are located in the LLC. This metadata is required to locate shared cache lines within all the private caches and removes the overhead of snooping-based coherency protocols (cf. Section 2.1.2). Non-inclusive cache designs relax inclusivity, and the data can but is not required to reside in the LLC.

Cache Replacement Policies. Due to the limited size of caches, when data is loaded into a cache, a cache line must be *evicted* to make space for the new data. The replacement policy defines which algorithm determines the cache line to evict. Different algorithms are utilized depending on the cache level and latency requirements [2]. The most common replacement policies are variants of the Least Recently Used (LRU) algorithm [2, 231]. LRU keeps track of the number of accesses to a cache line and evicts the one with the least accesses. The hardware usually does not implement the LRU algorithm but instead uses the Pseudo-LRU (PRLU) or Quad-age LRU (QLRU) algorithms [26]. The PLRU and QLRU algorithms approximate the LRU algorithm, where the QLRU implements LRU with a reduced complexity of four states [26]. L1 caches on Intel CPUs usually use the PLRU algorithm [2], and the LLC uses an adaptive QLRU algorithm [1].

2.1.3. Main Memory

A computer system's main memory is usually built using DRAM technology [83]. DRAM memory uses cells to store the information of single bits, where a cell consists of a transistor and a capacitor storing the encoded bit as a charge. Due to physical effects, the capacitor loses charge over time and the stored information has to be periodically refreshed, i.e., the values in the cells need to be refreshed [207]. The refresh interval until the charge has to be refreshed is usually 32 to 64 ms [109]. The DRAM Dual Inline Memory Modules (DIMMs) are structured to improve concurrent accesses, e.g., the structure contains channels, ranks, and banks [83, 123].

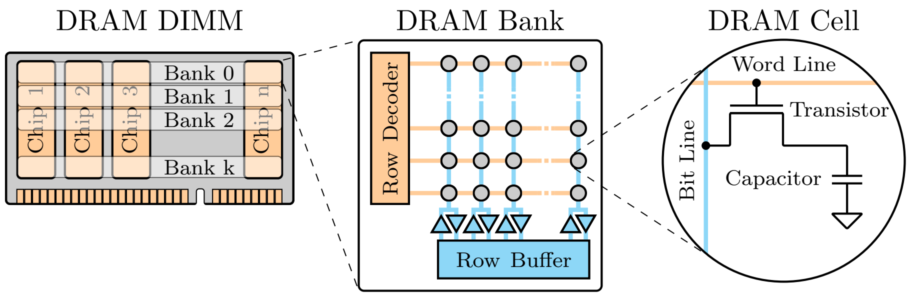


Figure 2.4.: The topology of a DRAM DIMM consists of multiple chips that work together to form banks. Each bank contains a grid of cells, where each cell includes a transistor and a capacitor that stores the encoded bit. Cells are grouped into rows, which are transferred to the row buffer via the sense amplifiers when the row's word line is active.

Figure 2.4 shows the schematic of a DRAM DIMM, bank, and cell. The gate of the cell's transistor is connected to the word line, which, when active, connects the capacitor over the bit line with the sense amplifier. Multiple cells inside a bank are combined into a *row*, i.e., the word lines of the cells are connected, and the data of the cells is forwarded over the individual bit lines all at once. The row size for Double Data Rate 3 (DDR3) memory is usually 8 KiB [109]. The sense amplifiers *sense* the charge of the cells and convert it to a *stable* signal, thereby depleting the charge. The *row buffer* stores the sensed data temporarily to allow the selection of the requested data and to speed up successive data accesses to the same row. Transferring the row's data from the cells to the row buffer is referred to as *activating* a row. Finally, the inverse process is performed, and the row is *closed* by recharging the cells' capacitors with the data value stored in the row buffer. To refresh the charge in the capacitors, the DIMMs perform from a top-level perspective, a similar operation to activating each row within a 64 ms window, and the refresh commands share the time window with regular data requests and commands [149].

The DRAM addressing function maps a physical address to the exact location within the topology of the DIMM, i.e., channel, rank, bank, and row. If a different row is still active, the active row is closed, and the requested row is opened. Depending on the row policy, the row can stay open for a short duration (open row policy), be closed immediately (closed row policy), or an adaptive combination of both policies [143].

2.2. Microarchitecture

In this section, we discuss the implementation of modern processors. Section 2.2.1 introduce superscalar pipelines and the optimizations that increase processor execution performance. Section 2.2.2 introduces the configuration and monitoring facilities of the processor and discusses the complexity of exhaustively identifying configuration options.

2.2.1. Microarchitectural Optimizations

CPUs deploy optimizations to speed up computations to meet today's performance requirements. The Instruction Set Architecture (ISA) defines the interface between software and hardware for a given architecture, e.g., x86-64 or ARM64. The exact hardware implementation is vendor-specific and is referred to as the microarchitecture [57]. The microarchitecture implements the CPU's superscalar pipeline, conceptually split into front-end and back-end. The front-end is responsible for *fetching* and *decoding* instructions. Compared to a more traditional pipeline, instructions on modern CPUs are not directly executed but split into smaller micro-operations (μ-ops). These μ-ops are issued from the front-end to the back-end, where they *execute*, have their effects *written back*, and are marked as retired. The back-end contains the arithmetic logic unit and the address generation to execute the μ-ops and interact with the memory subsystem (cf. Section 2.1). On superscalar CPUs, the back-end of the CPU can execute multiple instructions in parallel on individual execution units. Optimizations are built into such an execution pipeline to improve performance where superscalar out-of-order execution and predictors, like branch prediction and data prefetching, add the most significant performance improvements besides the CPU caches (cf. Section 2.1.2).

Out-of-Order Execution. Out-of-order execution uses the observation that in a program, instructions that do not have dependencies on each other can be rearranged to execute *out of order* [83, 218]. This optimization hides latencies as subsequent μ-ops can be executed in parallel while the CPU waits for other μ-ops to complete, e.g., a load is waiting to be served from memory. The CPU ensures that the program cannot observe this optimization architecturally and, therefore, retires the finished μ-ops in order, i.e., the architectural effects of the instruction become visible only after the previous instruction retires. However, if a given instruction cannot retire, e.g., due to a fault or an interrupt during execution, the temporary results are discarded, and the control flow is diverted to the handler. The μ-ops that are executed but never retire are named transiently executed instructions [147, 33, 126]. Architecturally, the CPU is always in a valid *in-order* state. However, the executed μ-ops still leave observable traces in microarchitectural buffers like caches, e.g., if a transiently executed μ-op loaded a value from the main memory and loaded it in the L1 cache.

Speculative Execution. Predictors, such as branch predictors, allow the hardware to predict the outcome of μ-ops before the dependencies are available or issue additional loads depending on previous access patterns. For instance, the branch predictor predicts the outcome of conditional control flow instructions like conditional jumps, indirect jumps, and returns before the branch condition is available [208]. The branch predictors use hardware structures like the Pattern History Table (PHT) [126, 249], Branch Target Buffer (BTB) [126, 138, 175], and Return Stack Buffer (RSB) [153] to store metadata of previous control flow changes and use this metadata to predict future jump targets. A successful prediction allows speculatively executing the jump target's code and hiding latency. However, if a prediction is wrong, the CPU has to discard all the μ-ops and revert the control flow to the correct execution path. Finally, the state of the branch predictor is updated for future predictions.

2.2.2. Microarchitectural Configuration and Monitoring

The processor exposes Model Specific Registers (MSRs) [95, 96, 7] to privileged software like the operating system and the BIOS. The software uses MSRs to configure the architecture and microarchitecture, like the entry point for the kernel (`sysenter`) or the configuration of the CPU operating frequency and adaptive power management [95]. Additionally, MSRs expose monitoring capabilities to track microarchitectural events like cache misses and power-related signals, such as the overall power consumption of the processor. Each MSR is a 64 bit registers and addressed with a 32 bit address, where functionally similar registers are grouped together [96]. Vendors only partially document MSRs for the public domain, and the complete documentation is only available for OEMs.

Therefore, although most of the MSR address space is empty and read or write attempts do not succeed, a large portion of the address space, whose functionality is unknown, still exists and could potentially expose security-critical information. Domas [55, 53, 54] explores the search space of undocumented system configuration registers on a Nehemiah CPU [55]. Due to the amount of registers, exhaustive analysis is not possible. Therefore, Domas utilizes a timing side channel to group system registers based on write timings, assuming that registers with similar behavior will exhibit similar timing signatures. The remaining search revealed a configuration register enabling instructions to interact with a deeply embedded instruction set in the CPU. Domas demonstrates that this instruction set can circumvent ring isolation. In Chapter 7, we analyze how MSR configuration bits affect the microarchitectural behavior of instructions groups.

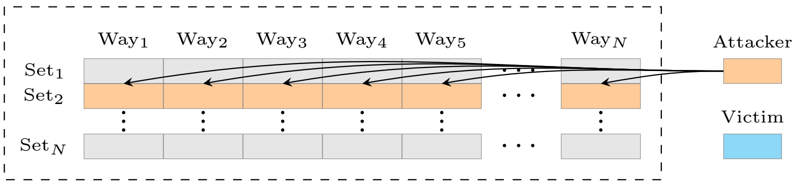
2.3. Side-Channel Analysis

This section introduces microarchitectural side channels and power analysis. Section 2.3.1 discusses the fundamentals of side channels and introduces cache attacks. Section 2.3.2 introduces power analysis with different analysis types and system interfaces replacing external measurement equipment.

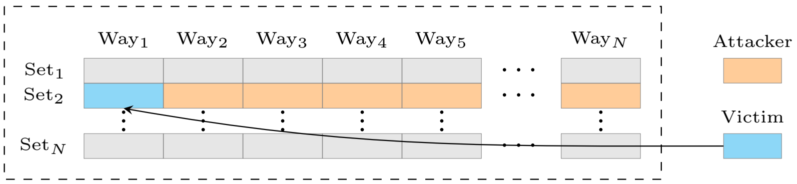
2.3.1. Microarchitectural Side Channels

Side channels extract secret information from unintended emissions of metadata from a concrete implementation [119]. The carrier signal for the *unintended* information leakage can be the power consumption [127, 25], electromagnetic emissions [5, 62, 85, 185], temperature [89], optical emissions [199], and execution time [125] of the implementation. The extracted metadata allows to reconstruct the secret information processed in the implementation with a specific success rate although the secret information is never directly exposed. The emission of information is a *side effect* of the implementation. Furthermore, timing side effects can originate from microarchitectural optimizations and components.

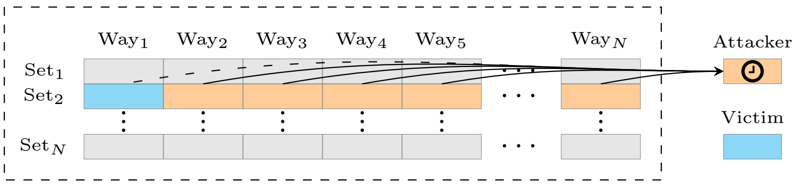
Cache Side-Channel Attacks. Cache side channel attacks exploit the timing difference between cached and uncached data (cf. Section 2.1.2) by measuring how long it takes to access data. Various cache side-channel techniques were proposed over the years, using different prerequisites and threat models. If an attacker and victim share code pages, e.g., when



(a) **Attacker Prime:** The attacker fills a cache set with data.



(b) **Victim Access:** The victim loads data and evicts a cache line.



(c) **Attacker Probe:** The attacker infers if the victim accessed data.

Figure 2.5.: The Prime+Probe attack fills a cache set with attacker accessible cache lines to infer if a victim application accessed the cache set.

using a shared library on the system, the operating system will map the same physical page into the virtual address spaces of both processes. Therefore, the attacker and victim will access the same cache line when accessing data or executing code on the page. Flush+Reload [248, 78] uses cache maintenance instructions to remove the shared cache line from the caches and measure the access time of the shared cache line. A fast timing indicates that the victim has accessed the cache line after the last flush. Evict+Reload [77, 145] replaces the explicit flush instruction with an eviction set to remove the data from the cache. Flush+Flush [75] utilizes that the flush instruction is faster if the cache line is not in the cache.

If no shared memory is available between victim and attacker, set-based attacks can be mounted to identify whether the victim accessed a cache line corresponding to a cache set. The Prime+Probe [14, 172, 174] attack exploits the limited number of ways in a set-associative cache (cf. Section 2.1.2). Each time uncached data is loaded into the cache, another cache line must be evicted from the cache to make space for the new data. Figure 2.5 shows the steps of a Prime+Probe attack. First, the attacker determines a targeted cache set and searches for a collection of virtual addresses that map to the same cache set, i.e., the cache set index is the same for all addresses. After finding the addresses, the attacker fills the cache set by accessing them. Second, the victim could access data and bring it into the cache, evicting one of the attacker's cache lines. Finally, the attacker measures the time it takes to access all addresses mapping to the same set. If the access time is increased, a cache miss has occurred, and the attacker can infer that the victim potentially accessed data that maps to this set. Otherwise, the victim did not perform an access to this cache set. Evict+Time [172, 14, 174] measures the time over the execution time of the victim instead of the time required to access the attacker cache set. Prime+Scope [180] and Reload+Refresh [26] extend Prime+Probe to optimize for the used cache replacement policy. Prime+Abort [50] uses Transactional Synchronization Extension (TSX) to observe transactional aborts when the victim interacts with the targeted cache set.

Side-Channel Attacks on KASLR. Kernel Address Space Layout Randomization (KASLR) randomizes the virtual addresses of the kernel's code and data during the boot process. This additional layer of security makes it harder for an attacker to identify the location of security-relevant data in the kernel. Microarchitectural side channels have been used to break KASLR [32, 73, 88, 105, 200]. Lipp et al. [144] present a KASLR break on AMD CPUs using the `prefetch` instruction. The execution time of the `prefetch` instructions leaks information about the number of mapped page-table levels of the given virtual address. An attacker can iterate over kernel address space, identify if the given kernel address is actually mapped, and break KASLR. In Chapter 7, we present how to prevent KASLR breaks that exploit prefetch instructions.

2.3.2. Power Analysis

The power consumption of a CPU depends on the executed operations due to the fundamental data-dependent power leakage of CMOS transistors [154, 35, 9]. Transistors act as switches, and their state, i.e., open or closed, represents bits. However, they expose a data-dependent power consumption depending on their switching behavior. Since transistors are fundamental for logic gates and hardware functionality, this data-dependent leakage is inherited by components in the CPU.

Power-analysis attacks [127, 25, 36, 230, 165, 48, 116, 177, 179] exploit this data-dependent power consumption and aim to reconstruct the secret inputs of, for instance, a cryptographic primitive. In such a scenario, an adversary uses measurement equipment to measure the power consumption during the cryptographic operations. Traditionally, these types of attacks required physical access to the device. However, vendors provide integrated energy measurement interfaces to monitor the processor's energy consumption. The Running Average Power Limit (RAPL) interface on Intel and AMD CPUs accumulates the consumed energy of individual processor components [47, 95, 7]. The RAPL interface has a reduced timing and value resolution compared to external measurement equipment [146, 61]. Where externally measured power traces have multiple power samples per instruction executed, the software-based traces record multiple executed instructions in a single sample. To counteract the low timing resolution, Hähnel et al. [79] and we [129] propose techniques to improve the quality and accuracy of the interface. Although these techniques improve the signal quality, the interface still does not match the resolution of external measurement equipment. Khan et al. [121] analyze the accuracy of the RAPL interface for extended measurement durations when compared with a ground truth. Benign use cases for energy measurements [13, 56] are justifiable, such as measuring the consumption of communication primitives [229] or evaluating the overhead of Meltdown and Spectre mitigations [84]. However, integrated energy measurement interfaces are a building block for power side-channel attacks (see Section 3.4).

Lipp et al. [146] demonstrate that the CPU's internal energy interface can replace the external measurement equipment and the need for physical access to the device to perform traditional power-analysis attacks. Due to the reduced timing resolution, software-based attacks usually require a replay primitive to amplify the power leakage [146, 131, 102]. Although traditional power analysis and software-based power analysis differ in measurement sample acquisition, the traditional attack methodology is applicable in both fields and is categorized into two analysis categories.

Simple Power Analysis. Simple Power Analysis (SPA) [127] considers direct influences of secret-dependent computations on the power consumption [127, 154]. In this scenario, the attacker is usually passive and uses

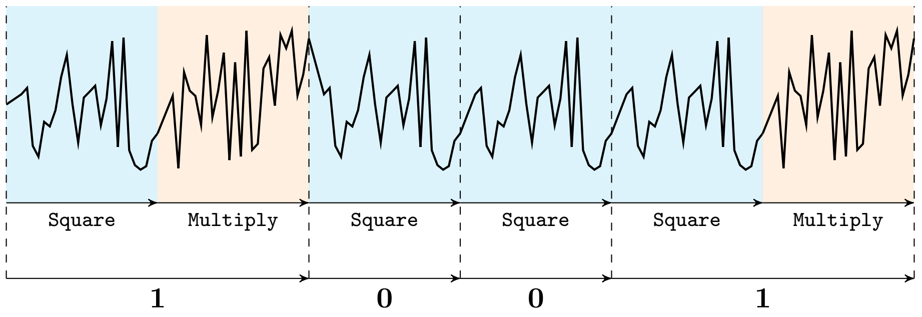


Figure 2.6.: Simple power-analysis attack on a non-constant time square-and-multiply RSA implementation. The difference in the energy consumption of the square and multiply operations leaks the secret exponent.

one or multiple recorded power traces of the same input to infer the secret information. Figure 2.6 shows an example of a SPA targeting a non-constant time square-and-multiply algorithm [93] used in RSA. The algorithm performs depending on the current exponent bit, either a single square operation or the square and the multiply operation. Assuming that the multiply operation has a distinct power signature compared to the square operation, the pattern of multiply invocations directly leaks the bits of the secret exponent. Therefore, visually inspecting the power trace is enough in this example to extract the secret exponent.

Differential Power Analysis. Differential Power Analysis (DPA) exploits *differential* information between power traces recorded with distinct inputs and a fixed key [127, 154]. Contrary to SPA, this attack category requires observing changing inputs and uses statistical measures to recover the secret information. The statistical methods range from the difference of means of two classes, e.g., if a bit is set or not set, to correlation-based analysis, usually referred to as Correlation Power Analysis (CPA) [25]. CPA searches for the maximum correlation coefficient between a modeled hypothetical power consumption and the actual power traces. The most common models used to compute the hypothesis are the Hamming weight of a single intermediate, e.g., the number of set bits, or the Hamming distance between two intermediates, e.g., describing the number of bit changes required to transition from one data value to another. The Hamming distance model is often used to model registers where the register value is overwritten with new data. By enumerating all possible values of the intermediate and computing each correlation coefficient, the hypothetical power consumption with the correct guess has the highest coefficient if the model is correctly representing the power consumption.

Profiled Analysis. Both SPA and DPA can be extended to profiled attacks by considering prerecorded traces with known secrets [154, 36]. These traces can be obtained if a test device of the targeted hardware is available, or if the targeted device allows the use of known keys. The idea of profiled attacks for SPA is to enumerate all possible values of an intermediate and match a prerecorded template with the attack trace. The gathered data can also be used to build a more accurate DPA model.

2.4. Fault Attacks and Defenses

Fault attacks change the behavior of well-defined procedures like computations [212, 167, 183, 184, 120], control-flow transitions [215], or directly manipulate data values [123]. Fault attacks targeting data values are categorized into attacks against data-in-use, i.e., data computed on, and data-at-rest, i.e., data stored in memory. Traditional fault attacks can use invasive and non-invasive methods. Invasive methods require opening the component under attack to inject the fault. Various techniques were established to induce faults, such as manipulating the operating voltage [28, 167, 183, 184, 120], clock source [212], or by using lasers [226].

To exploit fault attacks in a cryptographic context, techniques like differential fault analysis [16] and statistically ineffective fault attacks [52, 51] have been proposed. Fuhr et al. [60] present fault attacks recovering AES keys. Boneh et al. [19] discuss the importance of validating results of cryptographic algorithms to protect against fault attacks. Such a validation step can be computing backward from the output and validating that the computed input matches the actual input of the primitive. Computing a redundant path and comparing the outputs can also identify fault attacks. Additional countermeasures are proposed on the hardware level [117, 193] and for concrete cryptographic implementations like AES [12]. Mitigations against fault attacks are usually designed following a fault model, i.e., a model describing how the fault manifests in the implementation [191].

Secure boot ensures that the code responsible for initializing the hardware and booting the system until the operating system takes control is not tampered with. However, fault attacks targeting secure boot have been demonstrated on modern hardware [226, 225]. Timmers et al. [215] demonstrate how to control the program counter using fault attacks. Furthermore, Timmers et al. [214] use fault injection to escalate privileges.

Like power analysis, these attacks traditionally require physical access to the device to inject the faults. Nevertheless, over recent years, software-based fault attacks emerged. The Rowhammer attack [123] exploits the DRAM technology used in our main memory (cf. Section 2.1.3) and targets data-at-rest. Rowhammer exploits that frequently activating a DRAM row will induce bit flips in neighboring rows of the same bank. Walker et al. [232] analyze DRAM on a physical level and conclude that the bit flips are caused by electron injection, capture, and capacitive crosstalk. Hong et al. [86] have a similar conclusion of the root cause effect of electron spreading and injection. These effects are amplified due to the constant shrinking of the DRAM cells to increase storage density [168, 122, 86]. The density assumption is further supported by Kim et al. [122] as they find that the number of flips increases with each new DRAM generation when keeping the activations to a row constant. Lim et al. [141] analyze how bit flips in the memory are affected by proton radiation. Additionally, Hong et al. [86] discuss that depending on the activation time of a transistor, two effects induce bit-flips into neighboring rows: The Rowhammer and Passing Gate effect. We further detail the security implications and the state of the art of Rowhammer attacks in Section 3.2.

Dynamic Voltage- and Frequency-Scaling (DVFS) allows dynamic adjustment of a processor's operating point, i.e., the voltage and frequency of the underlying hardware. Adjusting frequency and voltage is crucial to control the power consumption, thermals, and performance of a CPU [111]. Modern processors provide privileged software interfaces to configure DVFS from the operating system, allowing for flexibility regarding the workload and system type, e.g., desktop or server system [95, 7]. Multiple software-based undervolting and overclocking attacks [212, 167, 183, 184, 120] use system interface to configure an unstable operating point that experiences faults during computations targeting data-in-use (see Section 3.3).

3 State of the Art

In this chapter, we introduce the state of the art in the research field relevant to the main contributions of this thesis. Section 3.1 discusses Meltdown-type and Spectre-type attacks. Section 3.2 discusses Rowhammer attacks and defenses. Section 3.3 discusses software-based Dynamic Frequency- and Voltage-Scaling (DVFS) attacks, focusing on undervolting and overclocking of processors. Section 3.4 discusses software-based power analysis using direct and indirect power interfaces.

3.1. Transient-Execution Attacks

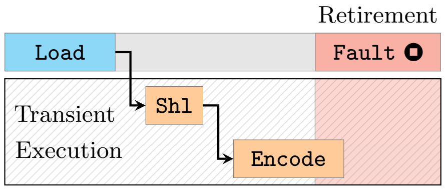
Transient-execution attacks exploit instructions that were executed but did not retire to encode information in side channels. These cases occur due to mispredictions, faults, or microcode assists in the out-of-order execution pipeline (cf. Section 2.2). Section 3.1.1 introduces Meltdown-type attacks and discusses how the main contributions of this thesis help to reduce MDS leakage. Section 3.1.2 introduces Spectre-type attacks and explains how we use Spectre as a data-corruption validation primitive.

3.1.1. Meltdown-type Attacks

The Meltdown [147] attack exploits faults and microcode assists during transient execution. Faults and microcode assists are handled in the retirement stage of a μ-op. However, due to out-of-order execution subsequent and non-dependent μ-ops can already compute on the transient intermediates for a short duration. Meltdown utilizes the execution of transient instructions to encode secrets extracted from another security domain into microarchitectural side effects. Multiple variants of Meltdown were discovered and categorized depending on the fault or microcode assist

```
1 # rdi = &kernel_address
2 # rsi = &encode_array
3 meltdown:
4   movzbl (%rdi), %eax
5   # never reached architecturally
6   shll $12, %eax
7   movb (%rsi,%rax), %al
8   ...
```

(a) Meltdown-US in assembly code.



(b) Sequence diagram of Meltdown-US.

Figure 3.1.: The assembly code and sequence diagram of the Meltdown-US variant. The attacker accesses an inaccessible kernel address in `rdi` (Line 4) and forces transient execution of the encoding instructions (Lines 6 and 7). The instructions access a specific page of the `encode_array` in `rsi`. The accessed page can be recovered using a side channel after the transiently executed instructions have been squashed, recovering the kernel value. During the retirement of the load instruction the fault is identified and raised to the software handler.

that triggers the transient execution [33]. For instance, the Metldown-US variant [33, 147] exploits that if an attacker accesses a kernel page, i.e., a page where the user-accessible bit is cleared (cf. Section 2.1.1), the data is transiently forwarded to subsequent μ-ops encoding the secret into a side channel. Figure 3.1 shows the assembly code of Meltdown-US where an inaccessible address is accessed, and the data is encoded in the cache during the transient window. This short duration between the load and the pipeline flush before handling the fault is enough to circumvent process isolation. To prevent Meltdown-US on a software level, the KAISER patch [71, 80] was deployed to unmap the kernel address space from all userspace address spaces. Therefore, the virtual-to-physical address translation during transient execution will fail, and no data can ever be loaded from the L1 cache (cf. Section 2.1.2).

The Foreshadow attack [236, 222], categorized as Meltdown-P [33], is similar to the original Meltdown-US variant. Metldown-P enables address-based leakage, i.e., the targeted data is leaked from the L1 cache via a wrongly translated physical address (cf. Section 2.1.2). Contrary to attacks targeting the L1 cache, Microarchitectural Data Sampling (MDS) attacks [200, 33, 201, 198, 163, 30, 209, 187] extend the Meltdown variants beyond L1 cache leakage [92]. MDS attacks target internal CPU buffers like the *load port*, *store buffer*, *line fill buffer*, and *staging buffer* in which data is relatively short-lived and *in-flight*. Therefore, the leakage from these buffers contains noise from other data traveling over the buffers.

Load Value Injection (LVI) [223] turns Meltdown-type attack primitives around to inject from an attacker to a victim. LVI uses microcode assists during the execution of a victim load to inject a value into the load instruction during transient execution. The victim transiently computes with the injected value, which can alter control flow or perform an out-of-bounds access to encode secret information into a side channel.

The first hardware mitigation for Metldown-type attacks replaced all transiently forwarded values with zeros [223, 68]. Although this prevents data leakage of the Meltdown variants, it enabled a new LVI variant [223, 68]. With LVI-Nullify [68], we propose a software-based solution to prevent exploitation of LVI-Null in Intel Software Guard Extension (SGX) enclaves. Vendors mitigate Meltdown-type attacks with additional hardware mitigations that render the KAISER patch [71, 80] obsolete, as data is no longer incorrectly forwarded during transient execution. Furthermore, new instructions and buffer flushes were added to prevent stale data leakage after context switches [98]. Nevertheless, these buffer flushes do not prevent attacks from sibling threads performed in parallel with the victim. Finally, the now mitigated Downfall [162] attack exploits optimizations of the Single Instruction Multiple Data (SIMD) gather instruction to leak SIMD registers from an internal optimization buffer.

Reducing MDS Leakage. MDS variants still pose a threat to CPUs that do not turn off simultaneous multithreading (SMT) and do not include hardware fixes for MDS attacks. Although microcode updates extended context switches to flush stale data from affected buffers, sibling cores share the same buffers and can mount MDS attacks in parallel if the buffers have not been flushed yet [98]. In Chapter 7 [133], we found mitigations against CrossTalk [187] and Medusa [163] that reduce the overall leakage of these MDS attacks without the requirement to disable SMT. To reduce Crosstalk leakage, we found a CPU configuration register that allows to intercept accesses to the `cpuid` instruction. The `cpuid` instruction is used in the Crosstalk attack to move the content of the *staging buffer* to the *line-fill buffer*. Intercepting this primitve stops transient execution and the attacker has to rely on other primitives that have reduced leakage rates [133, 187]. The Medusa MDS variant III [163] uses implicit write-combining `rep` string instructions to leak data from previous `rep` string instructions. We found a configuration register that changes the internal behavior of the string instructions, i.e., the memory type for these string operations, effectively preventing leakage from `rep` string operations [133].

Re-enabling Timers. Cache side-channel attacks usually utilize timing primitives to measure memory access latencies (cf. Section 2.3). Distinguishing cache hits from cache misses allows to infer secrets that were encoded during transient execution. Therefore, a proposed mitigation idea for the Xen hypervisor was to make guest-accessible timers less granular [227]. In Chapter 7 [133], we found an MSR exposing an undocumented timer to the guest whose resolution was not limited in the Xen hypervisor [242]. With the high-resolution timer, we distinguish cache hits from cache misses and mount a Foreshadow [236] attack on the Xen hypervisor. Additional methods to amplify cache timings [118, 159, 216] were proposed to circumvent mitigations that make timers less granular.

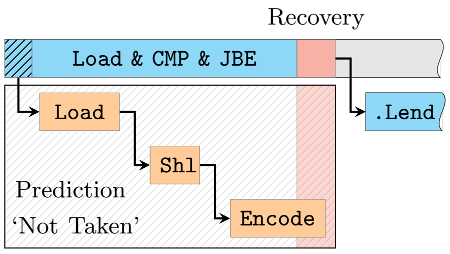
3.1.2. Spectre Attacks

Spectre [126] attacks exploit transient execution of incorrectly predicted control- or data-flow paths. The front-end relies on predictors for control-flow changes to issue μ-ops to the back-end. However, suppose a predictor returns the wrong control-flow prediction. In that case, the back-end must flush the execution pipeline and revert the control flow back to the correct position in the instruction stream, resulting in transiently executed instructions. During this transient window, traces in the microarchitecture can be recovered using side channels, similar to those in Meltdown-type attacks (cf. Section 3.1.1). Contrary to Meltdown, Spectre still has significant security relevance because Spectre is no CPU bug but rather an optimization with side effects [159]. One major problem with Spectre is that the structures responsible for the predictions can be mistrained from other processes or security domains. Therefore, an attacker process could mistrain the predictor and open a transient window in the victim process. Canella et al. [33] categorized the Spectre variants based on the microarchitectural predictor they use to start a transient window.

Spectre-PHT [126] mistrains the Pattern History Table (PHT) to influence the predictions for conditional control-flow changes. Spectre-PHT is especially problematic since it exploits a common programming pattern: the out-of-bounds check. For instance, if a program requires an array to

```
 1 # rdi = &array_address
 2 # rdx = array_index
 3 # rcx = &array_size
 4 # rsi = &lookup_table
 5 spectre_pht:
 6   cmpq %rdx, (%rcx)
 7   jbe .Lend
 8   movzbl (%rdi, %rdx), %eax
 9   shll $12, %eax
10   movb (%rsi,%rax), %al
11 .Lend
12   ...
```

(a) Spectre-PHT in assembly code.



(b) Sequence diagram of Spectre-PHT.

Figure 3.2.: The assembly code and sequence diagram of Spectre-PHT. The attacker has control over the index variable in rdx and can trigger the gadget repeatably to mistrain the PHT. The attacker specifies an out-of-bounds index, leading to transient execution until the value of the **array_size** (rcx) variable is available (Line 6). During transient execution the subsequent instructions will use the index, access the array out-of-bounds (Line 8), and encode the leaked value into the lookup table (Line 10). After the size variable is available, the misprediction is resolved, and the correct execution path is executed.

be indexed, the index is usually verified to be within the defined bounds of the array. However, during the transient window, the PHT can predict to take the *in-bound branch*, although the actual index variable is out of bounds. These mispredictions allow an attacker to access data beyond the defined array, i.e., any data belonging to the process. Figure 3.2 shows a Spectre-PHT gadget where the transiently accessed out-of-bounds data is re-encoded in a look-up table. This additional indirection allows an attacker to recover the secret value via a cache side channel (cf. Section 2.3.1).

Spectre-RSB [153, 135] exploits the Return Stack Buffer (RSB) to start transient execution. The RSB is responsible for predicting targets of return instructions that match a previously executed call instruction. Due to the implementation of the RSB, the RSB does not track modifications to return addresses on the stack directly. This insight allows the implementation of the *Retpoline* Spectre-BTB mitigation [221]. Instead of an indirect call or branch, the control flow is transferred by moving the jump address onto the stack and invoking the return instruction as depicted in Figure 3.3.

Canella et al. [31] and Xiong et al. [244] summarize research towards Spectre mitigations. We focus on actively deployed mitigations from CPU

```
 1  function:
 2      call retpoline(%rip)
 3  .Lloop:
 4      jmp .Lloop
 5  retpoline:
 6      lea call_target(%rip), %rax
 7      mov %rax, (%rsp)
 8      ret
 9  call_target:
10      ...
```

Figure 3.3.: The assembly code of the Retpoline mitigation. Due to the RSB not tracking direct modifications on the stack, indirect calls can be replaced by calling a Retpoline helper (Line 5) and modifying the return address on the stack (Line 7). During transient execution, the RSB will predict to continue at the return address of the previous call instruction (Line 3), and the transient execution will be trapped in an endless loop. After the misspeculation is resolved, the execution will continue to add the desired function (Line 9).

vendors and broadly applied software solutions. Intel and AMD introduced various mitigations to prevent predictions from different domains from influencing each other [100]. First, Indirect Branch Restricted Speculation (IBRS) prevents code running in a higher privilege level, e.g., the kernel, from reusing predictions from lower privilege domains like the user space. Second, the Indirect Branch Predictor Barrier (IBPB) prevents code following a barrier from being influenced by predictions made by the code before the barrier. Third, Single-Thread Indirect Branch Predictors (STIBP) prevent predictions from one logical CPU from influencing predictions of the other sibling. Finally, Retpoline [221] prevents Spectre-BTB by replacing indirect calls with an RSB misspeculation gadget to redirect transient execution. Most deployed hardware mitigations aim to prevent mistraining from one security domain to another. Therefore, mistraining within the process is still possible, leaving a large attack surface against in-process isolation, i.e., processes containing multiple security domains [40, 42]. Furthermore, some of these mitigations are ineffective against newer Spectre gadgets and mistraining techniques [10, 240, 220, 239].

Spectre as Data Corruption Validation. In Chapter 6 [132], we use a Spectre-PHT gadget to verify if the data corruption of our Rowhammer attack can be further exploited. We target the metadata of page-table entries (cf. Section 2.1.1). However, corrupting the reserved bits of the page-

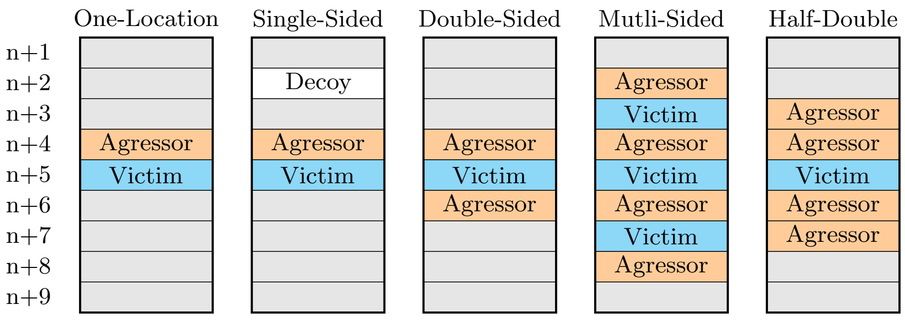


Figure 3.4.: Common Rowhammer hammering patterns. The patterns vary in the number of aggressor rows and their placement. Multi-sided hammering patterns surround aggressor rows with multiple victim rows.

table entry will result in an unrecoverable page fault that terminates the attacker process on our test device. Our Spectre verification gadget exploits that during transient execution, page faults of subsequent instructions are suppressed until retirement (cf. Section 2.2.1). Therefore, we access the memory pointed by the page-table entry in speculation and encode the loaded value into a cache side channel. The memory access will succeed if no reserved bits are corrupted. Afterward, the misspeculation is reverted. Therefore, if we observe the encoded value in the cache side channel, the access was successful in the transient domain, and the address is safe to access architecturally, i.e., no unwanted data corruption occurred. Otherwise, the page-table entry is corrupted, and accessing it will terminate the process. A similar primitive was used by Ravichandran et al. [189] to brute force ARM's pointer authentication codes during transient execution.

3.2. Rowhammer

Rowhammer [123] is a software-based fault attack discoverd in 2014 targeting data-at-rest inside the main memory. Seaborn et al. [205, 206] demonstrate the security implications of Rowhammer by exploiting bit flips in memory to gain kernel-level privileges. Additional attacks using Rowhammer have been used for privilege escalation [228, 132, 4, 22, 44, 72, 74, 190, 243, 253], target data integrity [107, 115, 213, 192, 106, 59, 15, 34, 58, 103, 104, 143, 178, 181, 237, 252], manipulate the accuracy of machine learning models [246], and as a side channel [136, 43, 217, 171].

Hammering Patterns. The amount of bit flips induced by a Rowhammer attack depends on the row access pattern within a DRAM bank. The rows accessed by an attacker are called *aggressor* rows and the exact row layout forms a hammering pattern. The *single-sided* hammering pattern uses one neighboring row and any additional row in the same bank to force frequent row activations [205, 206, 74] (cf. Section 2.1.3). The *double-sided* hammering pattern uses two aggressor rows directly adjacent to the victim row and maximizes the locality of disturbance errors in the victim row [205, 206]. The *one-location* variation uses only a single aggressor to induce bit flips [72]. Recent Rowhammer attacks use *multi-sided* hammering patterns consisting of multiple double-sided attacks in the same DRAM bank [59, 192]. We propose two novel hammering patterns in Chapter 6 [132], combining disturbance errors of aggressors beyond direct neighboring rows. Jattke et al. [106] introduce the Blacksmith fuzzer to find the optimal hammering pattern for a given DIMM that maximizes the number of bit flips. Figure 3.4 summarizes common Rowhammer hammering patterns.

DRAM addressing functions are crucial to map a physical address to the exact topology in the main memory (cf. Section 2.1.3). However, vendors usually do not document the addressing functions, resulting in additional challenges in placing the aggressor rows according to the hammering pattern. Pessl et al. [176] reverse-engineer the DRAM addressing functions of various DIMMs and found that the functions are usually composed of multiple xor functions. Furthermore, Pessl et al. found a bank-conflict side channel to identify if two addresses belong to the same bank. Additional methods were proposed to reverse-engineer the addressing functions using bank conflicts [11], temperature [114], and performance counters [82]. Helm et al. [82] found CPUs that assign distinct addressing functions to physical address ranges. Gerlach et al. [66] develop an efficient solver to recover non-linear addressing functions from side-channel measurements.

With the DRAM addressing function, an attacker can precisely place the aggressor rows in the main memory if the physical address is known. However, physical address information is no longer available to an unprivileged user [205, 206, 124]. Furthermore, huge pages like 2 MiB or 1 GiB pages allow to infer physical address bit directly from a virtual address (cf. Section 2.1.1) [72, 192]. Nevertheless, huge pages were restricted by operating systems and are no longer handed out to unprivileged processes. Islam et al. [103] use microarchitectural side channels to partially recover physical address bits and identify whether pages are contiguous in the main memory. Contiguous memory allows identifying the relative location of DRAM rows required for the hammering pattern despite not knowing the absolute position in the DRAM bank. Schwarz et al. [202] use the bank-conflict side channel and known xor-based DRAM addressing functions to recover partial physical address information. Kwong et al. [136] use contiguous memory to recover DRAM bank information. In Chapter 6 [132], we propose combining the bank-conflict side channel with insights into the Linux buddy allocator. With this technique, we identify whether a memory region could be contiguous to identify the relative positioning for the Half-Double hammering pattern.

Mitigations in Commodity Hardware. The first *temporary* Rowhammer mitigation was doubling the refreshes to the individual rows [8]. However, Kim et al. [123] and Aweke et al. [8] show that increasing the refresh rate does not prevent Rowhammer bit flips on some DIMMs. Commercial devices focus on two common Rowhammer mitigations: First, Error Correcting Codes (ECC) are established to prevent non-malicious data flips due to cosmic radiation or aging of the hardware. However, ECC offers limited protection against Rowhammer attacks. Second, to prevent Rowhammer attacks, the vendors deploy refresh-based mitigations. These mitigations count the number of activations to a row and issue additional refreshes to neighboring rows, preventing the accumulation of disturbance errors. LPDDR4(x) and DDR4 memory utilize a refresh-based mitigation technique called Target Row Refresh (TRR) [108]. Pseudo-TRR (pTRR) is a refresh-based Rowhammer mitigation implemented in the CPU's memory controller to track DRAM accesses and issue additional refreshes.

Beyond ECC and TRR. Cojocar et al. [44] reverse-engineer the ECC functions and find a timing side channel that indicates whether the hardware corrected a *correctable* bit flip. With this knowledge, they perform targeted Rowhammer attacks circumventing ECC and inducing *uncorrectable* bit flips in the main memory. Kwong et al. [136] present the Rambleed attack and use the ECC timing side channel to leak data from the main memory. Frigo et al. [59] demonstrate that early TRR mitigations can be overwhelmed by multi-sided Rowhammer attacks. Due to limited hardware inside the DIMMs to track aggressor rows, many of the aggressors' neighbors will not be refreshed. In Chapter 6 [132], we introduce the Half-Double effect, which combines the electrical disturbance of rows beyond the direct neighboring rows of the victim. We demonstrate in Chapter 6 [132] that TRR is not only ineffective against the Half-Double attack but fundamentally assists an attacker on commodity devices. Hassan et al. [81] propose a methodology to categorize, analyze, and reverse-engineer TRR implementations using a dedicated FPGA board. De Ridder et al. [192] synchronize multi-sided hammering patterns with the refreshes of the DRAM module, increasing the activation count within the refresh interval. A similar synchronization was used by Jattke et al. [107] to evade Rowhammer mitigations on Zen 2 and 3 CPUs. Kang et al. [115] found additional mitigations implemented in the memory controller of Intel CPUs and circumvented these mitigations by hammering multiple DRAM banks simultaneously. Luo et al. [151] present RowPress, which induces bit flips in the main memory using the passing gate effect instead of the Rowhammer effect [86] (cf. Section 2.4). Juffinger et al. [113] analyze RowPress with regards to single- and double-sided hammering patterns.

Academic Mitigations. Multiple research papers propose mitigations to prevent Rowhammer attacks since the original discovery in 2014 [123]. Rowhammer mitigations neutralize, detect, or eliminate bit flips in the main memory [72]. The following software-based mitigations were proposed. Anvil [8] is a detection-based mitigation tracking accesses to memory locations using performance counters (cf. Section 2.2.2). CATT [23] and Zebram [134] neutralize bit flips by distancing vulnerable data in the main memory. Copy-on-Flip [49] migrates pages that observe frequent ECC events to a new location to prevent targeted exploitation of Rowhammer bit flips. The subsequent mitigations require additional hardware changes. Graphene [173] proposes an optimized solution with a reduced area overhead compared to conventional counter-based Rowhammer mitigations. ProTRR [156] and Rega [157] optimize an adapted TRR algorithm, giving vendors additional configuration parameters to fine tune the mitigation. With CSI:Rowhammer [112], we propose a novel hardware-software co-design replacing ECC with a cryptographic MAC to detect bit flips. CSI:Rowhammer delegates error correction beyond single bit flips to the operating system, allowing for great flexibility when correcting file-backed data. Similarly, we propose PT-Guard [196] partitioning a MAC within the free bits of page-table entries. Additional mitigations dynamically remap rows within a DRAM bank to prevent contiguous hammering of neighboring rows [194, 241, 197]. Defenses building on the characteristics of Rowhammer bit flips have been demonstrated to be ineffective against modern Rowhammer variants [112]. For instance the underlying assumptions of CATT and Zebram were invalidated with the Half-Double effect [132] and further distanced Rowhammer flips [123].

3.3. Fault Attacks using DVFS Interfaces

The threat model of Trusted Execution Environments (TEEs) [45] allows an attacker to use Dynamic Voltage- and Frequency-Scaling (DVFS) interfaces and configure the processor to operate beyond *stable* voltage and frequency conditions (cf. Section 2.4). The CLKSCREW attack [212], induces faulty computation into ARM TrustZone by using a software-based overclocking interface. With these faulty computations, CLKSCREW demonstrates leaking cryptographic keys and loading self-signed applications into TrustZone. A similar interface was available on Intel CPUs to request a reduced operating voltage and potentially reduce the processor's power consumption. Plundervolt [167] uses the software-based undervolting interface to induce faulty computations into Intel Software Guard Extension (SGX) enclaves. Plundervolt demonstrates attacks to fault multiplications, leak RSA keys using the Bellcore and Lenstra fault-injection techniques [18, 139], and recover AES-NI keys, i.e., from a side-channel resistant AES hardware implementation. Similar attacks using undervolting interfaces have been shown on SGX [120, 184], TrustZone [183], and AMD Zen CPUs [186]. Due to the impact of these software-based DVFS attacks, Intel disabled the undervolting interface when using SGX [17, 130].

The removal of the software-based interfaces sparked research on low-cost hardware attacks within the physical access threat model. The Voltpillager [39] attack targets the exposed voltage regulators outside the CPU package. An attacker can inject voltage requests directly into the bus between the regulators and the CPU, re-enabling Plundervolt-style attacks. Newer Intel CPU generations deploy a fully integrated voltage regulator design that contains the voltage regulators in the CPU package and prevents low-cost hardware undervolting attacks [130, 29]. Buhren et al. [28] compromise AMD's Secure Encrypted Virtualization (SEV) using a voltage glitch to gain control over AMD's secure processor. Mahmoud et al. [152] exploit a CPU with integrated FPGA to construct a circuit on the FPGA that influences the CPU's operating voltage and induces faults in operations.

Mitigations against fault attacks like additional redundancy or backward computations (cf. Section 2.4) are applicable to protect against software-based attacks. However, these mitigations have a performance overhead, and backward computations are often only feasible for some computations. Furthermore, the fault injection capabilities of software-based attacks are weaker compared to a traditional hardware attacker [130]. Chen et al. [38] propose CAMFAS, an optimized redundancy fault attack mitigation that uses Single Instruction Multiple Data (SIMD) registers and instructions to compute the redundant paths with less performance overhead. However, the performance gains depend on whether the protected software does not already utilize SIMD registers and instructions. In Chapter 5 [130], we propose a probabilistic fault attack detection and mitigation for SGX enclaves. Our approach has no pre-requirements for the actual software to be protected. Instead, we place *trap* instruction inside the code and constantly verify if these instructions still compute the correct result. To find a suitable trap instruction, we analyzed the instructions of the x86 instruction set for their fault susceptibility with varying voltage and frequency operating points. We found that integer multiplication is the most susceptible instruction across the tested CPUs and faults most reliably before other instructions observe faulty computations.

3.4. Software-based Power Analysis

Software-based power analysis replaces the external measurement equipment of traditional power analysis with power-related signals or side effects obtainable directly via software (cf. Section 2.3.2). The Android operating system reported the power consumption to applications until Android 7 [182]. Yan et al. [245] demonstrate power side channels using the Android interface and distinguish keystrokes, password lengths, and applications. Michalevsky et al. [160] exploit power-consumption fingerprints to identify the device's location in a predefined set of possible locations. Chen et al. [37] use power profiles to fingerprint applications. Similarly, Qin et al. [182] fingerprint websites using energy readings.

Like the Android interface, the RAPL interface can be used to mount power side channels on x86 CPUs (cf. Section 2.3.2). Mantel et al. [155] use the RAPL interface to record the power consumption during RSA operations and distinguish RSA keys based on the Hamming weight of the keys. Fusi [61] demonstrate information leakage of 16 KiB large RSA keys if the keys contain large spaces of zeros between the individual ones. Larger RSA keys have the property of longer durations between the processing of each key bit and counteract the reduced timing resolution of the interface.

Gao et al. [63] use the available RAPL interface in a guest virtual machine to fingerprint the host. Zhang et al. [251] showcase a covert channel and perform website fingerprinting using the RAPL interface.

These power side channels show a similar pattern of fingerprinting distinct power signatures or partially recovering cryptographic keys. Traditional power-analysis attacks [127, 25] were considered an unrealistic attack vector for remote systems, as the timing and value resolution were reduced compared to external measurement equipment. With the Platypus attack [146], we advanced the field of software-based power analysis when using the RAPL interface. First, we show Correlation Power Analysis (CPA) on AES-NI, i.e., a side-channel resistant hardware implementation of AES, when performing block encryption. Second, we improve the relatively low signal quality of the interface by using microarchitectural replay techniques in the context of SGX [224]. This primitive allows us to record and amplify fine granular, per instruction energy readings and attack RSA, even if concealed within SGX. Fixes for the Platypus attack remove unpriviledged access to the interface and filter the interface to no longer report data-dependent energy consumption accurately when using SGX [99, 238]. Martínez et al. [158] introduce a systematization for remote power analysis and summarize techniques to measure the power consumption. Wang et al. [233] use similar techniques as in the Platypus attack to mount power side channels on AMD SEV. PowSpectre [102] proposes techniques to use Intel Transactional Synchronization Extension (TSX) to repeat the victim code and amplify power leakage.

Modern CPUs utilize DVFS to *dynamically* increase the operating frequency and performance for demanding workloads. However, the hardware enforces thermal and power limits to prevent physical damage to the CPU. If a workload reaches these limits, the CPU throttles the frequency to lower the power consumption, resulting in reduced performance. Wang et al. [234] and Liu et al. [148] discover that the timing variations due to throttling can be exploited as an indirect power-related signal, i.e., a signal that is not directly exposing the power consumption but correlates with it. Wang et al. [234] present the Hertzbleed attack, exploiting these timing differences remotely and leaking cryptographic keys from the SIKE post-quantum key exchange implementation. Liu et al. [148] mount a CPA attack (cf. Section 2.3.2) on AES-NI on Intel and AMD CPUs with these types of power-related signals. Wang et al. [235] and Taneja et al. [211] extend frequency-throttling attacks to integrated and dedicated GPUs. Cohen et al. [43] found another power-related signal in the number of bit flips a Rowhammer attack induces, i.e., the number of bit flips correlates to the power consumption of the DRAM DIMM (cf. Section 2.1.3).

In Chapter 8, we introduce Collide+Power [131] and show that data *collisions* in hardware expose the Hamming distance of the values in the power domain. With Collide+Power, we extend software-based power analysis to leak general-purpose data using threat models similar to Meltdown and MDS. Furthermore, we propose a measurement technique to improve the signal quality of the power-related signal.

Software-based power side channels were also explored by crossing hardware domains to construct energy reading primitives. Zhao et al. [254] use the FPGA of a heterogeneous computing system to construct a ring oscillator capable of measuring a power-related signal. They demonstrate attacks on components of the FPGA and attack on an RSA implementation running on the CPU of the SoC. O'Flynn et al. [170] show power side-channel attacks using an integrated analog-to-digital converter.

4 Conclusion

In this thesis and the corresponding publications, we explored the synergy of traditional hardware attacks with the microarchitectural world of transient execution, system interfaces, and components. We summarize the contributions of the thesis and publications with three key insights.

Transient execution can be exploited to enhance software-induced hardware attacks even in the post-transient era, where Meltdown and MDS variants have been fixed in hardware. We constructed a corruption validation primitive which acts as an oracle to validate if fault injection succeeded and suppress failed attempts (see Chapter 6 [132]). Similar primitives have been used without physical context to break pointer authentication codes [189] or exfiltrate physical address information [103]. Furthermore, we exploited transient execution to enhance software-based power analysis techniques in Chapter 8 [131]. Transient replay gadgets [131, 102] amplify subtle power leakage to make it measurable with power-related signals. We also showed that transient execution can be leveraged to expose data to power analysis in specific buffers [110, 131]. Previously, transient-execution attacks leaked targeted data directly. However, it is likely future work will shift away from transient-execution attacks as primary leakage primitive towards building blocks to move data or validate conditions during transient execution. A similar shift occurred with some side-channel attacks, using side channels as the primary data leakage primitive when targeting non-constant-time code. In more recent works, side channels are commonly used as building blocks, e.g., to encode secrets transiently.

Software-based fault attacks like Rowhammer and DVFS-based fault injection are significant threats to system security, and a single faulted data value or computation can hijack an entire system. Therefore, identifying the fault characteristics of these software-based fault attacks is crucial to developing efficient mitigations. In Chapter 6 [132] and Chapter 5 [130], we present methodologies to identify how faults manifest on the hardware. First, we demonstrated faults violating the existing assumptions of refresh-based Rowhammer mitigations. Second, we identified that some instructions are more susceptible to DVFS-based fault injection than others. Based on our fault-susceptibility analysis, we have presented a previously unknown Rowhammer variant and designed a novel DVFS fault detection mechanism. Finally, this analysis has already layed the foundation for follow-up work on improving energy efficiency by making undervolting more resilient [111]. In the future, we expect an increase in the number of faults experienced on systems as manufacturing processes shrink, as is already the case with DRAM. Furthermore, software-based fault attacks in other memory technologies like SRAM are likely to emerge as a new threat. Therefore, future research may shift towards principled mitigations that do not rely on specific fault characteristics [112, 196].

System interfaces are essential for monitoring and configuring the performance of complex processors. However, exposing interfaces to manage and monitor physical properties to unprivileged domains can have unforeseen security implications as evidenced by software-based power analysis [146] and fault attacks [212, 167]. Therefore, the interfaces to manage and monitor physical properties were restricted. In Chapter 7 [133], we presented a methodology to identify additional interfaces that expose exploitable signals of physical properties. Removing interface access to physical signals is a temporary solution, as physical properties are closely entangled with the system's power and thermal management. Both, the execution time, due to DVFS [234, 148], as well as the number of Rowhammer flips [43] is influenced by physical properties. Consequently, attacks can still observe physical properties without explicit interfaces. These signals form an additional layer of indirection to obtain the actual physical property. Therefore, one could consider these signals as *higher-order side channels*. The nature of these side channels varies widely, e.g., from flips in memory to execution time differences. Hence, significant future research is necessary to identify higher-order side channels that reintroduce leakage previously deemed security critical which therefore has been mitigated in its direct form.

Future Work Outlook

In this section, we summarize our outlook for future research. First, future research could deepen the understanding of the possibilities of microarchitectural primitives as building blocks for exploiting physical properties. For example, primitives like port contention [6] can likely force new *collisions* of data in the hardware, making the combined power leakage exploitable via software-based power analysis. Due to the increased throughput of execution ports, we expect the power leakage to be more substantial than that of the memory subsystem. Similarly, research to find new ways of speculatively interacting with data in the memory subsystem might also increase power leakage. For instance, newer CPUs implement data-dependent prefetchers that fetch additional memory into the caches by *observing* data values and deciding if the value points to memory.

Second, in the direction of software-based fault attacks, future research will likely shift toward new types of buffers in the hardware. For example, DRAM-based buffers might be used in more processor components, which might cause them to be inherently affected by Rowhammer. Furthermore, active workloads influence the highly complex power management and the integrated control systems. Future research could investigate how these systems monitor workloads and how the power management reacts to these workloads internally. The results of such an analysis could be essential to identify how to bring a processor close to the limits of stable operation and potentially enable new types of software-based fault attacks.

Third, research towards reverse engineering power management could also directly benefit power analysis attacks. Finding a *sweet spot* where the signal-to-noise ratio is at the maximum could drastically reduce the required traces and the attack runtime. Furthermore, a new research direction could harden software on existing hardware with additional instructions to explicitly prevent data collisions in a specific buffer or processor component and prevent certain power analysis attacks. Finally, future research could focus on identifying additional power-related signals and systematizing them based on their properties and leakage rates.

References

[1] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In: ISPASS. 2020 (p. 18).

[2] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In: ISPASS. 2014 (p. 18).

[3] Onur Acıiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES. In: IACR Cryptology ePrint Archive, Report 2006/138 (2006) (p. 3).

[4] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks. In: HOST. 2017 (p. 35).

[5] National Security Agency. TEMPEST: A Signal Problem. 1972. URL: `https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf` (p. 22).

[6] 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 (p. 45).

[7] AMD. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 00h-0Fh Processors. 2015. URL: `https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/48751_16h_bkdg.pdf` (pp. 21, 25, 28).

[8] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. In: ACM SIGPLAN Notices 51 (2016), pp. 743–755 (pp. 37, 38).

[9] John E Ayers. Digital Integrated Circuits: Analysis and Design. CRC Press, 2003 (p. 24).

[10] 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. 2022 (p. 34).

[11] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical DRAM mappings for rowhammer attacks. In: International Verification and Security Workshop (IVSW). 2018 (p. 36).

[12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: Workshop on Embedded Systems Security. 2010 (p. 27).

[13] Shajulin Benedict. Energy-Aware Performance Analysis Methodologies for HPC Architectures—An Exploratory Study. In: Journal of Network and Computer Applications 35 (2012), pp. 1709–1719 (p. 25).

[14] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 23, 24).

[15] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: CHES. 2016 (p. 35).

[16] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In: CRYPTO. 1997 (p. 27).

[17] Douglas Black. Intel & OEMs are disabling undervolting. Here's how to re-enable it. 2020. URL: `https://www.ultrabookreview.com/37095-dells-disabling-undervolting-on-their-laptops-heres-how-to-re-enable-it/` (p. 39).

[18] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the Importance of Eliminating Errors in Cryptographic Computations. In: Journal of cryptology 14 (2001), pp. 101–119 (p. 39).

[19] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In: EUROCRYPT. 1997 (p. 27).

[20] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In: CHES. 2006 (p. 3).

[21] 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. 2022 (pp. 6, 10).

[22] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P. 2016 (p. 35).

[23] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In: USENIX Security. 2017 (p. 38).

[24] 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 (p. 3).

[25] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In: CHES. 2004 (pp. 4, 22, 25, 26, 41).

[26] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In: USENIX Security. 2020 (pp. 18, 24).

[27] Billy Brumley and Risto Hakala. Cache-Timing Template Attacks. In: AsiaCrypt. 2009 (p. 4).

[28] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In: CCS. 2021 (pp. 27, 39).

[29] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. FIVR—Fully integrated voltage regulators on 4th generation Intel Core SoCs. In: APEC. 2014 (p. 39).

[30] 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 (pp. 4, 30).

[31] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In: GLSVLSI. 2020 (p. 33).

[32] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (p. 24).

[33] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security. 2019 (pp. 4, 21, 30, 32).

[34] Anirban Chakraborty, Sarani Bhattacharya, Sayandeep Saha, and Debdeep Mukhopadhyay. Explframe: exploiting page frame cache for fault analysis of block ciphers. In: DATE. 2020 (p. 35).

[35] Joseph S Chang, Antonio F Facchetti, and Robert Reuss. A Circuits and Systems Perspective of Organic/Printed Electronics: Review, Challenges, and Contemporary and Emerging Design Approaches. In: IEEE Journal on Emerging and Selected Topics in Circuits and Systems 7 (2017), pp. 7–26 (p. 24).

[36] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In: CHES. 2002 (pp. 25, 27).

[37] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. POWERFUL: Mobile app fingerprinting via power analysis. In: INFOCOM. 2017 (p. 40).

[38] Zhi Chen, Junjie Shen, Alex Nicolau, Alex Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. CAMFAS: A compiler approach to mitigate fault attacks via enhanced SIMDization. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2017 (p. 40).

[39] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In: USENIX Security. 2020 (pp. 5, 7, 39).

[40] Cloudflare. Cloudflare Workers. 2019. URL: `https://www.cloudflare.com/products/cloudflare-workers/` (p. 34).

[41] Cloudflare. Cloudflare Workers. 2021. URL: `https://www.cloudflare.com/products/cloudflare-workers/` (p. 11).

[42] Cloudflare. Mitigating Spectre and Other Security Threats: The Cloudflare Workers Security Model. 2020. URL: https://blog.cloudflare.com/mitigating-spectre-and-other-security-threats-the-cloudflare-workers-security-model/ (pp. 11, 34).

[43] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D Keromytis, Yossi Oren, and Yuval Yarom. HammerScope: Observing DRAM Power Consumption Using Rowhammer. In: CCS. 2022 (pp. 35, 41, 44).

[44] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In: S&P. 2019 (pp. 5, 35, 37).

[45] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (p. 39).

[46] 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 (p. 4).

[47] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In: ACM/IEEE International Symposium on Low Power Electronics and Design. 2010 (p. 25).

[48] Santos Merino Del Pozo, François-Xavier Standaert, Dina Kamel, and Amir Moradi. Side-Channel Attacks from Static Power: When Should we Care? In: DATE. 2015 (p. 25).

[49] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In: NDSS. 2023 (p. 38).

[50] 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. 2017 (p. 24).

[51] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical ineffective fault attacks on masked AES with fault countermeasures. In: ASIACRYPT. 2018 (p. 27).

[52] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. In: IACR Transactions on Cryptographic Hardware and Embedded Systems (2018), pp. 547–572 (p. 27).

[53] Christopher Domas. Breaking the x86 ISA. In: Black Hat USA. 2017 (pp. 8, 22).

[54] Christopher Domas. Continuing to Break the x86 Instruction Set. In: Shakacon (2018) (pp. 8, 22).

[55] Christopher Domas. Hardware Backdoors in x86 CPUs. In: Black Hat USA. 2018 (pp. 8, 22).

[56] Jack Dongarra, Hatem Ltaief, Piotr Luszczek, and Vincent M Weaver. Energy Footprint of Advanced Dense Numerical Linear Algebra using Tile Algorithms on Multicore Architectures. In: International Conference on Cloud and Green Computing. 2012 (p. 25).

[57] Agner Fog. The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2021. URL: `https://www.agner.org/optimize/microarchitecture.pdf` (pp. 15, 16, 20).

[58] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (p. 35).

[59] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In: S&P. 2020 (pp. 5, 8, 35–37).

[60] Thomas Fuhr, Eliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In: Workshop on Fault Diagnosis and Tolerance in Cryptography. 2013 (p. 27).

[61] Matteo Fusi. Information-Leakage Analysis Based on Hardware Performance Counters. MA thesis. Politecnico di Milano, 2017 (pp. 25, 40).

[62] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In: CHES. 2001 (p. 22).

[63] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In: DSN. 2017 (p. 41).

[64] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks. In: FC. 2024 (pp. 6, 11).

[65] 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 (pp. 6, 11).

[66] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. Efficient and generic microarchitectural hash-function recovery. In: S&P. 2024 (p. 36).

[67] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. Generic and Automated Drive-by GPU Cache Attacks from the Browser. In: AsiaCCS. 2024 (pp. 6, 12).

[68] Lukas Giner, Andreas Kogler, Claudio Canella, Michael Schwarz, and Daniel Gruss. Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX. In: USENIX Security. 2022 (pp. 6, 11, 31).

[69] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In: EuroSec. 2017 (p. 3).

[70] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In: USENIX Security. 2017 (p. 18).

[71] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 4, 30, 31).

[72] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (pp. 35, 36, 38).

[73] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 15, 24).

[74] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 35, 36).

[75] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 23).

[76] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-after-freemail: Generalizing the use-after-free problem and applying it to email services. In: AsiaCCS. 2018 (p. 3).

[77] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 4, 23).

[78] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P. 2011 (p. 23).

[79] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. In: ACM SIGMETRICS Performance Evaluation Review 40 (2012), pp. 13–17 (p. 25).

[80] Dave Hansen. KAISER: unmap most of the kernel from userspace page table. 2017. URL: `https://lkml.org/lkml/2017/10/31/884` (pp. 4, 30, 31).

[81] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications. In: MICRO. 2021 (p. 38).

[82] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters. In: Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE. 2020 (p. 36).

[83] John L Hennessy and David A Patterson. Computer Architecture: A Quantitative Approach. 6th ed. Morgan Kaufmann, 2017 (pp. 14, 18, 20).

[84] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level. In: EuroSys. 2021 (p. 25).

[85] Johann Heyszl, Stefan Mangard, Benedikt Heinz, Frederic Stumpf, and Georg Sigl. Localized Electromagnetic Analysis of Cryptographic Implementations. In: CT-RSA. 2012 (p. 22).

[86] Seungki Hong, Dongha Kim, Jaehyung Lee, Reum Oh, Changsik Yoo, Sangjoon Hwang, and Jooyoung Lee. DSAC: Low-Cost Rowhammer Mitigation Using In-DRAM Stochastic and Approximate Counting Algorithm. In: arXiv preprint (2023) (pp. 28, 38).

[87] Wei-Ming Hu. Lattice Scheduling and Covert Channels. In: S&P. 1992 (p. 3).

[88] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 24).

[89] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In: International Conference on Smart Card Research and Advanced Applications. Springer. 2013, pp. 219–235 (p. 22).

[90] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In: CHES. 2016 (p. 3).

[91] Intel. 5-level paging and 5-level EPT. 2017. URL: `https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html` (p. 14).

[92] Intel. Affected Processors: Transient Execution Attacks. 2020. URL: `https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model` (p. 30).

[93] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. 2019. URL: `https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations` (pp. 4, 26).

[94] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2023 (pp. 16–18).

[95] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (pp. 21, 25, 28).

[96] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers. May 2019 (p. 21).

[97] Intel. Intel Optane DC Persistent Memory. 2021. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html` (pp. 11, 12).

[98] Intel. Intel-SA-00233 Microarchitectural Data Sampling Advisory. 2019. URL: `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html` (p. 31).

[99] Intel. Intel-SA-00389. 2020. URL: `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html` (p. 41).

[100] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (p. 34).

[101] 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 (p. 18).

[102] Hafizul Islam, Zhenkai Zhang, and Fan Yao. PowSpectre: Powering Up Speculation Attacks with TSX-based Replay. In: AsiaCCS. 2024 (pp. 25, 41, 43).

[103] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security. 2019 (pp. 35, 36, 43).

[104] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (p. 35).

[105] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (p. 24).

[106] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Rowhammering in the Frequency Domain. In: S&P. Nov. 2021 (pp. 35, 36).

[107] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In: USENIX Security. 2024 (pp. 35, 38).

[108] JEDEC Solid State Technology Association. Low Power Double Data Rate 4. 2017. URL: `http://www.jedec.org/standards-documents/docs/jesd209-4b` (pp. 5, 37).

[109] Jedec Solid State Technology Association. Low Power Double Data Rate 3. 2013. URL: `http://www.jedec.org/standards-documents/docs/jesd209-4a` (pp. 18, 19).

[110] 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 (p. 43).

[111] Jonas Juffinger, Stepan Kalinin, Daniel Gruss, and Frank Mueller. SUIT: Secure Undervolting with Instruction Traps. In: ASPLOS. 2024 (pp. 5–7, 28, 44).

[112] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI: Rowhammer - Cryptographic Security and Integrity against Rowhammer. In: S&P. 2023 (pp. 6, 10, 38, 44).

[113] Jonas Juffinger, Sudheendra Raghav Neela, Martin Heckel, Lukas Schwarz, Florian Adamsky, and Daniel Gruss. Presshammer: Rowhammer and Rowpress without Physical Address Information. In: DIMVA. 2024 (p. 38).

[114] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse engineering of DRAMs: Row hammer with crosshair. In: International Symposium on Memory Systems. 2016 (p. 36).

[115] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism. In: USENIX Security. 2024 (pp. 35, 38).

[116] Matthias J Kannwischer, Peter Pessl, and Robert Primas. Single-Trace Attacks on Keccak. In: Cryptology ePrint Archive (2020) (p. 25).

[117] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware Designer's Guide to Fault Attacks. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 21 (2013), pp. 2295–2306 (p. 27).

[118] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In: USENIX Security. 2023 (p. 32).

[119] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. In: Journal of Computer Security 8.2/3 (2000), pp. 141–158 (p. 22).

[120] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In: USENIX Security. 2020 (pp. 5, 7, 27, 28, 39).

[121] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. In: ToMPECS 3 (2018), pp. 1–26 (p. 25).

[122] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In: ISCA. 2020 (pp. 5, 28).

[123] 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 (pp. 5, 8, 18, 27, 28, 35, 37–39).

[124] 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` (p. 36).

[125] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (pp. 3, 22).

[126] 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 (pp. 4, 11, 21, 32).

[127] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (pp. 4, 22, 25, 26, 41).

[128] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to Differential Power Analysis. In: Journal of Cryptographic Engineering 1 (2011), pp. 5–27 (p. 4).

[129] Andreas Kogler. Software-based Power Side-Channel Attacks. MA thesis. Graz University of Technology - Institute for Applied Information Processing and Communication, 2020 (p. 25).

[130] Andreas Kogler, Daniel Gruss, and Michael Schwarz. Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In: USENIX Security. 2022 (pp. 6, 7, 39, 40, 43).

[131] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In: USENIX Security. 2023 (pp. 6, 9, 25, 42, 43).

[132] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In: USENIX Security. 2022 (pp. 6, 8, 10, 34–39, 43).

[133] Andreas Kogler, Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz. Finding and Exploiting CPU Features using MSR Templating. In: S&P. 2022 (pp. 6, 8, 9, 31, 32, 44).

[134] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In: USENIX OSDI. 2018 (p. 38).

[135] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (p. 33).

[136] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (pp. 35, 37).

[137] Butler W Lampson. A note on the confinement problem. In: Communications of the ACM 16.10 (1973), pp. 613–615 (p. 3).

[138] Johnny KF Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. In: Computer 17 (1984), pp. 6–22 (p. 21).

[139] Arjen K Lenstra. Memo on RSA signature generation in the presence of faults. In: (1996) (p. 39).

[140] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel® Xeon 5500 processors. 2009 (p. 16).

[141] Chulseung Lim, Kyungbae Park, Geunyong Bak, Donghyuk Yun, Myungsang Park, Sanghyeon Baeg, Shi-Jie Wen, and Richard Wong. Study of proton radiation effect to row hammer fault in DDR4 SDRAMs. In: Microelectronics Reliability 80 (2018), pp. 85–90 (p. 28).

[142] Linux. Page Tables. 2024. URL: `https://docs.kernel.org/mm/page_tables.html` (p. 14).

[143] 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 (pp. 19, 35).

[144] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In: USENIX Security. 2022 (pp. 9, 24).

[145] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (pp. 3, 18, 23).

[146] 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 (pp. 4, 6, 9, 25, 41, 44).

[147] 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. 2018 (pp. 4, 9, 21, 29, 30).

[148] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In: CCS. 2022 (pp. 4, 9, 41, 44).

[149] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-aware intelligent DRAM refresh. In: ACM SIGARCH Computer Architecture News 40.3 (2012), pp. 1–12 (p. 19).

[150] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. Side-Channel Attacks on Optane Persistent Memory. In: USENIX Security. 2023 (p. 6).

[151] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In: ISCA. 2023 (p. 38).

[152] Dina G Mahmoud, David Dervishi, Samah Hussein, Vincent Lenders, and Mirjana Stojilović. DFAulted: Analyzing and exploiting CPU software faults caused by FPGA-driven undervolting attacks. In: IEEE Access 10 (2022), pp. 134199–134216 (p. 39).

[153] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 4, 21, 33).

[154] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer Science & Business Media, 2008 (pp. 4, 24–27).

[155] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. How Secure is Green IT? The Case of Software-Based Energy Side Channels. In: ESORICS. 2018 (p. 40).

[156] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In: S&P. 2022 (p. 38).

[157] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. Rega: Scalable rowhammer mitigation with refresh-generating activations. In: S&P. 2023 (p. 38).

[158] Macarena C Martínez-Rodríguez, Ignacio M Delgado-Lozano, and Billy Bob Brumley. SoK: Remote Power Analysis. In: ARES. 2021 (p. 41).

[159] 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) (p. 32).

[160] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In: USENIX. 2015 (p. 40).

[161] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (p. 4).

[162] Daniel Moghimi. Downfall: Exploiting Speculative Data Gathering. In: USENIX Security. 2023 (p. 31).

[163] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In: USENIX Security. 2020 (pp. 30, 31).

[164] Baker Mohammad. Embedded Memory Design for Multi-Core and Systems on Chip. Vol. 116. Springer, 2014 (pp. 15, 17).

[165] Amir Moradi. Side-Channel Leakage through Static Power: Should We Care about in Practice? In: CHES. 2014 (p. 25).

[166] Ira S Moskowitz. Noise effects upon a simple timing channel. In: NRL Memorandum Report (1990) (p. 3).

[167] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P. 2020 (pp. 5–7, 27, 28, 39, 44).

[168] Onur Mutlu. The RowHammer problem and other issues we may face as memory becomes denser. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). 2017 (p. 28).

[169] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. Springer Nature, 2020 (pp. 16, 18).

[170] Colin O'Flynn and Alex Dewar. On-Device Power Analysis Across Hardware Security Domains. In: CHES. 2019 (p. 42).

[171] Lois Orosa, Ulrich Rührmair, A Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. Spyhammer: Using rowhammer to remotely spy on temperature. In: arXiv:2210.04084 (2022) (p. 35).

[172] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 23, 24).

[173] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In: MICRO. 2020 (p. 38).

[174] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (pp. 23, 24).

[175] Chris H Perleberg and Alan Jay Smith. Branch target buffer design and optimization. In: IEEE transactions on computers 42 (1993), pp. 396–412 (p. 21).

[176] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security. 2016 (p. 36).

[177] Peter Pessl and Robert Primas. More Practical Single-Trace Attacks on the Number Theoretic Transform. In: LATINCRYPT. 2019 (p. 25).

[178] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In: EuroS&P. 2018 (p. 35).

[179] Robert Primas, Peter Pessl, and Stefan Mangard. Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption. In: CHES. 2017 (p. 25).

[180] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (p. 24).

[181] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In: HOST. 2016 (p. 35).

[182] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In: TrustCom/BigDataSE. 2018 (p. 40).

[183] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: CCS. 2019 (pp. 5, 7, 27, 28, 39).

[184] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In: AsianHOST. 2019 (pp. 5, 7, 27, 28, 39).

[185] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In: E-smart. 2001 (p. 22).

[186] Anja Rabich, Thomas Eisenbarth, and Luca Wilke. Software-based Undervolting Faults in AMD Zen Processors. In: its. uni-luebeck. de, no (2020) (p. 39).

[187] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In: S&P. 2021 (pp. 4, 30, 31).

[188] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (pp. 6, 12).

[189] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: attacking ARM pointer authentication with speculative execution. In: ISCA. 2022 (pp. 35, 43).

[190] 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. 2016 (p. 35).

[191] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Revisiting Fault Adversary Models–Hardware Faults in Theory and Practice. In: IEEE Transactions on Computers 72 (2022), pp. 572–585 (p. 27).

[192] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In: USENIX Security. 2021 (pp. 35, 36, 38).

[193] Teresa Riesgo and Javier Uceda. A Fault Model for VHDL Descriptions at the Register Transfer Level. In: European Design Automation Conference. 1996 (p. 27).

[194] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows. In: ASPLOS. 2022, pp. 1056–1069 (p. 38).

[195] 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 (p. 3).

[196] Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, Andreas Kogler, Daniel Gruss, and Moinuddin Qureshi. PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks. In: DSN. 2023 (pp. 6, 10, 38, 44).

[197] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime. In: MICRO. 2022 (p. 38).

[198] 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 (pp. 4, 9, 30).

[199] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple Photonic Emission Analysis of AES. In: CHES. 2012 (p. 22).

[200] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (pp. 24, 30).

[201] 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 (pp. 4, 9, 30).

[202] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 37).

[203] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. Dynamic Process Isolation. In: ESORICS. 2021 (p. 11).

[204] 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` (pp. 6, 11).

[205] Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. Mar. 2015. URL: `http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html` (pp. 35, 36).

[206] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat USA. 2015 (pp. 35, 36).

[207] John Paul Shen and Mikko H Lipasti. Modern processor design: fundamentals of superscalar processors. Waveland Press, 2013 (p. 18).

[208] James E Smith. A study of branch prediction strategies. In: ISCA. 1998 (p. 21).

[209] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (p. 30).

[210] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" Processor. In: IEEE Micro 40.2 (2020), pp. 45–52. DOI: 10.1109/MM.2020.2974217 (p. 16).

[211] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and ARM SoCs. In: USENIX Security. 2023 (p. 41).

[212] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security. 2017 (pp. 5, 7, 27, 28, 39, 44).

[213] 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 (p. 35).

[214] Niek Timmers and Cristofaro Mune. Escalating Privileges in Linux Using Voltage Fault Injection. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2017 (p. 28).

[215] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM using Fault Injection. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2016 (pp. 27, 28).

[216] Ben Titzer. What Spectre means for Language Implementers. 2019. URL: `https://pliss2019.github.io/ben_titzer_spectre_slides.pdf` (p. 32).

[217] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In: S&P. 2022 (p. 35).

[218] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In: IBM Journal of research and Development 11.1 (1967), pp. 25–33 (p. 20).

[219] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. In: Journal of Cryptology 23.1 (July 2010), pp. 37–71 (p. 3).

[220] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In: USENIX Security. 2023 (p. 34).

[221] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018. URL: `https://support.google.com/faqs/answer/7625886` (pp. 33, 34).

[222] 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. 2018 (pp. 4, 30).

[223] 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 (pp. 11, 31).

[224] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (p. 41).

[225] Jan Van den Herrewegen, David Oswald, Flavio D Garcia, and Qais Temeiza. Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. In: IACR Transactions on Cryptographic Hardware and Embedded Systems (2021), pp. 56–81 (p. 28).

[226] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adele Morisset, and Sébastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2017 (pp. 27, 28).

[227] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In: CCSW. 2011 (p. 32).

[228] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS. 2016 (p. 35).

[229] Akshay Venkatesh, Krishna Kandalla, and Dhabaleswar K Panda. Evaluation of Energy Characteristics of MPI Communication Primitives with RAPL. In: IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. 2013 (p. 25).

[230] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft Analytical Side-Channel Attacks. In: ASIACRYPT. 2014 (p. 25).

[231] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In: PLDI. 2020 (p. 18).

[232] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On DRAM Rowhammer and the Physics of Insecurity. In: IEEE Transactions on Electron Devices 68 (2021), pp. 1400–1410 (p. 28).

[233] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In: DIMVA. 2023 (p. 41).

[234] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In: USENIX Security. 2022 (pp. 4, 9, 41, 44).

[235] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W Fletcher, David Kohlbrenner, and Hovav Shacham. DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data. In: S&P. 2023 (p. 41).

[236] 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` (pp. 4, 30, 32).

[237] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. In: arXiv:1912.11523 (2019) (p. 35).

[238] Pu Wen. Add support for Hygon Fam 18h (Dhyana) RAPL. 2019. URL: `https://patchwork.kernel.org/patch/11123607/` (p. 41).

[239] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. In: USENIX Security. 2024 (p. 34).

[240] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In: USENIX Security. 2022 (p. 34).

[241] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J Nair. Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems. In: HPCA. 2023 (p. 38).

[242] XEN. XEN's MSR handling. 2021. URL: `https://github.com/xen-project/xen/blob/RELEASE-4.15.0/xen/arch/x86/msr.c` (p. 32).

[243] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: USENIX Security. 2016 (p. 35).

[244] Wenjie Xiong and Jakub Szefer. Survey of Transient Execution Attacks. In: arXiv:2005.13435 (2020) (p. 33).

[245] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. In: Symposium on Internetware. 2015 (p. 40).

[246] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips. In: USENIX Security. 2020 (p. 35).

[247] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. In: Cryptology ePrint Archive, Report 2014/140 (2014) (p. 3).

[248] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 3, 23).

[249] Tse-Yu Yeh and Yale N Patt. Alternative implementations of two-level adaptive branch prediction. In: ACM SIGARCH Computer Architecture News 20 (1992), pp. 124–134 (p. 21).

[250] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In: USENIX Security. 2024 (pp. 6, 10, 11).

[251] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. Red alert for power leakage: Exploiting intel rapl-induced side channels. In: AsiaCCS. 2021 (p. 41).

[252] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. In: arXiv:1912.03076 (2019) (p. 35).

[253] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In: MICRO. 2020 (p. 35).

[254] Mark Zhao and G Edward Suh. FPGA-based Remote Power Side-Channel Attacks. In: S&P. 2018 (p. 42).

[255] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. Cache Timing Attacks on Camellia Block Cipher. In: Cryptology ePrint Archive, Report 2009/354 (2009) (p. 3).

Part II.

Publications

List of Publications

During my PhD, I contributed to 15 peer-reviewed publications, of which 11 were accepted at tier 1 conferences. Out of these papers, 4 tier 1 papers are included as main contributions in this thesis as shown below.

Publications in this Thesis

[1] **Andreas Kogler**, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In: USENIX Security. 2023.

[2] **Andreas Kogler**, Daniel Gruss, and Michael Schwarz. Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In: USENIX Security. 2022.

[3] **Andreas Kogler**, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In: USENIX Security. 2022.

[4] **Andreas Kogler**, Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz. Finding and Exploiting CPU Features using MSR Templating. In: S&P. 2022.

Other Contributions

[1] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, **Andreas Kogler**, and Daniel Gruss. Remote Scheduler Contention Attacks. In: FC. 2024.

[2] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, **Andreas Kogler**, Daniel De Almeida Braga, and Daniel Gruss. Generic and Automated Drive-by GPU Cache Attacks from the Browser. In: AsiaCCS. 2024.

[3] Fabian Rauscher, **Andreas Kogler**, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024.

[4] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, **Andreas Kogler**, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In: USENIX Security. 2024.

[5] 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.

[6] Jonas Juffinger, Lukas Lamster, **Andreas Kogler**, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI: Rowhammer - Cryptographic Security and Integrity against Rowhammer. In: S&P. 2023.

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

[8] Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, **Andreas Kogler**, Daniel Gruss, and Moinuddin Qureshi. PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks. In: DSN. 2023.

[9] 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. 2022.

[10] Lukas Giner, **Andreas Kogler**, Claudio Canella, Michael Schwarz, and Daniel Gruss. Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX. In: USENIX Security. 2022.

[11] Martin Schwarzl, Pietro Borrello, **Andreas Kogler**, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. Dynamic Process Isolation. In: ESORICS. 2021.

5 Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks

Publication Data

Andreas Kogler, Daniel Gruss, and Michael Schwarz. Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In: USENIX Security. 2022

Contributions

Main author.

Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks

Andreas Kogler[1] Daniel Gruss[1] Michael Schwarz[2]

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

Abstract

Modern CPUs adapt clock frequencies and voltage levels to workloads to reduce energy consumption and heat dissipation. This mechanism, dynamic voltage and frequency scaling (DVFS), is controlled from privileged software but affects all execution modes, including SGX. Prior work showed that manipulating voltage or frequency can fault instructions and thereby subvert SGX enclaves. Consequently, Intel disabled the overclocking mailbox (OCM) required for software undervolting, also preventing benign use for energy saving.

In this paper, we propose Minefield, the first software-level defense against DVFS attacks. The idea of Minefield is not to prevent DVFS faults but to deflect faults to trap instructions and handle them before they lead to harmful behavior. As groundwork for Minefield, we systematically analyze DVFS attacks and observe a timing gap of at least 57.8 µs between every OCM transition, leading to random faults over at least 57 000 cycles. Minefield places highly fault-susceptible trap instructions in the victim code during compilation. Like redundancy countermeasures, Minefield is scalable and enables enclave developers to choose a security parameter between 0 % and almost 100 %, yielding a fine-grained security-performance trade-off. Our evaluation shows a density of 0.75, *i.e.*, one trap after every 1-2 instruction, mitigates all known DVFS attacks in 99 % on Intel SGX, incurring an overhead of 148.4 % on protected enclaves. However, Minefield has no performance effect on the remaining system. Thus, Minefield is a better solution than hardware- or microcode-based patches disabling the OCM interface.

1. Introduction

With a variety of use cases for modern computers, CPUs have an increasing number of features, also for security and performance. One feature available on Intel, AMD and ARM CPUs are trusted execution environments (TEEs). TEEs enable running code in a secure environment isolated from the rest of the system with the security goal of protecting code and data even from a compromised operating system or hypervisor.

To accommodate today's performance and efficiency goals, modern CPUs operate at various clock frequencies and voltage levels to adapt to the current workload. When the workload is low or energy must be saved due to thermal or battery constraints, the voltage level and the clock frequency are lowered. This mechanism, DVFS, is available on ARM, Intel, and AMD CPUs and can be controlled from privileged software. However, the modified voltage and frequency affect all security domains on the CPU.

Previous work [12, 11, 2, 34, 20, 21] has shown that an attacker can manipulate voltage and frequency using DVFS to inject faults into victim computations. The typical target of these attacks are TEEs since DVFS requires root privileges [12, 11, 34, 20, 21] or physical access [2], both of which are allowed in TEE threat models. The attacker uses DVFS to bring one or more CPU cores into a state where faults can occur with a very low probability. Thus, the CPU mostly still allows regular operation, *i.e.*, it does not cause a system crash. In this state, certain operations are more likely to experience a fault. Previous work has identified several of these operations, e.g., multiplication operations, pointer arithmetics, and AES-NI instructions. However, most instructions have not yet been analyzed for their fault probability.

In response to the DVFS attacks on Intel CPUs, Intel issued a CVE (CVE-2019-11157) and modified the SGX remote attestation process to verify that overclocking mailbox (OCM) and its model-specific registers (MSRs) allowing software-undervolting are disabled via a microcode update. The voltage regulators responsible for the core's voltage are connected to a bus receiving commands from CPU components, e.g., the OCM. In contrast, VoltPillager [2] directly sends these commands over the bus bypassing the CPU and the OCM. Therefore, disabling the OCM still leaves Intel CPUs without a fully-integrated voltage regulator (FIVR) design [42] vulnerable to VoltPillager [2] style attacks. Disabling the OCM led to complaints [16, 9] as the OCM is used to reduce overheating problems, thus increasing system performance and stability by undervolting the CPU. Some online guides even explicitly recommend reverting BIOS updates to get back the undervolting feature, breaking the ability to run SGX enclaves securely on these machines [9]. Disabling undervolting only if SGX is enabled impacts the entire system performance and stability if enclaves are used and is thus also not a desirable tradeoff. Selectively disabling undervolting while an enclave is active requires complex microcode changes, as the CPU has to ensure stable voltage levels on any reentry of the enclave. While it is unclear whether this is even possible with a microcode update, we also expect a high impact on the performance of enclaves.

In this paper, we propose the first software-level defense that probabilistically protects secure enclaves against all known DVFS attacks. As an empirical foundation for our defense, we systematically analyze DVFS attacks and categorize them based on the type of fault and its properties (e.g., spatial granularity, temporal granularity, and reproducibility). As part of our empirical analysis, we develop a framework to scan the x86 instruction set for DVFS fault susceptibility. Our analysis of the instructions with the highest fault probability confirms the implicit assumption from previous works that multiplications are most susceptible to faults. Hence, we rely on this instruction for our fault-deflection mechanism. We also analyze the temporal constraints and observe a timing gap of at least 57.8 µs between transitions from one voltage level to another. With the resulting weak control over the temporal fault location, state-of-the-art attacks have to repeat the victim operation millions of times [12, 20, 21, 11]. Currently, an attacker cannot precisely predict when and where during these operations the fault occurs.

Our defense, Minefield is a pure software-level defense implemented as a compiler extension. The basic idea of Minefield is not to prevent DVFS faults but to deflect them into trap instructions that are placed in the victim code during compilation, so that they cannot be weaponized anymore. The number of trap instructions scales as a security parameter from 0 % to almost amount for 100 % of the code base, yielding a fine-grained security-performance trade-off. Note that a security parameter of 100 % would refer to a program that consists only of trap instructions and no other instructions. Our evaluation shows that a trap density of 0.5, *i.e.*, one trap after every second instruction, mitigates the known DVFS attacks on Intel CPUs, namely Plundervolt [12], V0ltpwn [11], Voltjockey [21], and VoltPillager [2]. More specifically, in an attack on *mbedTLS* RSA-4096, a trap density of 2 mitigates more than 99 % of all attack attempts. Thus, Minefield is a viable defense against DVFS attacks on Intel SGX enclaves.

We carefully evaluate the performance impact on SGX enclaves with different Minefield security levels. Both runtime and memory overhead for the enclave scale up with the chosen security level. For a trap density of 0.75, which mitigates the known DVFS attacks in more than 99 % of the cases, Minefield incurs an overhead of 148.4 % on protected SGX enclaves on average. However, in some configurations Minefield even outperforms RSA redundancy protections, and the performance of normal-world applications remains entirely unaffected. Thus, Minefield is a better-suited mitigation against DVFS attacks on SGX enclaves than hardware- or microcode-based patches that disable the OCM entirely and also considers hardware-based undervolting attacks like VoltPillager [2].

While our evaluation focuses on Intel CPUs, we argue that the approach is applicable to ARM and AMD CPUs. Hence, Minefield can also be extended to prevent DVFS attacks on ARM TrustZone [34] and AMD SEV.

Contributions. The contributions of this work are:

1. We present a novel framework to systematically analyze the effects of DVFS faults on the entire x86 instruction set.
2. We propose a compiler extension Minefield, the first software-level defense against all known DVFS attacks.
3. We analyze different security levels and show that known DVFS attacks can be mitigated in 99 % of cases.
4. We evaluate the performance overheads of Minefield and show that the runtime overhead for SGX enclaves is below 150 % while mitigating 99 % of attacks.

Outline. Section 2 provides background, and Section 3 our threat model. Section 4 presents the high-level overview and poses research questions for Section 5. Section 6 details our implementation. Section 7 evaluates the security and performance. Section 8 discusses limitations. Section 9 concludes.

2. Background

In this section, we provide background on Intel SGX and DVFS, the mechanism behind the attacks we mitigate.

2.1. Intel SGX

Intel Software Guard Extension (SGX) [19, 36] is a trusted execution environment (TEE). To protect code and data on an untrusted system, an application is split into an untrusted and a trusted part, which is executed within a so-called SGX enclave. The enclave's execution state and memory cannot be accessed from other processes or the operating system. In the SGX threat model, only the CPU is trusted. Enclave memory is encrypted and integrity-protected in DRAM in a dedicated region called Enclave Page Cache (EPC), mitigating certain software-level and physical attacks. Thus, SGX even protects enclaves on systems compromised in software or hardware.

While these protections apply for the enclave's execution state (e.g., register values) and memory contents, scheduling, and page-table management are still performed by the untrusted operating system. Memory-safety violations [30], race conditions [39], or side channels [26, 32] can still lead to exploitation. Controlled-channel attacks [41], abuse, e.g., page-table entries or the APIC timer interrupt to precisely control the execution flow of a victim application [41, 27, 13]. Transient-execution attacks, e.g., Foreshadow [24], ZombieLoad [22], can precisely leak information from enclaves.

2.2. Power Management (DVFS)

Modern CPUs in smartphones, laptops, and servers, have different energy requirements. Especially mobile devices require constant energy balancing. Operating systems try to maximize the battery runtime while still providing sufficient computing power to handle the user's tasks. For dynamic adaption to the user's needs, modern CPUs implement Dynamic Voltage and Frequency Scaling (DVFS). DVFS allows changing the voltage and frequency from privileged software via model-specific registers (MSR) [19]. However, the overclocking mailbox (OCM) interface allows to change the alignment between voltage and frequency, e.g., reduce the operating voltage at a specific frequency.

Undervolting and overclocking have become important to personal computer owners, especially for gaming computers (overclocking) and laptops (undervolting). While system stability has always been a concern in these communities, only recently researchers discovered that these interfaces can be abused for attacks. The first DVFS-based fault attack [34] overclocked an ARM CPU, leading to fault injection in the TrustZone trusted execution environment. More recently, several works have explored undervolting as a means to inject faults into the Intel SGX trusted execution environment [21, 12, 2, 11]. These works have in common that they modify the operating voltage during the execution of critical instructions leading to a computational error propagating into the result of these instructions. These faulty results lead to incorrect behavior inside a (correct and bug-free) program. These results then lead to exposure of secret data from enclaves, e.g., by faulting index calculations of array accesses. Faulty results can also occur within cryptographic primitives, e.g., enabling differential cryptoanalysis on AES-NI.

The main difference between previous works is the way the operating voltage is changed. VoltJockey [21], V0ltpwn [11], and Plundervolt [12] assume the SGX threat model and use privileged access to the OCM. These attacks can be mounted purely from software and only require access to the OCM MSR. VoltPillager [2], on the other hand, uses additional hardware to send messages directly to the voltage regulator unit on the mainboard. Hence, currently, there is no software mitigation against it, leaving SGX enclaves unprotected.

3. Attacker Model

In this section, we provide the attacker model for Minefield. We base our attacker model on the previously published attacks [34, 20, 21, 12, 2, 11] and our own experiments.

Attacker Privileges. Our mitigation, Minefield, works under the widely adopted SGX threat model and assumes a privileged attacker who controls the operating system and the BIOS. As for the hardware, the attacker has direct physical access to the CPU and the mainboard, enabling the attacker to mount DVFS attacks [34, 12, 2, 20, 21, 11]. This includes attacks like VoltPillager [2] intercepting and issuing bus commands to the onboard voltage regulators circumventing the OCM. We assume that the faulting behavior of VoltPillager does not differ from the software issued undervolt as both approaches influence the core voltage (see Section 8).

The enclave does not require the OCM to be disabled by a given local attestation. Hence, if the enclave is built using Minefield, the attestation does not have to verify whether the microcode disabling the undervolting functionality is active. The attacker does not exploit bugs inside the enclave's code, nor the software surrounding the enclave initialization, nor side-channel attacks to extract secret information from the enclave. As our defense focuses on fault attacks, we consider side-channel attacks [13] (e.g., cache attacks on SGX) an orthogonal problem. However, we discuss the implications of the mitigation on side-channel robustness in Section 8.

Fault-Injection Capabilities. The attacker can attack the enclave execution with DVFS attacks and induce faults inside the results of machine instructions. We assume that the attacker controls the environment with the same precision as in known DVFS attacks [34, 12, 2, 20, 21, 11]. Importantly, no previous DVFS attacks was able to:

1. precisely target an arbitrary bit inside an instruction result (but mounted attacks that work with random bit flips),
2. precisely control how many bits flip (but report various faults from a single bit to multi-byte flips [12]),
3. precisely control the timing (undervolting windows are multiple microseconds),
4. precisely control which instruction is faulted (*i.e.*, many instructions are at risk of fault due to the length of the undervolting window). Certain instructions are found to be more susceptible to DVFS-based fault injection.

We assume that the attacker has the capabilities from these previous works since there is currently no indication that the OCM enables even stronger and more precise attacks.

No single-stepping. In particular, no known attack can combine single-stepping, e.g., using controlled-channel attacks to target a specific instruction, with a DVFS-based fault attack. Given the significant amounts of code executed during context switches in controlled-channel attacks, there is reasonable doubt that such an attack can be mounted reliably. Furthermore, controlled-channel attacks can also be mitigated using T-SGX [33],

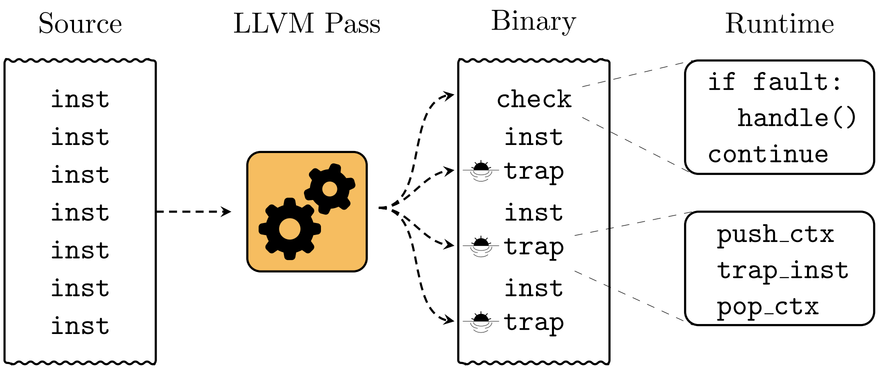


Figure 5.1.: An overview of Minefield. The compiler part of Minefield interleaves the instruction stream with trap instructions and code to detect faults in these instructions. A library is linked to the enclave handling detected faults at runtime.

entirely preventing single-stepping of SGX enclaves, or other interrupt-monitoring mechanisms [29, 15]. Hence, we assume that the attacker cannot target a single instruction this way. We discuss possible mitigations against a stronger attacker with single-stepping in Section 6.2.1.

4. High-Level Overview of Minefield

In this section, we provide a high-level overview of Minefield and the research questions we have to answer before designing it. The main goal is allowing the operating system to still control the undervolting of the CPU while ensuring that enclaves cannot be exploited. Minefield relies on an LLVM compiler extension to automatically place trap instructions in the code. Minefield has a security parameter to fine-tune the application-specific security-performance trade-off based on the required security guarantees. This makes Minefield an easily adaptable mitigation without changes in the protected software and with individual security levels per application.

Figure 5.1 shows an overview of Minefield and its components. Minefield consists of an LLVM compiler extension (Section 6.1) used to compile SGX enclaves, as well as a runtime library (Section 6.2) to check for faults and handle them. The compiler extension compiles unmodified source code and adds additional trap instructions to the binary. The trap instruction is an instruction highly susceptible to DVFS faults. Hence, the result of the trap instruction is used to detect faults that the enclave can then handle. Designing Minefield requires answering three research questions as follows.

4.1. Research Questions

While our approach may appear intuitive, a thorough analysis of DVFS-based fault attacks is necessary to ensure that Minefield is not built upon potentially wrong assumptions. Furthermore, even if the assumptions as outlined in the threat model hold, there remains a set of unanswered questions on the precise attacker capabilities. In the following, we ask three research questions we need to answer.

RQ1: Is there an **instruction** highly susceptible to faults, and if so, how can we find this instruction?

Although all the published attacks [11, 12, 34] show that instructions can be faulted based on concrete instances of instructions, there is no comprehensive analysis on which instructions can be faulted. V0ltpwn [11], Plundervolt [12], and VoltJockey [21] indicate that multiplications are highly susceptible to faults on all evaluated systems. However, without a comprehensive analysis, this remains an assumption that must be further analyzed, as we do in Section 5.1.

RQ2: What is the temporal and spatial **precision** of DVFS-based fault attacks?

Previous work induced single faults by repeating the target application until the fault hit the correct instruction [12, 11, 20, 21]. However, the precision for inducing faults is unknown. In Section 5.2, we analyze the capabilities of an attacker using DVFS to inject faults. We show that faults cannot be injected with arbitrary precision. Moreover, we show that there is a minimum time window between two undervolts.

RQ3: How can an enclave react when **detecting** a fault?

Detecting a fault is not sufficient. An enclave has to react to the fault as well. Without replay protection in SGX [31], attacks could be repeated at a high frequency, even if a fault is detected. Thus, it is insufficient to simply terminate the enclave, especially when an attacker can arbitrarily retry inducing a fault. We discuss possible solutions in Section 5.3.

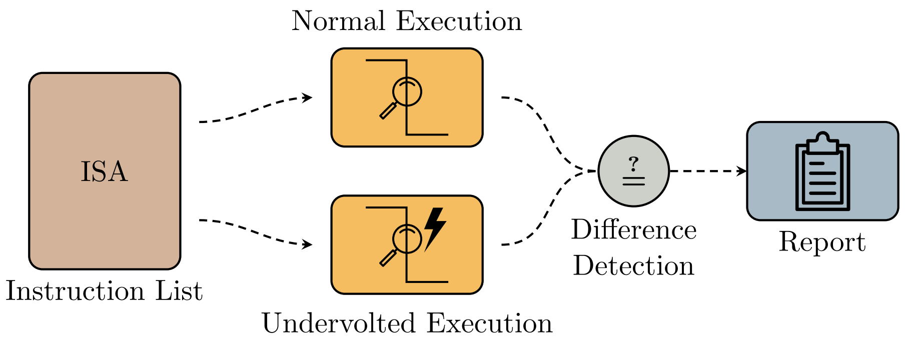


Figure 5.2.: The framework to find trap instructions. Based on a machine-readable list of instructions, the framework executes all unprivileged instructions with random arguments, once normally and once undervolted. If the output differs, the instruction is reported as being susceptible to faults.

Based on our analysis in Section 5, we present the design and implementation of Minefield in Section 6.

5. Analysis of Research Questions

In this section, we analyze the capabilities of software-based fault-injection attacks to answer the research questions asked in Section 4.1. We exhaustively test the fault susceptibility of x86 instructions (RQ1) using an automated framework in Section 5.1. Moreover, we analyze the capabilities of an attacker to inject faults (RQ2), *i.e.*, the type of fault, as well as the spatial and temporal precision, in Section 5.2. Finally, we discuss the handling of detected faults (RQ3) in Section 5.3.

5.1. RQ1: Instruction Susceptibility to Faults

To determine the fault characteristics, we analyze instructions of the x86 instruction set during critical undervolting conditions and monitor the faults injected into the results. The goal is to find a suitable trap instruction with the highest fault susceptibility that is used by Minefield to detect faults. Therefore, we implement an analysis framework to determine instructions that are usable as trap instructions and to get a more in-depth insight into how undervolting affects instructions.

Design. Our framework is designed to exhaustively test all unprivileged x86 instructions for multiple levels of undervolting. The design of our framework is illustrated in Figure 5.2. The basic idea is to test an instruction multiple times. In each test, the instruction is run once in a stable environment and once in an undervolted environment. For both runs, the same randomly-generated inputs are chosen. The framework records the output values for both runs and compares them. If the output differs, the undervolting led to a fault, and the instruction, its parameters, and the undervolting level are reported. As outlined, this test is performed multiple times for each instruction, namely with different input parameter values and undervolting levels and CPU frequencies.

Our test framework stores the bit difference of the expected result and the faulted result, *i.e.*, the bit location where the fault occurred. Furthermore, the framework analyzes the type of the fault, *i.e.*, whether it is a stuck-at-zero or stuck-at-one fault, or a bit flip. To measure the precision of the fault injection, we also record the temporal and spatial distance between two faulted operations. Consequently, our framework can determine the lowest observed temporal and spatial distance between two faults. This can later be used as a basis to determine appropriate security levels.

Implementation. To test the instructions defined in the x86 ISA, we use the list from Abel et al. [14], which contains all x86 instructions, including all ISA extensions, as well as the input, output, and side effects of the instruction. The framework automatically generates assembly code to parametrize these instructions. The generated assembly code is placed inside a loop to repeat the instruction multiple times. It is then compiled into a dynamic library for the test environment to load and evaluate. The framework allocates buffers for the instructions' state, e.g., registers and flags, and runs the instruction loop. The instruction in the loop then fills the buffers with all the changed state produced by the 1 000 000 iterations. Instructions that change the program's control flow are handled by setting the jump destination to an instruction after the jump that sets a flag to indicate that the jump was either taken or not.

The framework uses the same undervolting mechanism as Plundervolt [12], V0ltpwn [11], and VoltJockey [21], namely the OCM MSR 0x150. This MSR allows reducing the operating voltage for a short duration by modifying the voltage offset. When the undervolted execution is completed, the nominal voltage is restored, and the results are analyzed for bit errors. In this step, the framework compares the results of the undervolted instruction with the normal execution of the instruction. Each loop iteration is independent of the previous, so a fault inside one iteration is only visible in the iteration's outputs and does not influence other iterations.

To exhaustively test and analyze the instructions, we split the analysis into three distinct phases. First, we search for *faultable* instructions across all the tested CPUs at a fixed frequency and vary the undervolting offset until we see repeated system freezes or unrecoverable machine check errors. The framework monitors the response time and restarts over a remote power switch to recover from a system freeze. We store each reported faulted instruction into a global set of faultable instructions. Second, we use the set of faultable instructions to characterize the faulting behavior further. We execute each faultable instruction on each physical core of each CPU, with both varying frequency and undervolting offset. This analysis shows the minimum undervolt needed to observe a fault over the tested frequencies for each core. To evaluate the effect of other instructions on the faulting instruction, we tested each faultable instruction with all the other faultable instructions and evaluate if the faulting behavior is influenced.

Results. We analyzed 5 Intel CPUs with different microarchitectures, each running the same image with Ubuntu 21.04 with Kernel version 5.11. We list the exact CPUs in Table 5.2 (Section 10.2). For each CPU, we analyze each physical core, resulting in a total of 26 analyzed cores. Our experiments did not observe different faulting behavior for the sibling threads, but we observed differences between the physical cores. In the instruction finding phase, we executed 1258 instructions and instruction variants, *i.e.*, same instruction but with different mnemonics, from the base, SSE, SSE2, FMA, AVX, AVX2, and AES instruction set, fixed the frequency to 3000 MHz. This analysis revealed 71 faultable instructions variants with 12 unique instructions. We analyzed the first faulting point for each of these unique instructions by varying the frequency from 2000 MHz to 4000 MHz (if available) in 500 MHz steps. Table 5.2 (Section 10.2) shows the faulting point results.

From our experiment, we found that `imul` has the highest fault probability. Table 5.2 shows the analysis of `imul` in combination with different instructions. `imul` does not only fault well in isolation, but this behavior is also observable when combined with other instructions. The `imul` instruction faults in 92.1 % of all cases when other instructions also fault. For 1.5 % of the faulty instructions we need an additional `aesenc` instruction to detect the faults. On one CPU, we did not observe any faults with AES and

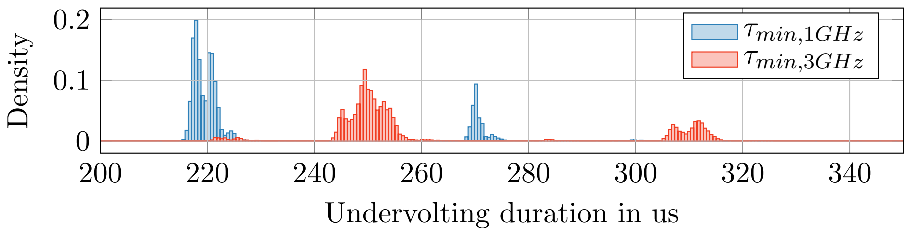


Figure 5.3.: The minimal undervolting time window for two distinct CPU frequencies.

hence used a `vorpd` instruction to detect the remaining 6.4 % faultable instructions.

Moreover, the `imul` instruction already suffers from faults at *smaller* undervolting offsets. This is in accordance with recent works [12, 11] that focus on `imul` as well. Hence, `imul` is ideal as a trap instruction to monitor if the CPU is driven near the specification limits. In Section 7.1, we also show that this property holds when using `imul` in full programs.

5.2. RQ2: Fault-Injection Capabilities

The security level of Minefield is related to the fault-injection capabilities of the attacker. Prior work [12, 11, 20, 21] did not comprehensively analyze properties, e.g., the precision, of the faults but simply measured the probability of being able to fault the target instruction at any point when running it in a long loop. However, to provide strong security guarantees, we analyze DVFS-based fault injection in more detail, *i.e.*, the fault model [47]. The fault model includes types of errors, temporal and spatial precision, and the number of faults that can be injected in one execution [49]. Our fault model is based on prior work [12, 11, 20, 21] and our own experiments, and considers spatial and temporal *precision* and the *type* of faults.

Temporal and Spatial Precision. Previous work [12, 11, 20, 21] did not analyze where faults can be injected. Typically, faults can occur at a random location [52], or can be induced for instruction sequences or surgically for single instructions. So far, no paper has shown that it is possible to induce DVFS faults with such surgical precision. Our experiments also indicate that targeting a single instruction with DVFS faults is impractical. Figure 5.3 shows a histogram for the experimentally measured minimum undervolting time at two different CPU frequencies. The average undervolting duration at 1 GHz is around 220 µs, which are 220 000 cycles. The shortest undervolting duration we observe is 57.8 µs, *i.e.*, 57 800 cycles. Hence, to target a single instruction, an attacker would have to target a code sequence where the victim instruction is the only instruction susceptible to a fault within this window.

The minimal length of the undervolting window also influences the timing of faults. First, the minimal duration of the undervolting limits the frequency in which an attacker can induce faults. The CPU requires time to change the voltage. This is true both for reducing as well as increasing the voltage. As shown in Figure 5.3, the durations are not constant but subject to variations in the microsecond range, depending on the CPU frequency and also the CPU itself. We do not have an explanation for this effect. However, as a consequence, undervolting precisely the same instruction sequence multiple times is infeasible. When undervolting, there is always a non-negligible probability that several instructions before or after the targeted instruction are undervolted as well.

Fault Types. In addition to the precision, it is also important what types of faults can be injected. Typical fault models consider stuck-at-zero, stuck-at-one [45], random faults [50], or flips in one or more bits [43, 44]. There are more specialized fault models, e.g., bits with a bias [40].

We based our analysis of the fault types on our experiments and the results in prior work [12, 21, 11]. Table 5.1 (Section 10.1) shows the detailed fault characteristics of each observed fault of our previous analysis. We confirm that a fault can influence one bit to multiple bytes. Further analysis revealed that we observe stuck-at-zero faults for instructions executing bitwise logical operations, *i.e.*, *VAND*, *VXOR*, and *VOR*. However, the faults of `imul` and further susceptible instructions behave randomly, *i.e.*, all observed bit positions can flip in both directions. Moreover, the affected bits differ between the physical CPU cores [11]. Hence, for the fault model, we assume that an attacker can flip between one and all bits of `imul`'s result to random values.

There is no difference if an ALU instruction is faulted or the address generation in a load or store instruction. In all cases, an attacker cannot choose the location of the bits, the number of bits, or the values of the bits.

5.3. RQ3: Handling Faults

While detecting a fault is a vital requirement for Minefield, it is not sufficient for protecting an enclave if the fault is not handled correctly. Hence, an important part of Minefield is the fault handler. We identified two different strategies for handling faults such that they cannot be exploited.

Cancel. The straightforward approach is to stop further execution as soon as a fault is detected. Aborting ensures that no further instructions are executed that potentially consume the faulted data. An abort handler does not require any change to the enclave code. To abort the enclave, the handler can either execute an illegal instruction, e.g., `ud2`, or simply stop execution by entering an endless loop. Note that Minefield does not suffer from false positives (cf. Section 7.1), *i.e.*, enclave execution is never wrongly aborted.

While abort handlers are straightforward, they may open new attack surface: An attacker could repeat the attack at a high frequency to increase the chance of bypassing our detection in one of the runs. Potentially, by knowing where the detection was triggered, the attacker might even improve the attack further. Without secure persistent storage and replay protection, the enclave developer must provide additional infrastructure to prevent the enclave from being started again. The SGX ecosystem already provides the EPID attestation method [38], allowing to identify a specific CPU, practically solving the replay protection problem if a remote trusted third party is available. We discuss different solutions in Section 8.

One possibility to reduce the frequency of restarts is to use monotonic counters [37]. These counters can only be read and incremented and are persistent across enclave restarts and also system restarts. Hence, by incrementing the counter on a fault, the enclave can track the number of total faults and decide not to start when a certain number of faults was detected. However, even with the counters, it is not possible to entirely prevent arbitrary execution of the enclave as the counters can be destroyed by re-installing the Intel PSW or by removing the BIOS battery [31]. Still, this at least slows down an attacker and might make an attack infeasible. We further discuss the availability of monotonic counters in Section 8.

Retry. A different approach is to try to “hide” the fault and prevent its weaponization by restoring the state before the fault and repeating the instruction. The retry handling is more complex, as instructions are not generally idempotent. Thus, the retry handler cannot simply re-execute the instruction before the fault or the current basic block. To use a retry handler, a developer has to define checkpoints in the enclave code to which a fault handler can safely jump back. Inside the retry handler, a developer can then choose to which checkpoint to return based on where the fault was detected. The implementation of such checkpoints could make use of the already existing `setjmp` and `longjmp` C functions.

The retry handler has the advantage that the enclave can continue execution in the presence of faults. Thus, this approach has similar advantages to multiple executions with a majority vote [47, 28], without the disadvantage of always executing code multiple times. The obvious disadvantage is that the developer has to take care of checkpoints at which execution can be retried. Moreover, the retry handler might provide an attacker with valuable information. As an attacker can monitor the execution time of the enclave, an attacker might learn that the fault was successfully injected. However, an attacker only learns that the fault definitely hit a trap instruction and not if the fault hit the target instruction. While this cannot be weaponized directly, it introduces a side channel (see Section 8).

5.4. Results

Based on the results from analyzing the research questions, we provide a solid fault model for software-based DVFS fault attacks. We show that there are indeed instructions that are more susceptible to faults than others. We confirm that `imul` instruction exploited in prior work [12] is indeed highly susceptible to faults, making it a perfect choice for Minefield's trap instruction. Furthermore, our analysis shows that an attacker cannot surgically induce faults. Both the temporal and spatial precision are limited by the minimal undervolting window of multiple microseconds. Hence, next to the instruction targeted by an attacker, there are always other instructions that are executed in the undervolted state as well. We can use this non-uniformity to place instructions with a higher chance to attract faults as trap instructions and enforce that the results of these trap instructions are not altered or faulted. Thus, these trap instructions enable us to protect the real instructions with a relatively simple mechanism against undervolting attacks. Depending on the number of inserted instructions, any induced fault is likely to also fault at least one of these inserted instructions.

6. Implementation of Minefield

In this section, we discuss the implementation details of Minefield. The implementation consists of two parts. The first part is a configurable LLVM compiler extension (Section 6.1) for adding additional trap instruction to enclave code at compile time. The second part is the runtime environment integrated into the enclave for detecting and handling induced faults (Section 6.2). Finally, Section 6.3 describes how the changes are integrated into the SGX toolchain.

6.1. Compiler Extension

The compiler extension implements a *Machine Function Pass* inside the LLVM 11 [51] backend. The *Machine Function Pass* allows inspecting each program's function on an x86 machine instruction level. Implementing Minefield in the backend ensures that it is language agnostic, as long as there is an LLVM frontend for the desired language. The compiler extension is responsible for placing trap instructions and generating the code that checks whether a fault was induced.

Trap Instructions. Based on our fault susceptibility analysis and the fault model (cf. Section 5), we select `imul` as a default trap instruction, as it has the highest probability to fault when undervolted on our tested systems. In addition to this default trap instruction, a developer can also provide a different trap instruction to the compiler extension, e.g., a pair for AES and multiplication instruction. To keep track of the current state of our `imul` instruction, we use two distinct instances of the trap that only differ in the register operand. By placing both of these instances in alternating order, we ensure that the values in the two registers are at most one execution of the trap instruction apart. This placement is independent of the chosen trap instruction. The compiler also ensures that the basic block contains an even number of traps by adding an additional trap if necessary. This ensures that the registers must always have the same content at the start of a basic block, eliminating the need to store additional information about the current correct value as Figure 5.4b shows.

Placing Trap Instructions. Generally, trap instructions are placed between existing instructions, as illustrated in Figure 5.4. However, several practical obstacles have to be handled by the compiler. The trap instruction

```
imul $11, input(%rip), %rax
cmp  %rax, limit(%rip)
ja   .L1
```

(a) unmodified

```
cmp   %r12, %r13
jne  __abort
imul __factor(%rip), %r12
imul $11, input(%rip), %rax
imul __factor(%rip), %r13
cmp  %rax, limit(%rip)
pushf
imul __factor(%rip), %r13
imul __factor(%rip), %r12
popf
ja    .L1
```

(b) modified

Figure 5.4.: Figure 5.4a shows the unprotected assembly instructions while Figure 5.4b shows the trap instruction sequence generated by Minefield with a placement density of 1.

modifies the content of a register. Hence, the compiler has to know that the register used in the trap instruction is clobbered. Moreover, both the trap instruction and the code for detecting fault might modify the CPU flags, *i.e.*, the `rflags` register.

In addition to the problems of inserting a trap itself, we also have to decide *when* to insert a trap instruction. Inserting more traps leads to better security guarantees, while it impacts the performance negatively. We provide tuning parameters to find a trade-off between the performance and the provided security by the mitigation. We denote this parameter as the *placement density*. This parameter defines the ratio of trap instructions to existing instructions, e.g., a density of 0 means that no trap instruction is placed, a density of 0.5 places a trap after every second instruction. If this parameter is chosen higher than one, we place multiple trap instructions after the original instruction. We also ensure that at least two trap instructions are placed inside a basic block such that the mitigation also works if the placement density is low.

As the compiler extension is implemented in the backend, inserting the trap instructions is straightforward. The compiler simply iterates over each of the instructions inside the basic blocks and can directly insert new instructions.

Handling Register Clobbering. If the placement density is greater or equal to 1, Minefield places a trap instruction basically after every

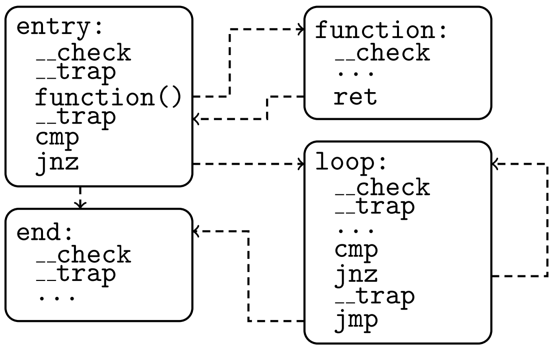


Figure 5.5.: Traps are inserted between normal instructions. To ensure the correctness of comparisons, no trap instruction is inserted directly between comparison and conditional jump but only at the two jump destinations. Trap instructions are checked at the beginning of each basic block.

instruction. As loading and storing a value from and to memory after every executed instruction incurs a high overhead, the trap instead operates solely on values stored registers. Minefield dedicates two general-purpose registers to fault checking to minimize the performance impact. For the general-purpose registers, we use `R12` and `R13`. These registers have no special use in the System V Application Binary Interface. Both registers are defined to be callee-saved. Thus, no other function can change the content of the registers. As a result, Minefield even supports calling functions that have not been compiled with the compiler extension, making it fully backward compatible with existing code. Reserving a general-purpose register inside the LLVM compiler infrastructure already excludes the register from the complete pipeline. Thus, no additional precautions are necessary.

In addition to the modification of the register, a trap can also modify the `rflags` register [18], thus changing the architectural state and, with that, potentially the semantics of the original program. To prevent saving and restoring the flags all the time, we rely on the liveness analysis of LLVM. The LLVM infrastructure records which registers are currently alive and in use and which instruction consumes a given register (variable liveness analysis). Therefore, we can monitor when the flags register is in use. Typically, the content of the flag register is only relevant between a comparison and a conditional operation, e.g., a conditional jump. Minefield can simply omit the placement of trap instructions between an instruction that modifies and an instruction that acts on the flags register. This approach increases the performance without significantly reducing the security guarantees, as faults are checked in any case at the beginning of a basic block. We evaluate the correctness of this approach in Section 7.3.

Without relying on the liveness, the `rflags` would have to be saved before and restored after executing the trap. However, saving and restoring the state is expensive, as it involves pushing the flags to the stack (`pushf`) and restoring it from the stack afterward (`popf`). While this ensures the correctness of the generated code, it adds a non-negligible performance overhead to the fault checks. For testing purposes, we provide an additional compiler option to fall back to this slower approach and not use the liveness analysis of LLVM.

Fault-detection Code. To detect faults in the trap instruction, the compiler extension creates code for the fault detection. Minefield supports two ways of checking whether a fault occurred in a trap instruction. This check can either be *immediate*, *i.e.*, the detection code is inserted after every trap instruction. Alternatively, the check can be *lazy*, *i.e.*, the check is only performed at the start of a basic block. Nevertheless, despite the used method, a check is always performed at the basic block's beginning by simply comparing `R12` and `R13`.

Both approaches have their advantages and disadvantages. *Immediate* checking results in a larger binary size and also a larger performance overhead. However, with *immediate* checking, the time between a fault and the detection of the fault is minimized. When using *lazy* checking, the trap instruction is verified at the beginning of each basic block. Hence, with lazy checking, the number of checks is reduced, increasing the performance but potentially increasing the time window in which a fault could be exploited.

Immediate checking seems intuitive. However, for instructions that only operate on registers and do not perform a memory access, *immediate* checking does not provide additional security since the faulted value is not visible outside of the registers of the CPU. With this observation, we can extend the *immediate* checking method to only check the trap instructions right before either a load or a store is executed. This extension ensures that neither the load's address nor the store's address or data was previously faulted.

Basic Blocks and Control Flow Changes. There are two main reasons we chose to verify the trap instructions at the beginning of basic blocks, as shown in Figure 5.5. First, as per definition, control flow changes can only target the beginning of a basic block and never target instructions inside the basic block. We can ensure that checks placed at the beginning of basic blocks are always executed, regardless of the control flow conditions [48, 46]. Second, a basic block has one entry point but can have multiple exit points. A basic block can be exited by either calling a different function, by returning, or by jumping to a different basic block. Therefore, checking all the possible exit paths requires more checks that impact the performance without providing any benefits. Nevertheless, we still have to perform checks before return instructions since in LLVM, call instructions can be placed inside basic blocks.

6.2. Runtime Fault Handling

The second part of Minefield is the runtime library, statically linked into the enclave, which handles the detected fault. By default, an abort-handler callback is called when a fault is detected. The implementation of the actual fault handler is the responsibility of the enclave developer. This allows maximum flexibility for the developer, as depending on the threat model, there are different reactions to a detected fault.

Minefield also provides two default fault handlers that can be used in many scenarios. These fault handlers can either retry or cancel the execution of the enclave when a fault is detected, as outlined in Section 5.3.

6.2.1. Monitoring Controlled-Channel Attacks

As already described in the attacker model (cf. Section 3), there is no combined controlled-channel DVFS fault attack and due to the significant amount of code, it is unlikely that such an attack could be implemented reliably. Controlled-channel attack frameworks, such as *sgx-step* [27] enable attackers to essentially single-step (and zero-step) an enclave. Thus, the attacker can step an enclave precisely to a single target instruction inside an enclave. If this technique could be combined with a DVFS attack, it could bypass our defense. There is a strong indication that such a combination cannot be mounted reliably, *i.e.*, the system freezes instead because of the substantial amount of micro-code executed during enclave entry. We also empirically validated this in our own experiments. When undervolting during enclave entry, the system easily freezes while the CPU restores the enclave state. We also emphasize that appropriate countermeasures against controlled-channel attacks (including single-stepping of SGX enclaves) already exist, e.g., T-SGX by Shih et al. [33].

Integrated mitigation. As T-SGX would incur additional overhead, we also propose a more integrated solution to prevent single-stepping-assisted DVFS fault attacks. Similar to previous work [29, 15], we can utilize the Save State Area (SSA) of the enclave to monitor any interruptions. When an enclave gets interrupted and the control flow is passed to the interrupt handler, the state of the enclave is stored inside the SSA of the enclave, including all the registers and additional enclave state. We can write a magic value to the position of the enclave `RIP` field inside the SSA and later check if this magic is still present or if it was overwritten by the CPU when exiting the enclave asynchronously. Frequent interruptions can be handled as described in previous works [29, 15].

6.3. Toolchain Integration

At the time of writing, Intel does not officially support LLVM to build SGX enclaves with their SGX SDK. Compiling the SDK with clang instead of gcc fails due to gcc-specific features used in the SDK. Hence, to use the SDK for evaluation with Minefield, we had to apply small changes to the SDK version 2.10 to make it compatible to LLVM.[1]

We only compile the trusted part of the SGX SDK with Minefield. This is sufficient, as only the trusted part is an attack target. The untrusted part is under the control of the attacker. Thus, there is no benefit in protecting this part. The protected part is responsible for the enclave initialization, the enclave entry calls, and the enclave exits. As a small part of the enclave entry and exit code is written in assembly, it needs to be manually patched, as the current prototype of Minefield does not support assembly files. In addition to the manual patching, we adopted the enclave entry function to set up the registers required for Minefield. There is no need for additional modifications inside the source code.

[1]Our changes to the SDK, the source of Minefield, and the test enclaves are provided at: `https://github.com/iaik/minefield`.

7. Evaluation

In this section, we evaluate the security (Section 7.1), performance (Section 7.2), and correctness (Section 7.3).

7.1. Security

We evaluate the security of Minefield by evaluating the probability to detect induced faults for two different applications, a victim highly susceptible to fault attacks as used by Murdock et al. [12], as well as a practical application based on *mbedTLS*. In both scenarios, we evaluate different undervolting levels and placement densities between 0 and 2 and show that Minefield can successfully protect these applications.

Setup. We use SGX enclaves built with Minefield. All experiments are conducted on an Intel Core i5-8265U running Ubuntu 20.04.1 LTS. We focus on two different enclaves, one enclave containing the Plundervolt multiplication proof-of-concept [12], and another enclave running *mbedTLS* [6]. As an abort handler, we use a function that reports faults without terminating the applications.

As a detection metric, we use the *recall* of Minefield. In our setup, the recall is calculated by dividing the number of experiments where the target was successfully faulted and the mitigation detected the fault ($\mathcal{F}\&\mathcal{D}$) by the number of the experiments where the target was faulted ($\mathcal{F}$). The *recall* is bounded between zero and one, where zero means that no fault was detected and one that all faults were detected. We use the *recall* instead of the *F-score* as a security metric since the *precision* of Minefield is consistently one. This is because the detection cannot observe *false negatives*. The check is entirely deterministic. Thus, it is not possible to detect a fault although there was no fault. Moreover, if we observe a fault inside a trap instruction, the system is already driven near the specification limits. Hence, even if the fault is only in a trap instruction, the execution of the enclave is no longer safe and should be terminated. As a consequence, there is no case where the trap instruction triggers without the system being compromised. For the two enclaves, we evaluate the *recall* for different voltage offsets and vary the placement density between 0 and 2 (cf. Section 6.1).

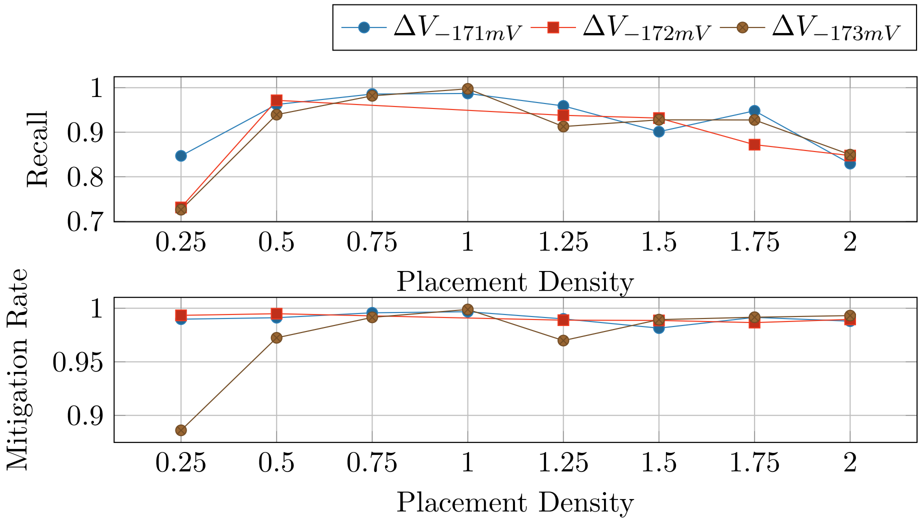


Figure 5.6.: The *recall* over the placement density for several undervolting offsets for the `imul` experiment.

7.1.1. Highly-susceptible Toy Victim

In the first scenario, we use a toy application inspired by the Plundervolt proof of concept [12]. In this application, we target four `imul` instructions executed 30 720 times inside a tight loop. The multiplications use the result of the previous iteration as input. Thus, any fault induced in a multiplication propagates to the final iteration's result. To detect a fault, it is sufficient to compare the final result to the ground truth. The fault propagation has the advantage that no additional fault-checking code has to be inserted. This toy application has a high probability that a fault can be induced. Moreover, every fault is effective, leading to a change in the final result. Hence, from a defender's perspective, this application is close to the worst case, as nearly every instruction has to be protected.

Figure 5.6 shows the recall when compiling this code with Minefield and inducing faults. We test different undervolting levels for which the rest of the system is still stable. We observe that the undervolting level itself does not significantly impact the probability of inducing a fault. As expected, the recall increases with the placement density of Minefield.

With a placement density of 0.5, we already recognize 80 % of the faults. If we further increase the placement density to 1, we can detect nearly all the faults for the different voltage offsets. We also observed that on one

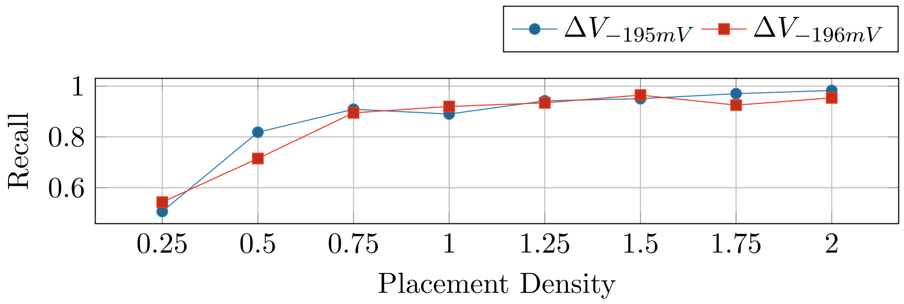


Figure 5.7.: The *recall* over the placement density for several undervolting offsets for the *mbedTLS* experiment.

of our cores, the recall started to decline when increasing the placement density above 1.25. However, this does not directly correlate with the security. In that case, the overall probability of inducing a fault decreased drastically. Hence, we were rarely able to induce a fault at all. To better visualize this effect, we show in Figure 5.6 the mitigation rate, *i.e.*, the inverse of the probability that an attacker faults the target instruction without the mitigation detecting it. For all placement densities above 0.75, the mitigation rate never dropped below 95 %.

This toy application shows that even if most instructions are susceptible to faults, and any fault is exploitable, Minefield can protect an application. Especially with a high placement density, there is a nearly arbitrarily adjustable trade-off between performance impact and security guarantees.

7.1.2. Real-world Victim

The second scenario uses a more realistic application. We protect *mbedTLS* [6] version 2.13 with Minefield. Due to its small codebase and simplicity, it can be easily used inside SGX enclaves [23]. As a constant target for side-channel attacks, *mbedTLS* also provides side-channel resilient implementations of the provided cryptographic algorithms [7].

For our evaluation, we focus on the RSA signature algorithm of *mbedTLS*. As shown in previous work [12], a fault during the signing can be sufficient to recover the private key. Hence, for the evaluation, we focus on the underlying *binary modulo exponentiation* function `mbedtls_mpi_exp_mod`, which is directly used inside the library's RSA algorithm. We choose the input parameters for the function to represent a 4096-bit key. After the execution of the algorithm, we check whether the result of the function was faulted and also determine whether Minefield detected the fault. For each placement density, we perform 2000 encryptions. The voltage is reduced for each of these 2000 encryptions before entering the function and restored after the return from the encryption.

Figure 5.7 shows the recall for the *mbedTLS* experiment. We observe a relatively high detection rate of 90 % with a low placement density of 0.75 across two voltage offsets of −195 mV, *i.e.*, the first offset where we observe faults and −196 mV, *i.e.*, the last offset where the system did not freeze. Compared to the toy application, the undervolting offset of the *mbedTLS* example is lower since the executed codebase is more extensive.

7.1.3. Results

For both our victims, Minefield reliably detects induced faults. Especially for higher placement densities, the probability of faulting a target instruction without triggering Minefield is very low. While the level of undervolting does not have a huge impact, we observe a trend that the mitigation performs slightly better if the CPU is driven more into critical conditions, *i.e.*, if we undervolt the CPU more.

For the analysis, we do not consider faults detected by Minefield that had no effect on the victim computations. In a real scenario, it is also desirable that these faults trigger the enclave's abort or retry mechanic, as a stable execution cannot be guaranteed. During our experiments, we observed on average 10 times more faults inside the trap instructions compared to the target `imul` instruction. This result also supports our choice for using `imul` as the default trap instruction.

7.2. Performance of Minefield

For the performance evaluation of Minefield, we evaluate three different metrics: the runtime overhead (Section 7.2.1), the increase in code size (Section 7.2.2), as well as the one-time compile-time overhead (Section 7.2.3).

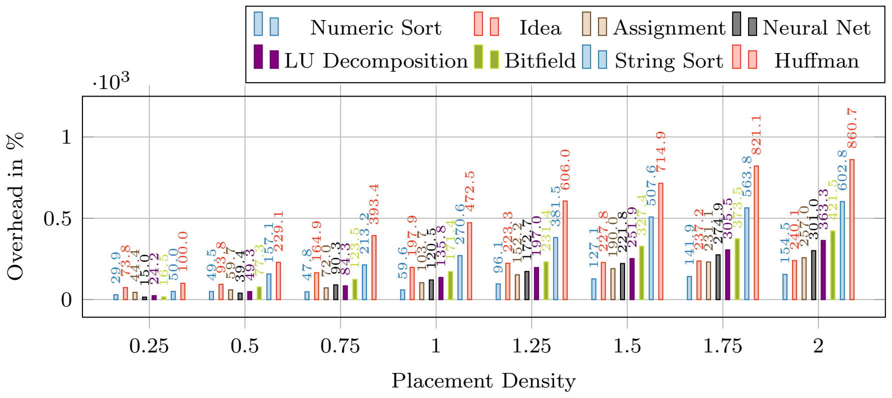


Figure 5.8.: The performance overhead of Minefield for the *sgx-nbench* benchmark over multiple placement densities. We observe a linear overhead with increasing placement density.

7.2.1. Runtime Evaluation

To evaluate the performance, we use the well-known *SGX nbench* benchmark suite for SGX. Additionally, we also evaluate the performance impact on *mbedTLS*, as we used this library for the security benchmark as well (see Section 7.1). For *mbedTLS*, we also compare the performance overhead of Minefield to the integrated fault-mitigation technique.

SGX nbench. *SGX nbench* [35] is an adoption of the traditional nbench benchmark suite for SGX enclaves. The benchmarks focus on classical benchmarks executed inside the enclave environment. We use this benchmark to evaluate the performance impact on actual performance code mitigated with Minefield including the SGX SDK. Each of the benchmarks is executed 25 times over a total duration of 2 h and 51 min. Figure 5.8 shows an average overhead for a placement density 1 of 191.51 %. The overhead linearly increases to 400.12 % for a placement density of 2. In all cases, the standard deviation was below 1 %.

mbedTLS. *mbedTLS* [6] already hardens the software implementation of its RSA algorithm against fault attacks. The `mbedtls_rsa_private` function used for encrypting data with the RSA key decrypts the complete ciphertext after encryption and compares if the decrypted message matches the provided function's input message. This is only possible since *mbedTLS* stores also the public key inside the private context.

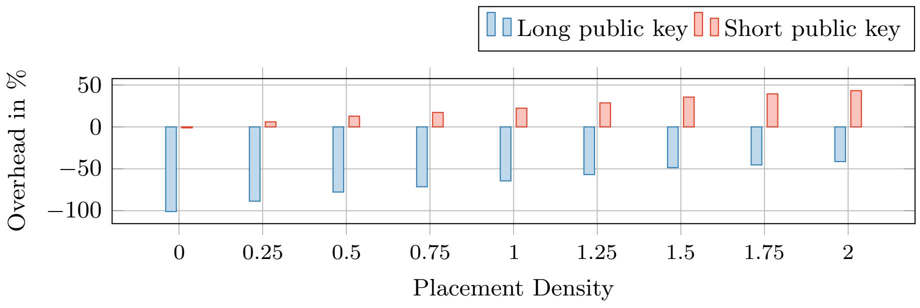


Figure 5.9.: The performance comparison between the *mbedTLS* RSA verification and Minefield-protected version.

With the built-in fault check, the RSA implementation takes on average 13.9 ms ($n = 1000$, $\sigma_{\bar{x}} = 0.064$) for one encryption with a 2048-bit key, where the public key is large, *i.e.*, it only has 6 leading zeros. When disabling the internal fault check, the same encryption takes on average 6.9 ms ($n = 1000$, $\sigma_{\bar{x}} = 0.045$). Hence, the runtime overhead of the internal check of *mbedTLS* is 100.99 %. When using the same parameters with a small public key, *i.e.*, the key has 2031 leading zeros, the overhead decreases to 1.13 %. For comparison with Minefield, we compile the version without the internal check with Minefield. Figure 5.9 shows the performance comparison over different placement densities. For large public keys (6 leading zeros), Minefield always performs better, regardless of the placement density. With a placement density of 0.75, we increase the performance by 71.42 % for full-length public keys compared to the internal verification of *mbedTLS*. We show in Section 7.1 that with a placement density of 0.75 we already achieve a recall of 90 %. For small public keys (2031 leading zeros) and at the same placement density, we only decrease the performance by 17.23 % compared to the internal verification of *mbedTLS*.

7.2.2. Code-Size Evaluation

As Minefield inserts additional code into an application, we compare the size of binaries created with Minefield and with the same compiler without any placed trap instructions. For evaluating the code size, we use the benchmarks used to evaluate the runtime overhead in Section 7.2.1. The code size is especially relevant for SGX, as the amount of physical memory usable by SGX is limited for all enclaves running on the system. We further discuss the memory impact in Section 8.

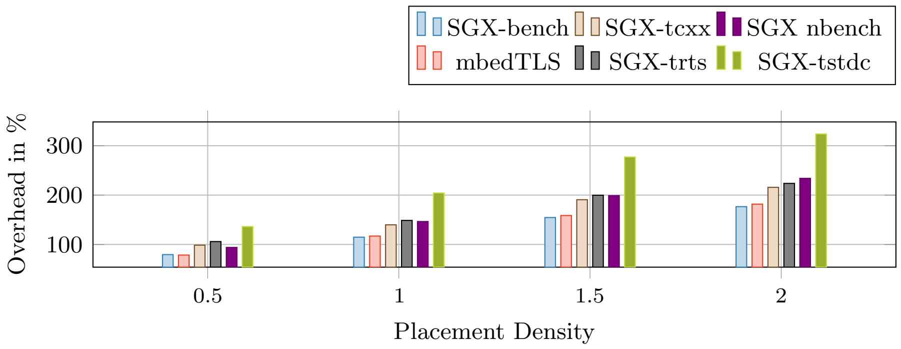


Figure 5.10.: The increase of the code size for the various benchmarks over the placement density parameter of Minefield.

The code size is increased by the constant size of the runtime library linked to the enclave code (cf. Section 6.2). Additionally, there is a variable increase based on the number of instructions in the enclave and the placement density. Figure 5.10 shows the code size increasing over the placement density parameter for the *SGX nbench*, *mbedTLS* and *SGX-bench* benchmarks. In addition, we also show the increase in code size for trusted SGX SDK components such as the runtime system, the C library, and the C++ library.

As expected, we observe a nearly linear increase of the code size when using Minefield. However, even for large applications such as *SGX nbench*, protected with a placement density of 1, the absolute increase is only 274.5 kB.

7.2.3. Compile-Time Evaluation

We analyze the impact on the compile time that Minefield has on enclaves. As a baseline, we compile the benchmarks without any mitigations enabled. We compare the compile time for different placement densities to this baseline.

Figure 5.11 shows the average increase in compile time for the benchmarks. The placement density does not have a significant impact on the compile time. For a placement density of 0.5, the overhead is negligible, with on average around 0.78 %. Even for a placement density of 2, the overhead is only around 1.6 %, which amounts to less than 0.5 s for *mbedTLS*.

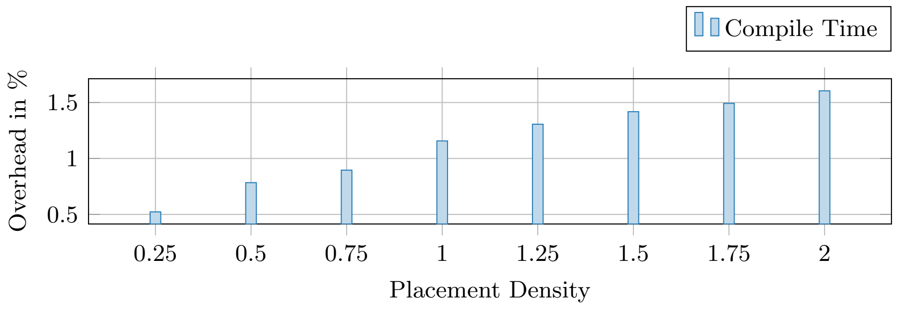


Figure 5.11.: The compile-time increase for the various benchmarks over the placement density parameter of Minefield.

We conclude that the overhead on the compilation time is negligible, especially as this is a one-time overhead for the developer. This small overhead also makes it feasible to use Minefield in the development process, and not only for the final compilation of an SGX enclave.

7.3. Correctness of Minefield

In addition to the performance and security evaluation, we also verify that our Minefield prototype does not introduce any correctness problems.

Compiler Correctness. We explicitly tested the compiler by running a C compiler test suite [8] to ensure that we did not introduce any bugs. The test suite confirmed that the compiler changes did not have any adverse effect on the correctness. We also confirmed that *mbedTLS* works correctly with Minefield by running it without undervolting both in SGX and as a native application. For SGX, the SGX-nbench benchmark verifies the correctness of the computed results in addition to the performance. We did not encounter any errors.

Integration Correctness. In addition to the correctness of the compiler itself, we also evaluated the correctness of our integration with the SGX SDK. We relied on SGX-bench [25] to test the enclave interactions. SGX-bench is a small test suite for SGX enclaves to determine the performance of, e.g., enclave entries, enclave initialization, and enclave ocalls. We did not run into any bugs or crashes when running the test cases, showing that Minefield successfully works with the SGX SDK. Moreover, running our test enclaves with *mbedTLS* without undervolting also showed that the integration works.

8. Discussion and Limitations

Current Mitigations. The currently active countermeasure against undervolting attacks on SGX enclaves prohibits the user from applying voltage offsets to the CPU. This removes a feature to gain additional performance or increase thermal thresholds against throttling. In addition, software-based fault mitigations either handwritten in cryptographic software or per compiler extension often focus on calculating results multiple times to check the correctness of the results against each other. While these mitigations provide a high level of security against fault attacks, they induce additional software complexity and performance overhead based on the type of instructions protected.

Hardware Undervolting. Starting from the 11th generation, Intel CPUs reuse fully integrated voltage regulator (FIVR) designs, previously abandoned after the 4th generation [42]. CPUs without a FIVR design expose the voltage regulators, allowing an attacker within our threat model (cf. Section 3) to mount VoltPillager [2] attacks. The voltage regulators are connected to a bus that receives commands from the CPU. VoltPillager directly sends these commands over the bus bypassing the CPU and the OCM. Although we performed the instruction analysis via the OCM, we argue that the observed fault behavior is independent of how the undervolt is issued. Therefore, we assume that Minefield is also applicable against hardware-based undervolting like VoltPillager, where the current mitigation to disable the OCM is ineffective.

Persistent Failing. Li et al. [5] show that AMD SEV's "security-by-crash" is exploitable, similarly using Minefield without hindering an attacker from arbitrarily often restarting an enclave might result in an undetected fault. Intel SGX does not support any local replay-protected persistent state that is also protected against an attacker with physical access. Hence, an enclave cannot securely store any data that could be used to detect how often the enclave has already been started, without using a trusted remote server. Thus, an attacker can always restart an enclave arbitrarily often. Even when relying on the monotonic counters [37] for counting restarts, an attacker can, e.g., remove the BIOS battery to destroy the counter, effectively resetting it [31]. The support for monotonic counters was discontinued in the Linux SGX-SDK [3]. However, the latest available documentation of the Windows SGX-SDK [10] (March 2020) still lists these functions.

If a trusted remote server is available, we can either implement the replay protection with Intel's EPID scheme or other rollback preventions. Intel's EPID group signature remote attestation scheme [38] can verify that enclaves are part of a certain CPU group. Moreover, EPID supports the *named-base* mode that allows linking two signatures, *i.e.*, the verifier can determine if two signatures originate from the same signer. Therefore, when using the named-base mode, the verifier can deny the data exchange with enclaves that repeatedly restart, observe faults, or do not terminate. Matetic et al. [31] present a rollback prevention for persistent state based on a distributed system. Hence, by relying on such a technique, Minefield could also implement persistent failing without requiring any hardware change. This would restrict an attacker to only a developer-defined number of induced faults per physical CPU.

Performance and Memory Overhead. The performance and memory overhead of Minefield is adjustable by the placement density (cf. Section 6.1) and affects only occasionally running SGX workloads, allowing the remaining system to benefit from undervolting and the resulting performance and energy gains. Unfortunately, adjusting the placement density also affects the security guarantees. We propose the following extensions for future work to reduce Minefield's overhead without affecting security.

First, Table 5.2 shows a margin between `imul` faults and faults of different instructions. Minefield can utilize this margin by protecting regular `imul` instructions with additional redundancy or replacing them with functional equivalents. Due to this margin, the trap `imul` instructions observes substantially more faults at these lower voltages than the other susceptible instructions. Thus, increasing the detection capabilities retaining the same security guarantees with lower placement densities, improving performance.

Second, we can reduce the impact on branch prediction by replacing the check's `cmp` and `jne` instructions with instructions generating a GP-fault if Minefield detects a fault. We propose using `xor` to calculate the registers' difference followed by `popcount` giving the number of bit errors. Adding this number to the higher 16 bit of a 64 bit address makes the address non-canonical if a fault was detected. When accessing a non-canonical address, the CPU raises a GP-fault causing an asynchronous enclave exit [19]. The enclave can only be resumed at the internal signal handler, stopping further faulty code execution [37].

Finally, the SGX driver ensures that enclaves that exceed the available EPC memory (usually 128 MB) can execute without limitations by swapping EPC pages [36]. Accessing a non-present EPC page introduces a latency of 13 103 cycles ($n = 1\,000\,000$, $\sigma_{\bar{x}} = 0.925$) to swap it back into the EPC on our Intel i5-8265U. We analyzed Intel and Synaptics production enclaves and found that their enclave sizes are below 3 MB. Furthermore, Intel's Ice Lake CPUs increase the available EPC memory up to 1 TB [4]. Therefore, we find Minefield's memory overhead for these enclaves tolerable.

Side Channels. Minefield does not protect against classical side-channel attacks on enclaves. Side channels are orthogonal to fault attacks and are thus out of scope for Minefield. Intel sees it as the developers responsibility to ensure that their code is free of side channels [17]. Importantly, Minefield does not introduce any new or additional side channels, as we decouple the instructions responsible for fault detection from the actual data processed by the enclave. However, if the enclave is already susceptible to side-channel attacks, Minefield might amplify the side-channel leakage. In the worst case, this can enable the exploitation of side channels that were previously considered not exploitable. For example, the inserted trap instructions might change a secret-dependent control flow within a cache line to a secret-dependent control flow on a cache line or even cache set granularity. Hence, for complete side-channel protection, developers have to ensure that all algorithms handling secrets are data oblivious [17].

Other Architectures. The idea of Minefield is not restricted to any given architecture and has two requirements. First, the architecture needs an instruction that is more susceptible to undervolting faults than others. Second, all targets that can be faulted must be compilable with Minefield. If the architecture meets these requirements, Minefield can probabilistically protect code running on the system. The performance depends on all the susceptible instructions of that architecture.

Minefield is also applicable to other TEE alternatives such as ARM TrustZone and AMD SEV. Qiu et al. [20] target ARM TrustZone with software undervolting faults and exploit faults in AES and RSA computations. Minefield could protect the target AES and RSA code if we port the compiler extension and the runtime library. Due to AMD's x86 instruction set, Minefield is directly applicable to AMD SEV workloads. As of writing this paper, there are no known software undervolting attacks against AMD. However, Buhren et al. [1] exploit AMD SEV by inducing hardware undervolting faults to compromise the secure coprocessor responsible for transparent encryption. In this case, the attack compromises AMD

SEV by exploiting code not protected by Minefield, breaking the second requirement and rendering the defense ineffective.

9. Conclusion

In this paper, we presented Minefield, the first software-level defense against DVFS attacks. We systematically analyze DVFS attacks and observe a timing gap of at least 57.8 µs between every OCM transition, leading to random faults over a sequence of at least 57 thousand cycles. The trap instructions Minefield places in the victim code during compilation are highly susceptible to faults. Our evaluation showed that a density of 0.75 traps per instruction, *i.e.*, 1-2 traps after every second instruction reliably mitigates the currently known DVFS attacks on Intel CPUs, namely Plundervolt, V0ltpwn, VoltJockey, and VoltPillager. Minefield allows fine-grained selection of the performance-security tradeoff. For this strong security level, we observe overheads of 94.4 % on average on protected SGX enclaves. The performance of the remainder of the system is entirely unaffected. Thus, we conclude that Minefield is an important alternative to a solution in hardware or microcode that comes with the prohibitive effect of disabling the OCM entirely.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, Dave Tian, for their guidance, comments and suggestions. Additional funding was provided by a generous gift from Amazon. Any opinions, findings, 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] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021, pp. 2875–2889 (p. 108).

[2] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In: 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 699–716. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai` (pp. 77–79, 81, 82, 106).

[3] Intel. Unable to find Alternatives to Monotonic Counter Application Programming Interfaces (APIs) in Intel Software Guard Extensions (Intel SGX) for Linux to Prevent Sealing Rollback Attacks. 2021. URL: `https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html` (p. 106).

[4] Intel. What Technology Change Enables 1 Terabyte (TB) Enclave Page Cache (EPC) size in 3rd Generation Intel Xeon Scalable Processor Platforms? 2021. URL: `https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html` (p. 108).

[5] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CROSSLINE: Breaking" Security-by-Crash" based Memory Isolation in AMD SEV. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021, pp. 2937–2950 (p. 106).

[6] ARM. mbed TLS. 2020. URL: `https:///tls.mbed.org` (pp. 98, 100, 102).

[7] ARM. Security Advisories - Tech Updates - Mbed TLS. 2020. URL: `https://tls.mbed.org/tech-updates/security-advisories` (p. 100).

[8] Andrew Chambers. c-testsuite. 2020. URL: `https://github.com/c-testsuite/c-testsuite` (p. 105).

[9] Douglas Black. Intel & OEMs are disabling undervolting. Here's how to re-enable it. 2020. URL: `https://www.ultrabookreview.com/37095-dells-disabling-undervolting-on-their-laptops-heres-how-to-re-enable-it/` (pp. 77, 78).

[10] Intel. Intel SGX SDK Developer Reference for Windows*. 2020. URL: https://software.intel.com/content/www/us/en/develop/download/sgx-sdk-developer-reference-windows.html (p. 106).

[11] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In: 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1445–1461. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar (pp. 77, 78, 81, 82, 84, 86, 88, 89).

[12] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. IEEE, 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057. URL: https://doi.org/10.1109/SP40000.2020.00057 (pp. 77, 78, 81, 82, 84, 86, 88, 89, 91, 98–100).

[13] Michael Schwarz and Daniel Gruss. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. In: IEEE Secur. Priv. 18.5 (2020), pp. 18–27. DOI: 10.1109/MSEC.2020.2993896. URL: https://doi.org/10.1109/MSEC.2020.2993896 (pp. 80, 82).

[14] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019. Ed. by Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck. ACM, 2019, pp. 673–686. DOI: 10.1145/3297858.3304062. URL: https://doi.org/10.1145/3297858.3304062 (p. 86).

[15] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating Speculative-Execution Attacks on SGX with HyperRace. In: 2019 IEEE Conference on Dependable and Secure Computing, DSC 2019, Hangzhou, China, November 18-20, 2019. IEEE, 2019, pp. 1–8. DOI: 10.1109/DSC47296.2019.8937682. URL: https://doi.org/10.1109/DSC47296.2019.8937682 (pp. 83, 97).

[16] Hacker News. Plundervolt: Software-Based Fault Injection Attacks Against Intel SGX. 2019. URL: https://news.ycombinator.com/item?id=21759683 (p. 77).

[17] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. 2019. URL: https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations (p. 108).

[18] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 2019 (p. 94).

[19] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 80, 107).

[20] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 195–209. DOI: 10.1145/3319535.3354201. URL: https://doi.org/10.1145/3319535.3354201 (pp. 77, 78, 81, 82, 84, 88, 108).

[21] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In: Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2019, Xi'an, China, December 16-17, 2019. IEEE, 2019, pp. 1–6. DOI: 10.1109/AsianHOST47458.2019.9006701. URL: https://doi.org/10.1109/AsianHOST47458.2019.9006701 (pp. 77, 78, 81, 82, 84, 86, 88, 89).

[22] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 753–768. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252 (p. 80).

[23] Fan Zhang. mbedtls-SGX: a TLS stack in SGX. 2019. URL: `https://github.com/bl4ck5un/mbedtls-SGX` (p. 100).

[24] 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: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 991–1008. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/bulck` (p. 80).

[25] Raul Quinonez. SGXBENCH framework for benchmarking SGX enclaves. 2018. URL: `https://github.com/sgxbench/sgxbench` (p. 105).

[26] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: 11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017. Ed. by William Enck and Collin Mulliner. USENIX Association, 2017. URL: `https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser` (p. 80).

[27] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017. ACM, 2017, 4:1–4:6. DOI: `10.1145/3152701.3152706`. URL: `https://doi.org/10.1145/3152701.3152706` (pp. 80, 96).

[28] Zhi Chen, Junjie Shen, Alex Nicolau, Alexander V. Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. CAMFAS: A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017. IEEE Computer Society, 2017, pp. 57–64. DOI: `10.1109/FDTC.2017.10`. URL: `https://doi.org/10.1109/FDTC.2017.10` (p. 91).

[29] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In:

26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 217–233. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss` (pp. 83, 97).

[30] Jae-Hyuk Lee, Jin Soo 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: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 523–539. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk` (p. 80).

[31] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 1289–1306. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic` (pp. 84, 90, 106, 107).

[32] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings. Ed. by Michalis Polychronakis and Michael Meier. Vol. 10327. Lecture Notes in Computer Science. Springer, 2017, pp. 3–24. DOI: `10.1007/978-3-319-60876-1\_1`. URL: `https://doi.org/10.1007/978-3-319-60876-1%5C_1` (p. 80).

[33] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017. The Internet Society, 2017. URL: `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx`

-eradicating-controlled-channel-attacks-against-enclave-programs/ (pp. 82, 97).

[34] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 1057–1074. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang (pp. 77, 79, 81, 82, 84).

[35] utds3lab. Adaptation of nbench-byte-2.2.3 for Intel SGX. 2017. URL: https://github.com/utds3lab/sgx-nbench (p. 102).

[36] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: IACR Cryptol. ePrint Arch. (2016), p. 86. URL: http://eprint.iacr.org/2016/086 (pp. 80, 108).

[37] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference. Rev 1.5. May 2016 (pp. 90, 106, 107).

[38] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: EPID provisioning and attestation services. 2016 (pp. 90, 107).

[39] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In: Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I. Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Vol. 9878. Lecture Notes in Computer Science. Springer, 2016, pp. 440–457. DOI: 10.1007/978-3-319-45744-4_22. URL: https://doi.org/10.1007/978-3-319-45744-4%5C_22 (p. 80).

[40] Sikhar Patranabis, Abhishek Chakraborty, Phuong Ha Nguyen, and Debdeep Mukhopadhyay. A Biased Fault Attack on the Time Redundancy Countermeasure for AES. In: Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers. Ed. by Stefan Mangard and Axel Y. Poschmann. Vol. 9064. Lecture Notes in Computer Science. Springer, 2015, pp. 189–203. DOI:

10.1007/978-3-319-21476-4_13. URL: https://doi.org/10.1007/978-3-319-21476-4%5C_13 (p. 89).

[41] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. IEEE Computer Society, 2015, pp. 640–656. DOI: 10.1109/SP.2015.45. URL: https://doi.org/10.1109/SP.2015.45 (p. 80).

[42] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. FIVR—Fully integrated voltage regulators on 4th generation Intel® Core™ SoCs. In: 2014 IEEE Applied Power Electronics Conference and Exposition-APEC 2014. IEEE. 2014, pp. 432–439 (pp. 77, 106).

[43] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa M. I. Taha, and Patrick Schaumont. Differential Fault Intensity Analysis. In: 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014. Ed. by Assia Tria and Dooho Choi. IEEE Computer Society, 2014, pp. 49–58. DOI: 10.1109/FDTC.2014.15. URL: https://doi.org/10.1109/FDTC.2014.15 (p. 89).

[44] Yoongu Kim, Ross Daly, Jeremie S. 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: ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014. IEEE Computer Society, 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210. URL: https://doi.org/10.1109/ISCA.2014.6853210 (p. 89).

[45] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013. Ed. by Wieland Fischer and Jörn-Marc Schmidt. IEEE Computer Society, 2013, pp. 108–118. DOI: 10.1109/FDTC.2013.18. URL: https://doi.org/10.1109/FDTC.2013.18 (p. 89).

[46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In: 2013 IEEE Symposium on Security

and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. IEEE Computer Society, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13. URL: https://doi.org/10.1109/SP.2013.13 (p. 96).

[47] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010. ACM, 2010, p. 7. DOI: 10.1145/1873548.1873555. URL: https://doi.org/10.1145/1873548.1873555 (pp. 88, 91).

[48] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005. Ed. by Vijay Atluri, Catherine A. Meadows, and Ari Juels. ACM, 2005, pp. 340–353. DOI: 10.1145/1102120.1102165. URL: https://doi.org/10.1145/1102120.1102165 (p. 96).

[49] Christophe Giraud and Hugues Thiebeauld. A Survey on Fault Attacks. In: Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS), 22-27 August 2004, Toulouse, France. Ed. by Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam. Vol. 153. IFIP. Kluwer/Springer, 2004, pp. 159–176. DOI: 10.1007/1-4020-8147-2_11. URL: https://doi.org/10.1007/1-4020-8147-2%5C_11 (p. 88).

[50] Ludger Hemme. A Differential Fault Attack Against Early Rounds of (Triple-)DES. In: Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 254–267. DOI: 10.1007/978-3-540-28632-5_19. URL: https://doi.org/10.1007/978-3-540-28632-5%5C_19 (p. 89).

[51] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA.

IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665 (p. 92).

[52] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In: Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51. DOI: 10.1007/3-540-69053-0_4. URL: https://doi.org/10.1007/3-540-69053-0%5C_4 (p. 88).

10. Appendix

10.1. Faulting Bits

Table 5.1 shows the analysis of which bit flips we observed for the instructions. We recorded each bitflip, analyzed the direction to which the bit flipped, and reported the overall flip tendency of the faulty bits. We observe that logical vector instructions have a high probability for stuck-at-zero bitflips. Furthermore, we also found that some vector operations introduce interesting faulting behavior, e.g., the vector comparison instruction shows bitflip within a given element.

10.2. Instruction Analysis

Table 5.2 shows the detailed first faulting points for all the faulted instructions we found. We tested the faulted instruction across all our available CPUs with SGX support. We compiled the faultable instruction with the trap instruction to verify that the generated faults are indeed detectable with a given trap instruction. We executed it with the reported undervolting offset.

Table 5.1.: The distinct faulted bit positions for each of our tested CPUs in mask form and the corresponding probability for each bit to fault towards one. Note that all the outputs of the instructions are present in the mask, *i.e.*, `imul` also contains the `rflags` register. We see that parallel vector operations show only stuck-at-zero flips, where other instructions like multiplications show a more random distribution.

	Core i9-9900K		Core i7-8700k		Xeon E3-1505M		Core i7-6700K (1)	
Instruction	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$	Mask	$P_{x\to1}$
IMUL	`0x000000000000000000000783fff000000`	46.8 %	`0x0000000000000000000000001fff800000`	48.3 %	`0x0000000000000000ff000fffc000000`	48.7 %	`0x0000000000000000000000007f8000000`	40.8 %
AESENC	`0x000000100000000000010000004188e04`	50.7 %	`0x000000000000000000000000004198e04`	49.4 %				
VANDN*	`0x000000a0022a0080000000a208a80000`	0.0 %			`0x10000050405455400000000000105c00`	0.0 %		
VAND*	`0x000000a002020080000000a000000000`	0.0 %			`0x00001010001455400000000500144400`	0.0 %		
VOR*	`0x000200a0022200820000002a02ac0020`	0.0 %			`0x00000010001414420000000000140e00`	0.0 %		
VPCLMULQDQ	`0x00000000000000000000000200000000`	0.0 %	`0x3fd7ff7ffffffffffffffffffffffbff`	51.1 %				
VPCMP*	`0x00a2aa82002000000000000000000000`	0.0 %	`0xffffffff00000000ffffffffffffffff`	33.2 %				
VPSRAD	`0x0aa2aaa0000200000000000000000000`	0.0 %						
VSQRTPD	`0x000000001f4ffff0000000000c3ffff`	52.6 %	`0x000c000000001dff000000000001fff`	47.0 %				
VXOR*	`0x000000a0222a0080000000a20a800000`	0.0 %			`0x10000050405455420000001500145e10`	0.0 %		
VPMAX*			`0xffffffffffffffffffffffffffffffff`	50.2 %				
VPADDQ			`0x00000101010001000000000000000000`	73.3 %				

Table 5.2.: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		`IMUL`	`AESENC`	`VANDN*`	`VAND*`	`VOR*`	`VXOR*`	`VPCLMULQDQ`	`VPSRAD`	`VPMAX*`	`VPCMP*`	`VSQRTPD`	`VPADDQ`
Core i9-9900K	2000	0	⬠ 244											
		2	⬠ 255	⬠ 255			⬠ 255							
		4					⬠ 255							
		5	⬠ 250											
		6	⬠ 250											
	2500	0	⬠ 187	⬠ 202	⬠ 201		⬠ 197	⬠ 197						
		1	⬠ 195	⬠ 197										
		2	⬠ 196	⬠ 197	⬠ 210									

Table 5.2.: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		`IMUL`	`AESENC`	`VANDN*`	`VAND*`	`VOR*`	`VXOR*`	`VPCLMULQDQ`	`VPSRAD`	`VPMAX*`	`VPCMP*`	`VSQRTPD`	`VPADDQ`
		5	⬠ 197											
		6	⬠ 197				⬠ 210							
		7	⬠ 201											
	3000	0	⬠ 154	⬠ 165	⬠ 165	⬠ 170	⬠ 165	⬠ 163		⬠ 172				
		1	⬠ 160	⬠ 159	⬠ 172	⬠ 200	⬠ 165	⬠ 166					⬠ 167	
		2	⬠ 160	⬠ 165	⬠ 180	⬠ 174	⬠ 169	⬠ 172		⬠ 185				
		3	⬠ 170		⬠ 172	⬠ 177	⬠ 172	⬠ 171	⬠ 180					
		4	⬠ 163	⬠ 173	⬠ 177	⬠ 202	⬠ 171	⬠ 175					⬠ 179	
		5	⬠ 164	⬠ 165	⬠ 175		⬠ 199	⬠ 176						
		6	⬠ 158	⬠ 176	⬠ 210		⬠ 177							
		7	⬠ 165	⬠ 172	⬠ 182		⬠ 209							
	3500	0	⬠ 156	⬠ 160	⬠ 167	⬠ 166	⬠ 162	⬠ 159		⬠ 167		⬠ 175	⬠ 166	
		1	⬠ 161	⬠ 152	⬠ 166	⬠ 166	☆ 156	⬠ 167					⬠ 161	
		2	⬠ 159	☆ 156	⬠ 200	⬠ 178	⬠ 165	⬠ 174		⬠ 180		⬠ 180	⬠ 172	
		3	⬠ 162	⬠ 169	⬠ 172		⬠ 170	⬠ 173	⬠ 167	⬠ 174			⬠ 175	
		4	⬠ 155	⬠ 165	⬠ 177	⬠ 180	⬠ 172	⬠ 204				⬠ 185	⬠ 167	
		5	⬠ 162	⬠ 165	⬠ 180	⬠ 207	⬠ 172	⬠ 204					⬠ 174	
		6	⬠ 154	⬠ 164			⬠ 177	⬠ 215	⬠ 175	⬠ 180			⬠ 176	
		7	⬠ 161	⬠ 166			⬠ 180			⬠ 176			⬠ 179	
	4000	0	⬠ 161	⬠ 161	⬠ 180	⬠ 170	⬠ 170	⬠ 168		⬠ 175			⬠ 166	
		1	⬠ 167	☆ 155		⬠ 200	⬠ 169	⬠ 195					☆ 155	
		2	⬠ 163	☆ 163			⬠ 175		⬠ 195				⬠ 162	
		3	⬠ 166	⬠ 170					⬠ 190				⬠ 166	

Table 5.2.: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of $-1\,\text{mV}$. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		`IMUL`	`AESENC`	`VANDN*`	`VAND*`	`VOR*`	`VXOR*`	`VPCLMULQDQ`	`VPSRAD`	`VPMAX*`	`VPCMP*`	`VSQRTPD`	`VPADDQ`
		4	⬠ 160	⬠ 163									⬠ 161	
		5	⬠ 175	⬠ 166									⬠ 169	
		6	⬠ 162	⬠ 166					⬠ 165				⬠ 168	
		7	⬠ 164	⬠ 194						⬠ 175				
Core i7-8700k	1500	0	⬠ 264	⬠ 264										
		1	⬠ 262											
	2000	0	⬠ 230	⬠ 230		⬠ 245	⬠ 240	⬠ 241	⬠ 242					
		1	⬠ 227				⬠ 242		⬠ 250					
		2	⬠ 242											
		5	⬠ 250											
	2500	0	⬠ 204	⬠ 207		⬠ 235	⬠ 230	⬠ 237	⬠ 214		⬠ 230	⬠ 225	⬠ 223	⬠ 227
		1	⬠ 200		⬠ 232		⬠ 229	⬠ 234	⬠ 215					
		2	⬠ 214			⬠ 241	⬠ 240							
		5	⬠ 221											
	3000	0	⬠ 194	⬠ 189	⬠ 227	⬠ 230	⬠ 222	⬠ 223	⬠ 195					
		1	⬠ 193	⬠ 222	⬠ 220	⬠ 220	⬠ 215		⬠ 195					
		2	⬠ 214	⬠ 220		⬠ 235	⬠ 224	⬠ 230						
		3	⬠ 223	⬠ 225			⬠ 235		⬠ 224					
		5	⬠ 214											
	3500	0	⬠ 209	⬠ 187	⬠ 230		⬠ 225	⬠ 228	⬠ 208				⬠ 229	
		1	⬠ 190	⬠ 221			⬠ 222	⬠ 223	⬠ 202				⬠ 230	
		2	⬠ 223	⬠ 220									⬠ 240	

Table 5.2.: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		`IMUL`	`AESENC`	`VANDN*`	`VAND*`	`VOR*`	`VXOR*`	`VPCLMULQDQ`	`VPSRAD`	`VPMAX*`	`VPCMP*`	`VSQRTPD`	`VPADDQ`
		3	⬠ 245										⬠ 245	
		5	⬠ 244											
	3700	0	⬠ 202	⬠ 192			⬠ 225	⬠ 229				⬠ 215	⬠ 226	
		1	⬠ 185	⬠ 220			⬠ 218		⬠ 198		⬠ 235		⬠ 220	
		2	⬠ 217	⬠ 230										
		3	⬠ 214								⬠ 240			
		5	⬠ 244											
Xeon E3-1505M	800	0	⬠ 277											
		1				⬠ 282								
	1500	0	⬠ 202		⬠ 222	⬠ 221	⬠ 219	⬠ 220						
		1	⬠ 212		⬠ 210	⬠ 213	⬠ 210	□ 211						
		2	⬠ 215		□ 217	□ 217	⬠ 209	□ 209						
		3	⬠ 208		□ 212	⬠ 215	□ 209	□ 209						
	2000	0	⬠ 190		⬠ 209	⬠ 211	⬠ 208	⬠ 207						
		1	⬠ 200		⬠ 200	⬠ 200	□ 195	⬠ 195						
		2	⬠ 205		⬠ 208	⬠ 207	□ 203	⬠ 205						
		3	⬠ 197		⬠ 200	⬠ 201	□ 195	□ 195						
	2500	0	⬠ 160				⬠ 175	⬠ 175						
		1					□ 159	□ 160						
		2	⬠ 165					⬠ 170						
		3	⬠ 160		⬠ 169		□ 150	□ 160						

Table 5.2.: The first faulting points for each of the susceptible instructions. We tested each instruction on each physical core across multiple CPUs and frequency operating points. The numbers represent the set undervolting offset in units of −1 mV. The symbols indicates with which type of trap can detect the fault (⬠ `imul`, ☆ `aesenc`, □ `vorpd`).

CPU	Frequency	Core	Instructions											
	MHz		`IMUL`	`AESENC`	`VANDN*`	`VAND*`	`VOR*`	`VXOR*`	`VPCLMULQDQ`	`VPSRAD`	`VPMAX*`	`VPCMP*`	`VSQRTPD`	`VPADDQ`
	3000	0	⬠ 172											
		1					□ 154	□ 155						
		2	⬠ 165											
		3				⬠ 173	□ 160	⬠ 159						
	3290	3					□ 158							
	3300	0	⬠ 175											
		1					□ 148	⬠ 154						
		2	⬠ 160											
		3	⬠ 170		⬠ 165		□ 156	⬠ 160						
Core i7-6700K (1)	2000	2	⬠ 249											
		3	⬠ 241											
	2500	3	⬠ 242											
Core i7-6700K (2)	No Faults Found													

6 Half-Double: Hammering From the Next Row Over

Publication Data

Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In: USENIX Security. 2022

Contributions

Main author.

Half-Double: Hammering From the Next Row Over

Andreas Kogler[1] Jonas Juffinger[1,2] Salman Qazi[3]
Yoongu Kim[3] Moritz Lipp[41] Nicolas Boichat[3]
Eric Shiu[5] Mattias Nissler[3] Daniel Gruss[1]

[1]Graz University of Technology
[2]Lamarr Security Research [3]Google
[4]Amazon Web Services [5]Rivos

Abstract

Rowhammer is a vulnerability in modern DRAM where repeated accesses to one row (the aggressor) give off electrical disturbance whose cumulative effect flips the bits in an adjacent row (the victim). Consequently, Rowhammer defenses presuppose the adjacency of aggressor-victim pairs, including those in LPDDR4 and DDR4, most notably TRR.

In this paper, we present Half-Double[2], an escalation of Rowhammer to rows beyond immediate neighbors. Using Half-Double, we induce errors in a victim by combining many accesses to a distance-2 row with just a few to a distance-1 row. Our experiments show that the cumulative effect of these leads to a sufficient electrical disturbance in the victim row, inducing bit flips. We demonstrate the practical relevance of Half-Double in a proof-of-concept attack on a fully up-to-date system. We use side channels, a new technique called *Blind-Hammering*, a new spraying technique, and a Spectre attack in our end-to-end Half-Double Attack. On recent Chromebooks with ECC- and TRR-protected LPDDR4x memory, the attack takes less than 45 minutes on average.

1. Introduction

Rowhammer is a widespread DRAM issue caused by the unintended coupling between its constituent rows [30]. By repeatedly accessing one row (*i.e.*, *aggressor*), an attacker can corrupt data in adjacent rows (*i.e.*,

[2]Named after a crochet stitch taller than a single but shorter than a double.

victims) by accelerating their charge leakage. As a powerful means of bypassing hardware and software memory protection, Rowhammer has been used as the basis for many different attacks (Section 2.3).

Previously, Rowhammer was understood to operate at a distance of one row: an aggressor would only flip bits in its two immediate neighbors, one on each side. This makes intuitive sense: as a coupling phenomenon[54], the Rowhammer effect should be the strongest at closest proximity. Indeed, this assumption underpins many countermeasures (Section 2.3) that have been proposed against Rowhammer, especially the ones that rely on detecting aggressors and refreshing the charge in their intended victims (e.g., [30, 34, 39]). In fact, *Target Row Refresh (TRR)*, a productionized countermeasure widely deployed as part of LPDDR4/DDR4 chips, falls into this *detect-and-refresh* category [14].

In this paper, we present *Half-Double*, a new escalation of Rowhammer where we show its effect to extend beyond just the immediate neighbors. Using Half-Double, we are able to flip bits in the victim by combining many accesses to a *far aggressor* (at distance two) with just a few to a *near aggressor* (at distance one). Both aggressors are necessary: accessing just the former does not flip bits in a row that's two away, whereas accessing just the latter devolves into a classic attack that's easily mitigated. Based on our experiments, the near aggressor appears to act as a bridge, *transporting* the Rowhammer effect of the far aggressor onto the victim. Concerningly, TRR facilitates Half-Double through its mitigative refreshes, turning their recipient row into the near aggressor that co-conspires with the far one that necessitated the refresh in the first place. In effect, the cure becomes the disease.

While the discovery and evaluation of Half-Double is the main contribution of this work, we also demonstrate its practical relevance in a proof-of-concept exploit. However, current systems limit the attacker's control, introducing 4 challenges: First (**C1**), the adversary needs to allocate memory contiguous in a DRAM bank. However, without access to physical addresses [48] and huge pages [18, 45], we have to introduce a novel approach combining *buddy allocator* information with a DRAM timing side channel to reliably detect contiguous memory. Second (**C2**), ECC-protected memory can make bit flips unobservable depending on the victim data which the attacker does not control, the adversary cannot template the memory like in previous Rowhammer attacks as hammering requires knowledge of the cell data. As state-of-the-art [18, 51, 8, 44, 14, 45] does not solve this problem, we introduce a novel technique called *Blind-Hammering* to induce bit flips despite the ECC mechanism of LPDDR4x. Third (**C3**), reduced address space sizes on recent ARM-based systems break the page table spraying mechanism from previous attacks [48, 43, 51, 18]. Therefore, we develop a new spraying technique that is still unmitigated. Finally (**C4**), without templating, we need an oracle telling whether Rowhammer induced an exploitable bit flip, without crashing the exploit. For this, we introduce a novel approach using a Spectre-based oracle for exploitable bit flips. We combine these techniques into an end-to-end proof-of-concept, the Half-Double Attack[3], which escalates an unprivileged attacker to arbitrary system memory read and write access, *i.e.*, kernel privileges. The Half-Double Attack runs within 45 minutes on a fully updated Chromebook with TRR-protected LPDDR4x memory.

To summarize, we make the following contributions:

1. We discover a new Rowhammer effect: Half-Double, and evaluate a set of devices and modules for susceptibility.
2. We perform a thorough root-cause analysis to empirically prove that TRR is responsible for the Half-Double effect.
3. We analyze the stop-gap mitigations present in today's systems and show that with a new exploit using Half-Double, we can bypass them and build an end-to-end attack.
4. Our end-to-end Half-Double Attack runs on up-to-date Chromebooks and combines the Half-Double effect with exploit techniques, side channels, and a Spectre attack.

Outline. We provide background in Section 2 and introduce a new Rowhammer pattern notation in Section 3. We overview the the Half-Double effect in Section 4 and empirically verify that it is a new effect in Section 5. We develop the end-to-end attack in Section 6. We discuss the related work and implications in Section 7 and conclude in Section 8.

Responsible Disclosure. Pre-existing contractual obligations between a subset of the authors and the memory vendors mean we cannot provide details as to the effectiveness or substance of efforts to remediate the flaws. We believe the work should nonetheless be published as the impact of disclosure is unlikely to impact consumer security substantially, as the presence of Rowhammer-type vulnerabilities is a known limitation in DRAM design prior to our publication [30, 14]. Therefore, we believe that

[3]Our open-source proof-of-concept implementations can be found at: `https://github.com/iaik/halfdouble`

publicly disclosing our new variant will help rather than hinder the safe deployment of systems.

We responsibly disclosed Half-Double by notifying the affected memory vendors, triggering a customary embargo. The vulnerability was made public via a blog post after the expiration of the embargo [42].

2. Background

In this section, we provide background on DRAM, the Rowhammer effect, and the broadly deployed TRR mitigation.

2.1. DRAM Organization

The main memory system consists of multiple *channels*, which are independent links between memory controller and DRAM chips. Since DRAM chips have a narrow data bus, several of them are grouped into a *rank* whose aggregated data-bus width matches that of the channel. Multiple ranks can time-share a channel. Chips in a rank run in lockstep, *i.e.*, organizationally, like a single larger chip. Hence, we use the terms “rank” and “chip” interchangeably. Each rank consists of *rows* of capacitor-based DRAM *cells*. To access a row, the voltage of its wordline must be raised, which connects its cells to their respective bitlines. Referred to as *activation*, this procedure then involves what’s called the *row-buffer* – situated at the other end of the bitlines – to sense the voltage perturbations and to amplify them to either ‘0’ or ‘1’. This brings us full circle as the cells are restored to their original state: fully discharged or fully charged. As long as the same row remains activated, subsequent accesses are served from the row-buffer. Such *row hits* are faster than *row conflicts* which must activate a different row. To increase the probability of a row hit, rows within a rank are partitioned into *banks* with dedicated row buffers. Capacitors and, thus, DRAM cells lose charge over time. Thus, all rows must be *refreshed* at a regular interval which is typically 32-64ms [27]. Refreshes are spread out evenly over time, *i.e.*, refreshing a small subset of rows with each *refresh* command. We emphasize that *refreshing a row is exactly the same as activating it* [37] (see Section 3).

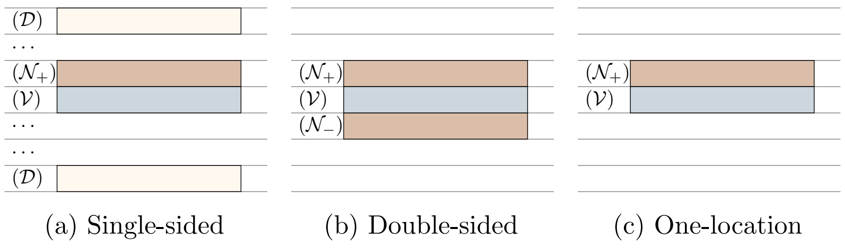


Figure 6.1.: Rowhammer access patterns: red rectangles (🟫) represent hammered rows, *i.e.*, the near aggressors $\mathcal{N}$, while blue rectangles (▨) represent the most likely row for bit flips, *i.e.*, the victim row $\mathcal{V}$. Single-sided hammering accesses a set of unrelated rows, which we call decoys $\mathcal{D}$ (□).

2.2. DRAM Address Reverse Engineering

Various attacks require specific placement of data in DRAM, motivating several works to reverse-engineer DRAM addressing functions. Pessl et al. [41] and later also Barenghi et al. [6] used the row buffer timing side channel. Jung et al. [28] reverse-engineered even the physical on-chip location via heat-based hardware-fault attacks. Helm et al. [20] used performance counters to measure row hits and misses. All of these methods group addresses into sets of addresses that see row conflicts or row hits with each other, *i.e.*, they are in the same bank. They then compute which combination of bits indicates the set, which is often a linear XOR-combination of bits. Helm et al. [20] showed that addressing functions can vary between different address ranges or channels. While older works found the row index to be just a subset of address bits [48, 56, 18, 41], more recent works also found XOR combinations to be used for index bits as well [49]. As a response to Rowhammer, physical addresses today are hidden from user programs [31], rendering approaches that rely on them unapplicable for attacks on up-to-date systems.

2.3. Rowhammer

At higher densities, chips are more likely to suffer from disturbance errors caused by intra-chip crosstalk [38]. In 2014, Kim et al. [30] demonstrated row-to-row disturbance errors in DRAM chips from memory accesses and called it *row hammer* [22]. Recently, Walker et al. [54] provided a comprehensive analysis of the underlying physics of Rowhammer.

Existing Rowhammer access patterns (Figure 6.1) vary depending on relative location of victim and aggressor(s). First, in the *single-sided* pattern [48], an attacker alternates accesses to two rows: an aggressor and what we call a *decoy*. Accesses to the decoy (an arbitrary row in the same bank) are needed to thrash the row-buffer to ensure that the aggressor is indeed activated. There is also an “amplified” variant [18] where two aggressors are placed next to each other. As the name suggests, they sometimes work to reinforce each other, yielding more bit flips in their respective victims than otherwise. Second, in the *double-sided* pattern [48], the victim is sandwiched between two aggressors. This is known to be the worst-case access pattern that induces the most bit flips. There is also a “many-sided” extension [14] that involves a larger number of aggressors and victims with varying degrees of sandwiching. Third, the *one-location* pattern [17] is similar to the single-sided one except that it eschews the decoy. Instead, it waits for the memory controller to clear the row-buffer before accessing the aggressor again to ensure that it is activated.

The Rowhammer vulnerability has been demonstrated in sandboxed environments [48], in native environments [48, 7, 17, 49], in virtual machines [56, 44, 25], in JavaScript [18, 8, 45], on mobile devices [51, 13], and over the network [50, 35]. Rowhammer exploits often borrow traditional exploit techniques such as memory spraying [48, 18, 56], grooming [51], and page deduplication [8, 44] to place the target data structure at the correct memory location. There are many proposals to improve hammering, with special instructions [43], load hazards [24], page table accesses [59], an onboard FPGA [55], and memory pressure from quality-of-service techniques [1].

Many defenses have been proposed [17], focused on detecting [23, 21, 40, 19, 10, 58, 53, 34, 39], neutralizing [9, 51, 18, 8, 44, 26], or eliminating [30, 4, 9, 29, 15] Rowhammer in software or hardware. A defense that has already been integrated into some DDR4 modules and the LPDDR4 standard [26] is Target Row Refresh (TRR), which we discuss in Section 2.4.

2.4. Mitigative Refreshes (a.k.a. “TRR”)

Starting from the (LP)DDR4 generation, vendors have implemented opaque and proprietary defenses inside their chips. Frigo et al. [14] found

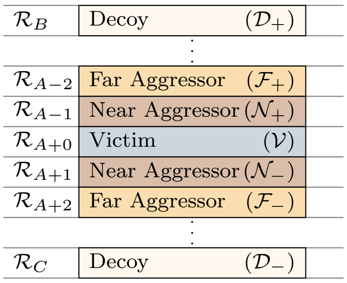

$\mathcal{R}_B$ Decoy $(\mathcal{D}_+)$
$\mathcal{R}_{A-2}$ Far Aggressor $(\mathcal{F}_+)$
$\mathcal{R}_{A-1}$ Near Aggressor $(\mathcal{N}_+)$
$\mathcal{R}_{A+0}$ Victim $(\mathcal{V})$
$\mathcal{R}_{A+1}$ Near Aggressor $(\mathcal{N}_-)$
$\mathcal{R}_{A+2}$ Far Aggressor $(\mathcal{F}_-)$
$\mathcal{R}_C$ Decoy $(\mathcal{D}_-)$

Figure 6.2.: Row annotations for the rows inside a single bank that surround the *victim* row.

that such measures appear to involve two main components: (i) a *sampler* identifying potential aggressors, and (ii) an *inhibitor* performing *mitigative refreshes* on their potential victims. Furthermore, the sampler is limited in its tracking capability and can be fooled when an attacker interleaves activations to multiple rows.

In contrast, our Half-Double Attack capitalizes on the shortcomings of the inhibitor, which is hardwired to perform mitigative refreshes on just the immediate neighbors without accounting for the longer-ranged effects of Rowhammer. In fact, we show how mitigative refreshes actually facilitate Half-Double Attack by turning their recipient row into a co-conspirator – more specifically, it becomes the near aggressor to the far one that necessitated the mitigative refresh.

In this paper, we use the terms “mitigative refresh” and “TRR” (Target Row Refresh) interchangeably. Despite its usage in previous works, the latter is a slight misnomer since it refers to a previously proposed (but never adopted) DRAM command that allows the CPU’s memory controller to send a row-address alongside a refresh command [5].[4]

3. A Systematic Rowhammer Pattern Notation

In this section, we introduce a new systematic notation for Rowhammer patterns, allowing us to categorize existing attacks and describe the Half-Double Attack effect in Section 4.

[4]“Pseudo TRR” is emulating that behavior by sending a pair of activation and precharge commands to refresh the desired row manually.

Our notation describes Rowhammer patterns and their locality concerning the actual row location inside a bank.We assume the row index represents the physical row position inside a bank. For the notation, we assume that rows with contiguous row indices are physically adjacent. Figure 6.2 denotes the rows inside a bank as follows. The *victim* row ($\mathcal{V}$) is the target of the Rowhammer attack, and the bit flips inside this row are used to measure the effectiveness of the pattern in the experiments. The direct neighbors of the *victim* row are the so-called *near aggressor* rows ($\mathcal{N}_+$, $\mathcal{N}_-$), which are directly followed by the *far aggressor* rows ($\mathcal{F}_+$, $\mathcal{F}_-$). These three types of rows are located in a contiguous range inside a bank. We denote rows further away from this range as *decoy* rows ($\mathcal{D}$). The absolute row position of the upper *far aggressor* row ($\mathcal{F}_+$) is denoted with $\mathcal{R}_{A-2}$, this allows us to address these rows with an index. To characterize the Rowhammer patterns, we use a special notation, e.g., $(\mathcal{A}_i \rightarrow (\mathcal{B} \rightarrow \mathcal{C})^{\beta})^{\infty}$, where i is the current repetition of the selected pattern. Hence, the first memory access goes to $\mathcal{A}_0$. Then, the pattern accesses the rows $\mathcal{B}$ and $\mathcal{C}$ and repeats these two accesses β times. After β accesses to $\mathcal{B}$ and $\mathcal{C}$, we continue with the next iteration, *i.e.*, row $\mathcal{A}_1$ is accessed next, and so on.

With this notation, we compare known Rowhammer patterns based on their row locality. Double-sided Rowhammer [48] uses two *near aggressors* to hammer the *victim* row, *i.e.*, we can express the pattern as $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^{\infty}$. Single-sided Rowhammer [48] effectively uses one *near aggressor* to hammer the *victim* row and 7 *decoy* accesses, *i.e.*, we can express the pattern as $(\mathcal{N}_+ \rightarrow (\mathcal{D}_i)^7)^{\infty}$. However, the purpose of these decoy rows is often to trigger a row conflict on DIMMs that use an open row policy, as otherwise, the accesses are served from the row buffer (cf. Section 2.1). More recent Rowhammer type attacks like TRRespass [14] and Smash [45], also use *near aggressor* and *decoy* rows. However, in both cases, multiple *victim* rows are targeted interleaved to exploit the limited TRR *sampler* size and deplete the number of protected rows. This allows the attack to induce flips in one of the *victim* rows hammered less often as the TRR mitigation no longer protects them.

In summary, all existing Rowhammer patterns use *near aggressor* rows to hammer, *i.e.*, they are *distance-1* patterns, directly surrounding single or multiple *victim* rows. The TRR mitigation is designed to mitigate these *distance-1* type attacks. TRR detects these repeated accesses to the *near aggressors* with the *sampler*, and then the *inhibitor* refreshes the *victim* row before a bit flip can occur (cf. Section 2.4). The detailed implementation of TRR refreshes is vendor specific and not publicly documented. We assume similar to Liu et al. [37], that TRR refreshes are implemented by closing the currently open row and then opening the *victim* row to load the row into the row buffer and, therefore, refresh the content of the victim. However, this raises the question of whether there are practically exploitable *distance-2* patterns.

4. The Half-Double Effect and Exploit

This section provides an overview of the Half-Double Attack, its new hammering patterns, and challenges for the attack.

4.1. The Half-Double Effect

With Half-Double Attack, we present two new Rowhammer patterns, the *Quad pattern* and the *Weighted pattern* (or more verbosely, Weighted Single Plus Decoys (WS+D)).

The *Quad pattern* (Pattern (6.1)) shifts the double-sided Rowhammer pattern outwards by one row:

$$(\mathcal{F}_{+} \rightarrow \mathcal{F}_{-})^{\infty}. \tag{6.1}$$

However, as this only drains a small amount of charge from the actual *victim* row, the *Quad pattern* incorporates the effects of TRR refreshes to hammer the *victim* row. The pattern uses the *far aggressors* ($\mathcal{F}_{+}$, $\mathcal{F}_{-}$) to hammer. Hammering the *far aggressors* is detected by the TRR *sampler*, and after a sufficient number of row activations, the TRR *inhibitor* issues (in an attempt to mitigate bit flips) a refresh command to the *near aggressors* ($\mathcal{N}_{+}$, $\mathcal{N}_{-}$). The TRR refresh mechanism closes the open row and activates the *near aggressor* rows successively to refresh them. These additional activations from the TRR refresh mechanism *assist* our hammering of the *victim* row by draining further charge from the *victim* row ($\mathcal{V}$).

The *Weighted pattern* (Pattern (6.2)) distributes half the hammers to the upper *far aggressor* ($\mathcal{F}_{+}$), and the others to the rows *below* the *victim* row. Thus, we represent it as

$$(\mathcal{R}_{A+4+3\cdot i} \rightarrow \mathcal{F}_{+} \rightarrow \mathcal{R}_{A+6+3\cdot i} \rightarrow \mathcal{F}_{+})^{\infty}. \tag{6.2}$$

The intuition of this pattern is to shift a double-sided Rowhammer pattern over the rows below the *victim*, while distributing half of the hammers to the *far aggressor* ($\mathcal{F}_+$). As the maximum number of rows inside a bank is limited, i is wrapped around to zero if $\mathcal{R}_{A+6+3\cdot i}$ is outside the physical row range, restarting the pattern below the *victim*. The first two repetitions of the *Weighted pattern* produce the following sequence: $\mathcal{R}_{A+4}$, $\mathcal{F}_+$, $\mathcal{R}_{A+6}$, $\mathcal{F}_+$, $\mathcal{R}_{A+7}$, $\mathcal{F}_+$, $\mathcal{R}_{A+9}$, $\mathcal{F}_+$. Similar to the *Quad pattern*, the *Weighted pattern* hammers the *far aggressor* ($\mathcal{F}_+$), but accesses *decoys* below the *victim*. The pattern accesses the *far aggressor* triggering the mitigative refresh mechanism (TRR) on the *near aggressor* ($\mathcal{N}_+$) *assisting* the hammering by draining further charge from the *victim* ($\mathcal{V}$).

We describe the effects of the Half-Double patterns with the following hypothesis $\mathcal{H}$ under which the Half-Double patterns induce flips into the *victim row*.

Hypothesis $\mathcal{H}$: Hammering *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$) triggers mitigative refreshes (TRR) on *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$), implicitly assisting the hammering of the *victim* row ($\mathcal{V}$), by draining charge from it. However, the refreshes of the *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$) cannot draw sufficient charge from the *victim* row without the activations of the *far aggressors* ($\mathcal{F}_+$, $\mathcal{F}_-$).

Compared with multi-sided Rowhammer [14, 45], Half-Double patterns do not rely on depletion of TRR resources. Instead, the patterns incorporate the TRR refresh mechanism such that is assists the Rowhammer attack. Therefore, this pattern is also applicable in a scenario where the TRR mechanism works perfectly against *distance-1* Rowhammer attacks. We evaluate and discuss the differences between Half-Double and state-of-the-art Rowhammer attacks in Sections 5.1.2 and 7.

4.2. The Half-Double Exploit

Rowhammer exploits typically involve solving several challenges beyond the bit flip. For Half-Double on state-of-the-art systems, we identified 4 challenges: **C1** the allocation of contiguous memory (without physical address information or huge pages), **C2** finding bit flips without templating (to bypass defenses against templating), **C3** memory spraying with constrained spraying resources, and **C4** bit flip verification (due to the uncertainty created by hammering blind). For Challenges 1 and 3 we can extend existing techniques. However, for Challenge 2, ECC memory hinders bit flip templating as the ECC code depends on the data in the corresponding cells unknown to the attacker. A novel approach we call *Blind-Hammering* circumvents this problem by not templating for bit flips. However, this introduces uncertainty, creating Challenge 4, which we resolve by combining *Blind-Hammering* with a Spectre attack. Hence, we put more focus on Challenge 2 and Challenge 4 as they require novel methods.

Challenge 1: Allocation of Contiguous Memory. The first challenge is to obtain access to adjacent rows in a bank. Physical address information is unavailable today due to previous Rowhammer attacks [31]. Huge pages or page fusion mechanisms are not available on all systems, making both approaches unapplicable for our attack. Therefore, we first design a novel contiguous memory detection, incorporating knowledge of the general structure of xor-based DRAM addressing functions to obtain information on the underlying physical addresses, even if the DRAM addressing functions of the device are unknown. Combined with knowledge of the behavior of the buddy allocator, we obtain information on the underlying physical addresses. Second, due to the precise row location requirements of the Half-Double patterns, we need to reverse-engineer the row indexing function of the bank using a timing side channel. Finally, we map a virtual address via the contiguous memory to a bank and row.

Challenge 2: An Alternative to Memory Templating. After controlling contiguous rows inside a bank, the next challenge is to find flippable locations inside the memory. Due to variances in the DRAM cells, some cells are more likely to flip than others [54]. The current state of the art is templating the memory in advance for locations that are susceptible to Rowhammer flips. However, some ECC memory hinder this step as bit flips are only reproducible in an attack if the templating was performed on the exact data or data that behaves identically for the ECC code. An attacker usually does not have this information, hindering the memory templating approach. Thus, we propose a new approach without memory templating, namely *Blind-Hammering*.

Challenge 3: Memory Massaging. This challenge focuses on filling the memory with targets that are exploitable with *Blind-Hammering*. The targets of our exploit are page table entries. We target the physical page number inside these page table entries. We use an approach where we map shared memory between multiple children of the parent process to fill the memory with additional page tables without filling the main memory with other non-exploitable data pages.

Challenge 4: Bit-Flip Verification. This challenge focuses on determining the location of an induced bit flip. Due to *Blind-Hammering*, we cannot directly check whether the hammering was successful or not, as accessing a potential corrupted page-table entry (PTE) is detected by the OS, terminating the exploit. We solve this problem with a novel Spectre oracle [32] determining whether the address is safe to access. We also develop an architectural alternative oracle and evaluate the advantages and drawbacks of both approaches.

We solve the above challenges in Section 6 and gain complete control over the system's main memory, proving that the Half-Double effect can be exploited on real-world systems.

5. Empirical Evaluation of Half-Double

To show that the hypothesis $\mathcal{H}$ holds and explains the Half-Double effect, we make the following observations:

1. We show that the Half-Double effect exists, *i.e.*, inducing bit flips on current TRR-protected systems (Section 5.1).
2. We show that Half-Double does not occur without (TRR-induced) refreshes on *near aggressors* and that there is a relation between TRR refreshes and the number of bit flips observed, *i.e.*, (counter-intuitively) more TRR refreshes lead to more bit flips in the *victim* row (Section 5.2).

To obtain noise-free observations, we use an FPGA board with full control over all refreshes and memory accesses, where we have no requirements on data retainment for stability (Section 5.3). Since the focus of this section is to show the above points and, thus, that the hypothesis $\mathcal{H}$ holds, we do not restrict ourselves to a specific threat model in this section.

5.1. Half-Double on TRR-protected LPDDR4x

In this subsection, we demonstrate Half-Double using the *Quad pattern* on TRR-protected systems.[5] We show that this pattern can generate bit flips and record the number of observed bit flips to measure the performance.

[5]We analyze the *Weighted pattern* in Section 5.2 and Section 5.3.

5.1.1. Test System and DRAM Addressing Functions

We use 10 commodity systems (see Table 6.9 for a full list). We reverse-engineer the DRAM addressing functions using the method by Pessl et al. [41] (cf. Section 2.2). Since their approach only maps a physical address to a given bank but does not recover the precise row index we need for the *Quad pattern*, we use an additional timing side channel between row hits and row conflicts within a bank [49] (cf. Section 2.2) to obtain information on the row indices. We discover that our two identical ARM-based Lenovo Chromebooks have the same row scrambling functions described by Tatar et al. [49], where bit 3 is XORed onto bits 2 and 1 in the row index. We illustrate the full DRAM addressing and indexing functions for the Chromebooks in Figure 6.3. With the device-specific functions, we map physical addresses to banks and row indices and, thus, test the Half-Double patterns from Section 4.1.

5.1.2. Evaluation of the *Quad pattern*

We test the *Quad pattern* with two strategies to reach DRAM: uncacheable memory [52] and memory flushing [30]. For uncacheable memory, we mark the *far aggressors*, the *near aggressors*, and the *victim* as uncacheable, allowing more hammering attempts within one refresh interval. The memory flushing approach is much slower, relying on the architecture's flush instruction to flush the *far aggressors* from the cache.

For our evaluation, we allocate a large chunk of memory and use the Linux `pagemap` interface [31] to extract physical addresses. We analyze the physical addresses of the chunk and group the virtual addresses corresponding to their banks. Afterwards, we search for addresses from the same bank mapping to a consecutive range of rows representing the *Quad pattern*, *i.e.*, we find rows $\mathcal{R}_{A-2}$ to $\mathcal{R}_{A+2}$, cf. Figure 6.2.

Modern memory controllers scramble data by XORing a mask onto the row's data. Cojocar et al. [11] showed that the data mask is the same across all rows. We empirically observed the data scrambling and, correspondingly, set all bytes of the *far aggressor* and *near aggressor* rows to `0x55` and fill the bytes of the *victim* row with `0xaa`. We found that this maximizes the number of bit flips we see in the *Quad pattern* attack across the tested devices.

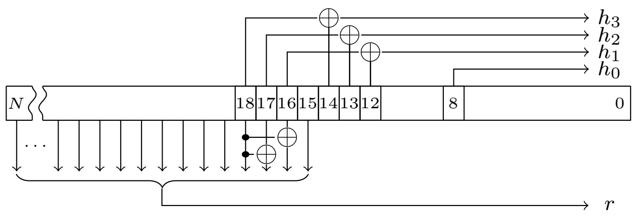


Figure 6.3.: The reverse-engineered DRAM addressing functions from our Chromebooks.

Table 6.1.: Performance of the *Quad pattern* with uncacheable memory and flush instruction on affected LPDDR4x systems.

System	$\mathbf{N}_{Hammers}$	$\mathbf{UC}_{0\to1}$	$\mathbf{UC}_{1\to0}$	$\mathbf{Flush}_{0\to1}$	$\mathbf{Flush}_{1\to0}$
Chromebook$_1$	23 274	27	40	2	5
Chromebook$_2$	23 586	235	2379	12	101
OnePlus 5T	25 687	2	30	1	24
Pixel 3	32 921	11	5	0	0
HTC U11	21 840	-	-	3	17

The hammering runs in a tight loop accessing the *far aggressors*. We run the *Quad pattern* for 20 000 000 iterations and check the *victim* row for bit flips. Table 6.1 shows the results of both approaches. The Chromebook$_2$ shows 36 times more flips than the identical Chromebook$_1$. The OnePlus 5T shows similar flip tendencies as the Chromebook$_1$. With uncacheable memory, we can induce 10 to 20 times more bit flips on the Chromebooks. However, the OnePlus 5T does not show a huge increase when using uncachable memory. We also observe more flips from 1 to 0, similar to Kim et al. [30]. However, we conclude that an attack is possible in either case, albeit faster if uncacheable memory is available.

Key Insight: Half-Double is capable of producing bit flips on TRR-protected memory.

To compare the Half-Double effect with current state-of-the-art multi-sided Rowhammer patterns, we performed three experiments using TR-Respass [14] with up to 20 aggressors on our most susceptible commercial system, *i.e.*, the Chromebook$_2$. First, we ported the publicly available TRRespass tool to ARM, including row scrambling and uncacheable memory support to search for bit flips. Second, we implemented the patterns in our hammering tool for cross-validation. We evaluated hammering uncachable memory with the *Quad pattern* with one of a 12 aggressor multi-sided pattern under the same conditions. We did not observe any bit flips with multi-sided patterns, whereas the *Quad pattern* induced 956 flips over the same time frame. This experiment concludes that there are commodity devices that are affected by Half-Double but not (or less) by other state-of-the-art patterns. Section 7 provides further discussion of Half-Double and multi-sided Rowhammer.

5.2. Determining the Role of TRR

With the experiments on the commodity systems, we cannot rule out that the observed flips are *distance-1* flips induced solely by TRR (or other row refreshes), or that they are actually *distance-2* bit flips induced by *far aggressor* hammering. In line with prior work [54], we observe a small number of *distance-2* bit flips, too infrequent to explain Half-Double (cf. Section 5.3). Furthermore, Helm et al. [20] found complex addressing functions that change depending on the actual physical location. To exclude this possible source of error, we use a commercial SoC platform with LPDDR4x memory to measure the influence of refreshes, e.g., those from TRR.

We obtained precise but confidential information from the vendor on the relationship between the actual physical row location inside a bank and the physical address on this SoC platform. For this system, we can switch memory refreshes off and on. However, this switch disables not only the TRR refreshes but also refreshes issued by the memory controller to conform to the refresh interval (e.g., 64 ms) or by pTRR. Completely disabling refreshes renders the system unusable, as DRAM cells lose charge and corrupt data after a short period. To still be able to run actual software, we build a duty cycle mechanism alternating between enabled and disabled refreshes, we denote as *dance* (*i.e.*, dancing between refresh on and off). In these *dance* experiments, we enable refreshes for 25 % of the time, *i.e.*, 64 ms enabled and 192 ms disabled.

The *dance* experiments allow limiting the number of refreshes temporarily and, therefore, observe the correlation between refreshes and the number of bit flips. While refreshes are disabled, they cannot unintentionally assist our Half-Double patterns, *i.e.*, no interfering TRR refreshes. While the window where refreshes are disabled is longer than the standard refresh period (e.g., 64 ms), it is short enough to avoid instabilities. We expect a

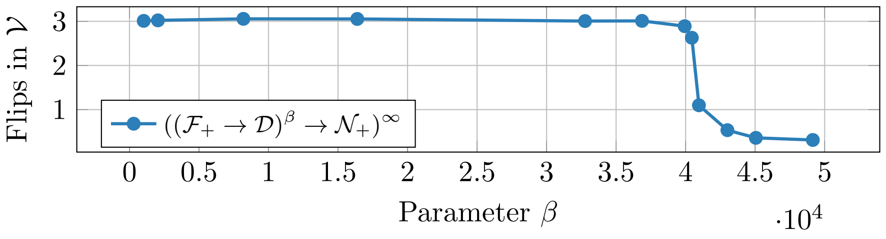


Figure 6.4.: Number of observed bit flips in the *victim* over the dilution parameter for the single-sided case.

reduction of the number of bit flips with a decreasing number of refreshes if our hypothesis $\mathcal{H}$ holds, *i.e.*, the TRR refreshes assist the Half-Double effect.

To show that TRR refreshes *assist* the observed bit flips from Section 5.1, we run the following experiment: We design **three** pattern categories to demonstrate the Half-Double effect: the first to show that the effect is not explained by *distance-2* hammering, the second to show that simulated TRR refreshes trigger the effect as well, and the third to show that only the simulated TRR refreshes alone do not trigger the effect. The patterns used to verify the hypothesis are constructed around the *victim row*. These observations show that only the combination of the TRR refreshes (or other accesses) to the *near aggressors* combined with our accesses to the *far aggressors* trigger the Half-Double effect, confirming our hypothesis $\mathcal{H}$. For our experiment we use a *victim* row that was reliably susceptible to Rowhammer attacks, *i.e.*, we usually were able to induce three bit flips with the *Weighted pattern*.

The **first** category verifies that the observed bit flips are not caused solely by *distance-2* hammering. The single-sided Pattern (6.3) accesses *far aggressor* ($\mathcal{F}_+$) and a *decoy* ($\mathcal{D}$).

$$(\mathcal{F}_+ \rightarrow \mathcal{D})^\infty \tag{6.3}$$

The double-sided Pattern (6.4) replaces this *decoy* with an access to the lower *far aggressor* ($\mathcal{F}_-$).

$$(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty \tag{6.4}$$

While with TRR, we observed a significant number of bit flips, in our experiments, both patterns do not show any considerable number of bit flips with TRR refreshes disabled. The number of bit flips observed is far

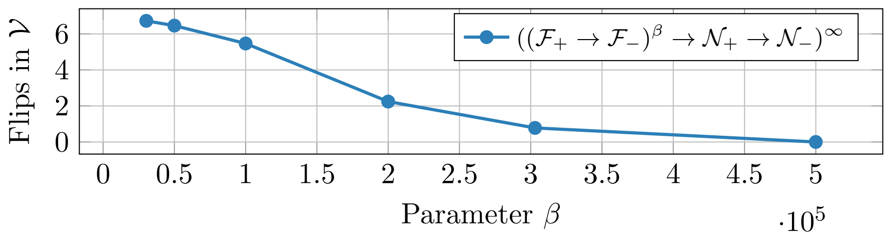


Figure 6.5.: Number of observed bit flips in the *victim* over the dilution parameter for the double-sided case.

too low to visualize them or to explain the Half-Double effect by *distance-2* Rowhammer bit flips (as we detail further in Section 5.3). Hence, this indicates that TRR refreshes contribute to the Half-Double effect, in support of our hypothesis $\mathcal{H}$.

The **second** category of patterns extend the Half-Double patterns with simulated TRR refreshes to the *near aggressors* while TRR is disabled. Patterns (6.3) and (6.4) are repeated β times before simulated TRR accesses to the *near aggressors* are performed. The single-sided Pattern (6.5) simulates TRR by accessing the *near aggressor* ($\mathcal{N}_+$).

$$((\mathcal{F}_+ \rightarrow \mathcal{D})^\beta \rightarrow \mathcal{N}_+)^\infty \tag{6.5}$$

The double-sided Pattern (6.6) simulates TRR by accessing both *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$).

$$((\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\beta \rightarrow \mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty \tag{6.6}$$

The parameter β is a *dilution* parameter allowing us to vary how many simulated TRR refreshes are performed, *i.e.*, we perform accesses to the *near aggressors* after a certain amount of accesses to the *far aggressors*.

For a low dilution parameter (*i.e.*, very high number of accesses to the *near aggressors*), these patterns behave like a traditional single-sided or double-sided Rowhammer attack without TRR protection, *i.e.*, inducing many bit flips. We confirmed this empirically, shown in Figure 6.4 for the singled-sided Pattern (6.5) and in Figure 6.5 for the double-sided Pattern (6.6). We see a correlation between the dilution parameter β and the number of bit flips, *i.e.*, fewer simulated refreshes lead to fewer bit flips. An alternative representation is also provided in Section 9.2. From these observations, we conclude that accesses to the *near aggressor* contribute

Table 6.2.: Number of bit flips observed in our FPGA setup (rows with bit flips in parentheses). The hammer duration determines the number of accesses (hammer count). The hammer duration has a stronger influence on the number of bit flips and affected rows than the dilution factor. Even a dilution factor of 3712, within one 64 ms refresh interval fits 950 272 accesses, 256 of which to the *near aggressors* simulating TRR (cf. Section 3), still induces bit flips in *all* 32 *rows.* Thus, only 256 accesses to the *near aggressors* combined with 950 016 accesses to the *far aggressors* are sufficient to attack any row.

	Accesses	296 960	356 352	415 744	475 136	534 528	593 920	653 312	712 704	772 096	831 488	890 880	950 272
	Duration	20 ms	24 ms	28 ms	32 ms	36 ms	40 ms	44 ms	48 ms	52 ms	56 ms	60 ms	64 ms
Dilution Factor	58	1 (1)	3 (3)	5 (5)	6 (6)	15 (12)	26 (19)	35 (20)	44 (23)	57 (28)	83 (30)	115 (32)	173 (32)
	116	1 (1)	3 (3)	4 (4)	6 (6)	14 (11)	24 (19)	32 (20)	40 (22)	51 (27)	73 (30)	117 (32)	152 (32)
	232	1 (1)	3 (3)	4 (4)	5 (5)	12 (10)	24 (19)	31 (20)	39 (21)	51 (27)	68 (30)	112 (32)	149 (32)
	464	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	24 (18)	32 (20)	39 (21)	49 (26)	70 (30)	109 (32)	148 (32)
	928	1 (1)	2 (2)	3 (3)	5 (5)	11 (8)	25 (18)	32 (20)	39 (21)	49 (25)	70 (29)	108 (32)	146 (32)
	1856	0 (0)	2 (2)	3 (3)	5 (5)	11 (8)	22 (17)	32 (20)	37 (21)	49 (25)	66 (29)	110 (32)	140 (32)
	3712	0 (0)	2 (2)	3 (3)	5 (5)	10 (7)	22 (16)	30 (20)	37 (21)	49 (25)	64 (27)	99 (31)	139 (32)
	7424	0 (0)	2 (2)	3 (3)	5 (5)	8 (6)	18 (15)	29 (19)	36 (20)	48 (25)	66 (27)	92 (31)	128 (31)
	14 848	0 (0)	0 (0)	2 (2)	4 (4)	7 (6)	15 (12)	22 (15)	32 (19)	40 (22)	58 (27)	80 (30)	109 (30)
	29 696	0 (0)	0 (0)	2 (2)	2 (2)	3 (3)	8 (7)	11 (9)	19 (14)	28 (18)	41 (25)	57 (27)	82 (29)

to the observed bit flips, supporting our hypothesis $\mathcal{H}$. However, we also need to test the null hypothesis, as we do in the following.

For this purpose, the **third** pattern category implements placebo patterns to verify that the bit flips in the second pattern category are not caused solely by the accesses to the *near aggressors* (*i.e.*, the null hypothesis). The patterns access *decoys* to keep the overall accesses rate to *near aggressors* the same as Patterns (6.5) and (6.6) for a given dilution. Thus, the single-sided Pattern (6.7) accesses one *near aggressor* ($\mathcal{N}_+$).

$$((\mathcal{D}_1 \to \mathcal{D}_2)^\beta \to \mathcal{N}_+)^\infty \tag{6.7}$$

The double-sided Pattern (6.8) accesses both ($\mathcal{N}_+$, $\mathcal{N}_-$).

$$((\mathcal{D}_1 \to \mathcal{D}_2)^\beta \to \mathcal{N}_+ \to \mathcal{N}_-)^\infty \tag{6.8}$$

When varying the dilution parameter, we observe a point at which the second category of patterns still produce bit flips in the *victim*, whereas the third category no longer does. More concretely, Pattern (6.7) shows a decrease in the bit flips before Pattern (6.5). We observe the same effect also for the double-sided case where the number of bit flips with Pattern (6.8) drops at a lower dilution parameter than the number of bit flips with Pattern (6.6). Since we only access *decoys* $\mathcal{D}_i$, unrelated to the *victim* row $\mathcal{V}$, this drop is explained by the missing accesses to the *far aggressors* $\mathcal{F}$, supporting our hypothesis $\mathcal{H}$. We conclude that

Table 6.3.: The modules used in the FPGA analysis. M_1 is not affected by Half-Double, M_3 is affected within default refresh windows (64 ms), M_2 is affected with longer windows.

Module	**Freq.**	**Size**	**Ranks**	**Banks**	**Pins**	**Half-Double**
M_1	2666	4 GB	1	8	x16	✗
M_2	3200	4 GB	1	8	x16	✓(>64 ms)
M_3	3200	8 GB	1	8	x16	✓

the additional accesses of the TRR *inhibitor* assist the hammering of the *victim* for the Half-Double effect.

Key Insight: TRR refreshes assist Half-Double but are not the root cause. They alone induce no bit flips in the *victim*.

5.3. Noise-free FPGA Experiments

To confirm our results without noise, we use the *ZCU104* FPGA platform[6] where we have full control over all refreshes and memory accesses and no requirements on data retainment for stability. In contrast to Section 5.2, we can disable all refreshes on the FPGA-based platform because the platform itself does not store any data in the DIMM, *i.e.*, the FPGA does not use the DIMM as system memory.

We analyzed three off-the-shelf DDR4 DIMMs listed in Table 6.3 and found that M_1 is not susceptible to the Half-Double effect. While the other two are affected, we could only demonstrate bit flips with the default refresh interval of 64 ms on M_3, whereas M_2 required a doubled refresh interval (128 ms). Therefore, for our analysis, we focused on M_3, the DIMM susceptible to Half-Double by default. For our experiments, we use the same patterns as in Section 5.2 and confirm our results in this highly controlled and noise-free setup.

$$((\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\beta \rightarrow \mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty \tag{6.9}$$

With Pattern (6.9), we hammer 32 rows individually and vary both dilution parameter β and total hammer duration (*i.e.*, hammer count, number of accesses) in this experiment. Furthermore, we vary the hammer duration from 20 ms to 64 ms with a step size of 4 ms. In contrast to the dilution

[6] `https://github.com/antmicro/litex-rowhammer-tester`

Table 6.4.: *Distance-1* double-sided hammering $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$ and the observed bit flips per cell and row.

Hammers	**Time (ms)**	**Cells**	**Rows**
18 000	1.212	2	1
24 000	1.616	23	18
30 000	2.020	136	31
36 000	2.425	495	32
42 000	2.829	1395	32
48 000	3.233	2870	32
54 000	3.637	5099	32
60 000	4.041	7749	32

parameter, we introduce the dilution factor d_f. This factor slightly changes the representation of β. The relation between the dilution factor and the dilution parameter for Pattern (6.9) is $d_f = \beta + 1$. A dilution factor of 58 refers to 1 *distance-1* hammer in every 58 hammers. Therefore, we can compute the accesses to the *near aggressors* directly by dividing the total hammers by the dilution factor. The dilution factor is varied from 58 to 29 696 by doubling it in each step.

Table 6.2 shows the results of this experiment and we can observe two effects, as expected: First, the number of bit flips increases with the hammer duration. We can induce bit flips in all 32 rows within one default refresh interval (64 ms) regardless of the tested dilution factor. Second, the number of bit flips decreases with a higher dilution. However, the decrease in bit flips is much flatter than for the hammer duration. Even with the highest tested dilution, we induce flips into 29 out of 32 rows within one default refresh interval.

To underline that the Half-Double effect is a different phenomenon than *distance-1* and *distance-2* Rowhammer effects, we test two more patterns on the FPGA system and compare them with the results obtained in the previous experiment. We use *distance-1* double-sided Rowhammer $(\mathcal{N}_+ \rightarrow \mathcal{N}_-)^\infty$ and the *distance-2* variant $(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty$. We again hammer 32 rows and measure the observed flips on a cell and row basis.

Table 6.4 shows the results of *distance-1* double-sided hammering, where the number of hammers required to induce flips into all 32 rows is only 36 000. This is 25 times smaller than with Half-Double, indicating that Half-Double is not just *distance-1* Rowhammer. However, 36 000 accesses are also much higher than what a TRR implementation could perform

Table 6.5.: *Distance-2* double-sided hammering $(\mathcal{F}_+ \rightarrow \mathcal{F}_-)^\infty$ and the observed bit flips per cell and row.

Hammers	**Time (ms)**	**Cells**	**Rows**
4 000 000	270	1	1
5 000 000	336	1	1
6 000 000	404	2	2
7 000 000	472	2	2
8 000 000	538	3	3
9 000 000	606	2	2
10 000 000	674	3	3

within the standard 64 ms refresh interval. Even at a low dilution factor like 58, this would require about 2 088 000 accesses within one refresh interval, *i.e.*, about twice as many accesses than fit in the standard refresh interval.

Table 6.5 shows *distance-2* double-sided hammering and we observe that we need 4 000 000 hammer accesses to obtain a *single distance-2* bit flip, *i.e.*, four times more accesses than fit within a 64 ms refresh interval. Hence, Half-Double can also not be explained with *distance-2* bit flips.

In line with Section 5.2, this again shows that $\mathcal{H}$ holds. With our results from Table 6.2, we can model when the Half-Double effect occurs. With a dilution factor of 3712 and 950 272 hammers, the total number of accesses to the *near aggressors* is 256. Therefore, only 256 accesses to the *near aggressors*, combined with 950 016 accesses to the *far aggressors* are sufficient to induce flips in each of the 32 rows. However, if we compare these numbers with the equivalent accesses in the *distance-1* and *distance-2* experiments, we are far below the required accesses to see even a single bit flip in both cases.

Key Insight: Both *distance-1* Rowhammer and *distance-2* Rowhammer effects would require more accesses than fit inside the standard 64 ms refresh interval if they would induce the Half-Double effect. Hence, we conclude that $\mathcal{H}$ is the most plausible explanation of Half-Double.

6. Half-Double Attack Exploit

In this section, we demonstrate the real-world attack capabilities of the Half-Double Attack. The attack is split into multiple phases, each tackling one of the challenges (see Section 4.2) to finally gain complete control over the system from within an untrusted executable. Our attack aims to induce a bit flip in the physical page-frame number of a PTE. If the corrupted page-frame number points to attacker-controlled data instead of the original page table, the adversary can forge additional page table entries. This grants the adversary arbitrary read and write access to the entire system memory.

Threat Model. We assume that the victim runs an untrusted executable or Android APP on either an ARM or x86 based system for our attack. Our attack does not exploit any software vulnerabilities in the OS or other running programs but only uses side-channel information and the provided interfaces by the OS. Furthermore, we assume that the LPDDR4x DRAM used on the system is both ECC- and TRR-protected. However, our attack does not rely on the exhaustion of TRR resources like previous Rowhammer attacks targeting TRR [14]. We evaluate the challenges on the Chromebooks, the OnePlus 5T and a Lenovo T490s to show the applicability across multiple architectures and operating systems (see Section 9.1).

From Virtual Memory Accesses to Half-Double Patterns. In the first step of our exploit, we map virtual addresses to actual physical row locations inside the DRAM banks, a building block to hammer with the Half-Double patterns. While we can use the *Quad pattern* and the *Weighted pattern*, we focus on the *Quad pattern* in our exploit as it induces bit flips faster. For the *Quad pattern*, we need to control at least five adjacent rows where the middle row is unmapped and used by the victim process. With DRAM addressing functions (cf. Section 5.1.1 and Figure 6.3), we can determine the physical location inside a bank and row. However, the required physical address information is not available to the unprivileged executable. We solve this challenge (**C1**) in Section 6.1.

Inducing Bit Flips. In the second step, we need to place potential bit flip targets at the right memory locations. Templated bit flips are very likely not reproducible during the actual attack, as the data in the victim rows differs between templating and attack phase, and the integrated ECC mechanisms of the DRAM depend on the actual data stored in the victim row. Consequently, bit flips during templating may not occur when attempting to fault the targeted data during the attack. Therefore, in Section 6.2, we present two new ways to solve **C2**. The first technique uses an alternative templating process that is ECC-aware. The second technique, called *Blind-Hammering*, is a versatile alternative to templating. However, verifying whether exploitable bit flips occurred becomes its own challenge then as we outline below.

Placing Exploitable Data. In the third step, we fill the system's memory with PTEs. However, due to address space limitations modern ARM-based devices enable for performance reasons, we cannot use the same spraying techniques as prior Rowhammer attacks. In Section 6.3, we solve this challenge **C3** by spawning child processes to increase the number of page tables in memory via multiple address spaces.

Bit-Flip Verification. We cannot directly access the hammered victim rows. Attempting to access a corrupted mapping is also fatal, as the Linux kernel detects corrupted PTEs upon faults and terminates the corresponding user-space process. In Section 6.4, we solve **C4** and use a Spectre attack to prevent irrecoverable crashes of our attacking app.

Combining all steps, we obtain a full **end-to-end exploit** with read and write access to the entire system's memory.

6.1. C1: Memory Allocation

To use the Half-Double patterns, the adversary needs access to at least 5 adjacent rows within the same bank. We present three distinct approaches to solve this challenge. Either via huge pages, using unique bank access patterns if the DRAM addressing functions are known, or by using the structure of unknown xor-based DRAM addressing functions

Via Huge Pages. The Chrome OS running on the Chromebooks as well as the Ubuntu running on the T490s has transparent huge pages activated. For 2 MB huge pages, the lowest 21 bits of virtual and physical address are the same. This covers all bits we need to find adjacent rows across the test systems, effectively solving this challenge (cf. Figure 6.3).

Via DRAM Addressing Functions. Disabling huge pages mitigates the aforementioned approach. Unfortunately, Half-Double requires specific row index information going beyond contiguency information from prior

Table 6.6.: *Page distance* patterns on the Chromebooks. Each pattern has one unique *page distance* highlighted in yellow.

P	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	...
$\mathbf{P}_0$	8	9	8	9	8	9	8	9	8	9	8	9	8	9	8	**1**	...
$\mathbf{P}_1$	8	7	8	11	8	7	8	11	8	7	8	11	8	7	8	**3**	...
$\mathbf{P}_2$	8	9	8	5	8	9	8	**13**	8	9	8	5	8	9	8	5	...
$\mathbf{P}_3$	8	7	8	7	8	7	8	**15**	8	7	8	7	8	7	8	7	...

work [47, 13, 33]. However, we can combine information on DRAM addressing functions (cf. Figure 6.3) and the *buddy allocator* [16] used in Linux and Chrome OS to detect contiguous memory blocks and reconstruct the additional physical address bits we require.

When dealing with a contiguous range of physical memory, the pages of the memory range are distributed over multiple banks due to the DRAM addressing functions. We now select only pages of a given bank and iterate over all the allocated pages to analyze the *page distances*. The *page distance* is the distance between two pages in the same bank. If we fix the bank and analyze the allocated memory, we observe that the *page distances* on the Chromebooks follow one of four patterns. Table 6.6 shows these four *page distance* patterns. The patterns have a period of 16, and each pattern has one unique *page distance*, which is highlighted in yellow. We only find four patterns since we can skip bit 8 of the DRAM addressing functions, as we always control it with the virtual address, leaving only eight remaining banks. However, we only observe four patterns as two banks share the same pattern.

All four *page distance* patterns have one unique distance per period (1,3,13, or 15). This unique value allows us to reconstruct the physical address bits 12 to 17, since the DRAM addressing functions start at bit 12 (cf. Figure 6.3). If we advance the unique *page distance* in pages (4 kB) in the physical memory, the underlying address bits 12 to 17 of the physical address change. However, due to the *page distance* analysis with the timing side channel [47, 33], we know that the page advanced by the unique *page distance* falls into the same bank. Therefore, changing the address bits did not influence the outcome of the bank from the DRAM addressing functions. This can only be the case once for each bank. Due to the unique *page distance*, we know that we fall into one of two banks, and therefore, this analysis only leaves bit 18 unknown. Table 6.7 shows the reconstructed physical address bits for each unique *page distance* value, depending on

Table 6.7.: Reconstruction of physical address bits via unique *page distances*.

Page	From							To						
Distance	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}
1	B	1	1	1	1	1	1	$\bar{B}$	0	0	0	0	0	0
3	B	1	1	1	1	1	0	$\bar{B}$	0	0	0	0	0	1
13	B	1	1	1	0	0	1	$\bar{B}$	0	0	0	1	1	0
15	B	1	1	1	0	0	0	$\bar{B}$	0	0	0	1	1	1

bit 18. This 50 % probability acts as a corresponding slow down for the attack.

Via DRAM Addressing Structure. We generalize the previous approach by formulating the general structure of xor-based DRAM addressing functions in the Z3 theorem prover [12]. Schwarz et al. [47] use a similar solver to recover physical address parts from *known* DRAM addressing functions and Kwong et al. [33] use a solver to recover bank information from *contiguous* memory. We in contrast retrieve *contiguous* memory information via bank access patterns without *knowing* the DRAM addressing functions.

In the exploit, we record the bank access pattern when iterating over each page of a virtual address range and determine the corresponding bank affinity via the row-conflict timing side channel [41]. The solver uses the pattern and the underlying structure of the xor-based DRAM addressing functions to implement the following question: *Can this bank access pattern be generated via a xor-based addressing function when walking over contiguous physical memory?* If the constraints are unsatisfiable, the underlying memory range is not contiguous or, the addressing functions are not xor-based. However, if the constraints are satisfiable, the solver found addressing functions and a physical start offset that generate this access pattern. The constraints are detailed in Section 9.3. We restrict the bits of the xor masks to only cover physical address bits 12 to 20, as the page offset controls bits below 12, and bits above 20 are not needed to find adjacent rows.

Evaluation. The memory scan takes less than 10 seconds with 2 MB pages and less than 3 minutes using the *page distance*, *i.e.*, 19.05 MB/s (n=10, $\sigma_{\bar{x}}$=0.002) on Chromebook$_1$, 13.03 MB/s (n=10, $\sigma_{\bar{x}}$=0.003) on Chromebook$_2$, 18.39 MB/s (n=10, $\sigma_{\bar{x}}$=0.006) on the OnePlus 5T and 46.55 MB/s (n=10, $\sigma_{\bar{x}}$=0.455) on the T490s.

Initial	0**1**1010**0**0	0**0**1010**0**0	0**1**1010**1**0
	↓	↓	↓
Flipped	0**0**1010**1**0	0**0**1010**1**0	0**0**1010**1**0
	↓	↓	↓
Corrected	0**0**1010**1**0	0**0**1010**0**0	0**1**1010**1**0
	(a) c_1 and c_6 flip	(b) c_1 flips but is corrected	(c) c_6 flips but is corrected

Figure 6.6.: Error correction of 8 memory cells with ECC (from c_7 to c_0). We effectively never see single bit flips.

Finally, we evaluated the correctness of the solver by generating physical address ranges of 512 pages consisting of uniformly generated contiguous memory blocks of up to 128 pages. This memory range is transformed via a DRAM addressing functions into a bank access pattern, where we additionally scrambled the bank index. We vary the pattern length that the solver receives as input and *slide* the solver over the whole memory range and compute the F-score metric. The solver achieves an average F-score of 0.97 with a bank access pattern length of 64 samples and an average scanning speed of 1.079 MB/s. Further details on performance and correctness of the functions are provided in Section 9.3.

6.2. C2: Alternative to Memory Templating

Due to semiconductor production variances, some cells are more susceptible to Rowhammer than others [54]. Because of that, most Rowhammer bit flips are reproducible, and their direction ($0 \rightarrow 1$ or $1 \rightarrow 0$) is fixed [30] as well. The affected systems (cf. Table 6.1) use LPDDR4x DRAM with ECC with a typical single-error-correction code[7]. We empirically verified this with our observation that we see no single, but only double bit flips. The reason is that the ECC memory requires at least two bit flips to show an effect within a code word. Otherwise, the bit flip is corrected and not exploitable. Figure 6.6 visualizes this effect with 8 data bits (parity bits are not shown). Therefore, we propose two techniques for the Half-Double Attack to work around this data dependency, an improved templating technique for ECC memory, and *Blind-Hammering*, which does not require any bit flip templating.

[7]We have not seen any freezes due to error detection, indicating that it is only single-error correction with no support for double-error detection.

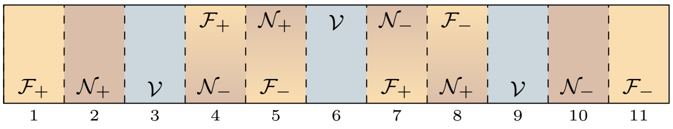


Figure 6.7.: The *zebra* pattern used for *Blind-Hammering*. Notice how the *near aggressors* and *far aggressors* change depending on the *victim* row.

ECC-aware Templating. Classic memory templating fills the aggressor rows with a specific byte value and fills the victim row with the inverse of this value. During our experiments, we find many weak cells when filling the aggressor rows with `0x55` and the victim row with `0xaa`. However, when moving to the next stage of the exploit where the victim row is filled with PTEs, we initially no longer observed any flips. The reason for this is the one outlined before and illustrated in Figure 6.6: Bit flips on ECC memory depend on the data that is stored in the cells [33]. Therefore, we need to adapt the templating phase to incorporate a presumed structure of the data we target, *i.e.*, fake PTEs when targeting page-frame numbers. Hence, during templating, we fill the victim rows with fake PTEs where the page-frame number is filled as in the regular templating approach, e.g., we fill the aggressor rows with `0x55555555555555` and victim rows with `0x68000AAAAAAFD3`. We keep track of all addresses that produce flips at the offset of the page-frame number field during the evaluation. Afterwards, we use these addresses to induce bit flips in a target page placed in the victim row.

Blind-Hammering. The disadvantage of ECC-templating is that it requires precise knowledge of the target data. *Blind-Hammering* generalizes our attack further and makes no assumptions about the target data in the victim row. Instead, it circumvents the ECC data dependency problem by not depending on the repeatability of the bit flips over changing victim data. *Blind-Hammering* skips the templating phase and hammers as many rows as possible with real page tables in potential victim rows. *Blind-Hammering* creates a *zebra* pattern (cf. Figure 6.7) by mapping contiguous memory (cf. Section 6.1) and then unmapping parts of the allocated memory to make room for the victim rows shown in blue. In essence, it performs the templating directly on the inaccessible victim rows and using the victim's own data for the attack. Consequently, the trade-off for *Blind-Hammering* is similar as for the templating phase. While *Blind-Hammering* enables targeting ECC memory, it has a clear drawback: The attacker cannot simply read the data anymore to check whether a bit has flipped. We elaborate this problem further in Section 6.3, motivating challenge **C4** that we then solve in Section 6.4, *i.e.*, the need to verify that a bit has flipped in an exploitable way without crashing the attacker process.

Evaluation. We evaluate *Blind-Hammering* on Chromebook$_2$ and observed 30 exploitable bit flips (13 flips $0 \rightarrow 1$, 17 flips $1 \rightarrow 0$) within 11.6 h. This gives us an average of 2.59 exploitable flips per hour, or 23.2 minutes on average to produce an exploitable flip. We used the ratio of overall bit flips to exploitable bit flips of the Chromebook$_2$, to estimate the exploitation time on our other devices. On the OnePlus 5T it takes approximately 6.4 h, on the HTC U11 4.0 h and on the Chromebook$_1$ 4.2 h, to flip an exploitable bit in a PTE.

6.3. C3: Memory Preparation (Spraying)

In this section, we fill the memory of the target systems with our attack targets, the PTEs. Modern ARM-based platforms, *i.e.*, mobile platforms, can reduce the levels of page tables from the default of 4 page-table levels to only 3 page-table levels to optimize the performance of page walks (on TLB misses). However, this also decimates the available virtual address space for every process by a factor of 512. Since the affected devices (cf. Table 6.1) use this approach, we only have a virtual address space of 512 GB available. While this is still much more than the amount of physical memory the device has, it severely limits the practicality of page-table spraying using file mappings. Previous work has used file mappings and other memory mappings to fill memory with page tables [48, 43, 51, 18]. However, with only 512 GB of virtual address space available, we can only create mappings requiring less than 262 144 page tables, *i.e.*, taking up 1 GB of memory. Thus, we can only occupy, e.g., 25 % of the available 4 GB on the Chromebooks. This increases the attack duration by a factor of more than 8 (since not all pages can be mapped).

To bypass this aggravating effect, we propose a new technique called *Child Spray*. Instead of spraying only our own virtual memory with mappings to allocate page tables, *Child Spray* spawns child processes that share memory with the parent process. The shared memory is only once in the physical memory, but each process has its own page-table hierarchy, effectively spraying the physical memory with page tables. The only disadvantage of

Child Spray is that a hammered PTE can point to a page table of a child process, leading to extra engineering steps for successful exploitation.

With this spraying approach and *Blind-Hammering*, bit flips can now occur in any page table any time, as we don't know which cells are vulnerable and where page tables are. We can check page tables periodically for changes by checking whether the shared memory mapping still has its expected content. However, as *Blind-Hammering* bit flips in page tables are, in contrast to templated bit flips, not predictable, PTEs often become invalid. Consequently, we need a method to test whether a bit flip in a page table occurred without crashing the attacker process. We solve this challenge in the following.

Evaluation. The *Child Spray* runs with two child processes at 79.39 MB/s (n=10, $\sigma_{\bar{x}}$=0.24) on Chromebook$_1$, 54.09 MB/s (n=10, $\sigma_{\bar{x}}$=1.437) on Chromebook$_2$, 25.42 MB/s (n=10, $\sigma_{\bar{x}}$=0.346) on the OnePlus 5T, and 99.88 MB/s (n=10, $\sigma_{\bar{x}}$=0.456) on the Lenovo T490s. Thus, the memory is filled with page tables within less than 1 minute on average. The *Child Spray* on the T490s is not required as the system supports 4 page table levels.

6.4. C4: Robust Bit-Flip Verification

With *Blind-Hammering* we cannot verify the success of a bit flip. Reading the corrupted data (as templating does) is not possible as it is not in our process. Accessing potentially re-mapped memory often crashes the attacker's process due to corrupted PTEs (e.g., mapped to an invalid physical memory region, or setting of a reserved bit), as the OS detects this corruption upon a faulting access. Consequently, we develop a new technique, combining the Half-Double Rowhammer attack with a Spectre [32] side-channel mechanism allowing us to safely determine whether an address can be accessed or whether accessing it would crash the attacker's process. With the *Speculative Oracle*, we can check whether a mapping is corrupted or not without triggering the OS's own detection.

6.4.1. Speculative Oracle

If the physical page-frame number points to an illegal memory location, a read or write to it raises a CPU fault, e.g., a Data Abort [2]. As CPU faults cannot succeed during speculative execution on systems that are

```
if (misprediction)
    access(probe + (pointer[0] & 1) + ... + (pointer[4] & 1));
if (flush_reload(probe) == CACHE_HIT)
    // Report valid address
```

Listing 6.1: Example code for our *Speculative Oracle*. The attacker learns whether `pointer[0]` till `pointer[4]` are accessible memory locations or would raise a CPU fault. If a fault is detected, the attacker probes each address individually.

not susceptible to Meltdown [36], we can use speculative execution to determine whether the bit flip corrupted the entry in a defective way or, otherwise, in an exploitable way. This allows us to avoid accesses that would make the OS terminate our attack process.

Our *Speculative Oracle* uses Spectre similar to Lipp et al. [36] in the Meltdown attack for *exception suppression* on an ARM-based mobile phone. Our Chromebooks use a Mediatek MT8183 SoC with ARM cores that are not vulnerable to Meltdown [3]. Thus, loads depending on the faulting load are not user-visible executed on this ARM microarchitecture.

Our *Speculative Oracle* uses exception suppression by mistraining branch predictors to execute the probing code transiently [36], cf. Listing 6.1. We transiently load *probe* with an offset based on *pointer*, which is the address to test. There are two possible cases for the *Speculative Oracle*: *pointer* is **valid** and, hence, its value forwarded to the *probe* load. Thus, *probe* is loaded into the cache. *pointer* is **invalid** and, hence, the *probe* load is not executed and, thus, not loaded into the cache. Using Flush+Reload [57], we determine whether *probe* is cached or not and, thus, whether *pointer* is valid or not.

As our probing gadget runs in the transient domain, the misspeculated branch may be corrected before the load is issued or the branch may not be mispredicted at all. Thus, *probe* may not be cached even when *pointer* is valid. Hence, we repeat the Spectre attack several times to more likely see a cache hit if *pointer* is valid. However, if *pointer* is invalid, *probe* can never be loaded into the cache and, hence, we infer *pointer* to be invalid with a high probability if we can not observe a cache hit after a certain number of repetitions. Additionally, we can probe multiple addresses at once by chaining them as additional dependencies to the *probe* address as shown in Listing 6.1. This significantly improves the runtime by allowing

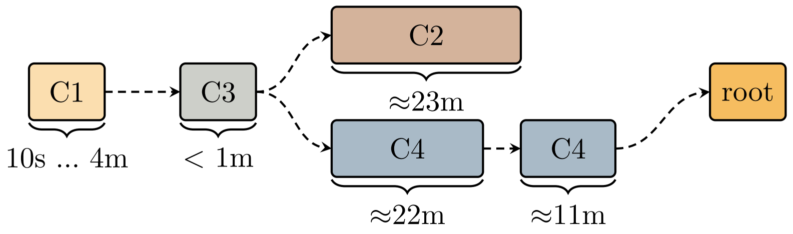


Figure 6.8.: The timing durations for the end-to-end exploit executed on the Chromebook$_2$. For Challenges **C1** and **C4** faster alternatives might be available (cf. Sections 6.1 and 6.4). The overall runtime is bound by Challenge **C2**, *i.e.*, the time it takes to induce an *exploitable* bit flip. Afterwards it takes on average halve **C4** to find the bit flip.

to coarsely scan multiple address candidates at once before probing the candidates separatetely.

Evaluation. We evaluate the success rate and runtime of our *Speculative Oracle*. Since a cache hit on *probe* can only be observed if the address under test is valid, our method has a false negative rate (classifying an invalid address as valid) of zero. Therefore, we can only misclassify a valid address as invalid if we do not observe a cache hit within our repeated trials of triggering the probe gadget.

We evaluated the success rate of the target address classification and the runtime for different numbers of Spectre attacks. We probe 5 addresses at once where either all of them are valid or one random one of them is invalid. We repeat the experiment 10 000 times, 5000 times for valid and 5000 times for invalid addresses. With a single probe try, we already achieve a success rate of 99.01 % with an average runtime of 0.008 ms (n=10 000, $\sigma_{\bar{x}}$=0.002) on the Chromebook$_1$. Chromebook$_2$ achieves 99.68 % success rate with two tries and an average runtime of 0.025 ms (n=10 000, $\sigma_{\bar{x}}$=0.006). On the OnePlus 5T we achieve a success rate of 99.24 % with three tries and a runtime of 0.034 ms (n=10 000, $\sigma_{\bar{x}}$=0.011). We also evaluated this technique on x86-based systems where we used the RSB misprediction for performance reasons. The Lenovo T490s achieves a success rate of 99.94 % with 20 tries and a runtime of 0.018 ms (n=10 000, $\sigma_{\bar{x}}$=0.004).

Hence, since almost all addresses remain valid and without bit flips, the verification with our *Speculative Oracle* consumes 19.0 minutes of CPU time to scan 2 GB of PTEs for bit flips. However, this scan can run on a second core in the background during *Blind-Hammering* (see Figure 6.8).

Table 6.8.: Overview of the challenges, their alternatives and availability across multiple platforms.

	Alternative	Requirement	Available on	Prior Work
C1	Physical Address Access	OS-enabled	Linux-based Systems	[48]
	Huge Pages	OS-enabled	Linux-based Systems	[18, 45]
	Bank Differences	Known Functions	DDR-based Memory	[47, 33]
	Solver	XOR-based Functions	DDR-based Memory	[47, 33]
C2	Templating	no ECC	Systems without ECC	[48, 43, 51, 18, 8, 14, 45]
	Blind-Hammering	Half-Double affected	see Tables 6.1 and 6.3	-
	ECC-Aware Templating	Half-Double affected	see Tables 6.1 and 6.3	[11, 33]
C3	Spray Children	`fork`	ARM64, x86	-
	Spray Page Tables	4 Page Table Levels	ARM64, x86	[48, 43, 51, 18]
C4	`vfork`	OS-enabled	Linux-based Systems	
	Speculative Oracle	Hardware	ARM64, x86	-

Architectural Alternative. As an alternative to the *Speculative Oracle*, we propose an architectural approach, namely using the `vfork` system call. `vfork` creates, similarly to `fork`, an exact copy of the calling process with the only difference that the page tables are not copied. Its primary purpose is to provide a faster version of `fork` for child processes that immediately execute another process via `exec`. Our `vfork` oracle creates a child process that scans 2 GB of PTEs for bit flips. The child process gets killed by the kernel if the page translation is corrupted and returns cleanly otherwise. By checking how the child process died, we know whether the address range is safe to access. If the child process was killed we use shared memory to communicate the last address accessed to the parent process. This minimizes the number of `vfork` invocations down to one per corrupted PTE.

Evaluation. The `vfork` alternative scans 19.45 GB/s (n=10, $\sigma_{\bar{x}}$=0.039) on the Chromebook$_2$, 256.47 GB/s (n=10, $\sigma_{\bar{x}}$=3.971) on the T490s. The bandwidth numbers approach nearly the maximum memory bandwidth of the systems. Hence, with this approach the verification consumes merely 56 s of CPU time on the Chromebook$_2$ and only 4.3 s on the T490s to scan 2 GB of PTEs for bit flips. The disadvantage of this approach is that it is trivial to mitigate by disabling our specific use of `vfork` in the kernel, e.g., as on the OnePlus 5T where the `vfork` instruction is aliased to `fork`. Nevertheless, the speculative oracle is still available.

6.5. End-to-End Attack Evaluation

Figure 6.8 shows the combined runtimes of all attack steps, with less than 5 minutes for contiguency (**C1**) and spraying (**C3**). *Blind-Hammering* (**C2**) takes on average 23.2 minutes on our Chromebook$_2$ to find an exploitable bit flip. Fourth, the *Speculative Oracle* (**C4**) runs in parallel to the *Blind-Hammering* and consumes 22 minutes. After the bit flip, the exploit cleans up, scans physical memory, and sets up page tables for convenient arbitrary read and write, in less than 3 minutes. Thus, the total runtime usually stays below 45 minutes but varies with the time until an exploitable bit flip occurs. On our other devices the total exploitation time is primarily determined by the time it takes to flip an exploitable bit, as the other exploit steps are negligible fast in comparison (cf. Section 6.2). Table 6.8 summarizes the challenges and the requirements for the solutions to be applicable.

7. Discussion

Recently, *TRRespass* [14] has fuzzed hammering patterns and showed that various TRR-protected DDR4 DIMMs are still susceptible to Rowhammer. The generated *many-sided* (3- to 19-sided) patterns worked on 13 out of 42 modules tested. The underlying effect they exploit is an optimization in TRR implementations, where DRAM modules count accesses only to a limited number of rows, which the attacker can exhaust. TRR then loses track of the number of accesses to *near aggressor* (*distance-1*) rows. Compared with the multi-sided Rowhammer patterns from TRRespass [14] and Smash [45], our Half-Double patterns do not rely on the depletion of TRR resources. Instead, the patterns directly incorporate the TRR refresh mechanism into the attack. Therefore, our patterns even work where the TRR mechanism works perfectly for detecting and mitigating *distance-1* Rowhammer.

Applicability to other Systems (including x86). We evaluated all building blocks of the end-to-end attack on other systems as well, in particular x86. Half-Double also exists on TRR-protected DDR4 memory on x86 systems, Some x86 processors, e.g., Xeon, use pTRR to introduce similar accesses to *near aggressors* and, hence, could be used in the Half-Double Attack attack. The contiguous memory detection is also applicable both on other Arm- and x86-based system. *Blind-Hammering* also works on both x86 and Arm. Spraying on x86 systems is easier as no child processes are required and techniques from prior work still apply. We show that the speculative oracle can either be adapted (e.g., using Spectre-RSB instead of Spectre-PHT, or adapting thresholds) to the specific system or be replaced by `vfork` which does not depend on microarchitectural behavior.

Mitigations. To mitigate Half-Double we discuss a short-term defense to protect the next generation of DRAM chips and a more generic Rowhammer defense. First, we propose (p)TRR±2, extending the existing victim-oriented refresh-based mitigations also to include *distance-2* aggressors. This mitigation would minimize the hardware changes as only the *inhibitor* of a TRR-based design needs to be adapted to refresh additional rows. Furthermore, as the results from Table 6.2 suggest, a more sophisticated design would refresh *distance-2* rows with a lower frequency than *distance-1* rows. Second, as DRAM cell density will further increase, we assume that the influence of the Half-Double effect will rise, increasing the need for a more generic Rowhammer protection. Saileshwar et al. [46] proposed to replace victim-oriented defenses like TRR with an attacker-oriented defense. They propose to swap attacker rows after a certain activation count is reached with another row within the same bank using a permutation layer. This mechanism statistically breaks the locality between aggressor and victim rows and makes it therefore highly unlikely to continuously hammer the same victim.

To harden affected systems against our end-to-end exploit, we propose to tackle the contiguous memory allocation, and the bit flip verification (cf. Sections 6.1 and 6.4). If the underlying system allocator ensures that the allocation never returns continuous pages, the attacker has to resort to a brute force approach to find the correct *far aggressors* to induce Half-Double bit flips. Furthermore, the `vfork` system call can be aliased to `fork` removing this gadget from the system.

8. Conclusion

We presented a new and unmitigated Rowhammer effect, Half-Double. Half-Double induces errors in a victim by combining a large number of accesses to a “far” aggressor (at distance two) with just a handful (dozens) to a “near” aggressor (at distance one). This is problematic on DRAM with mitigative refreshing as a Rowhammer protection (e.g., “TRR”), as protections implicitly access the near aggressors and, thus, instead of preventing Rowhammer assist Half-Double in inducing bit flips. We evaluate Half-Double thoroughly and demonstrate its practical relevance in an end-to-end Rowhammer attack. To overcome the challenges for an end-to-end attack on recent off-the-shelf devices, we used side-channel attacks, a novel technique called *Blind-Hammering*, a novel page table spraying technique, and a Spectre-based crash-resistant bit-flip verification. Our end-to-end proof-of-concept attack, the Half-Double Attack, gives an attacker arbitrary read and write access to the entire memory on fully up-to-date systems, as we showcase on Chromebooks with ECC- and TRR-protected LPDDR4x memory, in only 45 minutes average runtime.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, Shaanan Cohney, for their guidance, comments and suggestions. Part of the funding was provided by a generous gift from Amazon. Any opinions, findings, 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] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks. In: HOST. 2017 (p. 131).

[2] ARM. ARM Architecture Reference Manual ARMv8. ARM, 2013 (p. 154).

[3] ARM. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. 2018. URL: `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability` (p. 155).

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. In: ACM SIGPLAN Notices (2016) (p. 131).

[5] K.S. Bains, J.B. Halbert, C.P. Mozak, T.Z. Schoenborn, and Z. Greenfield. Row hammer refresh command. US Patent App. 13/539,415. Jan. 2014. URL: `https://google.com/patents/US20140006703` (p. 132).

[6] Alessandro Barenghi, Luca Breveglieri, Niccoló Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical DRAM mappings for rowhammer attacks. In: International Verification and Security Workshop (IVSW). 2018 (p. 130).

[7] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: CHES. 2016 (p. 131).

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P. 2016 (pp. 127, 131, 157).

[9] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In: USENIX Security Symposium. 2017 (p. 131).

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. ePrint 2015/1034. 2015 (p. 131).

[11] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In: S&P. 2019 (pp. 138, 157).

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. 2008, pp. 337–340 (pp. 150, 169).

[13] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (pp. 131, 149).

[14] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In: S&P. 2020 (pp. 127, 128, 131, 133, 135, 139, 147, 157, 158).

[15] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. ARMOR: A Run-time Memory Hot-Row Detector. 2015. URL: `http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer` (p. 131).

[16] Mel Gorman. Understanding the Linux Virtual Memory Manager. Prentice Hall Upper Saddle River, 2004 (p. 149).

[17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (p. 131).

[18] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 127, 128, 130, 131, 153, 157).

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 131).

[20] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters. In: Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE. 2020 (pp. 130, 140).

[21] 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 (p. 131).

[22] Rei-Fu Huang, Hao-Yu Yang, Mango C.-T. Chao, and Shih-Chin Lin. Alternate hammering test for application-specific DRAMs and an industrial case study. In: Annual Design Automation Conference (DAC). 2012 (p. 131).

[23] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. ePrint 2016/1196. 2017 (p. 131).

[24] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security Symposium. 2019 (p. 131).

[25] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (p. 131).

[26] JEDEC Solid State Technology Association. Low Power Double Data Rate 4. 2017. URL: http://www.jedec.org/standards-documents/docs/jesd209-4b (p. 131).

[27] Jedec Solid State Technology Association. Low Power Double Data Rate 3. 2013. URL: http://www.jedec.org/standards-documents/docs/jesd209-4a (p. 129).

[28] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse engineering of DRAMs: Row hammer with crosshair. In: International Symposium on Memory Systems. 2016 (p. 130).

[29] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in DRAM memories. In: IEEE Computer Architecture Letters 14 (2015) (p. 131).

[30] 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 (pp. 126–128, 130, 131, 138, 139, 151).

[31] 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 (pp. 130, 136, 138).

[32] 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 (pp. 137, 154).

[33] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (pp. 149, 150, 152, 157).

[34] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In: ISCA. 2019 (pp. 127, 131).

[35] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. In: arXiv:1711.08002 (2017) (p. 131).

[36] 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 (p. 155).

[37] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-aware intelligent DRAM refresh. In: ACM SIGARCH Computer Architecture News 40.3 (2012), pp. 1–12 (pp. 129, 134).

[38] Onur Mutlu. The RowHammer problem and other issues we may face as memory becomes denser. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). 2017 (p. 130).

[39] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In: MICRO. 2020 (pp. 127, 131).

[40] Matthias Payer. HexPADS: a platform to detect “stealth” attacks. In: ESSoS. 2016 (p. 131).

[41] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 130, 138, 150, 168).

[42] Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu, and Mattias Nissler. Introducing Half-Double: New hammering technique for DRAM Rowhammer bug. 2021. URL: `https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html` (p. 129).

[43] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In: International Symposium on Hardware Oriented Security and Trust. 2016 (pp. 128, 131, 153, 157).

[44] 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 (pp. 127, 131).

[45] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In: USENIX Security Symposium. 2021 (pp. 127, 131, 133, 135, 157, 158).

[46] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows. In: ASPLOS. 2022, pp. 1056–1069 (p. 159).

[47] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 149, 150, 157).

[48] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat Briefings. 2015 (pp. 127, 128, 130, 131, 133, 153, 157).

[49] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In: RAID. 2018 (pp. 130, 131, 138).

[50] 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 (p. 131).

[51] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS. 2016 (pp. 127, 128, 131, 153, 157).

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In: DIMVA. 2018 (p. 138).

[53] Saru Vig, Siew-Kei Lam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Rapid detection of rowhammer attacks using dynamic skewed hash tree. In: Workshop on Hardware and Architectural Support for Security and Privacy. 2018 (p. 131).

[54] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On DRAM Rowhammer and the Physics of Insecurity. In: IEEE Transactions on Electron Devices (2021) (pp. 127, 131, 136, 140, 151).

[55] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. In: arXiv:1912.11523 (2019) (p. 131).

[56] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: USENIX Security Symposium. 2016 (pp. 130, 131).

[57] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (p. 155).

[58] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: RAID. 2016 (p. 131).

[59] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. In: arXiv:1912.03076 (2019) (p. 131).

9. Appendix

Table 6.9.: All evaluated memory parts, including their production date, the underlying memory structure, and information of the test system or operating system we evaluated them on. We indicate parts evidently affected by Half-Double.

	Name	**Year-Week**	**CPU / SoC**	**RAM**	**Size**	**Manufacturer**	**Test System / Operating System**	**Half-Double**
DIMMs	M_1	2019-48	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✗
	M_2	2020-32	-	DDR4	4 GB	Confidential	ZCU104 FPGA Platform	✓
	M_3	2020-42	-	DDR4	8 GB	Confidential	ZCU104 FPGA Platform	✓
Mobile Devices	Chromebook$_1$	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Chromebook$_2$	2020-01	MT8183	LPDDR4x	4 GB	Unknown	Baseboard *Kukui* with Chrome OS Version 90.0.4430.218	✓
	Pixel 3	2018-40	SDM845	LPDDR4x	4 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.9	✓
	HTC U11	2017-18	MSM8998	LPDDR4x	4 GB	Unknown	Android 9 with Kernel Version 4.4	✓
	OnePlus 5T	2017-47	SDM835	LPDDR4x	6 GB	Unknown	Android 11 LineageOS 18.1 with Kernel Version 4.4	✓
	Samsung S9 (SM-G960F/DS)	2018-10	Exynos 9810	LPDDR4x	4 GB	Unknown	Android 10 with Kernel Version 4.9	✗
	Samsung S7 (SM-G935F)	2016-10	Exynos 8890	LPDDR4	4 GB	Unknown	Android 8 with Kernel Version 3.18	✗
PC	Lenovo T490s	2019-13	Intel i5-8265U	DDR4	16 GB	Samsung	Ubuntu 20.04.3 LTS with Kernel Version 5.11	✗
	Minisforum TL50 MiniPC	2021-43	Intel i5-1135G7	LPDDR4	16 GB	SK Hynix	Ubuntu 20.04.3 LTS with Kernel Version 5.13	✗
	Minisforum X35G MiniPC	2020-43	Intel i3-1005G1	LPDDR4	16 GB	Micron	Ubuntu 20.04.1 LTS with Kernel Version 5.4	✗

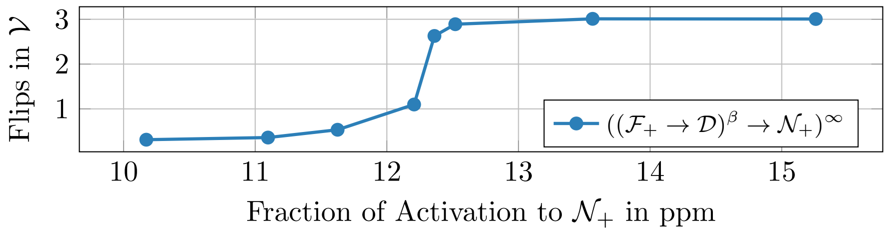


Figure 6.9.: Number of observed bit flips in the *victim* over the fraction of accesses to the *near aggressor* ($\mathcal{N}_+$) in 1 million accesses for the single-sided case.

9.1. Summary of the Evaluated Memory Parts

Table 6.9 provides a comprehensive list of all DDR4, LPDDR4 and LPDDR4x parts we obtained and evaluated in this paper. First, we divide the parts into the DIMMs analyzed in Section 5.3 via the ZCU104 FPGA platform. For these DIMMs we have complete control over DRAM addressing and refresh intervals to evaluate the performance of *distance-1*, *distance-2*, and Half-Double-based hammering (cf. Tables 6.2, 6.4 and 6.5). Second, we evaluate the mobile devices and the PC parts in Section 5.1, where we first reverse engineer the DRAM addressing functions [41] and use the *Quad pattern* for hammering. Table 6.1 shows the resulting bit flips of the affected devices. We observed overall 7 parts that are affected by Half-Double. For the unaffected mobile and PC parts, we can only speculate whether the underlying memory is Half-Double resistant or whether unknown row scrambling prevented mounting the *Quad pattern*. Hence, we cannot conclude that the underlying memory is indeed unaffected from Half-Double.

9.2. Alternative Representation for Bit Flips under Simulated TRR

In this section, we present a different representation for Figure 6.4 and Figure 6.5. Instead of using the β parameter, we plot the number of bit flips observed in the *victim* over the fraction of *near aggressor* in 1 million accesses. Figure 6.9 shows this data for the single-sided case (*i.e.*, the same data as Figure 6.4). Figure 6.10 shows this data for the double-sided case (*i.e.*, the same data as Figure 6.5),

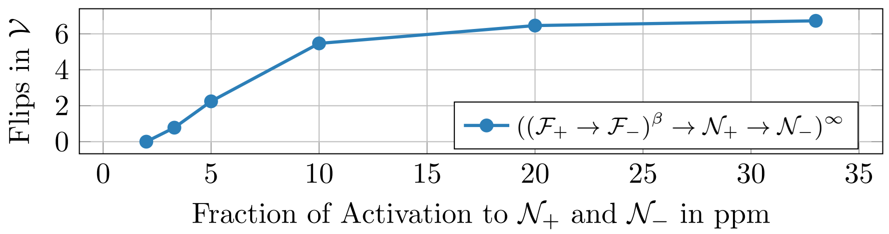


Figure 6.10.: Number of observed bit flips in the *victim* over the fraction of accesses to the *near aggressors* ($\mathcal{N}_+$, $\mathcal{N}_-$) in 1 million accesses for the double-sided case.

9.3. Contiguous Memory Solver

This section details the implementation of the solver and additional performance and correctness analysis based on the reconstructed DRAM addressing functions of the real devices.

Solver Implementation. The solver is implemented with the Z3 theorem prover [12]. To detect continuous memory regions, we first implement the structure of xor-based DRAM addressing functions as constraints. The solver solves for N xor masks we denote as M_i f or $0 \leq i < N$ and the base address of the contiguous physical range B. The general idea is to increment the base B for each of the given input samples, *i.e.*, the pages, as if the range would be contiguous, resulting in unsatisfiable constraints if not. Each of the input samples x_i comes from precisely one set $\mathbb{X}_i$, where x_i is the current sample index. First, we define the function $F_i(x)$ that computes the i-th set bit for the x-th page in the physical memory range:

$$F_i(x) = \bigoplus \left(M_i \wedge (B + x \cdot 0x1000)\right).$$

We denote $\bigoplus(x)$ as operation xor-ing all bits of x and $\wedge$ as the bitwise *and* operation. Second, we define a set index as a binary concatenation of each of the set's bits:

$$S(x) = F_0(x) \parallel \cdots \parallel F_{N-1}(x).$$

For each of the pages contained in one set we enforce that the set index is the same as of the first member of the set, *i.e.*, the first page of the set x_i^0:

$$\text{assert}(S(x_i) = S(x_i^0)) \forall x_i \in \mathbb{X}_i.$$

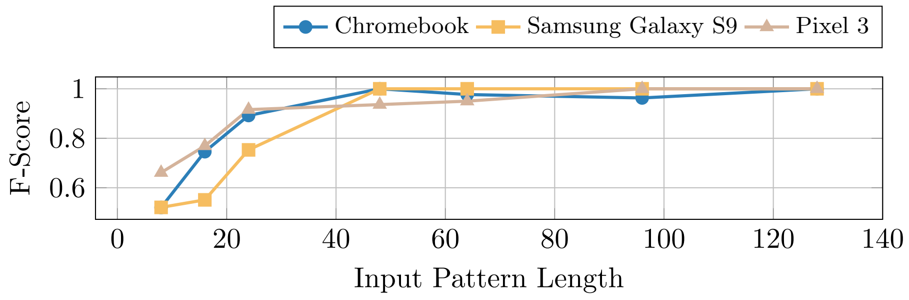


Figure 6.11.: F-Score of our solver-based contiguency detection for different pattern lengths.

Finally, we restrict that all other page offset not contained in one set must have a different set index:

$$\text{assert}(S(y) \neq S(x_i^0))\forall y \notin \mathbb{X}_i.$$

The underlying range can be generated via a xor-based DRAM addressing function if these constraints are satisfied. If unsatisfied, the memory region is not contiguous or the addressing functions are not xor-based.

Evaluation. We verify the correctness of the solver by randomly concatenating contiguous ranges with up to 128 pages and converting these physical pages into bank access patterns with real reverse-engineered DRAM addressing functions. In the evaluation, we *slide* the solver over this generated pattern and vary the number of input samples the solver receives. Figure 6.11 shows the resulting F-score metric for different DRAM addressing functions and various pattern lengths. We observe that the F-score increases with increasing pattern length. This is as expected since the solver internally has more constraints to rely on. We see that with a pattern length of 128 pages, the solver achieves an F-score of > 0.99 for each pattern.

7 Finding and Exploiting CPU Features using MSR Templating

Publication Data

Andreas Kogler, Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz. Finding and Exploiting CPU Features using MSR Templating. In: S&P. 2022

Contributions

Main author.

Finding and Exploiting CPU Features using MSR Templating

Andreas Kogler[1] Daniel Weber[2] Martin Haubenwallner[1] Moritz Lipp[3] Daniel Gruss[1] Michael Schwarz[2]

[1]Graz University of Technology [2]CISPA Helmholtz Center for Information Security [3]Amazon Web Services

Abstract

To ensure backward compatibility while adding new features to CPUs, CPU vendors enable a limited CPU configuration via so-called model-specific registers (MSRs). These MSRs have been introduced for various features, such as debugging, performance monitoring, or security. While many MSRs are documented, there is still a plethora of undocumented or sparsely documented MSRs in modern CPUs. Furthermore, with multiple hundred MSRs, each providing up to 64 configuration bits, it is tedious to find specific configuration options.

In this paper, we show that MSRs and their configuration bits can be detected automatically on Intel and AMD CPUs. We introduce MSRevelio, a framework to automatically detect bits that influence the behavior of instructions and semi-automatically find bits controlled by BIOS settings. We show that previously overlooked bits can harden systems against microarchitectural attacks such as Medusa, CrossTalk, and software-prefetch attacks. Additionally, we show that an undocumented lock bit allows disabling AES-NI at runtime, forcing mbedTLS to fall back to an AES implementation vulnerable to cache attacks. Exploiting this fallback inside an SGX enclave, we fully recover the AES key used by the enclave. With our detection approach, we show that security features retrofitted with microcode updates can be easily detected, even before the public documentation of the underlying vulnerability. In our analysis of the Xen hypervisor, we show that Xen's handling of MSRs was flawed for a long time, allowing guests to access undocumented and unhandled MSRs and fingerprint specific Xen versions. Using automated correlation analysis between documented and undocumented MSRs, we discover a previously undocumented MSR correlating with the CPU's timestamp counter. This

MSR is also accessible from Xen guests, and we demonstrate a Foreshadow attack when all other timers are unavailable or artificially deteriorated. Our results highlight that transparency is crucial for features interacting closely with CPU internals.

1. Introduction

With nearly every new CPU generation, CPU vendors add new features to their CPUs. While some of these features are architectural, such as new instruction-set extensions [19, 30], often features are more related to the microarchitecture, such as mitigation options for transient-execution attacks [37, 32]. Such microarchitectural features can often even be retrofitted to existing CPUs using microcode updates [19]. The non-architectural features typically require some form of interaction with the CPU. Typically, these CPU features are exposed via model-specific registers (MSRs). MSRs are special registers that can be read from and written to by privileged code, *i.e.*, the operating system (OS). Every MSR has a unique 32-bit address and a size of 64 bits. Generally, MSRs are used for interaction with the CPU, such as enabling and disabling CPU features, debugging, and performance monitoring.

While CPU vendors publicly document many MSRs, there are also undocumented MSRs or bits inside documented MSRs that are not documented. These MSRs might only be used internally to debug or reveal information that CPU vendors do not want to disclose [27, 23]. Undocumented MSRs have been shown to undermine CPU security. The AMD K8 CPU provided an MSR that enabled a debug mode [20]. Similarly, Domas [23] found an MSR on the VIA C3 CPU that allows enabling a so-called “god mode”. When enabling this mode, unprivileged applications can execute special CISC instructions that circumvent all privilege checks of the CPU. Some of these undocumented MSRs are mentioned in patents, but there is no clear description of what they do or how they can be used. Moreover, even for documented MSRs, not all bits are fully documented, *i.e.*, reserved bits that have effects.

MSRs can also be used to add security features to CPUs. For example, mitigations for Spectre [48], Foreshadow [73], ZombieLoad [71], RIDL [67], or CrossTalk [64] have been implemented using MSRs [32]. These MSRs control the speculation behavior and provide the possibility to clear several caches and buffers. All of them have been introduced with microcode updates to retrofit mitigations to older CPUs. In the case of OS support, it can query this functionality via the `cpuid` instruction or the IA32_ARCH_CAPABILITIES MSR and then use the features via the corresponding MSRs [32].

In this paper, we introduce MSRevelio[1], a framework that automatically detects available MSRs, regardless of whether they are documented or not. Our approach generates a list of readable and writable MSRs for a specific CPU. We compare the list of detected MSRs with the documented MSRs on Intel and AMD CPUs and classify the MSRs into documented, partly documented, and undocumented. This analysis reveals that all tested CPUs have a large number of undocumented MSRs. On the evaluated AMD CPUs, the number of undocumented MSRs even exceeds the number of documented MSRs. In addition to scanning MSRs, we also automatically analyze the detected undocumented MSRs. We sample the values of both documented and undocumented MSRs. Based on these samples, we automatically correlate undocumented with documented MSRs for a probabilistic classification of undocumented MSRs. Our approach is more robust than the timing-based approach suggested by Domas [23] that assumes similar MSRs expose similar access times.

We also use MSRevelio (Section 3) for a semi-automated analysis of different BIOS versions (Section 4). In this use case, we search for BIOS settings influencing the value of such (partly) undocumented MSRs, and indicating the MSR's purpose. Using MSRevelio, we scan for changes in undocumented MSR bits when modifying BIOS settings and measure potential impacts on instructions. By grouping instructions and collecting performance-counter readings, we identify possible effects of MSR bits on the group's instructions. This approach can also be used to search for all MSR configuration bits that impact a specific instruction. Based on these results, we present six security-relevant case studies.

We demonstrate that MSR bits influencing instruction behavior can mitigate but also introduce new security issues. While these bits could also be found in a manual analysis, MSRevelio alleviates the analysis substantially. We discover an MSR bit converting software-prefetch instructions to no-operations, mitigating software-prefetch attacks on AMD [26, 50]. Additionally, our approach finds a bit to trap `cpuid`, reducing the attack surface for CrossTalk [64]. Moreover, by templating BIOS features, we detect that fast-string support can be disabled at runtime, reducing the

[1]Find MSRevelio's source code at `https://github.com/IAIK/msrevelio`

impact of Medusa [60]. By unsetting the undocumented AES-NI lock bit, we can disable AES-NI at an arbitrary time within the SGX threat model, leading to a time-of-check-to-time-of-use vulnerability forcing mbedTLS [7] to fall back to a vulnerable AES implementation exploitable by a side-channel attack. We show the feasibility of this attack by recovering the full AES key from a single memory-access trace.

We use the found MSRs as a template for tracking the change of MSRs over different CPU microcode versions. Microcode cannot only modify but also add entirely new MSRs. We automatically detect which MSRs have been added in specific microcode versions and whether the MSR was, later on, removed again with a microcode update. Based on our analysis, we can clearly detect which microcode version added mitigations for transient-execution attacks. We even discover CPUs for which such MSRs have been introduced months before the vulnerability was publicly disclosed and the MSR was documented. We cross-check all detected MSRs with official documentations to discover that all added MSRs are related to security features, showing that this approach can leak information about potential embargoed security vulnerabilities.

We also show that the Xen hypervisor just recently prevented the guest OS from accessing undocumented MSRs [16, 62]. Instead of using an allow list for MSRs that should be accessible to a guest, Xen relied on a block list to allow access to all MSRs except a few. With our automated approach, we show how this blocklist evolved over different versions of Xen. This allows fingerprinting of the Xen version even if the hypervisor prevents access to that information or if anti-VM detection methods are applied [47]. In the final case study, we show that the blocklist-based approach allows guests on older Xen versions to potentially access security-relevant MSRs of the host system. Our correlation analysis reveals a previously unknown MSR available to Xen guests that correlates with known timers. Leveraging this MSR yields a high-resolution timer, even if other timers are unavailable, e.g., because the hypervisor uses Fuzzy time [29, 76] or restricts parallel execution, preventing counting threads. To verify that the discovered MSR can act as a timer, we demonstrate a Foreshadow attack [73] using this MSR.

Our results show that MSRs directly impact the system's security. Access to certain MSRs can have negative consequences in the cloud, as they might re-enable attacks that were thought mitigated. Other MSRs, however, can also be used to mitigate attacks for which only costly software workarounds are available, such as prefetch-based attacks [26, 50]. As we found these

MSRs on all systems affected by the corresponding vulnerability, they can be used as a short-term solution until the vulnerability is fixed in hardware.

To summarize, we make the following contributions:

1. We demonstrate an automated approach to detect undocumented MSRs and MSR bits on Intel and AMD CPUs and their impact on instructions and system functionality.
2. We show how our detected MSRs harden systems against microarchitectural attacks but also enable new attacks.
3. We show that the block-list approach used in the Xen hypervisor poses a risk to the system security by allowing guest access to undocumented MSRs, demonstrating a new timing primitive for side-channel attacks.
4. We analyze the evolution of MSRs over microcode versions, showing silent additions of security-related MSRs.

Responsible Disclosure We responsibly disclosed our findings to Intel on August 3rd, 2021. Intel acknowledged our findings.

Outline Section 2 provides background. Section 3 introduces MSRevelio, a framework to automatically find and classify MSRs. Section 4 extends MSRevelio with “BIOS templating” to pinpoint features in MSRs. Section 5 demonstrates security implications of MSRs in six case studies. Section 6 discusses limitations. Section 7 concludes.

2. Background

In this section, we provide background about MSRs, Intel SGX, microcode, and transient-execution attacks.

2.1. Model Specific Register (MSR)

MSRs are special CPU registers, allowing interaction with low-level CPU features and advanced configuration of the CPU’s behavior. Modern x86 CPUs have hundreds of MSRs [35, 61]. However, there is usually only sparse public documentation [23], *i.e.*, many MSRs are not publicly documented, and for many MSRs, the function of specific bits is not mentioned or not precisely defined. MSRs are accessed using the two privileged instructions `rdmsr` and `wrmsr` for reading and writing the 64-bit MSRs. Each register is addressed using a unique 32-bit address. Hypervisors restrict MSRs to prevent the guest systems from taking over control of the host. As MSRs are typically implemented in microcode, they can be removed or added, and their behavior can be updated via CPU microcode updates. For instance, recently, MSRs have been used to add security mitigations against Spectre [48], Foreshadow [73], ZombieLoad [71], and CrossTalk [64] attacks.

2.2. Intel SGX

Intel SGX (Software Guard Extensions) is an instruction set extension providing a trusted-execution environment (TEE) for Intel CPUs. The SGX threat model, similar to other TEEs, assumes that even privileged software such as the OS, administrative users, and peripheral hardware may be compromised and behave maliciously. The trusted code is separated from the untrusted code into a so-called enclave. Enclaves operate within an encrypted and isolated memory region so that even the OS or a physical attacker cannot access the unencrypted memory contents. However, Intel considers vulnerabilities in enclaves the responsibility of the enclave developer, including software side channels [12, 70], and software bugs like race conditions [80, 69]. Enclaves are launched within a regular application and can be interrupted by the OS at any point.

2.3. Micro-op Performance Profiling

With the rising complexity of out-of-order execution CPUs, profiling the performance of actual executed code is non-trivial. To get insight into the resources allocated and events triggered inside a CPU, vendors introduced Performance Monitoring Counters (PMCs). With these counters, a user can monitor the execution of instructions more precisely. However, mapping the observed events to a given instruction is complex, as the CPU splits instructions into smaller micro-ops.

NanoBench [1] is a framework designed to measure the exact PMCs of single instructions. The framework compiles measurement code from a given assembly snippet which allows minimizing the external measurement noise. In addition to the automatic measurement code generations, the framework also handles filling the CPU pipeline with a known state to allow for the same base conditions for all measurements.

2.4. Transient-execution Attacks

With out-of-order and speculative execution, a CPU can lazily handle exceptions and predict the outcome of computations, e.g., the target of an indirect jump, to reduce pipeline stalls. When the CPU has to handle an exception or mispredict a computation, the pipeline's state is rolled back to the instruction causing the exception or misprediction. As rolled back instructions are never committed to the architecture, they are referred to as transiently executed [15, 44].

Spectre [48] and Meltdown [53] showed that attackers can abuse transiently-executed instructions to leak data, *i.e.*, so-called transient-execution attacks [15]. An attacker encodes the results of a transient computation into a microarchitectural element that is not rolled back, e.g., the CPU cache. After discovering transient-execution attacks, multiple such attacks have been published [28, 15, 55, 60, 71, 67, 13, 73, 81, 64, 74]. Microarchitectural data sampling (MDS) attacks [71, 67, 13, 60, 64] are a subclass of transient-execution attacks leaking values from internal components of the CPU, e.g., the line fill buffer. In these attacks, an attacker brings the CPU into a state where transiently executed instructions compute with stale or incorrect values of internal buffers or caches. Leaking the values of these components allows leaking data across all security boundaries.

2.5. Microcode

Modern CPUs frequently receive updates to react to security concerns or bugs. Hence, manufacturers need a mechanism to update the behavior of CPU instructions or components. The microcode is an additional layer of abstraction between the actual hardware and the Instruction Set Architecture (ISA), which allows altering the internal behavior of CPU instruction to a certain extent [49]. Furthermore, some complex instructions require so-called microcode assists, which then execute a sequence of micro-ops read from the microcode [19, 71].

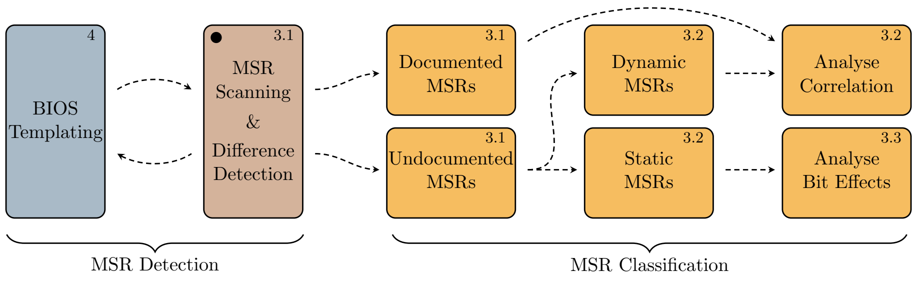


Figure 7.1.: The general structure of MSRevelio's analysis steps.

3. MSRevelio

This section describes MSRevelio, a framework to automatically find, classify, and analyze MSRs. MSRevelio aims to find undocumented features of MSRs that ultimately impact the security of the system. The overview of the steps of MSRevelio is shown in Figure 7.1. First, MSRevelio scans the potential MSR address space (cf. Section 3.1). This scan obtains a list of available MSRs that are automatically classified into read-only, write-only, and read-writable MSRs. This list is filtered based on the official documentation of the CPU, resulting in documented, undocumented, or partly documented, *i.e.*, some bits are undocumented, MSRs. The MSRs that are not or only partly documented are recorded to classify them into two groups. Dynamic MSRs that change over time are correlated with documented MSRs to find similarities (Section 3.2), or even aliased MSRs. For static, unchanging MSRs, MSRevelio analyzes the bits to determine whether toggling the bit impacts the behavior of instructions (Section 3.3). In such a case, we can manually investigate whether such a change is security-relevant. Finally, we extend MSRevelio to find the influence of certain BIOS configurations on MSRs (Section 4) to gain additional information on undocumented MSR bits.

3.1. Detecting Undocumented MSRs

The MSR range of modern CPUs is continuously extended to provide additional functionality or adapt to new security flaws. Due to the large 32-bit address space, most of the addresses are either not used and do not provide any functionality or are reserved for future extensions. However, there are more MSRs available than officially documented. To find these undocumented MSRs, MSRevelio uses both the `rdmsr` and `wrmsr` instruction. Both instructions raise a General Protection Fault (GP fault) if the CPU does not physically back the MSR address. The reason that we use both instructions is that there are 4 different MSR types: read- and writable, read-only, write-only, and non-present MSRs. By combining reads and writes to these MSRs, MSRevelio can detect all types of MSRs over the entire 32-bit address space of the MSRs.

3.1.1. Design

Detecting the presence of an MSR is not influenced by the MSR scope or the current privilege level. The `rdmsr` and `wrmsr` instructions already require OS privileges, and each core has the same set of accessible MSRs [34]. Our approach cannot access certain restricted MSRs that are only readable in SMM mode, e.g., MSR `0x9e`. However, as not even a ring-0 attacker can use them, we do not consider this a huge limitation. As every CPU core exposes the same MSRs, MSRevelio can search the MSR address space in parallel, significantly decreasing the execution time of the profiling. The framework first tries to read an MSR and catches any generated GP faults. The second test is a write to the MSR, again catching generated GP faults. Based on occurred faults, MSRevelio distinguishes between read-only, read- and writable, write-only, and non-existing MSRs. The framework stores the addresses of existing MSRs for later analysis.

To compare the discovered list of available MSRs to the official documented MSRs, MSRevelio additionally implements a PDF parser for the PDF-only documentation. As the structure of the MSR tables in both the Intel and AMD documentation is consistent, we can automatically find and extract the information from these tables. In addition to all documented MSRs, the parser extracts undocumented and reserved bits of documented MSRs. MSRevelio compares the discovered MSRs with the parsed documentation to determine undocumented or only partly documented MSRs.

3.1.2. Implementation

MSRevelio is implemented as a Linux kernel module with an additional user-space library. It uses the `rdmsrl_safe` and `wrmsrl_safe` kernel functions to catch GP faults when reading and writing MSRs. MSRevelio tries not to alter the content of the MSRs by writing the value that was read before. Only for write-only MSRs, this is not possible and MSRevelio conservatively tries to write ‘0’ to such MSRs. For most of the writeable MSRs, the documentation [36, 61] states that when writing to the MSR, undocumented bits must be ‘0’ to prevent a GP fault. This behavior is also necessary, as MSRs can only be overwritten with a 64-bit value and not bitwise. A complete scan when using multiple cores on our *AMD Ryzen Threadripper 1920x* with 5244 accessible MSRs takes 5.74 min ($\sigma_{\bar{x}} = 0.0005$, $n = 10$).

3.2. Classifying MSRs

The classification functionality of MSRevelio is divided into two parts, namely the detection of static and dynamic MSRs and the further analysis of undocumented dynamic MSRs based on the correlation with other documented dynamic MSRs. Note that for this analysis, we can only use MSRs that are readable. We define *static MSR* as an MSR with a fixed value that only changes if the MSR is actively written to, e.g., MSRs containing configuration bits. A *dynamic MSR* is an MSR that is continuously updated by the CPU, e.g., counters or sensor values. To distinguish static from dynamic MSRs, MSRevelio samples the values of the respective MSRs for a certain amount of time to detect if the value changes at some point. While an MSR classified as a dynamic MSR is always a dynamic MSR, MSRevelio might classify some dynamic MSRs as static MSRs if the value updates only with a very low frequency. However, for analyzing the impact on instruction behavior (cf. Section 3.3), we are only interested in static MSRs as well as all write-only MSRs, as they are static in its nature, *i.e.*, they do not change their value. Hence, as static MSRs are always classified as static MSRs, this only results in some additionally tested MSRs, but no missed MSRs.

We further classify the found undocumented dynamic MSRs by cross-correlating them with all documented dynamic MSRs. For the correlation, we continuously sample all, *i.e.*, documented and undocumented, dynamic MSRs for 10 s while executing a CPU stress test in parallel. The stress test triggers spikes in electricity and temperature sensor readings and triggers changes in the power states. As a result, undocumented MSRs exposing such states are easier to correlate with existing documented MSRs, as they contain more features for the correlation. Each resulting sample set of every undocumented dynamic MSR is then correlated with every documented dynamic MSR using the Spearman and the Pearson coefficient. For every un-

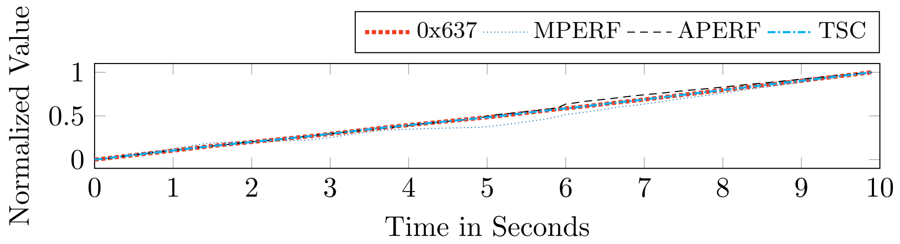


Figure 7.2.: Normalized values of the undocumented MSR(`0x637`) and the highly-correlating MSRs MSR(`0xe7`) (MPERF), MSR(`0xe8`) (APERF), and MSR(`0x10`) (TSC) (all monotonic counters) over 10 s.

documented MSR, MSRevelio generates a list of documented MSRs, sorted by the correlation coefficient, and a plot of all the sampled values. Figure 7.2 illustrates such a result, showing the undocumented MSR(`0x637`) and the documented MSR(`0xe7`) (IA32_MPERF), MSR(`0xe8`) (IA32_APERF) and MSR(`0x10`) (IA32_TIME_STAMP_COUNTER). While there are more computationally expensive methods to compare time series [9, 65], we do not require such complex algorithms, as the recorded data points are aligned. Hence, a simple correlation analysis is sufficient. The correlation analysis results in a list of similar MSRs, allowing to judge the likely information source of the MSR, e.g., whether it contains thermal readings or a counter. In Section 3.4, we show that this approach can find undocumented MSRs and detect the type of values exposed by the found MSR.

3.3. Impact on Instruction Behavior

To further analyze static MSRs, MSRevelio analyzes the impact of MSR bits on instructions. The goal of this scan is to find undocumented or reserved bits that influence the behavior of instructions. The framework performs the scan on the static MSRs (cf. Section 3.2), as fluctuating MSR bits are usually not used as feature-control bits.

3.3.1. Design

To automatically detect changes in instruction behavior, MSRevelio uses performance counters for templating. To ensure that MSRevelio not only finds effects triggered by single bit flips but instead also finds effects that

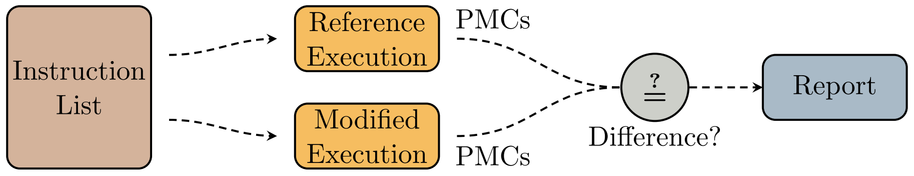


Figure 7.3.: MSRevelio changes bits in the undocumented MSR and measures various instruction groups' performance counters to see if the bit influenced the instructions.

result from enum MSR fields, *i.e.*, groups of bits inside an MSR that belong to the same configuration option, we rely on optimized *flipping masks*. The masks tests all possible values of all possible enum fields with size $\leq W$ within an MSR by only performing 2^W writes to the MSR. For each observed effect the MSR is further analysed to find the exact enum field. Figure 7.3 shows the general concept of MSRevelio's bit scan. In the first step, a ground truth is recorded: MSRevelio executes a set of instructions on the CPU and records instruction-related performance counters. If for a set of instructions there is a change in the performance-counter values, it is an indication that one of the altered MSR bits affect the instruction. An alternative design could iterate over all possible bit values of an MSR instead of only considering the bounded enum fields. We did not choose this approach for two reasons.

First, such a design would require significantly more measurements to be performed. For example, on the tested Intel Core i5-4570, we found 3612 writable bits spread across 177 undocumented MSRs. In this case, our approach (with $W = 4$) only needs to test 2832 MSR values and additionally 496 tests per observed side effect to further find the root cause of the effect, whereas a full search would require $16\,853 \times 10^{12}$ tests (the calculations are shown in Appendix 8.1). Testing the entire search space for the writable reserved bits of the documented MSRs on this processor would require only $16\,712 \times 10^6$ tests. One could argue that for the latter amount of tests, it is feasible to parallelize the tests using a cloud provider's resources. However, doing so is costly. At the time of writing, the cheapest on-demand CPU that AWS offers for its data center in Frankfurt costs $\$\,0.0047$/hour [4]. As our current implementation takes around 3 seconds per test, we can run 1200 tests for $\$\,0.0047$ and hence the complete search space enumeration for the reserved bits would cost $\frac{16712 \cdot 10^6}{1200} \cdot \$\,0.0047 = \$\,65\,455$. Note that these costs are only for testing a single CPU and abstract away the challenge to find a cloud provider offering the target CPU.

Second, it is reasonable to assume that flipping additional bits does not hide the effect of other bits in most cases. It is not beneficial to have one bit for enabling a feature and another for disabling it again. Instead, a practical implementation would use a single bit to toggle the activation of a feature, as the structure of most documented MSRs shows [36, 61]. An exception for this are enum MSR fields consisting of multiple bits to encode more than two values. Thus, we choose *flipping masks* to find enum MSR fields of up to W bits. This approach is not a perfect fit for every possible scenario, e.g., if an enum field is spread over more than W bits or contains unrelated bits in the middle. However, our approach acts as a trade-off between the scenarios we cover and the search space.

3.3.2. Implementation

To execute the instruction and record the performance counters, MSRevelio uses the nanoBench framework [1]. The nanoBench framework allows recording performance-counter events of a given assembly code snippet. The framework takes care of compiling the snippet and repeating it multiple times without introducing additional overhead. We test 124 common instructions divided into 16 groups based on their semantics. We show the groups and link the used performance-counter config in Appendix 8.2. MSRevelio calls the nanoBench framework for each group separately. Hence, MSRevelio knows precisely which category of instructions the bit flip inside the MSR influenced. To cope with the large search space of undocumented MSR bits combined with the nanoBench framework invocation, MSRevelio pre-filters the undocumented bits in three phases based on their behavior.

Phase 1: Detecting Writable Bits. In the first phase, MSRevelio iterates over all the undocumented MSRs in parallel and records the bits inside the MSRs that can be toggled. This is done by reading the original MSR value once and then toggling one bit at a time and detecting if the `wrmsr` instruction executes successfully. While reserved or unimplemented bits typically raise a GP fault, some bits silently ignore the write. Hence, MSRevelio rereads the MSR value and checks if the bit was toggled. If we can successfully write to an MSR but reading from the MSR always faults, we consider the MSR as write-only and the bit as modifiable. Since setting

Table 7.1.: MSRevelio's results for different microarchitectures, including the number of found and undocumented MSRs and the categorization of the undocumented MSRs. The number of similar MSRs indicates if an undocumented dynamic MSR can be correlated with a likeliness of more than 85% to a documented dynamic MSR.

CPU	μ-Arch	μ-Code	# Found (RW, RO, WO)	# Undocumented (RW, RO, WO)	# Static (RW, RO)	# Dynamic (RW, RO)	# Similar
AMD Ryzen Threadripper 1920X	Zen	0x8001137	5244 (5223, 17, 4)	4876 (4873, 2, 1)	4873 (4871, 2)	2 (2, 0)	0
Intel i7-6700k	Skylake	0x9e	477 (363, 108, 5)	105 (68, 35, 2)	99 (68, 31)	4 (0, 4)	2
Intel i7-8700k	Coffee Lake	0xb4	517 (388, 122, 7)	126 (89, 35, 2)	121 (89, 32)	3 (0, 3)	3
Intel i9-9900k	Coffee Lake	0xde	537 (413, 117, 7)	136 (99, 35, 2)	132 (99, 33)	2 (0, 2)	2
Intel Xeon Silver 4208	Cascade Lake	0x5003102	1109 (957, 142, 10)	647 (591, 52, 4)	601 (553, 48)	42 (38, 4)	42

arbitrary undocumented bits inside an MSRs can lead to various system freezes and undefined CPU states, the framework relies on a blocklist of such MSRs. This phase is also used to filter out bits that cause system freezes to enhance later execution time. Note that this process can also be fully automated using a remotely-controllable power switch [23].

Phase 2: Initial Recording of the Changed State. In the second phase, MSRevelio starts recording possible effects on the instruction groups. Our implementation tests all possible combinations of enum MSR fields consisting of up to 4 bits. We assume that multiple undocumented enum MSR fields are independent with respect to effects on instructions. With this assumption, this phase does not test each individual enum MSR field, but instead alters the undocumented MSR by toggling different enum fields at the same time. In fact, it is possible to test all combinations of 4 consecutive bits at an arbitrary position inside the MSR using only $2^4 = 16$ different MSR values. We use *flipping masks* containing a 1 in case a bit has to be flipped at that respective position in the MSR. We create the 16 masks by flipping the bits at position n after having generated $2^{n \ mod \ 4}$ masks, *i.e.*, we flip bit 0 after every generated mask, bit 1 every second, bit 2 after every fourth, and bit 3 after every eight masks, e.g., the second mask has the following representation: `0x1111111111111111`.

Due to this construction, given an arbitrary position of 4 consecutive bits, the 16 bitmasks represent all 16 values of these 4 consecutive bits (see Appendix 8.3). This optimization reduces the search time by a factor of $\frac{16+60 \cdot 8}{16} = 31$ compared with an optimized sliding window, *i.e.*, testing all the enum values at a given position and then shifting the window by one while reusing previous results. The results of this phase are candidate MSRs where at least one of the changed bits affects the instructions. These MSRs are the basis for the third phase.

Phase 3: Finding the Origin of the Effects. After the second phase, MSRevelio has a list of MSR candidates that have observable effects on the instructions groups. In this phase, MSRevelio iterates over each of these candidates and sequentially tests all enum MSR fields to pin down the effects observed in the performance counters to a specific enum field and its value. This phase's results contain detailed information about the MSRs and which bits influence a certain instruction group. We further analyze the automated results of this scan in Section 3.4 and in six case studies in Section 5.

3.4. Results

We conducted an exhaustive search for undocumented MSRs on a total of 5 CPUs. The overall results of these CPUs are shown in Table 7.1. We found 5890 undocumented MSRs on AMD and Intel CPUs, with most of the discovered MSRs (4876) on AMD CPUs. However, 96.8 % of the found static, read-and-writeable, undocumented MSRs on AMD do not raise a GP-fault when written but ignore the written value, restricting further bit behavior analysis. We also observe similar behavior for 54.1 % of the Intel MSRs. We analyzed all these undocumented MSRs for correlations with documented MSRs. We found 53 undocumented MSRs that expose continuously changing values correlating with existing MSRs. For example, the dynamic MSR(`0x637`) exposes a monotonic counter correlating with documented counters, and we further explore this counter in Section 5.6. For the static MSRs, we conducted the enum field search to find bits that influence specific instructions. We found 1 undocumented and 6 partially documented bits that affect instructions such as `cpuid` and `prefetch`. The effects of these MSR bits are analyzed in Section 5. To determine the specific functionality of the MSR, manual analysis is necessary.

4. Detecting OS-Configurable BIOS Features

In this section, we extend the MSR scanner of MSRevelio to detect differences in static MSRs that are caused by changes in BIOS settings. In line with Intel's documentation [41], BIOS refers to firmware, regardless if it is an actual BIOS or UEFI. The BIOS is responsible for configuring multiple CPU settings on boot, e.g., available features [19] and settings related to the power management. While some settings can only be changed by the BIOS at boot time, other features can also be modified by the OS. Many BIOS versions, e.g., on consumer systems, expose only a small subset of settings to the user. With this approach MSRevelio can template BIOS versions to hint the user on how to reenable *unlocked* features from the OS. Additionally, it can be used to analyze whether undocumented or poorly-described BIOS features impact MSRs.

4.1. BIOS Templating

Our approach produces a list of MSRs and bits inside these MSRs configured by the BIOS, including documented and undocumented bits. For this purpose, we template BIOS features that modify MSRs by changing a BIOS setting manually and automatically scanning all MSRs' values (cf. Section 3.1). As we target features, we are only interested in static MSRs. After toggling a BIOS setting, we compare the values of all static MSRs to the values of the initial scan of all MSRs (cf. Section 3.2). If an MSR has a different value, we assume that the BIOS setting led to the change of this MSR. We can increase the certainty that this MSR value depends on the BIOS setting by repeating the process multiple times.

To further focus on undocumented settings, MSRevelio uses the list of documented MSRs and their bits to check whether the changed bits are documented (cf. Section 3.1). If either an undocumented MSR or an undocumented bit change, MSRevelio reports this as an undocumented feature. This undocumented feature is analyzed for impacts on the instruction behavior in the same way as MSRs obtained from a full MSR scan (cf. Section 3.3). Again, to reduce the likelihood of uncorrelated changes, repeating the process multiple times increases the probability that the reported MSR bit is indeed related to the BIOS feature.

4.2. Setup

We evaluate 5 systems with BIOSes exposing a rich set of features. The tested CPUs are a Celeron J4005, Core i7-6700K, Core i7-8565U, and Core i7-10510U, with an Intel JYGKLCPX.86A.0053.2019.1015.1510, AMI 2.17.1246, HP R93 01.01.06, and an AMI 2.21.1277 BIOS, respectively. These BIOSes include options that are not documented in the BIOS manual and for which we did not even find any unofficial documentation, e.g., “Strong Weak Leaker” or “K1 off”. We initially set all the BIOS values to their defaults and use MSRevelio to get the difference in the MSR availability and the MSR values after changing specific settings.

4.3. Results

Our scans reveal several BIOS options that directly affect MSRs. While several MSRs are locked after the BIOS initialization, some of them can be modified by the OS to emulate these BIOS settings. We also discovered BIOS settings that affect undocumented MSRs or MSR bits.

4.3.1. Documented Settings

Some of the changed MSRs are documented and simply expose the read-only status of the BIOS setting. Most settings, including VTx/VTd, turbo boost, fast-string support, or execute disable, are reflected in the MSR(`0x1a0`), documented as IA32_FEATURE_CONTROL. While the BIOS locks the configuration bits for VTx/VTd, the fast-string support and turbo boost can be changed by the OS. The execute-disable feature can theoretically also be modified by the OS. However, this only led to OS crashes on our machines. To point out the impact of the unlocked bits, we show in Section 5.4 that the OS can harden a system against Medusa [60] using the feature bit for fast-string support.

Another documented setting is the enabling and disabling of hardware prefetch features. For this setting, the BIOS simply modifies MSR(`0x1a4`) (MSR_MISC_FEATURE_CONTROL), which is also writable by the OS [77]. However, our tested BIOS versions only disabled the L2 prefetcher and not the L1 prefetcher when setting “Hardware Prefetcher” to “disable”. While we consider this at least misleading, if not a bug, we do not see any security problem in that behavior.

4.3.2. Unofficial or Undocumented Settings

Our approach detected MSRs that are either entirely undocumented, not officially documented, or not documented for the microarchitecture on which we found the MSR. Such MSRs include MSR(`0x621`), MSR(`0x35`) (MSR_CORE_THREAD_COUNT), MSR(`0x7a`), and MSR(`0xe2`) (MSR_-PKG_CST_CONFIG_CONTROL). MSR(`0x35`) provides information on the state of hyperthreading on Intel Xeon CPUs. However, while it is not documented for Intel Core CPUs, it also works on our Intel Core machines. MSR(`0x621`) is not publicly documented, but mentioned as MSR_UNCORE_PERF_STATUS in a book by Gough et al. [25] for Xeon E3/E5 CPUs without further details. Based on the description of the BIOS setting that modified bit 0 of this MSR, we learn that it is the state of Intel SpeedStep. We discovered another potential bug in one of our BIOS versions concerning bit 0 of MSR(`0x7a`). The BIOS provides an option "MachineCheck" which toggles exactly this bit. While we did not find any official documentation, the CoreBoot source [17] and a Linux kernel patch [66] suggest that this bit enables Intel SGX, which is also supported on our machine. A "Timed MWAIT" feature in our BIOS enables bit 31 in MSR(`0xe2`), which is officially reserved. We assume that this enables an `mwait` extension to continue execution after a specified number of CPU cycles have elapsed, similar to AMD's `mwaitx` instruction [5]. We leave it as future work to reverse engineer how this feature can be leveraged and its impact. When enabling and disabling AES-NI via the BIOS, it changes bit 1 in MSR(`0x13c`). For this MSR (MSR_FEATURE_CONFIG), the documentation states that 2 bits are used to represent the AES-NI state, without providing a detailed description. On all analyzed machines, bit 0 was always set to '1'. To showcase the security impact of this finding, Section 5.1 shows that bit 0 is actually a lock bit that can be exploited to attack SGX enclaves by disabling AES-NI at runtime.

5. Case Studies

This section presents six case studies demonstrating the security impact of previously overlooked MSR bits. We show that undocumented MSRs can prevent existing attacks and re-enable mitigated attacks in certain scenarios. We also detect security-relevant MSRs in microcode distributed before the vulnerability is disclosed. Finally, we show that specific hypervisors version expose distinguishable MSR fingerprints and provide access to security-relevant undocumented MSRs.

5.1. Exploiting the AES-NI Lock Bit

MSRevelio's BIOS templating (cf. Section 4) revealed that MSR(`0x13c`) (MSR_FEATURE_CONFIG)'s lowest two bits, which enable AES-NI, either contain the value '1' or '3'. The Intel SDM [36] documents that if these bits are ‘3‘, the AES instructions are not available until the next reset, otherwise, they are available. Also, the SDM notes that if the bits are not equal to ‘1‘, the instructions can be misconfigured. However, the individual behavior of these bits is not documented.

We observe that the MSR value cannot be changed via the `wrmsr` instruction from within the OS. This indicates that the BIOS locks the MSR after finishing the initialization. As bit 0 is always set on all tested machines, we assume that this bit is the lock bit of the MSR, set by the BIOS to restrict further changes. We verify this assumption by modifying the BIOS to not set the bit in this MSR, which is in line with the threat model of SGX [19]. With the unlocked MSR, we show that an attacker can modify the feature-detection mechanism of a securely-designed SGX enclave using mbedTLS to fall back to an insecure cryptographic-algorithm implementation, allowing the full extraction of the AES key via Prime+Probe.

5.1.1. Threat Model

Where available, the AES-NI instructions are used in cryptographic libraries to implement the AES algorithm securely and efficiently [30, 7]. These libraries are often combined with Trusted Execution Environments (TEEs) to protect the implementation of cryptographic algorithms and establish secure communication with other parties. Furthermore, the threat model of SGX protects an enclave from a malicious OS and even malicious BIOS firmware [19]. In this threat model, an attacker can modify the BIOS [8].

We consider two distinct attack scenarios. First, we consider a system under complete attacker control. Here, the attacker tries to extract a secret key used by a targeted enclave. In this scenario, the attacker modifies the BIOS only on the attacker's machine to remove the lock bit. Second, we envision a scenario where the MSR is not initialized at all, e.g., because the BIOS developer was not aware of this MSR. While we did not encounter such a BIOS on any of our tested machines, there is a chance that such a BIOS exists due to a large number of BIOS vendors and the wide variety of BIOS versions.

5.1.2. BIOS Modification

To verify that the first bit of MSR(`0x13c`) is the actual lock bit of the AES-NI instructions, we patch the BIOS of our test system. For this case, we use a *MINISFORUM X35G* mini-PC with an *AMI* BIOS and an *Intel Core i3-1005G1* CPU. Dumping and flashing the BIOS is possible via the official AMI Firmware Update tool [6]. Alternatively, an attacker can simply use a low-price SPI flasher such as a CH341A if no software tool is available or if such a tool does not allow flashing a modified image. We extract all 253 binaries of the BIOS image using *UefiTool* [68] and disassemble them using Ghidra with the firmware utilities plugin [45]. In our BIOS, the MSR(`0x13c`) is initialized in the silicon init (SiInitFsp) module. Depending on the BIOS version (cf. Appendix 8.4), the `wrmsr` is either inlined or encapsulated in a wrapper function. In both cases, we simply patch the initialization of the `EAX` register to not set bit 0. The patch for multiple BIOS versions is provided in Appendix 8.4. For the second scenario, we replace the `wrmsr` (or the call to the wrapper) with `NOP` to leave the MSR uninitialized.

5.1.3. Behavior Verification

After booting both images, bit 0 of the MSR reads as '0'. For both BIOS modifications, the `cpuid` instruction still reports that AES-NI is available. Writes with the `wrmsr` instruction to the second bit of MSR(`0x13c`) are reflected by the `rdmsr` instruction, verifying that with a bit 0 cleared, bit 1 is not locked. Furthermore, setting bit 0 using `wrmsr` prevents any subsequent changes to the MSR. Hence, the bit 0 is indeed the lock bit of this MSR. In addition to acting as a lock bit, bit 0 is also the "apply" bit. Changes to the second bit only take effect after the lock bit is set. Therefore, the two BIOS modifications behave the same because the CPU ignores the second bit until the lock bit is set. If both bits are set, AES-NI is disabled, and the instructions raises an illegal instruction exception as expected.

5.1.4. AES-NI inside Intel SGX

We demonstrate the security implications of changing the AES-NI availability at runtime on the mbedTLS library [7]. Due to its small codebase and side-channel resistant AES-NI implementation, it is often used inside

Intel SGX [7]. In mbedTLS, the CPU feature detection is performed over the `cpuid` instruction. Due to the restricted SGX environment, the `cpuid` instruction is not available inside enclaves. Therefore, the SGX-SDK uses an OCALL to retrieve the CPUID information from outside the enclave [39], leading to a potential attack vector manipulating the read CPUID leaf. To enable a robust CPU feature check, enclave developers can rely on executing a potentially not supported instruction and configuring an exception handler to catch the exception [57]. If the instruction executes without raising an exception, the hardware supports the given CPU instruction, otherwise, the exception handler is used to continue execution safely. With this mechanism, the feature detection is encapsulated inside the enclave and does not rely on untrusted data.

Second, a developer might know about the limitations of mbedTLS's fallback algorithm and check the availability of AES-NI with the secure feature-detection in the enclave's initialization phase and abort if AES-NI is not enabled. Furthermore, we assume developers do not account for changing feature bits like the AES-NI-enable bit during runtime, as the possibility of such behavior is not documented.

5.1.5. Proof-of-concept Attack

The attacker enables AES-NI in the BIOS and leaves it enabled during the initialization of the enclave. Hence, any feature check for the availability of AES-NI, be it through an OCALL to `cpuid` or using the trusted CPUID library [57], detects the availability of AES-NI. Even if the presence of AES-NI is enforced through some kind of attestation, the default enabled AES-NI state without the lock bit passes this check. However, an attacker can disable the AES-NI instruction set at any point by simply interrupting the enclave and modifying the MSR. With precise execution control of SGX enclaves, e.g., using SGX-Step [75], an attacker can target a specific instruction after which the AES-NI instructions are disabled. As a result, this leads to a time-of-check-to-time-of-use vulnerability for SGX enclaves that check for AES-NI and later on use it, as is the case for the mbedTLS library (Version 2.26.0). In case AES-NI is not available, mbedTLS falls back to a software-based AES implementation, which is not side-channel resistant [72], as it uses key-dependent memory accesses (cf. Appendix 8.5).

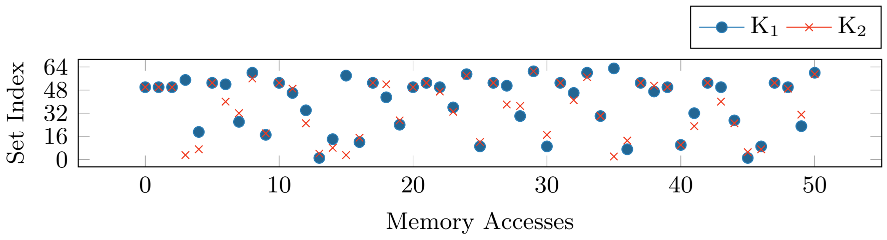


Figure 7.4.: Truncated secret-dependent cache-set accesses of mbedTLS' AES-128 implementation for two different keys.

5.1.6. Prime+Probe on SGX

This case study shows that an attacker can extract secret information via a cache attack from an SGX enclave. Since the SGX environment ensures flushing of the L1 cache during an enclave exit, and the SGX attestation can verify that hyperthreading is disabled, an attack using the L1-cache is unlikely. Therefore, an attacker needs to perform Prime+Probe on the last-level cache (LLC). We assume that an attacker uses precise execution control, e.g., SGX-Step [75], for Prime+Probe on the LLC [59, 12].

We simulate a Prime+Probe attack on mbedTLS, to show that a single trace suffices to exploit the AES-NI misconfiguration and leak the secret key. We accurately simulate cache sets, using the Intel Pin tool [43] to record the memory accesses and extract the corresponding cache set. As we only consider virtual addresses, we only extract the lower 6 bits of the cache set. However, this is already sufficient to recover the key. Figure 7.4 shows the cache-set accesses for two AES keys.

MbedTLS' AES implementation leaks the key in two different functions. First, in the `mbedtls_aes_setkey_enc` function responsible for the key schedule. Second, in the `mbedtls_aes_encrypt_internal` function performing the T-table-based encryption. We consider two attacks, both with a *single* simulated Prime+Probe trace. If the attacker only records the encryption function, we require a known plaintext to recover the key. However, if the key schedule is included, we extend the attack to even recover parts of the plaintext. Note that we can exchange the encryption and known-plaintext attack with the decryption counterparts.

We analyze the trace with the Z3 solver [21] over 10 000 simulated Prime+Probe attacks on mbedTLS' AES-128 implementation with randomly generated keys and plaintexts. The solver recovers all keys using a known-plaintext attack, where each recovery only takes seconds. With unknown plaintext, the solver needs at most 74.08 min ($\sigma_{\bar{x}} = 0.367$, $n = 1200$) to recover the key and additionally 10 bytes of the plaintext. The solver can recover even more plaintext bytes, however, the performance depends on the used key and plaintext. The performance evaluation used as the known plaintext's bytes, the higher bytes 10 to 15. The solver always finds the correct key without additional candidates. Hence, if an attacker can disable AES-NI at runtime, they can force mbedTLS to a path vulnerable to side-channel attacks, and extract the key.

5.2. Mitigating Software Prefetch Attacks on AMD

In this case study, we present the first software mitigation on AMD systems against prefetch-based side-channel attacks [26, 50]. Prefetch-based KASLR breaks exploit the runtime difference of the prefetch instruction for mapped and unmapped addresses, effectively derandomizing the kernel location. The prefetch KASLR break is an important part in the recent "Spectre in the Wild" exploit [79] to find the addresses of targeted kernel structures. As this is the only known full microarchitectural KASLR break on AMD CPUs, it is desirable to prevent this type of attack. Furthermore, Lipp et al. [50] exploit prefetch on AMD to break fine-grained KASLR, monitor kernel activity and to leak kernel memory using Spectre. With MSRevelio, we search and discover an MSR that disables the prefetch instructions on AMD systems. The prefetch-disabling bits can be set by the OS or a privileged user to prevent all prefetch-based side-channel attacks and, therefore, remove a building block for sophisticated attacks.

5.2.1. Threat Model

We assume a system without software bugs in the kernel and enabled KASLR. We further assume an unprivileged attacker with code execution on the system. The system does not expose the kernel offset via system interfaces, e.g., `/proc/kallsysms`. Thus, the attacker uses the prefetch-based KASLR break to mount attacks or extract data from the kernel, e.g., using a Spectre attack [79, 50].

Table 7.2.: The bits of MSR 0xc0011029 were found with the instruction analysis of MSRevelio.

MSR 0xc0011029	Description	Effect
Bit 2	disable `PREFETCHNTA`	-1 LdDispatch
Bit 3	disable `PREFETCHT0`	-1 LdDispatch
Bit 4	disable `PREFETCHT1`	-1 LdDispatch
Bit 5	disable `PREFETCHT2`	-1 LdDispatch
Bit 6	disable `PREFETCHW`	-1 LdDispatch
Bit 7	disable `PREFETCH`	-1 LdDispatch

5.2.2. MSR Discovery

With the knowledge that hardware prefetchers can be disabled [3], we suspect that there might also be a possibility to disable software prefetchers. Hence, we leverage MSRevelio to automatically find MSR configuration bits that influence the software-prefetch instructions (cf. Section 3.3). On an AMD Ryzen Threadripper 1920X CPU, MSRevelio discovered the MSR(`0xC0011029`). As Table 7.2 shows, bits 2 to 7 inside the MSR alter the behavior of the prefetch instructions as detected by MSRevelio. MSRevelio found these bits due to the reduced `LsDispatch.LdDispatch` performance counter by exactly one load compared to the reference (cf. Section 3.3). This MSR is a "tweak" MSR that is used in errata to circumvent CPU bugs [78]. Although this MSR is not documented for the Zen microarchitecture (family 17h), we find the MSR in the extensive list of documented MSRs for the Bulldozer microarchitecture (family 15h) [3] (page 590), where these bits are documented as disabling the software prefetch instructions. Hence, by setting these bits, the OS can disable each of the six variants of the `prefetch` instructions individually.

5.2.3. Mitigate Prefetch Attacks

For our experimental setup, we use the file-based Linux MSR interface to disable all the software prefetch instructions. For evaluation, we build a PoC implementation of a prefetch-based KASLR break. The kernel is located in one of 512 possible virtual address offsets [14]. The PoC measures the execution time of the `prefetch2` instruction for all these possible virtual addresses. For every address, the KASLR break measures

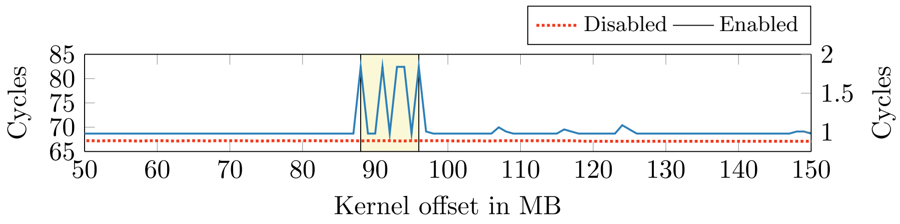


Figure 7.5.: The prefetch-based KASLR break with enabled (left axis) and disabled (right axis) prefetch instructions. The yellow box indicates the kernel's location at start offset 88.

the execution time of 10 000 prefetch invocations executed in a loop. The loop is repeated 100 times, and the minimum of all the tries is recorded. If the kernel is mapped at the prefetched location, the execution time is higher (cf. Figure 7.5).

We execute the KASLR break on an AMD Ryzen Threadripper 1920x @ 3.8GHz with Ubuntu 20.04 LTS and Linux 5.4.0-74, with prefetch instructions enabled and disabled. Figure 7.5 shows the difference between the two invocations of the KASLR break. We observe that the KASLR break can precisely locate the kernel (offset 88) in the enabled case and fails to locate the kernel otherwise. Furthermore, we compare the execution time of `prefetch` when disabled with the execution time of a single byte `nop` instruction. The `nop` instruction takes 0.886 cycles ($\sigma_{\bar{x}} = 0.0092$, $n = 512\,000\,000$) on average, and the disabled `prefetch` instruction takes 0.885 cycles ($\sigma_{\bar{x}} = 0.0090$, $n = 512\,000\,000$) on average. From this experiment, we conclude that the disabled `prefetch` instruction is indeed equivalent to a `nop` instruction [3]. Thus, the disabled prefetch instruction stops loading data into the cache and does not translate the provided virtual address.

We compare the disabling of the prefetch instructions with the recent FLARE [14] KASLR mitigation, which hardens the kernel against microarchitectural KASLR breaks. However, as all other microarchitectural KASLR breaks only apply to Intel CPUs, disabling the prefetch instructions leads to similar security and performance guarantees as FLARE without modifying the kernel or additional memory overhead. Furthermore, a privileged user, such as a system administrator, can directly activate this mitigation over the MSR interface without any additional requirements. The runtime overhead is also not directly visible as most applications do not use prefetch instructions. On our Ubuntu 20.04 installation, less than 1 % of the installed binaries (300 out of 30 842) contain any software prefetch instructions. We evaluate the performance impact of disabling these instructions with the SPEC CPU 2017 benchmark. Table 7.2 shows the benchmark results, where the baseline is with the prefetch instructions enabled. The average performance overhead is only 0.04 %, and thus negligible.

5.3. Intercepting CPUID to Reduce CrossTalk Leakage

In this case study, we present a software-based mitigation to reduce CrossTalk [64] leakage. The CrossTalk attack leaks data from the staging buffer via the line-fill buffer. To get the data from the staging buffer to the line-fill buffer, CrossTalk uses `cpuid`, `rdrand`, `rdseed`, and `rdmsr` as *leaking primitives* leaking confidential data such as random numbers. However, the `rdmsr` instruction is only available in a privileged attacker model. Ragab et al. [64] assume that the used instructions cannot be trapped, but that it would hypothetically also hinder exploitation. We challenge this assumption by using MSRevelio to search for an MSR bit that can trap the `cpuid` instruction. Our evaluation shows that `cpuid` is the most reliable unprivileged leakage primitives even without TSX. Indeed, MSRevelio successfully discovered such a bit, leading to the first pure software mitigation to practically mitigate unprivileged CrossTalk attacks. By trapping `cpuid`, the OS can ensure that no confidential data, such as random numbers, are leaked from the staging buffer.

5.3.1. Threat Model

For the pure software mitigation, we assume that an attacker can run unprivileged programs on a CPU affected by CrossTalk. The attacking program controls on which logical core the code is executed and can invoke the unprivileged `cpuid` instruction. The system does not deploy the microcode patch against CrossTalk [64], e.g., for performance reasons or because none is available.

5.3.2. MSR Discovery

MSRevelio found MSR(`0x140`) on an Intel i5-4570 using the instruction-behavior analysis (cf. Section 3.3) that allows trapping the `cpuid` instruction. On this microarchitecture, this MSR is undocumented in the Intel

Table 7.3.: SPEC CPU 2017 benchmark for the performance overhead when disabling the software prefetch instructions.

Benchmark	**SPEC Score** Baseline	No Prefetch	**Overhead** [%]
600.perlbench_s	4.02	4.02	0.00
602.gcc_s	7.32	7.34	−0.27
605.mcf_s	6.57	6.56	+0.15
620.omnetpp_s	2.89	2.88	+0.35
623.xalancbmk_s	4.34	4.33	+0.23
625.x264_s	4.79	4.79	0.00
631.deepsjeng_s	3.19	3.20	−0.31
641.leela_s	3.29	3.28	+0.30
648.exchange2_s	9.10	9.11	−0.11
Average			+0.04

SDM [36]. The only mention of this MSR is on the Xeon Phi, where it is referred to as `MISC_FEATURE_ENABLES`. Still, even on Intel Core, Xeon, and Celeron CPUs, setting bit 0 results in `cpuid` raising a GP fault. This feature was apparently introduced with the Intel Ivy Bridge microarchitecture as it does not exist on our tested Sandy Bridge machine (i5-2520M). On all our tested machines starting with Ivy Bridge (i5-3230M) to Ice Lake (i3-1005G1) and Jasper Lake (N4500), the MSR exists, and setting bit 0 allows trapping `cpuid`. Therefore, this allows us to harden systems against unprivileged CrossTalk attacks. While designing mitigations on top of undocumented features seems unwise, Intel CPUs with the 10^{th} generation already contain silicon fixes [31].

5.3.3. CrossTalk-Mitigation Implementation

We implemented a proof of concept to show that this attack is indeed prevented by trapping the `cpuid` instruction. Our PoC consists of a kernel module and a user-space shared library. The kernel module provides a single `ioctl` that is used by the user-space library to set and clear bit 0 of MSR(`0x140`), *i.e.*, to trap `cpuid` or allow it. The user-space library sets up the kernel module and transparently handles the GP faults. It is simply preloaded for binaries using the `LD_PRELOAD` environment variable. Note that this is just for the sake of simplicity for the proof-of-concept implementation. When implemented for a production system, the entire implementation would either be in the kernel, or partly in the kernel and partly in the dynamic linker and loader, such that it is applied to every binary on the system. The preloaded library installs a signal handler for GP faults. This signal handler analyzes the memory location at the faulting instruction pointer. If the address of the instruction pointer is accessible (can be verified by abusing the `access` syscall [58]), the library checks if the opcode of `cpuid` (`0xA20F`) is found. If not, any potential other signal handler can be called. However, if the cause of the fault is the `cpuid` instruction, the library ensures that no sensitive data can be leaked from the staging buffer. We evaluated two variants: (1) only executing `cpuid` once before the application starts, and returning cached values for all other calls, (2) first overwriting the staging buffer with other information by calling `rdrand` and then executing the `cpuid` instruction. In both cases, `cpuid` does not transfer the targeted sensitive values from the staging buffer to the line-fill buffer. Moreover, caching the output of `cpuid` is not a functional problem, as the output typically does not change during the runtime.

5.3.4. Evaluation

We evaluated both variants' security and performance overhead on an Intel Xeon E3-1505M v5 with Ubuntu 20.04 and Linux 5.4.0-90. To verify that our method indeed hinders CrossTalk, we implemented the two PoCs from the paper, leaking the CPU brand string and the last generated `rdrand` random number. Both PoCs leak the targeted data from the staging buffer. We verified that preloading our library without enabling the trap does not negatively impact the PoC. With the `cpuid` trap enabled, we do not observe any leakage anymore. For the leakage mitigation, we do not observe any difference in the methods, *i.e.*, whether we use cached `cpuid` values or overwrite the staging buffer.

To measure the performance overhead of the `cpuid` trap, we used a microbenchmark that simply executes `cpuid` in a loop. In the normal case, *i.e.*, without the `cpuid` trap, the execution takes 182 cycles ($\sigma_{\bar{x}} = 0.6$, $n = 100\,000$). Trapping the `cpuid` instruction, overwriting the staging buffer, and re-executing it takes on average 9932 cycles ($\sigma_{\bar{x}} = 6.1$, $n = 100\,000$). Caching the `cpuid` instruction improves the performance slightly, with an average execution time of 8204 cycles ($\sigma_{\bar{x}} = 5.7$, $n = 100\,000$). As `cpuid` is typically only called at program startup, e.g., in the libc for feature detection [24], this overhead is negligible for the overall system performance. In contrast, the microcode patch for CrossTalk introduces an overhead of factor 12 for the `rdrand` instruction [64], which is usually used more often than the `cpuid` instruction.

5.3.5. Leakage Analysis

We validate that `cpuid` has the highest leakage rate of the unprivileged instructions, by extending the Crosstalk PoC to allow evaluation of the leakage rates when using either signal handling, TSX, or TAA as an exception-suppression method. First, we evaluate leaking `rdrand` with the `cpuid` instruction. Second, we exchange `cpuid` with `rdseed` in the attacker and evaluate the leakage again. Our evaluation found that exchanging `rdseed` with `rdrand` and vice versa did not influence the leakage rates. Therefore, we focused on leaking the more commonly used `rdrand` values. For the evaluation, we use an Intel i7-6700k with Ubuntu 20.04 and Linux 5.4.0-40 with disabled mitigations. The victim generates a random number every 187.5 ms and repeats this 100 times, generating overall 800 B of random data. We perform each experiment 10 times and count the correctly-leaked bytes and how often the entire eight-byte random number is successfully leaked.

For `cpuid`, we observe a leakage of 711.3 B ($\sigma_{\bar{x}} = 4.185$ B) with signal handling, 741.3 B ($\sigma_{\bar{x}} = 4.770$ B) with TSX, and 416.3 B ($\sigma_{\bar{x}} = 3.556$ B) with TAA. With `rdseed` as primitive, we observe a leakage of 3.4 B ($\sigma_{\bar{x}} = 0.544$ B) with signal handling, 2.1 B ($\sigma_{\bar{x}} = 0.368$ B) with TSX, and 553.3 B ($\sigma_{\bar{x}} = 18.296$ B) with TAA. Furthermore, when using signal handling, `cpuid` leaks on average 51.8 times the entire random number whereas `rdseed` is unable to leak the entire random number. For the overall byte-wise leakage rate, the leakage rate of `cpuid` is 211.4 times higher than `rdseed`'s when using signal handling. With recent microcode patches disabling TSX and, therefore, mitigating TAA, the `cpuid` trap is a viable option to further harden systems against unprivileged CrossTalk attacks.

5.4. Disabling Fast-String Support to Reduce Medusa Leakage

In this case study, we show a software-based approach to reduce the leakage of the Medusa attack [60]. Medusa is a Microarchitectural Data Sampling

Table 7.4.: SPEC CPU 2017 benchmark performance overhead when disabling the fast-string optimization.

Benchmark	**SPEC Score**		**Overhead**
	Baseline	No Fast-Strings	[%]
600.perlbench_s	5.17	5.16	+0.19
602.gcc_s	8.76	8.06	+8.06
605.mcf_s	6.95	6.91	+0.58
620.omnetpp_s	3.61	3.62	−0.28
623.xalancbmk_s	4.50	4.51	−0.37
625.x264_s	4.47	4.46	+0.15
631.deepsjeng_s	3.92	3.67	+6.37
641.leela_s	3.56	3.57	−0.19
648.exchange2_s	9.82	9.82	0.00
Average			+1.61

(MDS) attack, leaking data from the line-fill buffer on Intel CPUs. Medusa leaks data from Write Combining (WC) operations or memory operations backed by WC memory. These write-combining instructions use a part of the line-fill buffer to combine writes to the same cache line to reduce requests sent over the memory bus.

The Medusa attack uses implicit WC instructions like non-temporal moves, `rep movs`, `rep stos` instructions, or explicit WC memory to leak data from the WC buffer. However, as WC memory requires a special memory type, an attacker needs privileges to acquire it, which is only realistic when attacking SGX. We focus on implicit WC operations, available to an unprivileged attacker. By reducing the likelihood that sensitive data ends up in the WC buffer, the probability of a successful Medusa attack is also reduced.

5.4.1. Threat Model

We assume an unprivileged attacker is exploiting implicit WC instructions for the Medusa attack on an affected CPU. The OS does not mitigate the Medusa attack, e.g., with the adapted `verw` instruction to clear the fill buffer or group scheduling [60]. Therefore, an attacker can be co-located on the same physical core as the victim program, sharing the core's fill buffer with the victim.

5.4.2. MSR Discovery

With the BIOS templating approach (cf. Section 4), MSRevelio automatically detected the documented fast-string enable bit (bit 0) inside MSR(`0x1a0`) (IA32_FEATURE_CONTROL). Intel [36] documents this bit as fast-string enable bit, but the internal effects are only sparsely documented [33, 35]. Based on the instruction-behavior analysis of MSRevelio, we see that clearing the fast-string-enable bit changes the associated instructions to no longer perform WC memory writes. Therefore, this bit is suitable to reduce leakage of the unprivileged Medusa attacks.

5.4.3. Evaluation

We evaluate the impact of the fast-string enable bit on the Medusa attack on an Intel Core i7-6700K CPU that is affected by Medusa with Ubuntu 20.04 LTS and Linux 5.4.0-40. For the evaluation, we rely on the public Medusa PoCs [60]. Specifically, we focus on the PoC variants using fast-string operations that an unprivileged attacker can use [60]. We run the victim using fast-string operations with sensible data and test against these attack variants. We fix the test system's frequency to 3 GHz and pin the attacker and victim applications to sibling hyperthreads. We first verify that the PoC successfully leaks the targeted data. When disabling fast-string operations using MSR(`0x1a0`), the leakage is entirely gone, successfully preventing these variants of Medusa. We evaluate the performance overhead when disabling fast-string operations with the SPEC CPU 2017 benchmark and observe an average performance overhead of 1.61 %. Table 7.4 shows the benchmark results, where the baseline is with the optimized fast strings operations enabled.

5.4.4. Discussion

Similar to the CrossTalk mitigation (cf. Section 5.3), this is only a short-term solution for affected CPUs. Newer CPUs, e.g., 10^{th} generation, are not affected by Medusa anymore, hence this mitigation is only necessary on older CPUs for which this MSR bit successfully reduces the leakage. Additionally, CPUs received security updates by repurposing the `verw` instruction to clear microarchitectural buffers on context switches [60]. While `verw` does not prevent attacks from a hyperthread, disabling the write combining instruction mitigates these attacks. Finally, as the leakable data for an attacker depends on the instructions executed within a victim application, the only remaining sources for leakage are non-temporal moves, as well as the upper 128 bit of AVX stores, which can be disabled via the `XCR0` register.

5.5. Tracing Microcode-introduced MSRs

In this case study, we show that MSRevelio can trace the evolution of MSRs of a CPU over multiple microcode versions. Tracing the addition of MSRs allows determining the microcode version where vendors deployed patches relying on additional MSRs. Moreover, we can analyze the time between deployment and documentation of MSRs. We show that all MSRs retrofitted using microcode on our tested machines are security-related. Thus, detecting MSRs before they are documented hints that there is a CPU vulnerability currently under embargo, as the fixes should, in the best case, already be deployed when the embargo ends. If a security patch introduces undocumented MSRs, we assume that the undocumented MSR exposes additional configuration bits for the mitigation. With this information, an adversary can determine the effects corresponding to the added MSR and potentially infer the reason for the security patch. This approach is similar to *patch diffing* where an attacker extracts the vulnerability by inspecting the patches provided to a system or component.

Since the discovery of Spectre [48] and Meltdown [53], many security patches have included new MSRs to mitigate vulnerabilities. To help OSs and hypervisors mitigate the impact of Spectre, a microcode update added MSR(`0x49`) (IA32_PRED_CMD) and MSR(`0x48`) (IA32_SPEC_CTR). These MSRs allow configuring the branch predictor and flushing it's state [32]. Similarly, due to Foreshadow [73], the MSR(`0x10B`) (IA32_FLUSH_CMD) was introduced to flush the L1 cache [37]. Recently, Intel also introduced MSR(`0x123`) (IA32_MCU_OPT_CTRL) with a microcode update to change the behavior of `rdseed` and `rdrand`, mitigating the CrossTalk attack [64].

5.5.1. Threat Model

We assume a sophisticated attacker is tracking the evolution of MSRs over multiple released microcode versions for different CPU generations to find new undocumented MSRs. The attacker uses the classification approaches shown in Section 3.2 and Section 3.3 to determine the effects of the MSR and potentially the vulnerability's source. This leaves the attacker with additional time until the public disclosure to mount attacks on unpatched systems.

5.5.2. Implementation

To trace the evolution of the MSRs over the microcode version, we use the late-loading mechanism of Linux [83]. The late-loading mechanism updates the CPU microcode to a newer version without rebooting the system. As a source for the microcodes, we rely on two GitHub repositories. First, the official Intel microcode repository [38], containing the microcode versions back to March 2019. Second, a collection of microcode versions from Plato Mavropoulos [56] dating back to 1996. A signature from Intel ensures the integrity of all microcode files. For each microcode version available for our CPU, MSRevelio extracts the list of MSRs available after applying the microcode update. As microcode can only be replaced by microcode with a newer version, we start at the initial microcode version hardcoded in the BIOS, and gradually update to newer versions. As a test system, we chose a CPU with the Sandy Bridge microarchitecture, which is the oldest second-generation Intel Core microarchitecture, released in 2011. Additionally, we use a CPU with the Broadwell microarchitecture (released 2014) and a CPU with the Coffee Lake microarchitecture (released 2017). For all published transient-execution attacks [31, 15], at least one of these microarchitectures is affected. With 28 microcode versions, ranging from 2011 to 2021, we found a large number of different microcodes to test.

5.5.3. Results

As expected, MSRevelio detects the introduction of MSR(`0x48`), MSR(`0x49`), MSR(`0x10B`) and MSR(`0x123`). Interestingly, there is a significant time difference between the first occurrence of these MSRs and their documentation. For the Sandy Bridge machine, MSR(`0x48`) and MSR(`0x49`) were introduced with microcode 0x2d on February 7th, 2018, more than a month after the disclosure of Spectre [48]. In contrast, on Broadwell, these MSRs were introduced with microcode version 0x28 already on November 17th, 2017, *i.e.*, nearly 7 weeks *before* the public disclosure. On the same machine, MSR(`0x10B`) was introduced with microcode 0x2b on March 22nd, 2018 nearly 5 months *before* the public

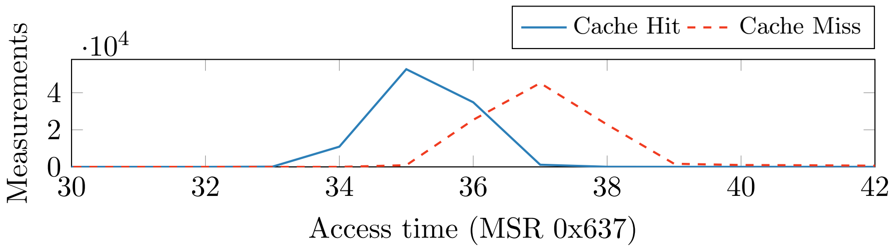


Figure 7.6.: Cache hits and misses measured with MSR(`0x637`).

disclosure of Foreshadow on August 14th, 2018. The Sandy Bridge machine received this MSR with microcode 0x2e on May 10^{th}, 2018, 3 months *before* the disclosure. MSR(`0x123`) was introduced on the Broadwell with microcode 0x2f from November 12th, 2019, while the public disclosure was on June 9th, 2020. Skylake-S and Coffee Lake received the MSR with the microcode update from January 9th, 2020. We use MSRevelio to analyze the CrossTalk mitigation's MSR(`0x123`) and observe differences for `rdrand` and `rdseed`, directly revealing the affected instructions. These results show that microcode updates containing new MSRs are distributed before the MSR is officially documented. None of our tested microcode updates introduced any non-security-related MSRs. We assume that new non-security-related MSRs are only introduced with new microarchitectures. Hence, by using MSRevelio, it is possible to reveal the existence of CPU vulnerabilities or errata before they are publicly documented.

5.6. Exploiting Xen

In this section, we show that MSRevelio is also applicable in cloud environments to enumerate MSRs accessible to guest virtual machines. This does not only expose access to MSRs that can constitute a security threat but also allows fingerprinting the hypervisor. The Xen hypervisor checks the availability of an MSR inside the `guest_rdmsr` and `guest_wrmsr` functions and decides if access should be *simulated*, *allowed*, or trigger a *GP-fault* [82]. Until recently, the Xen hypervisor used a blocklist prohibiting guests from accessing MSRs. Due to the nature of a blocklist approach, the hypervisor does not restrict access to undocumented MSRs and, thus, allows guests to read and write them. With version 4.15 (April 2021), Xen changed the default behavior from a blocklist to an allowlist [16, 62], preventing access to undocumented MSRs.

5.6.1. Threat Model

We assume a privileged attacker running in a Xen VM re-enacting a scenario of a low-priced cloud provider offering single-core virtual machines. We assume that the attacker's virtual machine is pinned to a logical core of the machine sharing the physical core with other guests. Furthermore, we assume that the Xen hypervisor disables access to MSRs enabling power side-channel attacks [52] and traps calls to the timestamp counter to reduce its resolution [76, 54]. As the victim machine only runs on a single core, alternative timing approaches like counting threads [51] are unavailable.

5.6.2. Alternative Timer in Xen

With MSRevelio, we discovered the undocumented MSR(`0x637`) on Intel CPUs that continuously increments its value and correlates with the timestamp counter (cf. Figure 7.2). While not officially documented in Intel's SDM [36], we found a reference to MSR(`0x637`) in the coreboot project [18] as MSR_COUNTER_24_MHZ. We exploit this timer to mount a Foreshadow attack [42, 81] from a Xen virtual machine. We evaluate the attack on an Intel Core i7-8650U running Ubuntu 20.04.2 LTS with Linux 5.4.0-52 and Xen 4.14.2. The attacker runs in a PVH virtual machine with Debian 10 and Linux 4.19.0-16 with a single virtual CPU assigned to a logical core.

In our experiment, the attacker uses the MSR(`0x637`) as a timestamp to distinguish cache hits from misses. In contrast to the cycle-accurate timestamp accessible via `rdtsc`, this timestamp has a lower resolution of only 41.67 ns. Thus, when measuring the access time to cached and uncached memory using the MSR, there is a slight overlap, as shown in Figure 7.6.

In our attack, the attacker sets the PFN of one of its EPT pages to a physical location of a victim page and marks the page as non-present, triggering an L1 Terminal Fault [42] on access. As the page is not present, the CPU aborts the page translation and uses the attacker-controlled PFN to lookup data in the L1 cache. We use Intel TSX to suppress the fault and encode the leaked values by caching corresponding addresses in a look-up table. Using Flush+Reload with the MSR as a timestamp, we successfully recover the leaked values.

Table 7.5.: Accessible MSRs in different Xen versions.

Xen Version	# Accessible	# Static	# Dynamic
4.7 (IBM cloud)	618	600	18
4.11.4 (Ubuntu)	521	496	25
4.11.4 (Debian)	505	486	19
4.14.1	452	434	18
4.15	203	202	1

In our attack, we leak a 50-byte string from a victim running on the sibling hyperthread. Our unoptimized proof-of-concept implementation has an average runtime of 107.4 ms ($\sigma_{\bar{x}} = 1000$, $n = 1.756$) and an average leakage rate of 214 B/s ($\sigma_{\bar{x}} = 1000$, $n = 4.176$). When using `rdtsc` as the timestamp, we achieve an average runtime of 0.38 ms ($\sigma_{\bar{x}} = 1000$, $n = 0.003$) and an average leakage rate of 49 147 B/s ($\sigma_{\bar{x}} = 1000$, $n = 498.66$). The difference is mainly caused by the measurement imprecision caused by the lower resolution and also because querying the MSR value has a heavy performance impact [2].

5.6.3. Fingerprinting Xen Versions

In our analysis of various Xen versions, we detected that the number of visible MSRs is different for each Xen version, allowing an attacker to infer the Xen version if this information is (partly) blocked, as e.g., on the IBM cloud. We evaluated MSRevelio within different Xen versions on an Intel i7-8650U CPU running Ubuntu 20.04.2 LTS with Linux 5.40-52 and on the IBM cloud.

For our evaluation, we first fingerprint the hypervisor by using the MSR detection mechanism of MSRevelio, including the analysis for static and dynamic MSRs. We show the results of the detected MSRs for 5 different XEN versions in Table 7.5. It is noticeable that with an increasing version number, the number of exposed MSRs decreases as the blocklist is extended with additional entries. The implemented allowlist approach of version 4.15 significantly reduces the number of MSRs from 452 (version 4.14.1) to 203.

It is worth highlighting that while reporting the same Xen version (4.11.4), the number of accessible MSRs between the latest available version on Ubuntu and Debian differs by 16 MSRs. By comparing the detected

MSRs, we observed that the MSRs exploited by the Platypus attack [52] are still accessible on the Ubuntu installation as the security patches of Xen have not been applied to Ubuntu. Thus, despite reporting the same version number, virtual machines can be exposed to different security risks depending on the patch level.

6. Discussion and Limitations

6.1. Related Work

One of the first MSR scanners dates back to 2001 [46] using the file-based Linux MSR interface to search for readable MSRs. In contrast to MSRevelio, this scanner can only detect readable MSRs. Furthermore, recent changes to the Linux kernel restrict accesses to MSRs from userspace [63], making this approach less reliable.

Domas [23, 22] focused on the execution time of reading MSRs to identify ones with unique functionality to detect a potential CPU backdoor. With MSRevelio, we do not focus on unique MSRs changing the ISA, *i.e.*, introducing new instructions or changing the architectural functionality of instructions.

Haruspex [10] scans the x86 instruction set with speculative execution and performance counters. Bölük also applied this approach to detect undocumented MSRs [11]. While the performance of this approach is unclear, the stated detected MSRs match our findings, e.g., for MSR(`0x2e6`) mentioned as `LT_LOCK_MEMORY_MSR` in an Intel errata [40]. In addition, MSRevelio found that one can only write 0 to this MSR.

6.2. Indirect Effects

There are also MSR bits that affect the system without directly affecting instructions. Examples are the configuration of hardware prefetchers or fixes for CPU errata. These bits only have a measurable effect in corner cases that cannot be triggered automatically. To detect the effect of such bits, targeted test cases would be required. If such a targeted test case exists, e.g., because someone has the intuition that an undocumented MSR affects a specific feature (e.g., disabling hardware prefetchers), MSRevelio can also be extended to analyze MSR bits concerning this particular test.

7. Conclusion

With MSRevelio, we automatically detect undocumented MSRs and their effects on Intel and AMD CPUs. We demonstrate that undocumented MSRs can not only hint at the existence of CPU vulnerabilities but that they can also introduce new attack vectors or re-enable mitigated attacks. Furthermore, we show that undocumented MSRs can also be used to mitigate various microarchitectural attacks.

In this paper, we show that undocumented or sparsely documented MSRs have a non-negligible effect on system security, not only on native systems but also in the cloud.

Acknowledgments

We thank the anonymous reviewers for their guidance, comments, and suggestions. Additional funding was provided by a generous gift from Amazon. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties. Daniel Weber thanks the Saarbrücken Graduate School of Computer Science. Andreas Kogler thanks Claudio Canella, Lukas Giner, and Martin Schwarzl for the discussions and feedback.

References

[1] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In: ISPASS. 2020 (pp. 177, 184, 217).

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS. 2019 (p. 207).

[3] Advanced Micro Devices Inc. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. 2013 (pp. 195, 196).

[4] Amazon. Amazon EC2 On-Demand Pricing. 2021. URL: `https://aws.amazon.com/ec2/pricing/on-demand/` (p. 183).

[5] AMD64 Architecture Programmer's Manual. Advanced Micro Devices Inc. 2017 (p. 189).

[6] AMI. BIOS/UEFI Utilities. 2021. URL: https://www.ami.com/products/firmware-tools-and-utilities/bios-uefi-utilities/ (p. 191).

[7] ARM. mbed TLS. 2020. URL: https:///tls.mbed.org (pp. 175, 190–192).

[8] Jean-Philippe Aumasson and Luis Merino. SGX Secure Enclaves in Practice: Security and Crypto Review. In: Black Hat Briefings. 2016 (p. 190).

[9] Donald J. Berndt and James Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In: Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining. 1994 (p. 182).

[10] Can Bölük. Haruspex. 2021. URL: https://github.com/can1357/haruspex/ (p. 208).

[11] Can Bölük. Undocumented MSRs with Haruspex. 2021. URL: https://twitter.com/_can1357/status/1427511999550959628 (p. 208).

[12] 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 (pp. 177, 193).

[13] 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 (p. 178).

[14] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 195, 196).

[15] 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 (pp. 178, 204).

[16] Andrew Cooper. x86/hvm: disallow access to unknown MSRs. 2020. URL: https://xenbits.xen.org/gitweb/?p=xen.git;a=commitdiff;h=84e848fd7a162f669cf8248ce502ca864f869447 (pp. 175, 205).

[17] CoreBoot. CoreBoot - Bios Update Trigger. 2021. URL: https://github.com/coreboot/coreboot/blob/master/src/soc/intel/common/block/include/intelblocks/msr.%5C#L17 (p. 189).

[18] coreboot. coreboot: Fast, secure and flexible OpenSource firmware. 2019. URL: https://www.coreboot.org/ (p. 206).

[19] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (pp. 173, 178, 186, 190).

[20] Czernobyl. Super-secret debug capabilities of AMD processors ! 2014. URL: http://www.woodmann.com/collaborative/knowledge/index.php/Super-secret_debug_capabilities_of_AMD_processors_! (p. 173).

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. 2008, pp. 337–340 (p. 193).

[22] Christopher Domas. Breaking the x86 ISA, v. 2017-07-27. In: Black Hat US (2017) (p. 208).

[23] Christopher Domas. Hardware Backdoors in x86 CPUs. In: Black Hat US (2018) (pp. 173, 174, 176, 185, 208).

[24] Free Software Foundation. GLIBC Feature Detection. 2021. URL: https://github.com/bminor/glibc/blob/master/sysdeps/x86/cpu-features.c (p. 200).

[25] Corey Gough, Ian Steiner, and Winston Saunders. Energy Efficient Servers. Apress, 2015 (p. 189).

[26] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 174, 175, 194).

[27] G. Glenn Henry and Terry Parks. Apparatus and method for limiting access to model specific registers in a microprocessor. US Patent 8,341,419 B2. Dec. 2012 (p. 173).

[28] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (p. 178).

[29] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In: Journal of Computer Security (1992) (p. 175).

[30] Intel. Advanced Encryption Standard (AES) Instructions Set: White Paper. 2008 (pp. 173, 190).

[31] Intel. Affected Processors: Transient Execution Attacks. 2020. URL: `https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model` (pp. 198, 204).

[32] Intel. Deep Dive: CPUID Enumeration and Architectural MSRs. May 2019. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-cpuid-enumeration-and-architectural-msr%5C#MDS-CPUID` (pp. 173, 174, 203).

[33] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (p. 202).

[34] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 2019 (p. 180).

[35] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 176, 202).

[36] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers. May 2019 (pp. 181, 184, 189, 198, 202, 206).

[37] Intel. Intel Analysis of Speculative Execution Side Channels. 2018. URL: `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf` (pp. 173, 203).

[38] Intel. Intel Linux Processor Microcode Data Files. 2021. URL: `https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files` (p. 204).

[39] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference. Rev 1.5. May 2016 (p. 192).

[40] Intel. Intel Xeon Processor Scalable Family. 2020 (p. 208).

[41] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (p. 186).

[42] Intel. L1 Terminal Fault SA-00161. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault (p. 206).

[43] Intel. Pin - A Dynamic Binary Instrumentation Tool. 2012. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool (p. 193).

[44] Intel. Refined Speculative Execution Terminology. 2020. URL: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology (p. 178).

[45] Alex James. ghidra-firmware-utils. 2021. URL: https://github.com/al3xtjames/ghidra-firmware-utils (p. 191).

[46] Dave Jone. Dave Jone's MSR scanner. 2001. URL: https://www.mail-archive.com/linuxbios@listman.lanl.gov/msg02813.html (p. 208).

[47] Phillip Kemkes. Techniques: Current Use of Virtual Machine Detection Methods. 2020. URL: https://www.gdatasoftware.com/blog/2020/05/36068-current-use-of-virtual-machine-detection-methods (p. 175).

[48] 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 (pp. 173, 177, 178, 203, 204).

[49] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. Reverse engineering x86 processor microcode. In: USENIX Security Symposium. 2017 (p. 178).

[50] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In: USENIX Security Symposium. 2022 (pp. 174, 175, 194).

[51] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (p. 206).

[52] 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 (pp. 206, 208).

[53] 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 (pp. 178, 203).

[54] Weijie Liu, Debin Gao, and Michael K Reiter. On-demand time blurring to support side-channel defense. In: ESORICS. 2017 (p. 206).

[55] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (p. 178).

[56] Plato Mavropoulos. CPUMicrocodes. 2021. URL: `https://github.com/platomav/CPUMicrocodes` (p. 204).

[57] John Mechalas. Trusted CPU Feature Detection Library. 2019. URL: `https://github.com/intel/sgx-cpu-feature-detection` (p. 192).

[58] Matt Miller. Safely Searching Process Virtual Address Space. 2004 (p. 199).

[59] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (p. 193).

[60] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In: USENIX Security Symposium. 2020 (pp. 175, 178, 188, 200–202).

[61] Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh. 3.03. Advanced Micro Devices Inc. July 2018 (pp. 176, 181, 184).

[62] Roger Pau. x86/pv: disallow access to unknown MSRs. 2020. URL: `https://xenbits.xen.org/gitweb/?p=xen.git;a=commitdiff;h=322ec7c89f6640ee2a99d1040b6f786cf04872cf` (pp. 175, 205).

[63] Borislav Petkov. [RFC PATCH] x86/msr: Filter MSR writes. 2020. URL: `https://lkml.org/lkml/2020/6/12/273` (p. 208).

[64] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In: S&P. 2021 (pp. 173, 174, 177, 178, 197, 200, 203).

[65] David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, eds. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations. MIT Press, 1986. ISBN: 0-262-68053-X (p. 182).

[66] Jarkko Sakkinen. Add SGX Launch Control MSR definitions. 2018. URL: `https://patchwork.kernel.org/project/intel-sgx/patch/20181106134758.10572-14-jarkko.sakkinen@linux.intel.com/` (p. 189).

[67] 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 (pp. 173, 178).

[68] Nikolaj Schlej. UEFI firmware image viewer and editor. 2020. URL: `https://github.com/LongSoft/UEFITool` (p. 191).

[69] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS. 2018 (p. 177).

[70] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 177).

[71] 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 (pp. 173, 177, 178).

[72] Sofiane Takarabt, Alexander Schaub, Adrien Facon, Sylvain Guilley, Laurent Sauvage, Youssef Souissi, and Yves Mathieu. Cache-timing attacks still threaten IoT devices. In: Codes, Cryptology and Information Security. 2019 (p. 192).

[73] 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 (pp. 173, 175, 177, 178, 203).

[74] 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 (p. 178).

[75] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (pp. 192, 193).

[76] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In: CCSW. 2011 (pp. 175, 206).

[77] 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` (p. 188).

[78] Denys Vlasenko. Better document AMD "tweak MSRs". 2017. URL: `https://lore.kernel.org/patchwork/patch/783107/` (p. 195).

[79] Julien Voisin. Spectre exploits in the "wild". 2021. URL: `https://dustri.org/b/spectre-exploits-in-the-wild.html` (p. 194).

[80] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In: ESORICS. 2016 (p. 177).

[81] 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` (pp. 178, 206).

[82] XEN. XEN's MSR handling. 2021. URL: `https://github.com/xen-project/xen/blob/RELEASE-4.15.0/xen/arch/x86/msr.c` (p. 205).

[83] Fenghua Yu and Borislav Petkov. The Linux Microcode Loader. 2019. URL: `https://www.kernel.org/doc/html/latest/x86/microcode.html` (p. 204).

8. Appendix

8.1. Number of Tests

Equation (7.1) shows the number of tests N_T with MSRevelio. For each writable MSR, we first perform 16 tests in *phase 2* to find an MSR with side-effects. To determine the exact position of the enum within the MSR, we perform the optimized sliding window search, where we re-use the previous results. For the case with $W = 4$, we require 496 tests per observed effect for *phase 3*.

$$N_T = \underbrace{\left(\sum_{writeable} 2^W\right)}_{\text{phase 2}} + \underbrace{\left(\sum_{observed} \left(2^W + (64 - W)2^{W-1}\right)\right)}_{\text{phase 3}} \tag{7.1}$$

In comparison, Equation (7.2) shows the number of tests N_T required for an exhaustive search.

$$N_T = \sum_{msr \in writeable} 2^{writableBits(msr)} \tag{7.2}$$

8.2. Instruction Groups and PMC Events

Table 7.6 shows the instructions groups used during the instruction behaviour analysis (cf. Section 3.3). The full configuration groups including the detailed instructions are available in MSRevelio's GitHub repository[2]. For the performance counters we use the default configurations of *nanoBench* [1] for Intel[3] and AMD[4] CPUs. These configuration contain all performance counters for the given microarchitecture. However, most of the counters are also available on similar Intel or AMD microarchitectures, and we therefore relied on the Skylake configuration for all tested Intel CPUs (cf. Table 7.1).

[2]MSRevelio's repository: `https://github.com/IAIK/msrevelio`

[3]Intel PMC config: `https://github.com/andreas-abel/nanoBench/tree/fc038541dfd0de3428c7521131a548a85e923f7c/configs/cfg_Skylake_all.txt`

[4]AMD PMC config: `https://github.com/andreas-abel/nanoBench/tree/fc038541dfd0de3428c7521131a548a85e923f7c/configs/cfg_Zen_all.txt`

Table 7.6.: The used *instruction groups* for the instruction behavior analysis of the MSR bits (cf. Section 3.3).

Group Name	**# Instructions**	**Group Description**
AES	6	AES-NI instructions
CPUID	1	`cpuid` instruction
FENCES	3	`sfecne`, `mfence`, and `lfence` instructions
FLUSH	1	`clflush` instruction
FP_ARITH	9	x87 floating-point instructions
FP_VARITH	7	AVX2 vector floating-point instructions
INT_ARITH	12	x86 integer instructions
INT_VARITH	11	AVX2 vector integer instructions
LOAD	1	AVX2 vector load instruction
STORE	1	AVX2 vector store instruction
MOVES	35	various memory `mov` instructions
MISC	11	`xchg`, `bswap` and string instructions
PREFETCH	6	`prefetch` instructions
RANDOM	1	`rdrand` instruction
STRIDED	17	*strided* memory loads
TIME	2	`rdtsc`, `rdtscp` instructions
Total	124	

```
# AMI Aptio V BIOS/UEFI 2.21.1277 (Core Version 1.010)
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 10 P:83C801EB0383C80389442410:83C800EB0383C80289442410
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 12 P:83C801EB0383C80389442410:83C800EB0383C80289442410

# AMI Aptio V BIOS/UEFI 2.18.1263 (Core Version 5.12)
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 10 P:83C801EB0383C8035250683C010000:83C800EB0383C8025250683C010000
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 12 P:83C801EB0383C8035250683C010000:83C800EB0383C8025250683C010000
```

Figure 7.7.: The patch for AMI Aptio V BIOS to disable the AES-NI lock bit. The patch can be applied to a BIOS image using UEFIPatch.

8.3. Flipping Masks

Table 7.7 shows the *flipping masks* for enum fields with a length of up to 4 bits covering all possible 4 bit combinations, regardless of the position of the enum field within the 64 bit long MSR. When shifting the enum field inside the MSR, the *flipping mask* mimics the `rotate` operation.

8.4. BIOS Patch

Table 7.7.: *Flipping masks* for enum fields with length of up to 4 bits. Selecting 4 consecutive bits within these 16 masks always covers all possible enum bit combinations by mimicking the `rotate` operation (cf. Section 3.3).

Flipping Mask	**Enum[3:0]**	**Enum[4:1]**
`0x0000000000000000`	`0b...0000`	`0b..0000.`
`0x1111111111111111`	`0b...0001`	`0b..1000.`
`0x2222222222222222`	`0b...0010`	`0b..0001.`
`0x3333333333333333`	`0b...0011`	`0b..1001.`
`0x4444444444444444`	`0b...0100`	`0b..0010.`
⋮	⋮	⋮
`0xffffffffffffffff`	`0b...1111`	`0b..1111.`

Figure 7.7 is the patch in the UEFIPatch[5] format. We tested it with two different BIOS images, an AMI Aptio V 2.21.1277 with build date 2020, and an AMI Aptio V 2.18.1263 with build date 2021. The motherboard of the all-in-one PC is the BESSTAR TECH IB9. As we chose the pattern-based patch format, it should also be applicable to other AMI BIOS versions. The difference in the two patches is that in 2.18.1263, there is a call to a function wrapping `wrmsr`, whereas in 2.21.1277, the `wrmsr` instruction is inlined.

8.5. mbedTLS Leak

Listing 8.1 shows an excerpt of the relevant code from the mbedTLS[6] library where the implementation falls back to the T-table implementation if AES-NI is not available.

[5]UEFIPatch tool: `https://github.com/LongSoft/UEFITool/tree/master/UEFIPatch`

[6]MbedTLS's source code: `https://github.com/ARMmbed/mbedtls/blob/dd57b2f240c597e4cf6cc2492d5c03d067f234f9/library/aes.c#L587`

```
 1 #if defined(MBEDTLS_AESNI_C) && defined(MBEDTLS_HAVE_X86_64)
 2     if( mbedtls_aesni_has_support( MBEDTLS_AESNI_AES ) )
 3         return( mbedtls_aesni_setkey_enc( (unsigned char *) ctx->rk,
   -> key, keybits ) );
 4 #endif
 5
 6     for( i = 0; i < ( keybits >> 5 ); i++ )
 7     {
 8         GET_UINT32_LE( RK[i], key, i << 2 );
 9     }
10
11     switch( ctx->nr )
12     {
13         case 10:
14
15             for( i = 0; i < 10; i++, RK += 4 )
16             {
17                 RK[4]  = RK[0] ^ RCON[i] ^
18                 ( (uint32_t) FSb[ ( RK[3] >>  8 ) & 0xFF ]       ) ^
19                 ( (uint32_t) FSb[ ( RK[3] >> 16 ) & 0xFF ] <<  8 ) ^
20                 ( (uint32_t) FSb[ ( RK[3] >> 24 ) & 0xFF ] << 16 ) ^
21                 ( (uint32_t) FSb[ ( RK[3]       ) & 0xFF ] << 24 );
22
23                 RK[5]  = RK[1] ^ RK[4];
24                 RK[6]  = RK[2] ^ RK[5];
25                 RK[7]  = RK[3] ^ RK[6];
26             }
27             break;
28         /* additional cases for different key lengths */
29     }
```

Listing 8.1: An excerpt of the relevant code from the mbedTLS library that is susceptible to cache attacks. If AES-NI is not available (Line 2), mbedTLS falls back to a T-table implementation with secret-dependent key lookups in the array FSb.

8 Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels

Publication Data

Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In: USENIX Security. 2023

Contributions

Main author.

Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels

Andreas Kogler[1] Jonas Juffinger[1] Lukas Giner[1]
Lukas Gerlach[2] Martin Schwarzl[1] Michael Schwarz[2]
Daniel Gruss[1] Stefan Mangard[1]

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

Abstract

Differential Power Analysis (DPA) measures single-bit differences between data values used in computer systems by statistical analysis of power traces. In this paper, we show that the mere co-location of data values, e.g., attacker and victim data in the same buffers and caches, leads to power leakage in modern CPUs that depends on a combination of both values, resulting in a novel attack, Collide+Power. We systematically analyze the power leakage of the CPU's memory hierarchy to derive precise leakage models enabling practical end-to-end attacks. These attacks can be conducted in software with any signal related to power consumption, e.g., power consumption interfaces or throttling-induced timing variations. Leakage due to throttling requires 133.3 times more samples than direct power measurements. We develop a novel differential measurement technique amplifying the exploitable leakage by a factor of 8.778 on average, compared to a straightforward DPA approach. We demonstrate that Collide+Power leaks single-bit differences from the CPU's memory hierarchy with fewer than 23 000 measurements. Collide+Power varies attacker-controlled data in our end-to-end DPA attacks. We present a Meltdown-style attack, leaking from attacker-chosen memory locations, and a faster MDS-style attack, which leaks 4.82 bit/h. Collide+Power is a generic attack applicable to any modern CPU, arbitrary memory locations, and victim applications and data. However, the Meltdown-style attack is not yet practical, as it is limited by the state of the art of prefetching victim data into the cache, leading to an unrealistic real-world attack runtime with throttling of more than a year for a single bit. Given the different variants and potentially more practical prefetching methods, we consider Collide+Power a relevant threat that is challenging to mitigate.

1. Introduction

Power analysis attacks exploit differences in the power consumption of hardware circuits for different operations or data operands [26, 17, 2]. Most power analysis attacks use external measurement equipment on small devices such as smart cards [29, 30] to extract their internal secret information (e.g., cryptographic keys). Without physical access, an attacker can still resort to software-based side channels, exploiting, e.g., timing [15], or the cache state [42, 28]. While the attack techniques are generic, these physical and software-based side-channel attacks are typically specific to a victim application, e.g., a cryptographic implementation. Hence, for all of these side-channel attacks, there is clear guidance on how developers can mitigate them, e.g., constant-time implementations, blinding, masking, or adding randomness [24].

Meltdown [20] and MDS [34, 35, 3] can still leak secret information even when developers followed best practices on mitigations. In this sense, they are generic attacks that are not tailored to a specific algorithm but rather a CPU, meaning they can leak arbitrary data from a victim context regardless of the algorithm executed. However, Meltdown and MDS are mitigated through hardware and software patches.

Recently, software-based power side channels gained more traction [21, 38, 19, 27, 25, 33, 8, 9, 41], especially after the discovery that software-level interfaces are precise enough to mount power analysis attacks on cryptographic implementations [19]. One stop-gap solution against these attacks is making the corresponding software-level interfaces privileged (e.g., Intel RAPL). However, adaptive power management leads to constant frequency adjustments to comply with energy and heat limits. When the CPU works with data operands that consume more energy, the CPU reaches these limits more frequently. Consequently, power consumption variations translate directly into timing differences [38, 21]. Hence, software-based power side channels are still practical. Still, so far, they targeted specific applications and have not been able to demonstrate Meltdown-style or MDS-style generic leakage from arbitrary memory locations and victim contexts.

In this paper, we present Collide+Power, a novel attack showing that software-based power side channels constitute a much more fundamental and generic security threat. Our central observation is that the mere co-location of data values, e.g., attacker and victim data in buffers and caches, in modern CPUs introduces subtle but exploitable power leakage that depends on the combination of both values. This combination has several components, including e.g., the Hamming distance between attacker and victim data. Thus, we can exploit this combined leakage by varying the attacker-controlled data value and learning the precise victim value from the combined power leakage. Consequently, Collide+Power overcomes all isolation boundaries on modern systems, enabling practical attacks leaking 4.82 bit/h from other security domains.

The foundation of our end-to-end attacks is amplifying the subtle leakage signal, which is far below other components of the power consumption. For this purpose, we develop a novel differential measurement technique where each sample is based on two measurements with inverted attacker-controlled values. Due to the nature of the power leakage in the CPU's memory hierarchy, this approach cancels out unwanted noise terms and amplifies the desired signal by a factor of 8.778 on average compared to a straightforward DPA approach.

Another building block for our end-to-end attacks is a precise power leakage model of the CPU's memory hierarchy. Prior models [19, 38, 21] have not captured the subtly combined leakage we exploit and cannot be used in our attack to leak data from arbitrary memory locations. We develop precise leakage models for various attack scenarios, depending on the microarchitectural element where the attacker and victim data are *colliding*, e.g., leakage models considering L1 cache eviction when targeting L1 caches. Based on our precise models, we demonstrate that Collide+Power can even leak single-bit differences with fewer than 23 000 measurements.

In our end-to-end attacks, Collide+Power varies attacker-controlled data during differential power analysis. In this work, we demonstrate two powerful attack scenarios: The first is MDS-Power, an MDS-style attack [34, 35], leaking arbitrary data accessed by the victim (data in use), without assumptions on the algorithm run by the victim. Our end-to-end MDS-Power attack leaks 4.82 bit/h from another security domain co-located on a sibling hardware thread. The second scenario is Meltdown-Power, a Meltdown-style attack [20], leaking data at rest, with 0.136 bit/h (with amplification) from arbitrary memory locations in the kernel, using the same attack mechanism as Meltdown to interact with the cache hierarchy [36, 13]. Nevertheless, our Meltdown-Power proof-of-concept has severe practical limitations due to the state-of-the-art of prefetching data into the memory hierarchy in a real-world scenario, leading to an unrealistic attack

Table 8.1.: Collide+Power fills a significant gap, broadening software-based power analysis from attacks on specific algorithms to generic attacks like Meltdown and MDS.

Attack	**Target**	
	Software Implementation	Generic (CPU and Hardware)
Traditional Microarchitectural Side Channel	Prime+Probe [28] Flush+Reload [42] BranchScope [7]	Meltdown [20] Foreshadow [39] MDS [35, 34]
Software-based Power Side Channel	Platypus [19] Hertzbleed [38] FTS-CA [21]	**Collide+Power (our work)**

runtime of more than a year per bit with throttling. However, discovering faster ways to prefetch data into the memory hierarchy improves the leakage rates of Collide+Power.

Collide+Power is a generic attack that works on any modern CPU that co-locates attacker and victim data in the microarchitecture, e.g., caches. Thus, conceptually, it is the same step as from cache side channels to Meltdown applied to power side channels (cf. Table 8.1). Our Collide+Power attack framework is agnostic to the type of leakage traces and works with traces from the RAPL interface and timing differences without any modifications alike[1]. Our evaluation shows that an end-to-end Collide+Power attack instantiated with timing side-channel traces only requires 133.3 times more samples than direct power measurements. To facilitate the genericity and reproducibility of our results, we perform most of the evaluation and analysis with the generic RAPL interface.

We conclude that software-based power analysis attacks are more generic and extend beyond the leakage of well-known and structured attack targets from cryptographic contexts. In contrast to attacks relying on the design of microarchitectural elements such as Spectre or Flush+Reload, Collide+Power, like Rowhammer, exploits fundamental physical properties present in CPUs. Therefore, mitigating these attacks poses a much larger challenge than previous works anticipated, and fully mitigating Collide+Power, regardless of whether the victim performs side-channel hardened cryptographic or general-purpose operations, remains a significant challenge.

[1]The source code of the framework and the proof-of-concepts can be found at: `https://github.com/iaik/CollidePower`

To summarize, we make the following contributions:

1. We systematically analyze the leakage of different operations on the memory hierarchy and develop a leakage model including the attacker-victim combined leakage.
2. We present a novel and generic differential measurement technique combining multiple guess measurements per victim data value, amplifying the leakage by 8.778x.
3. We demonstrate unprivileged end-to-end Collide+Power attacks with throttling, leaking arbitrary secret data without targeting the specific algorithm the victim uses.
4. We evaluate Collide+Power in theoretical and practical end-to-end attacks and, in the more practical attacks, observe average leakage rates of 4.82 bit/h, with a success rate of 98.9 % (n=1000, $\sigma_{\bar{x}}$=0.32 %).

Outline. Section 2 provides background, and Section 3 the high-level idea. Section 4 presents the leakage analysis, and Section 5 our novel differential measurement. We discuss the implementation in Section 6, the evaluation in Section 7, and mitigations and limitations in Section 8. Section 9 concludes.

Responsible Disclosure. We disclosed our findings to Intel on November 23, 2022, and ARM and AMD on February 9, 2023. Collide+Power was assigned CVE-2023-20583 and was held under embargo until August 1, 2023. The vendors responded with advisories and guidelines to mitigate the risk.

2. Background

In this section, we present background on memory within CPUs, transient-execution, and power-analysis attacks.

2.1. The CPU's Memory Hierarchy

Modern CPUs have an internal hierarchy from small memories close to the execution pipeline to large memories such as an SRAM last-level L3 cache or even a DRAM-based victim L4 cache. Data is always served from the fastest hierarchy level that holds the data, dramatically lowering average memory access latencies. Buffers typically serve a dedicated purpose, e.g., load and store buffer (the memory order buffer) track load and store operations. Caches are the next larger storages, following a similar design across different CPUs: They are organized in n-way sets with cache-line sizes of 64 B. The smallest, L1 cache, is split between data (which we focus on) and instructions. The L2 cache is slightly larger and slower. The last-level L3 cache is significantly larger, organized in independent cache slices, and shared across cores.

Caches have been a popular target of side-channel attacks, such as Prime+ Probe [31, 28, 22]. In a Prime+Probe attack, the attacker creates and uses an *eviction set* to constantly *prime* an entire cache set. When the victim accesses a cache line mapping to the primed cache set, an attacker-controlled entry is evicted, which the attacker can observe by the access latency to its own eviction set during the subsequent priming.

When data is used, it travels through the memory hierarchy and is placed in buffers and caches, involving busses and fill buffers to transmit or temporarily store the data. Besides caches, the line-fill buffer (LFB), a temporary storage for, e.g., data loads and evictions, uncacheable, and non-temporal accesses, has been exploited in different attacks [20, 35, 34].

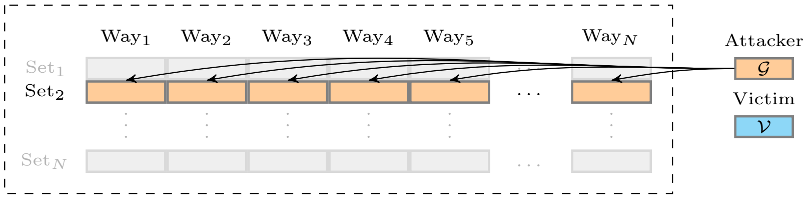
2.2. Transient-Execution Attacks

Out-of-order and speculative execution contribute to performance substantially. Instructions are retired in order, and the outcomes of predictions and faults are checked before committing the results. Mispredictions are rolled back and undone. Instructions executed out-of-order or speculatively which are never committed, are called *transient* [4, 16]. Transient execution may change the microarchitectural state, e.g., cache accesses. If these state changes depend on secret data, attackers can extract the secrets by leveraging a side channel [4, 16]. Canella et al. [4] systematized transient-execution attacks into Spectre-type attacks and Meltdown-type attacks. Meltdown-type attacks [20, 4] leverage transient execution in out-of-order execution since exceptions are raised in the retirement phase of out-of-order execution. In contrast, Spectre-type attacks [16, 4] exploit transient execution caused by speculatively executing mispredicted branches. Various attacks have been demonstrated exploiting different prediction mechanisms in modern CPUs [16, 4, 14, 10, 23]. Spectre attacks on the kernel require code snippets (gadgets) in the kernel.

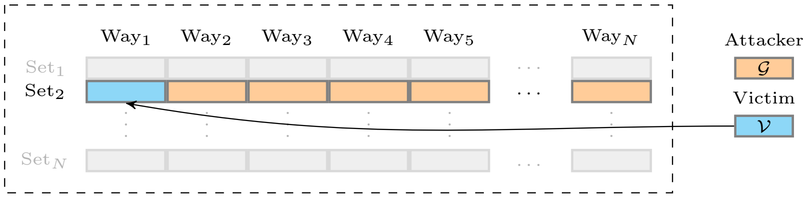
2.3. Power-Analysis Attacks

Power analysis attacks exploit differences in the energy consumption of a hardware circuit when computing with secret data. The switching behavior of CMOS circuits (the building block of CPUs), *i.e.*, the transitions between '0' and '1' bits, dictates the power consumption in a data-dependent way. There are mainly two analysis methods: First, Simple Power Analysis (SPA) [17] uses direct observations in a power trace to identify secret information, e.g., different energy signatures of secret-dependent control flow. Second, Differential Power Analysis (DPA) [17] uses statistical methods like the difference of means or correlations (then also called Correlation Power Analysis, CPA [2]) to infer secret information when SPA is insufficient. Regardless of the power analysis technique, external measurement equipment is typically used to measure energy consumption.

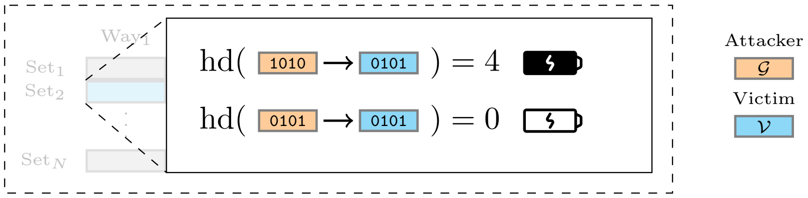
Lipp et al. [19] transformed traditional power-analysis attacks into software-based attacks on modern x86 CPUs using the Running Average Power Limit (RAPL) [11] interface. They demonstrate SPA and CPA on modern x86 CPUs and extract AES-NI and RSA keys despite the comparably low sampling rate of the software interface. In response to this attack, the RAPL interface is no longer accessible to unprivileged users. Furthermore, to protect Intel's Trusted Execution Environment (SGX), the interface no longer reports the exact energy consumption when SGX is enabled. However, as modern CPUs use adaptive power management to comply with thermal and power limits, the CPU frequency and performance depend on energy consumption. Computing on data that consumes more energy leads to more frequent throttling, observable by an unprivileged attacker, even in remotely measurable timing differences [38]. Statistical methods reduce the noise far enough to enable inference of processed data. Wang et al. [38] demonstrated software-based power side-channel attacks based on the CPU frequency as a proxy and replacement for energy consumption interfaces. Liu et al. [21] later also showed that frequency and timing could replace direct power consumption measurements. Both works leak cryptographic keys from other security domains, Wang et al. [38] even remotely across the network.



(a) **Step 1:** The attacker primes each cache line of the target cache set with the attacker-controlled guess $\mathcal{G}$.



(b) **Step 2:** The victim accesses the secret $\mathcal{V}$ and forces a cache line to change from $\mathcal{G}$ to $\mathcal{V}$.



(c) **Step 3:** The energy consumption during this change is proportional to the number of bit changes between $\mathcal{G}$ and $\mathcal{V}$.

Figure 8.1.: Collide+Power uses the attacker-controlled cache lines filled with $\mathcal{G}$ to recover the victim value $\mathcal{V}$.

3. High-Level Overview of Collide+Power

In this section, we present the high-level idea of Collide+Power. Collide+Power is a software-based power side-channel attack exploiting the fundamental design of how modern CPUs handle data. We exploit that the mere co-location in the memory hierarchy, e.g., attacker and victim data in a cache, introduces subtle but exploitable leakage in the power consumption. As illustrated in Figure 8.1 for the example of caches, an attacker can influence the power consumption with known and attacker-controlled values. When replacing a value in the cache, the power consumption of the CPU depends on the Hamming distance between old and new values stored in the cache, *i.e.*, the number of different bits between the two values. The smaller the Hamming distance between the two values, the less energy is consumed, reaching its minimum for identical values and the maximum for inverse values. Thus, an attacker varying its own data value accordingly can infer the precise victim's data value without being able to access it.

3.1. A Precise Model for DPA

Exploiting subtle power differences is challenging. Collide+Power uses DPA, and more specifically CPA, which can exploit arbitrarily small power differences as long as the number of measurements can be increased. This is the case in our attack scenario, as attacker-controlled data and victim data are co-located in the cache for an arbitrarily long time. Previous work by Lipp et al. [19] showed that repeating byte-wise loads on x86 CPUs follow the Hamming weight model. This Hamming *weight* model does not capture the leakage components that combine attacker-controlled and victim data, *i.e.*, the Hamming *distance.* However, for Collide+Power, the Hamming distance between two distinct cache line values $\mathcal{G}$ and $\mathcal{V}$ of different security domains is the relevant *signal* for the attack. Furthermore, the power consumption is much more significantly influenced by other constant and non-constant factors, outweighing the *signal* by orders of magnitude. These factors include *data-dependent noise*, *i.e.*, components depending on the attacker-controlled value $\mathcal{G}$ and the victim value $\mathcal{V}$ but not in a combined and exploitable way. These factors include *independent noise*, *i.e.*, from other processes or environmental influences. Our generalized power model (cf. Section 4) for the CPU's memory hierarchy,

$$\mathcal{P}(\mathcal{G},\mathcal{V}) \approx \underbrace{a_0 \cdot \mathrm{hd}(\mathcal{G},\mathcal{V})}_{signal} + \underbrace{w_0 \cdot \mathrm{hw}(\mathcal{G}) + w_1 \cdot \mathrm{hw}(\mathcal{V})}_{data-dependent noise} + \underbrace{\omega}_{noise}, \tag{8.1}$$

includes all of these factors as well as the Hamming weights and the Hamming distance, where the *guess* $\mathcal{G}$ is attacker-controlled, and $\mathcal{V}$ is the targeted *constant* secret *victim* value.

From Toy Example to Real-World Attack. As the power observations directly only reveal a combination of Hamming distance between $\mathcal{G}$ and $\mathcal{V}$ and their Hamming weights, an attacker needs to vary the parameter $\mathcal{G}$ to infer the exact value of $\mathcal{V}$. Figure 8.2 shows a simplified instance of this problem. By choosing guesses $\mathcal{G} = 2^i$, $i \in \mathbb{N}$ with constant Hamming

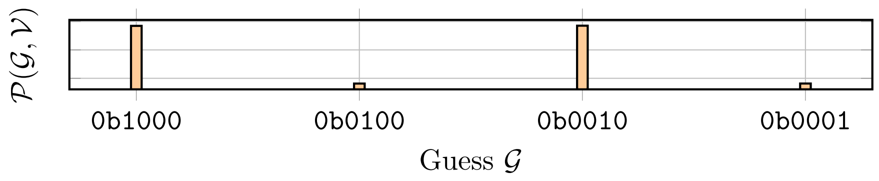


Figure 8.2.: High-level intuition for Hamming-distance-based leakage. Leaking the secret $\mathcal{V}$ by using guesses with constant $\mathrm{hw}(\mathcal{G})$ isolates $\mathrm{hd}(\mathcal{G},\mathcal{V})$ in the leakage model (Equation (8.1)). Here binary guesses ($\mathrm{hw}(\mathcal{G}) = 1$) are used to infer the inverted bits of $\mathcal{V}$ (`0b0101`) due to the changes in $\mathrm{hd}(\mathcal{G},\mathcal{V})$.

weights, *i.e.*, $\mathrm{hw}(\mathcal{G}) = 1$, the *data-dependent noise* of Equation (8.1) is constant, but the Hamming distance $\mathrm{hd}(\mathcal{G},\mathcal{V})$ changes based on $\mathcal{G}$. Thus by finding guesses that reduce $\mathcal{P}$, we can infer that the corresponding bit is also set in the secret victim value $\mathcal{V}$. While this toy example provides a great intuition of the basic idea, Collide+Power uses a more sophisticated and robust correlation-based approach.

Leaking the value $\mathcal{V}$ is a search for the maximum correlation between the model and observations. Our CPA uses the recorded samples as observations $\mathcal{O}$ and maximizes the correlation for a given model $\mathcal{M}(\mathcal{G},\mathcal{V})$. Crucial for the success rate of the CPA is the accuracy of the model and its component and the ratio between the signal and the noise (cf. Equation (8.1)), *i.e.*, the signal-to-noise ratio. Therefore, we design in-depth experiments to recover the structure in Sections 4.1 and 4.2 and the coefficients for the different components in Section 4.3.

Differential Measurement. To increase the signal-to-noise ratio further, we exploit that the power consumption can be influenced by two extremes: the *maximum* Hamming distance and the *minimum* Hamming distance. Instead of just one measurement, with one attacker-controlled value and an unknown victim value, we perform two measurements in a *differential measurement* technique (cf. Section 5). The first measurement takes a chosen $\mathcal{G}$, yielding $\mathcal{P}_{\mathcal{G}}$, and the second takes the inverted guess $\tilde{\mathcal{G}}$, yielding $\mathcal{P}_{\tilde{\mathcal{G}}}$. By using the difference between $\mathcal{P}_{\mathcal{G}}$ and $\mathcal{P}_{\tilde{\mathcal{G}}}$ as a single combined sample, this measurement technique increases the signal-to-noise ratio by a factor of 8.778 on average. It not only doubles the weight of the Hamming distance in the power leakage but also reduces other constant and non-constant factors.

3.2. End-to-End Attacks

Collide+Power is a generic attack for various scenarios and environments. The commonality is that the attacker triggers a situation where attacker-controlled values $\mathcal{G}$ and victim-controlled values $\mathcal{V}$ compete for storage in the same shared microarchitectural element. Depending on the target, this may involve cache eviction or flushing, loading of cache lines, or read and write accesses without cache interaction. Consequently, Collide+Power has no more requirements than typical cache attacks, *i.e.*, the ability to perform memory accesses. We build two end-to-end attack variants on Collide+Power:

In **MDS-Power**, we target data in use by a victim program. The victim constantly uses a secret value, *i.e.*, in a loop, which effectively keeps it in the memory hierarchy of the core (e.g., buffers like the LFB or the L1 cache). In this scenario, identical to MDS attacks, Collide+Power leaks precise secret values from a victim co-located on a sibling thread with 4.82 bit/h.

In **Meltdown-Power**, we target data at rest from arbitrary memory addresses. The attacker uses the same mechanisms as in Meltdown to pull victim data into caches, *i.e.*, leakage of data is facilitated by prefetch gadgets in the kernel [36], which are still present in kernels today [13, 40, 4]. Meltdown-Power leaks amplified data values with 0.136 bit/h in exactly the same scenario, making it a drop-in replacement for the now mitigated Meltdown attack. Finally, mounting Meltdown-Power in a real-world setting, *i.e.*, using frequency throttling attacks [38, 21], we estimate that an attacker requires 2.86 years to leak an unamplified bit from the kernel. However, this low security risk might drastically change if new architectural or microarchitectural ways of prefetching victim data in co-location with attacker-controlled data are discovered.

3.3. Threat Model

We assume the attacker runs unprivileged native code. If specific interfaces, e.g., RAPL, are unavailable [19], Collide+Power has the minimal requirement that the attacker can measure time (e.g., with `rdtscp`), serving as a proxy for the power consumption [38, 21]. Collide+Power is agnostic to the type of leakage traces and works identically with any direct or indirect power trace. Furthermore, similar to prior works on software-based power side channels [38], we either *stress* the other CPU cores, *i.e.*, a multi-threaded attack, or use the throttling effect through default or adjusted power limits, translating energy consumption into timing differences. The only additional assumption MDS-Power makes is that attacker and victim are co-located on sibling threads of a physical core, identical to the threat model of RIDL and ZombieLoad [34, 35].

Meltdown-Power has no core co-location requirement. As in Meltdown, we assume the attacker has a target address to attack [20]. When targeting the CPU's caches with Meltdown-Power, interaction with the caches is required, *i.e.*, regular memory accesses to load and evict data from L1 and L2 cache. Meltdown-Power also requires the presence of a prefetch gadget in the kernel, which Meltdown also requires for non-L1 cache data leakage [36]. Recent work confirmed that prefetch gadgets are still present in kernels today [13, 40, 4]. Like Meltdown, Meltdown-Power uses the prefetch gadget to load the victim cache line $\mathcal{V}$ into the cache. We evaluate Meltdown-Power with two different gadgets: an artificial Spectre-RSB prefetch gadget for a comprehensive evaluation and a real-world Spectre-PHT prefetch gadget to demonstrate the signal. We detail and evaluate all attacks in Sections 6 and 7.

4. Memory-Hierarchy Leakage Analysis

In this section, we analyze the power leakage of the CPU's memory hierarchy and derive a precise power leakage model. We find the general structure of the model in Section 4.1, analyze the effect of data location and bus widths in Section 4.2, and compute the precise coefficients in Section 4.3.

4.1. Determining the Structure of the Leakage

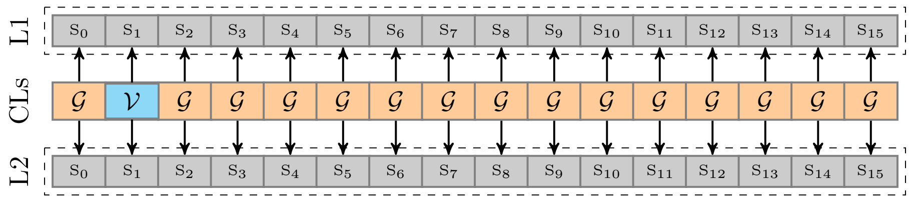
In this section, we design experiments to generate activity in certain cache levels. We analyze all pairwise combinations of *Hamming distance* and *Hamming weight* between slices of values within attacker-controlled and victim cache lines, revealing the components of the power leakage. We use an *Intel Core i7-8700K* CPU for our analysis. Section 7.1 shows that this analysis applies to a broad range of CPUs. We find three zones within a cache line that influence the leakage strength and structure, which can be represented by *Hamming distance* and *Hamming weight* expressions.

```
func record_sample(𝒢, 𝒱) -> Sample {
  fill(cache_line[0..15], 𝒢, 𝒱);
  Measurement start = measure();
  repeat (L) { access(cache_line[0..15]); }
  return measure() - start;
}
```

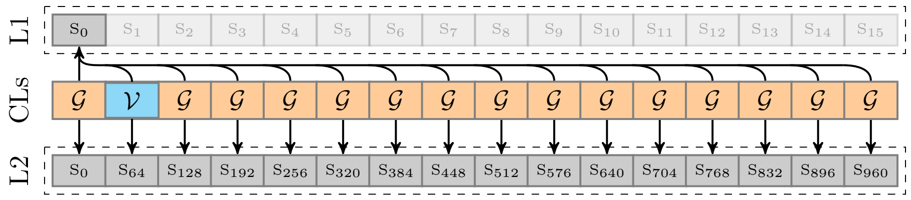
Listing 8.1: Pseudo code measuring the power consumption over a loop accessing 16 distinct cache lines.

Accessing and Filling the Cache Lines. For our analysis, we focus on two different values placed in the cache lines: the attacker controlled guess $\mathcal{G}$ and the victim value $\mathcal{V}$. Listing 8.1 shows the code used to access the cache lines with `movq` *loads*, *i.e.*, 8 B loads, and generate one sample for a randomly chosen $\mathcal{V}$ and $\mathcal{G}$. Each sample is generated by measuring the RAPL energy consumption over a tight loop running $\mathcal{L}$ iterations of 16 accesses interacting with the memory hierarchy. The number of iterations determines the measurement duration. According to our analysis, hardware prefetchers also have a minor influence on the leakage. However, we disable hardware prefetchers to derive a precise leakage model that the attacker can utilize. We detail the influences of the hardware prefetchers in Section 7.6 and emphasize that the attacks also work with hardware prefetchers enabled. MDS-Power does not trigger any prefetching as all data accesses are already cached and served from the cache (cf. Section 6). Meltdown-Power is influenced by the prefetcher loading the adjacent victim cache line, which we analyze in Section 7.6.

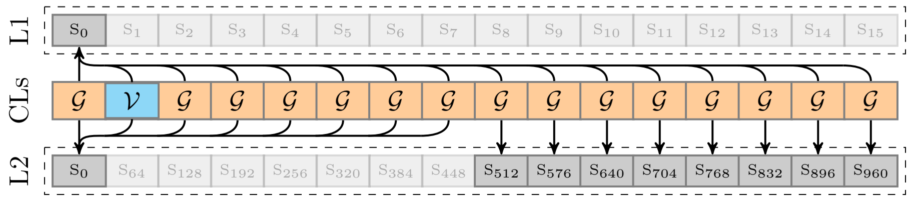
Cache Line Eviction. The i7-8700K we use for our analysis has an 8-way L1 cache and a 4-way L2 cache with pseudo-LRU replacement [1], allowing us to precisely determine the cache line to be evicted from the cache. For the 8-way L1 cache design, we use a setup with 16 distinct cache lines. Out of the 16 cache lines, we use one as the *victim* cache line filled with $\mathcal{V}$. The remaining 15 cache lines represent attacker-controlled lines, each filled with the *guess* $\mathcal{G}$. To determine the influence of accessing these 16 cache lines on the leakage model, we evaluate three eviction types with chosen L1 and L2 cache sets as shown in Figure 8.3, one without eviction, one with L1 eviction only, and one with L1 and L2 eviction: *No Eviction:* All cache lines are in different L1 sets; all 16 accesses are served from the L1 cache without self-eviction. *L1 Eviction:* All 16 cache lines are in one L1 but different L2 cache sets, resulting in constant L1 but no L2 cache eviction. *L1+L2 Eviction:* 8 cache lines are in one L1 and L2


L1
CLs
L2
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15}
$\mathcal{G}$ $\mathcal{V}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15}

(a) **No Eviction:** All cache lines are in individual L1 and L2 sets.


L1
CLs
L2
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15}
$\mathcal{G}$ $\mathcal{V}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$
S_0 S_{64} S_{128} S_{192} S_{256} S_{320} S_{384} S_{448} S_{512} S_{576} S_{640} S_{704} S_{768} S_{832} S_{896} S_{960}

(b) **L1 Eviction:** All cache lines are in the same L1 set.

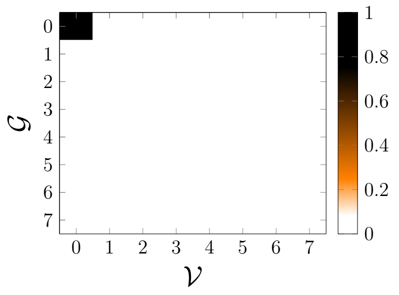

L1
CLs
L2
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15}
$\mathcal{G}$ $\mathcal{V}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$ $\mathcal{G}$
S_0 S_{64} S_{128} S_{192} S_{256} S_{320} S_{384} S_{448} S_{512} S_{576} S_{640} S_{704} S_{768} S_{832} S_{896} S_{960}

(c) **L1+L2 Eviction:** All cache lines are in the same L1 set. The first eight are in one L2 set. The remaining are in individual L2 sets.

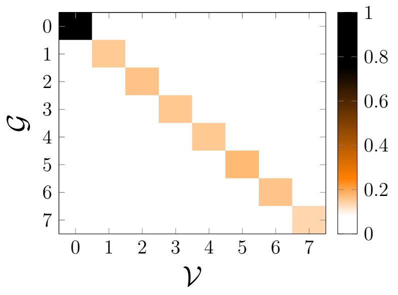
Figure 8.3.: Eviction patterns used to derive the leakage model.

cache set; the other 8 are in one L1 but unique L2 cache sets. Based on these experiments, we determine precisely how these cases influence the power leakage of the CPU.

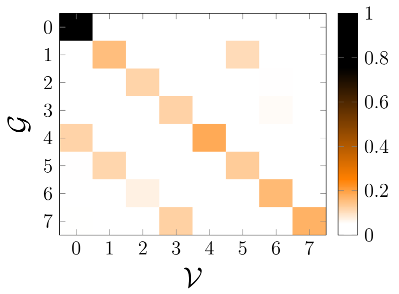
Leakage Structure Analysis. We split the values $\mathcal{V}$ and $\mathcal{G}$ into consecutive slices and determine the impact on the sliced components $\mathrm{hd}(\mathcal{V}_i, \mathcal{G}_j)$, $\mathrm{hd}(\mathcal{V}_i, \mathcal{V}_j)$, $\mathrm{hd}(\mathcal{G}_i, \mathcal{G}_j)$, $\mathrm{hw}(\mathcal{G}_i)$, and $\mathrm{hw}(\mathcal{V}_i)$ on the power consumption. We perform a linear regression with all the resulting variables and visualize the coefficients, *i.e.*, a non-zero coefficient indicates if a slice is relevant for the power leakage function. Overall, we record 24 130 228 samples with the test code (cf. Listing 8.1) for the three eviction types. Figure 8.4 shows the results for 8 B slice sizes. This indicates that the Hamming distance and weight model the leakage components very well. Visually, we see three distinct effects: First, regardless of the eviction technique the



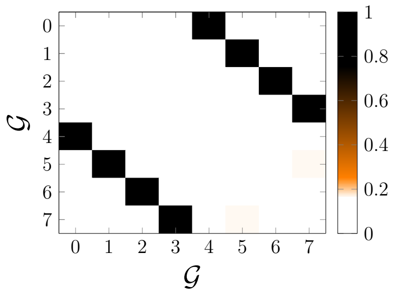
(a) **No Eviction:** hd($\mathcal{V}_i, \mathcal{G}_j$)



(b) **L1 Eviction:** hd($\mathcal{V}_i, \mathcal{G}_j$)



(c) **L1+L2 Eviction:** hd($\mathcal{V}_i, \mathcal{G}_j$)



(d) **L1+L2 Eviction:** hd($\mathcal{G}_i, \mathcal{G}_j$)

Figure 8.4.: Coefficients for the sliced components hd($\mathcal{V}_i, \mathcal{G}_j$) for all eviction techniques and hd($\mathcal{G}_i, \mathcal{G}_j$) for L1+L2 eviction. We see that the structure of the power model changes based on the eviction technique and additional *unaligned* effects.

first 8 B component hd($\mathcal{V}_0, \mathcal{G}_0$) shows a clear signal. Second, hd($\mathcal{V}_k, \mathcal{G}_k$), $k \in \{1 \text{ to } 7\}$ expose a leakage signal for the L1 and L1+L2 eviction case, albeit weaker than hd($\mathcal{V}_0, \mathcal{G}_0$). Finally, when considering L1+L2 eviction, an additional shifted signal components appear for hd($\mathcal{V}_k, \mathcal{G}_h$), hd($\mathcal{V}_k, \mathcal{V}_h$), hd($\mathcal{G}_k, \mathcal{G}_h$) with $k \in \{0 \text{ to } 7\}$ and $h = k + 4 \bmod 8$. The period of this shift is 32 B, indicating that this effect could originate from a bus-size change from 64 B to 32 B, e.g., the interconnect between L2 and L3, meaning that the two halves of the cache line are transmitted *after* each other. Combining these observations that relate to the widths with which data is moved through the memory hierarchy, we can distinguish the influence of three zones within a cache line on the power leakage model: **(1)** the bytes accessed by the `movq` instruction, **(2)** the *lower* half of a cache line, and **(3)** the *upper* half of a cache line. Reducing the slice sizes down to

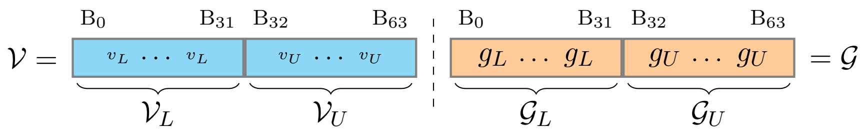


Figure 8.5.: We fill the guess value $\mathcal{G}$ and victim value $\mathcal{V}$ with repeating nibbles. We use four distinct nibbles for both the *upper* and *lower* half of each 64 B value.

single bits, *i.e.*, all possible cases, confirms that the leakage still follows the same structure, confirming the suitability of the model.

The power leakage is influenced by the bytes accessed, the lower and upper cache line half, in distinct ways that can be modeled by *Hamming distance* and *Hamming weight*.

4.2. Modelling the Leakage Function

In this section, we derive and quantify a power leakage model that predicts the power consumption of loads, stores, and evictions from different points in the cache hierarchy, depending on the data. This model is the basis of our correlation power analysis attack and our end-to-end attacks (see Section 6). We extend the initial experiments with *stores* (`movq`) and *prefetches* (`prefetcht0`) to distinguish a load that fills a register from a prefetch that only brings data into the cache. To measure the effects of *dirty* cache lines, we clear the lowest 8 bytes of a cache line, marking it as dirty. Based on our previous insights into influence factors within a cache line, we optimize the regression analysis by operating with repeating nibbles (4-bit values), reducing the runtime by several orders of magnitude. Figure 8.5 shows the structure of two 64 B values: the attacker controlled guess $\mathcal{G}$ and the victim value $\mathcal{V}$. Each value is split into the *lower* ($\mathcal{G}_L$,$\mathcal{V}_L$) and *upper* ($\mathcal{G}_U$,$\mathcal{V}_U$) 32 B parts, resembling the discovered zones. Finally, we randomly sample four nibbles, *i.e.*, v_L, v_U, g_L, and g_U, to fill $\mathcal{V} = \mathcal{V}_L|\mathcal{V}_U$ and $\mathcal{G} = \mathcal{G}_L|\mathcal{G}_U$ respectively. We consider 4-bit aligned nibbles, resulting in 128 nibbles per value. Our experiments reveal that the power model consists of four distinct leakage components depending on the cache eviction strategy used. Forwarding data to a register adds additional leakage.

The Full Leakage Model. Due to the different energy consumption of the instructions and the eviction strategies, we use the average *power*

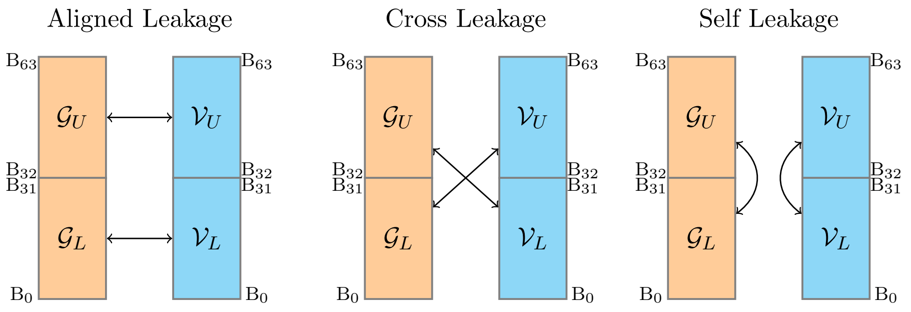


Figure 8.6.: *Aligned*, *cross*, and *self* leakage between $\mathcal{G}$ and $\mathcal{V}$.

($\mathcal{P}$) as our measurement, *i.e.*, the energy over time. This measurement compares how substantial the leakage is and optimizes for the fastest leakage method. Furthermore, we determine the influence of the Hamming distance and the Hamming weights of $\mathcal{G}$ and $\mathcal{V}$ on the average power consumption. We perform a linear regression with the model shown in Equation (8.2) to get the exact scaling factors that model the μW changes due to single-bit changes. We choose a least-squares linear regression as it minimizes the error between the noisy measurements and the model. The model is visualized in Figure 8.6 and features the following components. First, the model contains the *aligned* leakage terms (a_i), *i.e.*, the leakage between aligned bits in the cache line. Second, the model contains the *cross* leakage terms (c_i), *i.e.*, the leakage across the *lower* and *upper* parts of different cache lines. Third, we also include the *self* leakage terms (s_i), *i.e.*, the leakage between the upper and the lower parts of the same cache line. Finally, we add the Hamming weights (w_i) of each upper and lower cache-line part yielding the model

$$\begin{aligned}\mathcal{P} = &\underbrace{a_0 \cdot \mathrm{hd}(\mathcal{V}_L, \mathcal{G}_L) + a_1 \cdot \mathrm{hd}(\mathcal{V}_U, \mathcal{G}_U)}_{\text{aligned leakage}} + \underbrace{c_0 \cdot \mathrm{hd}(\mathcal{V}_L, \mathcal{G}_U) + c_1 \cdot \mathrm{hd}(\mathcal{V}_U, \mathcal{G}_L)}_{\text{cross leakage}} \\ &+ \underbrace{s_0 \cdot \mathrm{hd}(\mathcal{V}_L, \mathcal{V}_U) + s_1 \cdot \mathrm{hd}(\mathcal{G}_L, \mathcal{G}_U)}_{\text{self leakage}} \\ &+ \underbrace{w_0 \cdot \mathrm{hw}(\mathcal{V}_L) + w_1 \cdot \mathrm{hw}(\mathcal{V}_U)}_{\text{victim weight}} + \underbrace{w_2 \cdot \mathrm{hw}(\mathcal{G}_L) + w_3 \cdot \mathrm{hw}(\mathcal{G}_U)}_{\text{guess weight}}. \end{aligned} \tag{8.2}$$

Table 8.2.: The results of the linear regression, the correlation coefficients, and SNR$_A$ for different types of evictions and instructions.

		Effectiveness		Aligned Leakage		Cross Leakage		Self Leakage		Weights			
Inst.	Evict.	$\hat{\rho}$ $\cdot 1$	SNR$_A$ $\cdot 10^{-3}$	hd(v_L, g_L) a_0 in μW	hd(v_U, g_U) a_1 in μW	hd(v_L, g_U) c_0 in μW	hd(v_U, g_L) c_1 in μW	hd(v_L, v_U) s_0 in μW	hd(g_L, g_U) s_1 in μW	hw(v_L) w_0 in μW	hw(v_U) w_1 in μW	hw(g_L) w_2 in μW	hw(g_U) w_3 in μW
Load	None	0.118	6.384	159.2	3.0	0.0	2.2	0.0	2.4	0.0	0.0	173.8	0.0
	L1	0.737	2.645	219.9	108.7	0.0	0.0	0.0	0.0	186.9	90.1	3198.8	1399.1
	L1+L2	0.633	1.957	119.2	54.9	48.9	34.5	52.8	548.8	118.3	43.7	1679.5	637.4
Prefetch	None	0.003	0.001	0.0	2.5	0.0	2.1	4.3	3.2	1.6	0.1	3.2	0.0
	L1	0.191	0.861	27.1	30.1	0.0	0.0	2.8	0.0	8.7	10.1	183.6	184.8
	L1+L2	0.218	0.491	17.0	17.8	10.5	10.2	12.2	177.2	7.7	5.9	118.1	114.8
Store	None	0.003	0.001	1.5	0.6	0.5	0.0	0.0	0.0	5.4	0.0	4.8	0.1
	L1	0.103	0.376	18.6	21.4	1.5	0.9	3.4	45.3	6.4	11.8	78.9	115.5
	L1+L2	0.280	0.644	64.5	83.7	42.0	41.2	89.1	946.0	21.7	66.7	188.1	630.4

4.3. Quantifying the Leakage Function

Table 8.2 shows the results of the regression analysis (cf. Section 4.2). The results indicate that eviction strategy and access instruction influence the power leakage model of the cache hierarchy. Furthermore, the correlation and model coefficients change based on the data placement. The results account for the repeating nibbles and show the coefficients for single nibbles, allowing us to quantify and compare the bit leakage in μW. Furthermore, we introduce the aligned signal-to-noise ratio (SNR$_A$) between the *aligned* leakage and all other components, *i.e.*, the noise, for simple comparison across techniques. This is not the overall signal-to-noise ratio of the model but rather the part important for our attacks.

Instructions and Data Placement. In line with our regression analysis (cf. Section 4.1), we observe that the position of the data and the instructions used to access the data influence the model coefficients. First, using no eviction never yields a significant signal for the *upper* cache line parts $\mathcal{V}_U$ and $\mathcal{G}_U$, as data is never moved between L1 and L2 cache. Furthermore, for prefetch with no eviction, we never observe any significant signal for any of the cache line parts, as no part of the cache line is moved into a register, and prefetch moves no data when it is already present in the L1 cache. For the store instruction, we overwrite the lower 8 B with zeros. We observe no signal with no eviction, as the dirty cache line never leaves the cache, and no actual victim or guess nibbles are stored. Therefore, we conclude that explicitly loading data introduces observable leakage, e.g., 159.2 µW per bit difference for our test CPU.

The model coefficients are influenced by the instruction performing the access and from where to where data is transmitted in the CPU's internal memory hierarchy.

Cache Eviction. Model coefficients are influenced by different types of cache eviction, influencing the aligned signal-to-noise ratios of the techniques. The SNR_A for load instructions decreases by a factor of 2.413 when switching from no eviction to L1 eviction. We see a further decrease of factor 1.351 when switching from L1 to L1+L2 eviction, similar to the factor of 1.753 when using prefetch instructions. However, for store instructions, we see an increase of factor 1.712 in the aligned signal-to-noise ratio when performing L1+L2 eviction. We conclude that the mechanism to *write back* dirty cache lines add additional leakage (which we exploit in Section 7.5). Overall, we observe that for higher cache eviction activity, the correlation coefficients are increasing, indicating that the overall model represents the power consumption more accurately, but the desired signal is not increasing as strongly.

The model is more accurate with more cache eviction. However, the aligned signal-to-noise ratio decreases for the load and prefetch instructions, whereas for store instructions, it increases when using additional evictions.

Memory Bus Effects. The results in Table 8.2 support the leakage patterns from our regression analysis (cf. Section 4.1). First, we observe that *aligned leakage*, *i.e.*, coefficients a_0 and a_1, is present regardless of the eviction used. This is fundamental for our two attack variants MDS-Power and Meltdown-Power (cf. Section 6). When using L1+L2 eviction we observe additional effects: *cross-leakage* and *self-leakage* (cf. Figure 8.6). The self-leakage of the attacker-controlled guess in the case of the load instructions is a factor of 4.604 times stronger than the aligned leakage, 9.955 times for prefetches, and 11.302 times for stores, respectively. However, our novel differential measurement technique removes all the influences of self-leakage in the signal, as we demonstrate in Section 5. Finally, we observe cross-leakage between the upper and lower parts of $\mathcal{V}$ and $\mathcal{G}$. In all cases, the cross-leakage terms c_0 and c_1 are smaller than the aligned leakage terms. Therefore, the main influence in the CPA is still the aligned leakage.

L1+L2 eviction adds additional leakage effects. The self-leakage of the attacker-controlled guess overshadows all other components. However, we address this with our differential measurement technique. The aligned leakage terms outweigh the cross-leakage terms.

We conclude that an attacker has several different ways to influence and model the leakage. By using memory accesses, cache loads and stores,

Table 8.3.: The coefficients and statistics of the differential measurement technique for different types of evictions and instructions.

Inst.	Evict.	Effectiveness		Aligned Leakage		Cross Leakage		Self Leakage		Weights			
		$\hat{\rho}$ $\cdot 1$	SNR$_A$ $\cdot 10^{-3}$	hd(v_L, g_L) a_0 in μW	hd(v_U, g_U) a_1 in μW	hd(v_L, g_U) c_0 in μW	hd(v_U, g_L) c_1 in μW	hd(v_L, v_U) s_0 in μW	hd(g_L, g_U) s_1 in μW	hw(v_L) w_0 in μW	hw(v_U) w_1 in μW	hw(g_L) w_2 in μW	hw(g_U) w_3 in μW
Load	None	0.311	72.004	544.5	4.2	1.1	0.5	0.0	0.0	0.0	0.0	362.6	0.0
	L1	0.907	7.873	598.3	278.8	0.0	0.0	0.0	0.0	0.0	0.0	6124.4	2696.9
	L1+L2	0.822	5.632	339.3	141.7	106.6	89.4	0.0	0.0	0.0	0.0	3750.7	1435.0
Prefetch	None	0.003	0.000	0.0	0.8	0.0	5.7	0.0	0.0	0.0	0.0	1.7	2.8
	L1	0.370	11.365	136.7	133.9	1.9	0.1	0.0	0.0	0.0	0.0	454.1	455.5
	L1+L2	0.300	5.294	80.5	86.9	40.9	43.0	0.0	0.0	0.0	0.0	334.0	332.5
Store	None	0.003	0.000	0.0	0.0	0.0	3.1	0.0	0.0	0.0	0.0	7.0	0.0
	L1	0.241	3.876	63.3	74.5	4.9	9.6	0.0	0.0	0.0	0.0	204.6	303.2
	L1+L2	0.450	6.457	133.7	169.0	84.7	86.2	0.0	0.0	0.0	0.0	347.1	1130.5

and eviction, the attacker can precisely control and optimize the leakage rate for victim data in different locations in the CPU's internal memory hierarchy.

5. Differential Measurement

As outlined in Section 3.1, increasing the signal-to-noise ratio is crucial to make Collide+Power practical. We propose a *differential* measurement technique to eliminate some noise influences, amplify the leakage, and reduce the required samples. We exploit that the power consumption can be influenced by two extremes: the maximum and minimum Hamming distance between attacker and victim value. However, the measurements are affected by additive noise ω that, based on our analysis, is near constant between temporally close samples.

Masking Victim Data. An attacker cannot influence the victim data value $\mathcal{V}$ in a real-world scenario. Therefore, we introduce the mask m, a 64 B value, indicating which bits of $\mathcal{G}$ should be inverted; all remaining bits are unchanged. We measure one sample $\mathcal{P}_\mathcal{G}$ for the guess $\mathcal{G}$ and directly afterward the sample $\mathcal{P}_{\tilde{\mathcal{G}}}$ with the inverse guess $\tilde{\mathcal{G}} = \mathcal{G} \oplus m$ and subtract the two samples $\Delta\mathcal{P}(\mathcal{G}, \tilde{\mathcal{G}}) = \mathcal{P}_\mathcal{G} - \mathcal{P}_{\tilde{\mathcal{G}}}$. The mask m selects only a fraction of the cache line of $\mathcal{V}$ for the differential measurement, reducing the search complexity in a divide-and-conquer-style approach. The differential measurement changes the simplified leakage model of

$$\mathcal{P}(\mathcal{G}, \mathcal{V}) = a_0 \cdot \mathrm{hd}(\mathcal{G}, \mathcal{V}) + w_0 \cdot \mathrm{hw}(\mathcal{V}) + w_2 \cdot \mathrm{hw}(\mathcal{G}) + \omega, \tag{8.3}$$

by subtracting the model for $\mathcal{P}(\tilde{\mathcal{G}}, \mathcal{V})$ to

$$\Delta\mathcal{P} = 2a_0 \cdot \mathrm{hd}(\mathcal{G}_m, \mathcal{V}_m) + 2w_2 \cdot \mathrm{hw}(\mathcal{G}_m) - (a_0 + w_2) \cdot \mathrm{hw}(m), \tag{8.4}$$

where $\mathcal{G}_m = \mathcal{G} \wedge m$ and $\mathcal{V}_m = \mathcal{V} \wedge m$ are the masked cache lines with which we mask non-targeted data. The noise term ω and the unknown but constant value $\mathrm{hw}(\mathcal{V})$ cancel out. Furthermore, subtracting $\mathrm{hd}(\tilde{\mathcal{G}}, \mathcal{V})$ from $\mathrm{hd}(\mathcal{G}, \mathcal{V})$ yields a 2x amplification and a constant offset $\mathrm{hw}(m)$. Finally, $\mathrm{hw}(\mathcal{G}) - \mathrm{hw}(\tilde{\mathcal{G}})$ also results in a 2x amplification and another offset $\mathrm{hw}(m)$. Thus, the final model amplifies the leakage by a factor of 2 and eliminates some additive noise. We show the full derivation of Equation (8.4) in Section 10.1.

Quantifying the Differential Measurements. Table 8.3 shows the results with our differential measurement technique. We see that the derived differential model $\Delta\mathcal{P}(\mathcal{G}, \tilde{\mathcal{G}})$ in Equation (8.4) holds, and we gain a twofold amplification in a_0, a_1, w_2, and w_3 compared to Table 8.2. We can also see that w_0 and w_1, *i.e.*, the influence of $\mathrm{hw}(\mathcal{V})$, are reduced to 0. Similarly, using L1+L2 eviction completely removes the self-leakage effects s_0 and s_1. The correlation coefficients between our model and the actual measurements increase up to 0.907 in the case of loads with L1 eviction, *i.e.*, a 23 % increase, indicating that this significantly reduces the overall noise ω of the measurements. Finally, we also see that the aligned signal-to-noise ratio is increased by up to a factor of 11.27 (8.778 on average). Therefore, the measured results support our derivation of the differential model and its properties.

Our differential measurement model amplifies the signal by a factor of 2 and eliminates a significant part of the additive noise and self-leakage effects.

CPA Model Coefficients. In this section, we discuss how to minimize profiling for the CPA model used for Collide+Power (cf. Section 3). We derive the leakage structure and coefficients in Equation (8.2) and Table 8.3. As these exact coefficients require additional time to profile, we discuss three different approaches. First, we can profile the coefficients for the target CPU once and use a full model of Equation (8.2). An attacker could target its own cache lines to obtain these coefficients. Second, we try to approximate the coefficients without profiling the target. The correlation coefficient $\hat{\rho}$ is scaling and location invariant, meaning that linear scaling $\cdot a$ and offsets $+b$ of the model do not influence the CPA attack. Due to the scaling invariance, we only need the ratio between some of the coefficients based on the used setup, e.g., w_2/a_0. If we consider Table 8.3, we observe that this ratio can be approximated based on the attack technique used. The ratio is approximately 0.7 for no eviction, 10.2 for L1 cache eviction, and 11.1 for L1+L2 cache eviction when using load instructions. The resulting model, then, is

$$\mathcal{M}(\mathcal{G}_m, \mathcal{V}_m) = \mathrm{hd}(\mathcal{G}_m, \mathcal{V}_m) + \tfrac{w_2}{a_0} \cdot \mathrm{hw}(\mathcal{G}_m). \tag{8.5}$$

Finally, we fix the attacker-controlled parameter $\mathrm{hw}(\mathcal{G}_m)$ to a constant value simplifying the differential model to

$$\mathcal{M}(\mathcal{G}_m, \mathcal{V}_m) = \mathrm{hd}(\mathcal{G}_m, \mathcal{V}_m). \tag{8.6}$$

Due to the mask m, and the differential measurement, we can partition the brute-force approach of recovering $\mathcal{V}$ into smaller problems by only recovering $\mathcal{V}_m$. The attacker can choose an arbitrary mask m and, according to Equation (8.4), only the selected bits will produce a measurable Hamming distance $\mathrm{hd}(\mathcal{G}_m, \mathcal{V}_m)$ because the unmasked parts cancel out. We further verify that the unmasked portions do not influence the CPA success probability in Section 7.2.

6. End-to-End Attack Implementation

In this section, we describe the implementation of our two Collide+Power end-to-end attacks: MDS-Power and Meltdown-Power. While Meltdown-type attacks are mitigated in recent CPU generations, Collide+Power forms a drop-in replacement, achieving leakage rates of 4.82 bit/h in the MDS-Power case and 0.136 bit/h in the amplified Meltdown-Power case. The MDS-Power variant targets data-in-use, whereas the Meltdown-Power variant targets data-at-rest. Generally, Collide+Power is agnostic to the power side channel used. Depending on the system configuration and hardware, high-accuracy channels like Intel RAPL may be available to the attacker. However, for our end-to-end attacks, we rely on timing-based power side-channel attacks, which are not mitigated on x86 systems and, thus, can be mounted by an unprivileged attacker, as we confirm in our evaluation (cf. Section 7.4).

6.1. MDS-Power Implementation

MDS-Power follows the exact scenario and threat model of MDS attacks like RIDL and ZombieLoad [34, 35] (cf. Section 3.3). Listing 8.2 shows the

```
while (true) {
  access(&victim_cache_line);
}
```

Listing 8.2: In the RIDL PoC, a victim program frequently accesses the victim cache line $\mathcal{V}$. Collide+Power extracts the accessed data without relying on any MDS vulnerabilities.

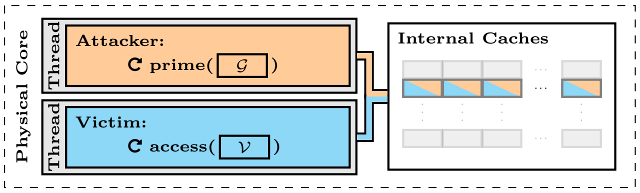


Figure 8.7.: MDS-Power: Attacker and victim constantly reload $\mathcal{G}$ and $\mathcal{V}$ respectively while being co-located on hyperthreads.

RIDL PoC[2] where the victim program accesses a cache line in a loop while being co-located with the attacker on the same physical core. Although a real-world victim program is unlikely to perform secret accesses in a loop, it offers a fair comparison of MDS-Power with RIDL and ZombieLoad as they use similar victim programs. Intel fixed this MDS hardware flaw in the 9th CPU generation. With MDS-Power, we demonstrate MDS-style leakage without relying on any hardware MDS vulnerability.

The basic setup of MDS-Power is illustrated in Figure 8.7. MDS-Power exploits that due to the victim's own load, the victim's secret value constantly moves through the CPU's memory hierarchy, e.g., in and out of internal buffers. The attacker simultaneously repeatedly loads guesses $\mathcal{G}$, which then move through the same parts of the CPU's memory hierarchy. MDS-Power then instantiates Collide+Power either with direct (e.g., the Intel RAPL interface if available) or indirect (e.g., via timing differences due to throttling) power side-channel leakage. MDS-Power is evaluated in Section 7.3.

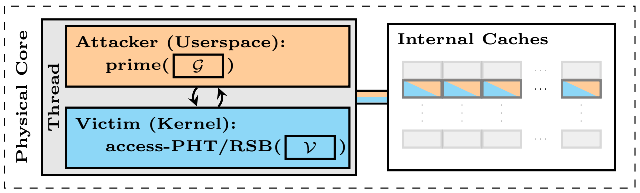


Figure 8.8.: Meltdown-Power: The attacker primes the cache with $\mathcal{G}$ and uses an RSB or PHT gadget in the kernel to load $\mathcal{V}$. The address of $\mathcal{V}$ can be any kernel address.

6.2. Meltdown-Power Implementation

Meltdown-Power follows the scenario and threat model of the original Meltdown attack [20], leaking arbitrary kernel data from userspace (cf. Section 3.3). Like Meltdown on non-L1 data, Meltdown-Power depends on the victim code to load the data, possibly triggered by the attacker. We use a Spectre prefetch gadget inside the kernel [36]. They are more widespread [4, 13] than regular Spectre gadgets, as they only load data but do not leak it. The attacker then alternatingly primes the cache with 15 distinct cache lines filled with the guess $\mathcal{G}$ (cf. Section 4) and then reloads the victim cache line $\mathcal{V}$ using a prefetch gadget, as shown in Figure 8.8. We evaluate Meltdown-Power with two prefetch gadgets, an artificial one for an in-depth evaluation and a real-world gadget to show the practicality of Meltdown-Power.

Artifical Spectre-RSB Prefetch Gadget. We start our evaluation with the Spectre-RSB gadget shown in Listing 8.3, located in the `ioctl` entry function of a kernel module. During misspeculation, it dereferences `rdx` containing an attacker-controlled pointer to the victim cache line $\mathcal{V}$. Similar to the retpoline mitigation [37], the gadget first calls a function (Line 3), which in Lines 8 and 9, modifies the return address on the stack to point to the `ioctl` return. Then, the CPU misspeculates the return address and transiently accesses the victim cache line $\mathcal{V}$ (Line 5). This brings $\mathcal{V}$ into the cache, exposing it to Collide+Power. Identical to the prefetch gadget exploited by Meltdown [36], this gadget can fetch any kernel-accessible memory location into the cache, including all physical memory, via the direct-physical map.

[2]RIDL PoC source code: `https://github.com/vusec/ridl/blob/be77e2bd16df8a1ec78c4bf9b82912c230971dc2/pocs/ridl_basic.c`

```
# rdx = &victim_cache_line
module_ioctl:
    call retpoline(%rip)
misspeculation:
    mov (%rdx), %rax
    ud2
retpoline:
    lea retpoline_end(%rip), %rax
    mov %rax, (%rsp)
    ret
retpoline_end:
    xor %eax, %eax
    ret
```

Listing 8.3: The artificial Spectre-RSB prefetch gadget in a kernel module used for the first Meltdown-Power evaluation.

Real-World Spectre-PHT Prefetch Gadget. To demonstrate the practicality of Meltdown-Power, we also evaluate Meltdown-Power with a real-world Spectre prefetch gadget in Linux kernel 5.14, in the function `find_keyring_by_name` of the `KEYCTL_JOIN_SESSION_KEYRING` syscall, discovered by Johannesmeyer et al. [13]. Section 10.2 shows the relevant part of the code and details how misspeculation causes type confusion to bring victim data into the CPU's memory hierarchy to expose it to Collide+Power.

For Collide+Power, we need to *minimize* the activity in the kernel to obtain a high signal-to-noise ratio. Instead of using a single process rapidly creating new keyrings, we use child processes in the same namespace that hold their keyrings for a longer time frame. This *avoids* the frequent clean-up operations for unused keyrings in the kernel, minimizing kernel activity. With the number of child processes, we control the loop iterations, *tuning* the mistraining of the branch prediction. By naming all keyrings differently, we *reduce* the operations performed within the loop by continuing it early. The last keyring in the iterated list satisfies all conditions to join, greatly *reducing* the code executed in `KEYCTL_JOIN_SESSION_KEYRING` after the misspeculation. Thus, we minimize the noise substantially compared to prior work, enabling us to use this real-world prefetch gadget in an end-to-end Meltdown-Power attack. Meltdown-Power is evaluated in Section 7.5.

Table 8.4.: Evaluation of the leakage model for Collide+Power on different CPUs and microarchitectures.

	CPU	**Microarchitecture**	**Microcode**	**Stepping**	**Release Year**	**L1 Ways**	**L2 Ways**	$\hat{\rho}$	**SNR**$_A \cdot 10^{-3}$
Intel	Core i5-2520M	Sandy Bridge	0x1b	7	2011	8	8	0.011	0.107
	Core i3-7100T	Kaby Lake	0xec	9	2017	8	4	0.024	0.416
	Xeon E-2176M	Coffee Lake	0xf0	10	2018	8	4	0.820	17.997
	Core i7-10510U	Comet Lake	0xc6	12	2019	8	4	0.549	7.188
	Core i7-10710U	Comet Lake	0xe0	0	2019	8	4	0.130	0.008
	Core i7-1185G7	Tiger Lake	0x72	1	2020	12	20	0.728	7.643
	Core i9-12900K	Alder Lake	0xf	2	2021	12	10	0.352	3.513
	Core i9-9900	Coffee Lake	0xd6	12	2019	8	4	0.725	10.992
	Core i7-8700K	Coffee Lake	0xf0	10	2017	8	4	0.907	72.004
	Core i9-9980HK	Coffee Lake	0xaa	13	2019	8	4	0.799	79.132
AMD	Ryzen 5 2500U	Zen	0x810100b	0	2017	8	8	0.428	5.429
	Ryzen 5 3550H	Zen+	0x8108102	1	2019	8	8	0.585	3.025
	EPYC 7252	Rome	0xa50000c	0	2019	8	8	0.160	0.178
	Ryzen 9 5900HX	Zen 3	0x8301055	0	2021	8	8	0.650	7.269

7. Evaluation

In this section, we evaluate Collide+Power on multiple CPUs to demonstrate that cache-hierarchy leakage is a widespread problem. Furthermore, in an end-to-end scenario, we demonstrate MDS-Power leaking data from the sibling thread with both the RAPL interface and throttling side channels. Finally, we evaluate Meltdown-Power in one artificial setting to demonstrate that we can extract single bits from the kernel and measure an amplified real-world prefetch gadget to verify that we observe a signal useable for Collide+Power.

7.1. Affected CPUs

We systematically analyze on which CPUs the differential model Equation (8.4) of Collide+Power works. We perform the same experiments as in Section 5 and report the maximum observed correlation coefficient and the maximum SNR$_A$, *i.e.*, the significance of the aligned Hamming distance leakage, which is the foundation of Collide+Power. Table 8.4 shows our result on 14 CPUs from both AMD and Intel, spanning a release period from 2011 until 2021, for nearly all of which we can demonstrate to be affected by Collide+Power. We observe that 12 out of 14 CPUs have a maximum correlation coefficient above 0.1, reaching 0.907 on the Intel Core i7-8700K (cf. Table 8.3). Furthermore, we found a maximum signal-to-noise ratio of $79.132 \cdot 10^{-3}$ on the Intel Core i9-9980HK, which is 9.9 % stronger than the Intel Core i7-8700K. For the CPUs with lower

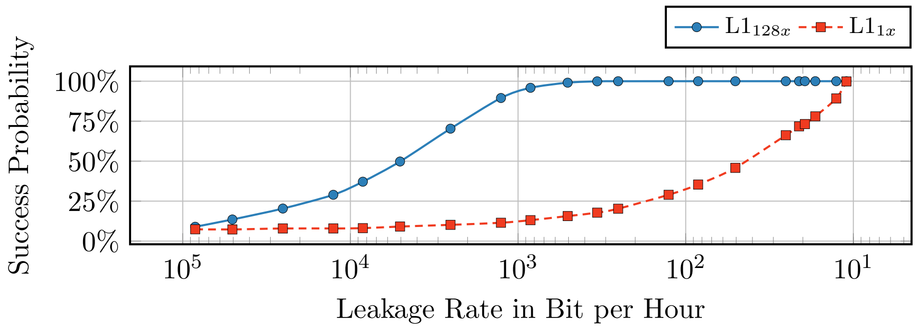


Figure 8.9.: The CPA success probability for the raw channel when using loads to evict the L1. We target both amplified (128x) and single nibble (1x) victim values. The probability increases with a lower leakage rate, *i.e.*, with more samples.

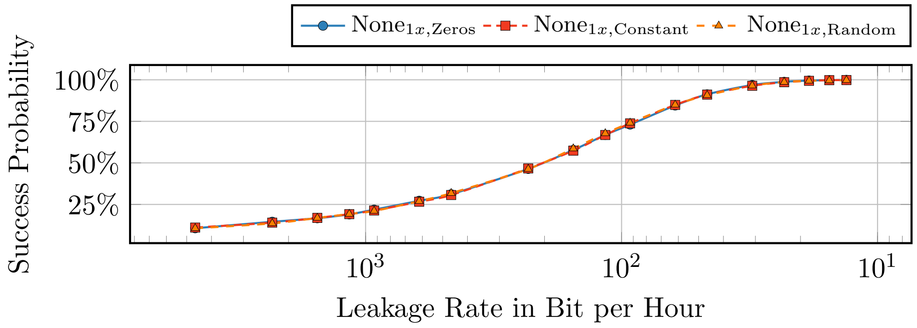


Figure 8.10.: The CPA success probability is not influenced by the non-targeted data in the victim cache line. We target the unamplified (1x) victim nibble and fill the remaining parts with zero, constant, and randomly changing data.

metrics, we cannot exclude that a slightly different cache eviction (cf. Figure 8.3) could increase leakage. Finally, we conclude that Collide+Power is widespread due to how we build and design CPU memory hierarchies.

7.2. Evaluation of the Model for the Channel

Our evaluation of the CPA models using our differential measurement shows the general capabilities of Collide+Power. We focus on load instructions with L1 cache eviction in two settings. First, we fill the complete values $\mathcal{V}$ and $\mathcal{G}$ with the repeating nibbles v and g, respectively (128x).

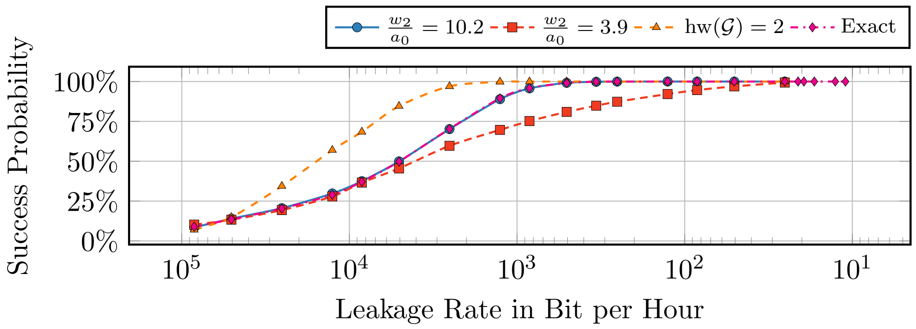
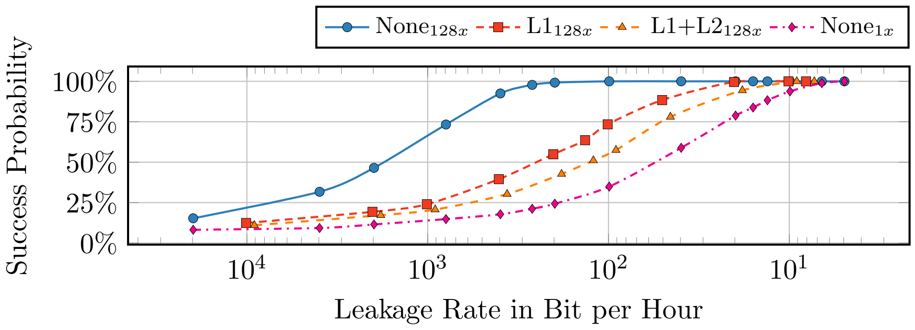


Figure 8.11.: Comparison between different CPA models and the influence of the wrong ratio on the success probability. The wrong ratio requires reducing the leakage rate by a factor of 20 to achieve the same success rate as the correct ratio.

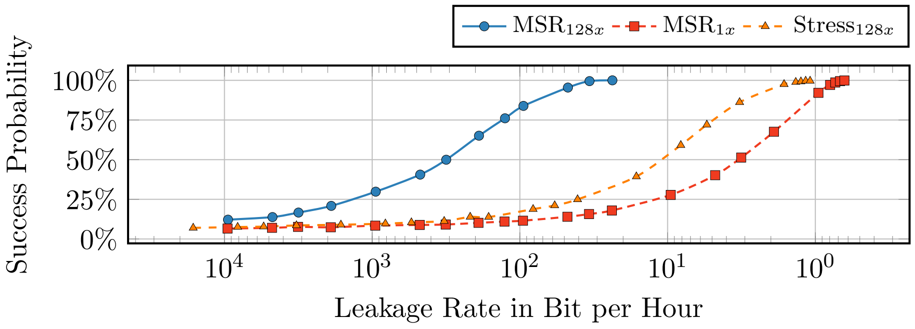
Second, we use a single nibble v and g and zero the remaining parts, representing no-amplification (1x). To compute the CPA success probability (cf. Figure 8.9), we repeatedly take a specific number of random samples from our measured samples, perform the CPA, and compute how often v was recovered without any bit errors. The number of samples used determines the leakage rate. In the amplified case, we can recover 505.81 bit/h of v with a success probability of 99.0 % (n=1000, $\sigma_{\bar{x}}$=0.31 %). Without amplification, we still achieve a leakage rate of 10.99 bit/h with 99.8 % (n=1000, $\sigma_{\bar{x}}$=0.14 %), corresponding to 23 000 differential measurements per nibble.

The differential measurement technique allows the masking of specific data within a cache line. To verify that the differential measurement is unaffected by the unmasked data, we perform an experiment that targets the unamplified (1x) victim nibble v in three distinct scenarios: We fill the remaining parts of the cache line with either zero, random data that stays constant, or random data that changes between measurements and compute the influences on the CPA success probability. We use `movq` *loads* with the *no eviction* pattern (cf. Figure 8.3) for this analysis and show in Figure 8.10 that the unmasked victim data does not influence the CPA success probability. This matches the derivation of Equation (8.4) where the unmasked terms cancel out. We discuss changing the masked data as mitigation against Collide+Power in Section 8.

In Figure 8.11, we compare coefficients of the CPA model (cf. Section 5) based on the amplified (128x) L1 eviction experiment from the first



(a) MDS-Power using RAPL. The *no-eviction* technique works best.



(b) MDS-Power using the timing-based throttling side channel and the *no-eviction* technique. MSR uses a reduced power limit. Stress uses stressors on the other CPU cores to limit the power budget.

Figure 8.12.: **MDS-Power** probability to leak the nibble v error-free for different strategies and measurement methods.

paragraph. First, when only using samples with fixed Hamming weight ($\text{hw}(\mathcal{G}) = 2$), we achieve a 2.5 times higher leakage rate than the exact model to achieve a success rate of 99.8 % (n=1000, $\sigma_{\bar{x}}$=0.14 %). Second, fixing the ratio $^{w_2}/_{a_0}$ to 10.2 does not result in an observable difference to the exact model coefficients. Finally, using the incorrect ratio $^{w_2}/_{a_0}$ of 3.9 requires reducing the leakage rate 20 times to reach a success rate of 99.3 % (n=1000, $\sigma_{\bar{x}}$=0.26 %). Therefore, we conclude that small inaccuracies in the model coefficients can be compensated with more samples.

7.3. Evaluation of MDS-Power

We evaluate MDS-Power on an Intel Core i9-9900K CPU with Ubuntu 20.04.5 LTS and Linux kernel 5.4. To compare the leakage of the different approaches, we report the leakage rate in bits per hour with the CPA success probability of recovering the victim nibble v without errors. We disabled the hardware prefetchers during this experiment. However, as we discuss in Section 4.1, the best attack scenario is unaffected by hardware prefetchers. Figure 8.12a compares the leakage for different eviction strategies when using load instructions to access the *guess* cache lines, measured with RAPL. We observe that the *no eviction* technique achieves a leakage rate of 188.80 bit/h and recovers the amplified nibble (128x) with 99.2 % (n=1000, $\sigma_{\bar{x}}$=0.28 %) success rate. *No eviction* achieves 9.33 times the leakage rate of the L1 cache eviction and 18.07 times the leakage rate of the L1+L2 cache eviction. When targeting a single *unamplified* nibble (1x), *no eviction* achieves a leakage rate of 4.82 bit/h with a success rate of 98.9 % (n=1000, $\sigma_{\bar{x}}$=0.32 %).

Figure 8.12b evaluates the *no-eviction* technique with throttling attacks [21, 38]. With adaptive power management [11], the CPU regulates its frequency and voltage to meet its power targets. However, the time-stamp counter frequency, read by `rdtscp`, is fixed, and therefore, throttling-based attacks can be mounted with this simple primitive. We evaluate the two distinct methods to achieve frequency throttling, as described in Section 3.3. First, we set the power limit of the CPU to 5.625 W over 0.977 ms in the `MSR_PKG_POWER_LIMIT` [12]. We observe that when using the MSR to set the power limits, the *no-eviction* technique achieves a leakage rate of 33.78 bit/h with a success probability of 99.5 % (n=1000, $\sigma_{\bar{x}}$=0.22 %) leaking the amplified nibble compared to the leakage rate of 0.68 bit/h with a success probability of 99.5 % (n=1000, $\sigma_{\bar{x}}$=0.22 %) when targeting the unamplified nibble.

Second, we run the *stress* program on the remaining logical threads reducing the available thermal and energy budget. We achieve a leakage rate of 1.16 bit/h with a success probability of 99.6 % (n=1000, $\sigma_{\bar{x}}$=0.19 %) in the amplified and a leakage rate of 0.065 bit/h with a success probability of 95.3 % (n=1000, $\sigma_{\bar{x}}$=0.66 %) for the unamplified case, respectively. We summarize the results of MDS-Power in Table 8.5. We conclude that MDS-Power can extract secret information with throttling side channels, albeit with a strongly reduced leakage rate compared to RAPL.

Table 8.5.: Summary of MDS-Power using load instructions for the different measurement variants and eviction techniques.

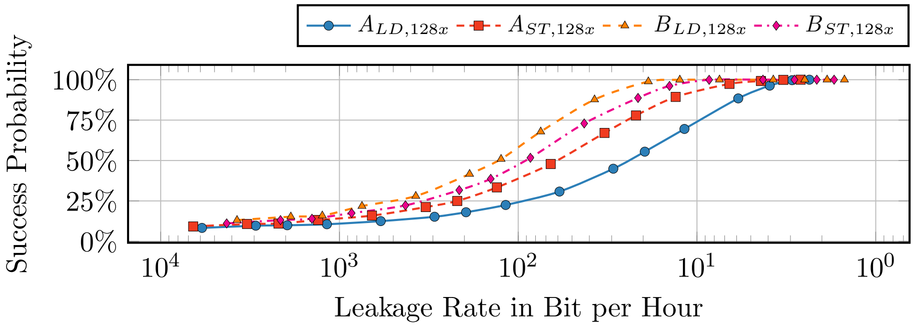
	Eviction	Ampl.	Leakage Rate $\cdot 1$ bit/h	Measurement Duration $\cdot 1$ ms	Samples
RAPL	None	128x	188.80	36.3	1050
	L1	128x	20.24	71.3	4998
	L1+L2	128x	10.45	79.2	8705
	None	1x	4.82	36.3	40 000
Limit	None	128x	33.78	76.1	2800
	None	1x	0.68	75.6	140 000
Stress	None	128x	1.16	44.2	140 000
	None	1x	0.065	44.5	2 500 000

Table 8.6.: Comparison between the model coefficients for none-evicting loads for RAPL and throttling side channels.

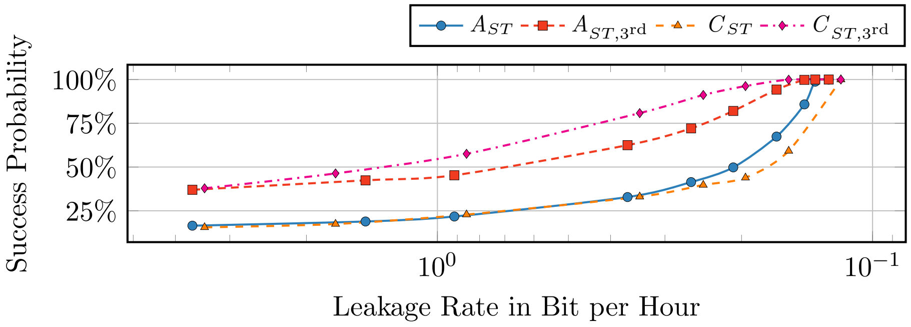
	$\hat{\rho}$	SNR_A	$\mathrm{hd}(\mathcal{V}, \mathcal{G})$	$\mathrm{hw}(\mathcal{V})$	$\mathrm{hw}(\mathcal{G})$
Interface	$\cdot 1$	$\cdot 10^{-3}$	a_0	w_0	w_2
RAPL	0.378	39.5	275.0 µW	14.7 µW	453.5 µW
MSR	0.189	13.3	1840.5 ns	−70.8 ns	2390.3 ns
Stress	0.029	0.3	62.2 ns	−0.9 ns	87.4 ns

7.4. Collide+Power through Power and Time

We verify that the leakage models derived in Section 5 holds when exchanging the average power measurements with timing measurements. Table 8.6 shows the correlation coefficients, the aligned signal-to-noise ratio, and the model coefficients for the RAPL interface, throttling attacks via the energy limit MSR, and throttling via stress (cf. Section 3.3). We denote that the unit changes from Watt to Seconds due to the different measurements. We reuse the recorded data of the MDS-Power experiment on our Core i9-9900 CPU (cf. Section 7.3). In line with the differential leakage model, we observe that the $\mathrm{hw}(\mathcal{V})$ component is minimal compared to the other components due to the differential measurements. Furthermore, we see a decrease in the correlation coefficient of factor 2 when switching from RAPL to the power limit MSR and a decrease of 13.034 when switching from RAPL to stress. We conclude that although the signal-to-noise



(a) Meltdown-Power using the artificial Spectre-RSB gadget on CPUs A and B with either loads (LD) or stores (ST).



(b) Meltdown-Power using the real-world kernel Spectre-PHT gadget. The plots donated with 3^{rd} show the probabilities of finding the correct v in the first three candidates predicted by the model.

Figure 8.13.: **Meltdown-Power** probability to leak the nibble v error-free for different strategies and prefetching methods.

ratio and the correlation coefficients are decreasing, we still observe a measurable signal usable for MDS-Power as shown in Section 7.3.

7.5. Evaluation of Meltdown-Power

Meltdown-Power has a significantly lower performance than MDS-Power due to the amount of code executed for the prefetch gadget and the reliability of its speculation. Generally, this activity drastically increases the time to obtain a single measurement sample and reduces the signal-to-noise ratio. Therefore, to enable a fair and robust comparison to MDS-Power, we evaluate Meltdown-Power primarily with the RAPL interface. Furthermore, we disable the hardware prefetchers during this experiment. We discuss and evaluate the resulting implications in Section 7.6. We estimate the number of samples required to observe the same leakage with throttling side channels (cf. Section 7.4), based on our RAPL measurements, taking the evaluation of MDS-Power into account (see Section 7.3). Some Meltdown mitigations switch the `cr3` register during context switches [20], which implicitly flushes the TLB. When the kernel flushes the TLB upon entry, the real-world prefetch gadget found by Johannesmeyer et al. [13] has a low prefetch rate on our test machine. As Meltdown-Power is particularly relevant on systems not affected by Meltdown, we assume that such mitigations are not in place, and the TLB is not flushed after entering the kernel, which is commonly the case on newer microarchitectures.

We evaluate Meltdown-Power with Ubuntu 20.04 LTS on an Intel Core i7-8700K (CPU A), an Intel Core i9-9980HK (CPU B), and an Intel Core i7-6700K (CPU C). CPU A and C run Linux kernel version 5.4, and CPU B version 5.13. We find that L1+L2 eviction with dirty cache line works best for Meltdown-Power and did not observe a signal with no-eviction, in line with our assumptions as the guess $\mathcal{G}$ will not *reach* the value $\mathcal{V}$. We disabled the hardware prefetchers. However, in this scenario, we show in Section 7.6 that the influences of the hardware prefetchers are minimal.

First, we evaluate the artificial Spectre-RSB gadget from Listing 8.3 located within a custom kernel module in Figure 8.13a. In the amplified nibble (128x) case, we achieve a leakage rate of 12.47 bit/h with 99.9 % (n=1000, $\sigma_{\bar{x}}$=0.10 %) success probability on CPU B which is 2.843 times higher than the leakage rate of CPU A with 99.2 % (n=1000, $\sigma_{\bar{x}}$=0.28 %) success probability. Furthermore, in an unamplified scenario (1x), we achieve a leakage rate of 0.84 bit/h with a success probability of 99.7 % (n=1000, $\sigma_{\bar{x}}$=0.17 %) on CPU B.

Second, we evaluate Meltdown-Power with the real-world Spectre-PHT gadget. The real-world Spectre-PHT gadget loads the cache line with a 90.51 % (n=10 000, $\sigma_{\bar{x}}$=0.03 %) probability on CPU C when using 20 keyrings (cf. Section 6.2). We fix the victim nibble v, *i.e.*, the target of the attack, to 7 and use the 128x amplification to fill the complete victim value $\mathcal{V}$. The attacker configures the prefetch gadget to load the victim's direct physical map address and uses 20 keyrings for the loop. Figure 8.13b shows that on CPU A we achieve a leakage rate of 0.136 bit/h with a success probability of 98.8 % (n=1000, $\sigma_{\bar{x}}$=0.34 %) to recover the victim nibble v. On CPU C, we achieve 0.147 bit/h with a probability of 96.7 % (n=1000, $\sigma_{\bar{x}}$=0.56 %). Table 8.7 summarizes the results of Meltdown-Power, and

Table 8.7.: Summary of Meltdown-Power when using L1+L2 eviction for the different variants across machines.

	CPU	Inst.	Ampl.	Leakage Rate ·1 bit/h	Measurement Duration ·1 ms	Samples
RSB	A	Load	128x	2.94	122.5	20 000
	A	Store	128x	4.39	109.4	15 000
	B	Load	128x	12.47	192.4	3000
	B	Store	128x	8.55	168.5	5000
	B	Store	1x	0.84	519.3	16 500
PHT	A	Store	128x	0.136	1968.8	27 000
	C	Store	128x	0.147	2099.8	23 370

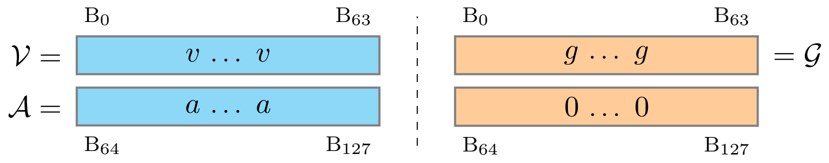


Figure 8.14.: We fill the guess value $\mathcal{G}$, the victim value $\mathcal{V}$, and the adjacent cache line of the victim $\mathcal{A}$ with repeating nibbles.

we conclude that Meltdown-Power is capable of extracting data across privilege boundaries.

Finally, we estimate the leakage rates for additional Meltdown-Power scenarios. First, we compute the leakage rate when targeting an *unamplified* nibble in the kernel with RAPL. We use the leakage conversion factor of 46.02 (cf. Section 7.2) resulting in a duration *estimate* of 14.1 days/bit. Second, we estimate the leakage rates for throttling attacks when using both the power limits and the stress program to leak the unamplified nibble. Based on Section 7.3 and Table 8.5, we compute a leakage rate conversion factor of 7.09 between RAPL and the power limits and 74.15 between RAPL and stress, respectively. This results in a leakage duration of 99.95 days/bit with power limits and 2.86 years/bit with stress-induced throttling, indicating a reduced SNR. We conclude that the real-world leakage rates with the Spectre-PHT gadget are impractical. Future work is required to determine whether potential improvements and optimized prefetch gadgets exist and can bring the attack runtime down to a level where Meltdown-Power poses a significant security risk.

Table 8.8.: Coefficients and statistics of the differential measurement technique modeling the adjacent cache line prefetcher.

HWPF	$\hat{\rho}$ $\cdot 1$	SNR_A $\cdot 10^{-3}$	$\mathrm{hd}(v, g)$ μW	$\mathrm{hd}(a, g)$ μW	$\mathrm{hw}(v)$ μW	$\mathrm{hw}(a)$ μW	$\mathrm{hw}(g)$ μW
Enabled	0.891	9.263	351.26	25.69	0.00	0.00	3250.24
Disabled	0.902	12.055	307.10	0.00	0.76	0.00	2514.21

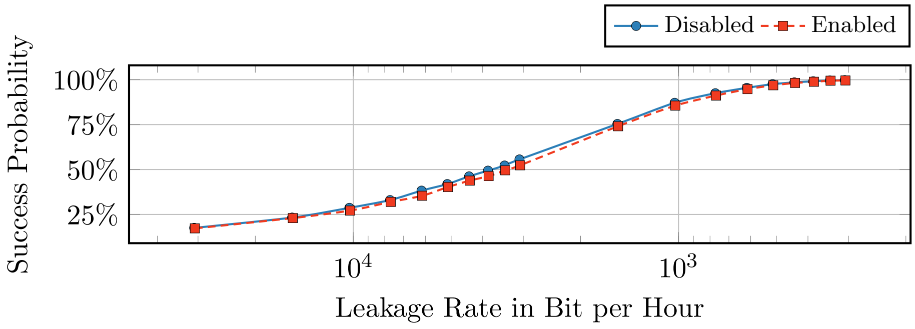


Figure 8.15.: The CPA success probability of the raw channel when using loads with L1+L2 eviction for (128x) amplified nibbles for enabled and disabled hardware prefetchers.

7.6. Evaluation of Hardware Prefetchers

We identified that Meltdown-Power is influenced by the prefetcher loading the adjacent victim cache line. Therefore, we analyze the influences of the *adjacent cache line* prefetcher on the leakage when this prefetcher is the most active with L1+L2 eviction as this pattern triggers the most evictions and reloads (cf. Figure 8.3). Figure 8.14 introduces a new fill value $\mathcal{A}$, filled with the repeating nibble a, resembling the data located in the adjacent cache line to the victim data $\mathcal{V}$. We fill the attacker-controlled adjacent guess cache lines with zeros and perform the leakage analysis with the load instructions and L1+L2 eviction on our Core i7-8700K (cf. Sections 4.3 and 5). First, we identify the strength of each leakage component for v, a, and g in our leakage model. Table 8.8 shows that the desired leakage component $\mathrm{hd}(v, g)$ is 13.67 times stronger than the undesired leakage component $\mathrm{hd}(a, g)$ of the adjacent cache line prefetcher. The coefficient for $\mathrm{hd}(a, g)$ drops to zero if the hardware prefetchers are disabled. Second, we determine the influences of the hardware prefetchers on the CPA success probability when not modeling any prefetchers in the model, *i.e.*, ignoring the prefetching effects. Figure 8.15 shows the CPA success probability with hardware prefetchers enabled or disabled, respectively. We compute an average decrease of 1.33 % when the hardware prefetchers are enabled. However, in the success probabilities above 95 %, we observe an average decrease of only 0.297 %. We conclude that hardware prefetchers only have minimal influence on Collide+Power.

8. Mitigations and Limitations

In this section, we discuss potential mitigations and limitations of Collide+Power. A principled mitigation against Collide+Power should eliminate or reduce the root cause of the leakage. We discuss mitigation ideas on the hardware and software level, although entirely preventing power-based leakage is still an open problem for general-purpose CPUs.

Hardware Mitigations. Power analysis attacks have been studied for decades on smaller form factor devices, such as smart cards [24, 17, 2]. Hence, hardware-based mitigations were researched and deployed in the wild for these systems. Such mitigations include blinding, masking, or adding randomness [24]. However, these mitigations require a fundamental hardware redesign and are usually tailored to protect cryptographic algorithms. Furthermore, these mitigations often require additional hardware for partitioning the computation into multiple parts, adding a significant performance impact [32]. Therefore, while it is theoretically possible with newer CPUs, such effective but costly mitigations are unlikely to be added to consumer CPUs.

Software and Operating System Mitigations. Meltdown-Power shown in Section 6.2 relies on a prefetch gadget in the kernel. Hence, a potential mitigation is eliminating all prefetch gadgets in the kernel. However, orthogonal research from Johannesmeyer et al. [13] on gadget finding suggests that a plethora of such gadgets exist in the kernel. Exacerbating the problem further: The prefetch gadgets required for Collide+Power are simpler than traditional transient-execution gadgets as they do not require the encoding part, e.g., a secret-dependent cache access. Orthogonally, MDS-Power currently requires co-location with the victim on a hyperthread, which could be prohibited if a group scheduling policy is implemented. However, the effects described in this paper likely apply to additional shared buffers in the CPU. Following the suggestions of Wang et al. [38], Turbo Boost and SpeedStep on Intel or Cool'n'Quiet on AMD CPUs can be disabled to curb userspace attacks, causing the CPU to reach the power limit less frequently. However, the question arises of which other power-related signals an attacker could use instead of the throttling side channels. Finally, in Section 7.2, we show that changing data co-located in the victim cache line does not impact the attacker's success probability. However, dynamically changing the victim values, e.g., cryptographic keys, breaks the assumptions of Collide+Power that the victim data is constant during the attack, effectively preventing leakage due to the relatively low leakage rates. Nevertheless, in contrast to traditional rekeying, the changing interval must depend on wall-clock time, not usage count, as unused secrets could be reachable with Collide+Power. Another alternative mitigation for MDS-Power is to *sandwich* secret data loads between victim-controlled loads, preventing the collisions of the attacker-controlled guesses and the victim value. However, this mitigation is ineffective against Meltdown-Power.

Limitations. While Collide+Power exploits the energy differences induced by cache loads, our primitives are not limited to the cache. In theory, the contents of any microarchitectural element with data-dependent energy consumption can be leaked. In practice, we require that the energy consumption becomes observable via performance counters for a privileged attacker or with frequency scaling in an unprivileged scenario. Future work could explore the impacts of Collide+Power on other microarchitectural buffers. The current Meltdown-Power proof-of-concept has severe practical limitations reflected in the low security risk when using the Spectre-PHT prefetch gadget. Therefore, Collide+Power benefits from research identifying optimized prefetch gadgets.

9. Conclusion

Collide+Power shows that mere co-location of data values in microarchitectures introduces combined leakage in the power domain. Our systematic analysis of the CPU's memory hierarchy led to precise leakage models that enable the exploitation of this combined leakage. We demonstrated that Collide+Power works with power consumption interfaces or throttling-induced timing variations alike. Our novel differential measurement technique amplifies the signal-to-noise ratio by a factor of 8.778 on average, compared to a straightforward DPA approach. We demonstrated that Collide+Power can even leak single-bit differences from the CPU's memory hierarchy with fewer than 23 000 measurements. In MDS-style end-to-end attacks, Collide+Power leaks 4.82 bit/h in the same scenario as RIDL and ZombieLoad but without relying on the MDS hardware flaw. However, in real-world Meltdown-style attacks, we encounter practical limitations leading to leakage rates of more than a year per bit with throttling. Future work is required to find more practical prefetching methods to replace the current Spectre-PHT gadget and to reevaluate the potential security risk of Meltdown-Power. Since Collide+Power is a generic attack with different variants, applying to any modern CPU, it poses a significant challenge for future work to develop mitigations against this threat. For commodity systems, mitigating Collide+Power is more challenging, as it exploits the very basics of microarchitecture design.

Acknowledgments

We thank the anonymous reviewers, especially our shepherd, for their guidance, comments, and suggestions. We thank Robert Primas, Daniel Weber, and Moritz Lipp for engaging discussions. Furthermore, we thank Brian Johannesmeyer and Jakob Koschel for their help and expertise with the Spectre-PHT prefetch gadget. This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe F8504), and the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS). A generous gift from AWS, Red Hat, and Google provided additional funding. Any opinions, findings, 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] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS. 2019 (p. 234).

[2] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In: CHES. 2004 (pp. 223, 228, 257).

[3] 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 (p. 223).

[4] 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. Extended classification tree and PoCs at https://transient.fail/. 2019 (pp. 227, 232, 233, 245).

[5] Jonathan Corbet. Moving the kernel to modern C. Feb. 2022. URL: `https://lwn.net/Articles/885941/` (p. 265).

[6] Jonathan Corbet. Toward a better list iterator for the kernel. Mar. 2022. URL: `https://lwn.net/Articles/887097/` (p. 265).

[7] 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 (p. 225).

[8] Matteo Fusi. Information-Leakage Analysis Based on Hardware Performance Counters. MA thesis. Politecnico di Milano, 2017 (p. 223).

[9] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In: DSN. 2017 (p. 223).

[10] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (p. 227).

[11] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 228, 251).

[12] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers. May 2019 (p. 251).

[13] 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 (pp. 224, 232, 233, 245, 246, 254, 257, 265).

[14] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (p. 227).

[15] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 223).

[16] 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 (p. 227).

[17] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In: CRYPTO. 1999 (pp. 223, 228, 257).

[18] Jakob Koschel. [RFC PATCH 00/13] Proposal for speculative safe list iterator. Feb. 2022. URL: `https://lwn.net/ml/linux-kernel/20220217184829.1991035-1-jakobkoschel@gmail.com/` (p. 265).

[19] 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 (pp. 223–225, 228, 230, 232).

[20] 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. 2018 (pp. 223–225, 227, 233, 245, 254).

[21] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In: CCS. 2022 (pp. 223–225, 228, 232, 251).

[22] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 227).

[23] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (p. 227).

[24] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer Science & Business Media, 2008 (pp. 223, 257).

[25] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. Vulnerabilities Introduced by Features for Software-based Energy Measurement. 2017. URL: `https://tubiblio.ulb.tu-darmstadt.de/104085/` (p. 223).

[26] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In: CHES. 2000 (p. 223).

[27] Colin O'Flynn and Alex Dewar. On-Device Power Analysis Across Hardware Security Domains. In: CHES. 2019 (p. 223).

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 223, 225, 227).

[29] David Oswald and Christof Paar. Breaking Mifare DESFire MF3ICD40: Power analysis and templates in the real world. In: CHES. 2011 (p. 223).

[30] David Oswald, Bastian Richter, and Christof Paar. Side-channel attacks on the Yubikey 2 one-time password generator. In: RAID. 2013 (p. 223).

[31] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (p. 227).

[32] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic: DPA-resistance without routing constraints. In: CHES. 2005 (p. 257).

[33] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In: TrustCom/BigDataSE. 2018 (p. 223).

[34] 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 (pp. 223–225, 227, 233, 243).

[35] 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 (pp. 223–225, 227, 233, 243).

[36] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In: FC. 2021 (pp. 224, 232, 233, 245).

[37] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018. URL: `https://support.google.com/faqs/answer/7625886` (p. 245).

[38] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In: USENIX Security. 2022 (pp. 223–225, 228, 232, 233, 251, 257).

[39] 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` (p. 225).

[40] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In: USENIX Security. 2022 (pp. 232, 233).

[41] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. In: Symposium on Internetware. 2015 (p. 223).

[42] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 223, 225).

10. Appendix

10.1. Differential Leakage Model Derivation

We derive Equation (8.4) by starting with the power leakage model of Equation (8.3), defined as

$$\mathcal{P}(\mathcal{G},\mathcal{V}) = a_0 \cdot \mathrm{hd}(\mathcal{G},\mathcal{V}) + w_0 \cdot \mathrm{hw}(\mathcal{V}) + w_2 \cdot \mathrm{hw}(\mathcal{G}) + \omega.$$

We add an adaptive noise term ω to model a realistic measurement. We model the power consumption of two distinct attacker-controlled guesses, the normal guess $\mathcal{G}$ and its masked inverse $\tilde{\mathcal{G}} = \mathcal{G} \oplus m$. We assume that the noise between two successive samples is constant due to time locality. The victim data $\mathcal{V}$ stays constant during both of these guesses. This yields the two equations

$$\mathcal{P}_{\mathcal{G}} = a_0 \cdot \mathrm{hd}(\mathcal{G}, \mathcal{V}) + w_2 \cdot \mathrm{hw}(\mathcal{G}) + w_0 \cdot \mathrm{hw}(\mathcal{V}) + \omega \quad \text{and}$$
$$\mathcal{P}_{\tilde{\mathcal{G}}} = a_0 \cdot \mathrm{hd}(\tilde{\mathcal{G}}, \mathcal{V}) + w_2 \cdot \mathrm{hw}(\tilde{\mathcal{G}}) + w_0 \cdot \mathrm{hw}(\mathcal{V}) + \omega.$$

Subtraction of the equations $\mathcal{P}_{\mathcal{G}}$ and $\mathcal{P}_{\tilde{\mathcal{G}}}$ yields the power difference between both of the attacker's guesses,

$$\begin{aligned}\mathcal{P}_{\mathcal{G}} - \mathcal{P}_{\tilde{\mathcal{G}}} = a_0 \cdot \Big(\mathrm{hd}(\mathcal{G}, \mathcal{V}) - \mathrm{hd}(\tilde{\mathcal{G}}, \mathcal{V})\Big) \qquad & \cancel{+w_0 \cdot \mathrm{hw}(\mathcal{V}) + \omega} \\ + w_2 \cdot \Big(\mathrm{hw}(\mathcal{G}) - \mathrm{hw}(\tilde{\mathcal{G}})\Big) \qquad & \cancel{-w_0 \cdot \mathrm{hw}(\mathcal{V}) - \omega}.\end{aligned}$$

First, we derive the results of the subtraction of the Hamming difference $\mathrm{hd}(\mathcal{G}, \mathcal{V})$ with the Hamming difference where one parameter is the mask inverse $\mathrm{hd}(\tilde{\mathcal{G}}, \mathcal{V})$. We consider the following equation $\mathrm{hd}(x, y) - \mathrm{hd}(\neg x, y)$. To change a value from x to y, $\mathrm{hd}(x, y)$ bits need to be flipped. Similarly, to change from x to $\neg x$, the number of bits in a word (N) need to change. Therefore, we can flip from $\neg x$ to x and *undo* the flips we require to get to y, yielding $\mathrm{hd}(\neg x, y) = N - \mathrm{hd}(x, y)$. Therefore,

$$\mathrm{hd}(x, y) - \mathrm{hd}(\neg x, y) = 2 \cdot \mathrm{hd}(x, y) - N.$$

If we consider $\mathrm{hd}(\mathcal{G}, \mathcal{V}) - \mathrm{hd}(\mathcal{G} \oplus m, \mathcal{V})$, we observe that only the bits selected by m are active as the other bit differences cancel out due to the Hamming distance. Therefore, the word size is reduced to $N = \mathrm{hw}(m)$, and we derive

$$\mathrm{hd}(\mathcal{G}, \mathcal{V}) - \mathrm{hd}(\mathcal{G} \oplus m, \mathcal{V}) = 2 \cdot \mathrm{hd}(\mathcal{G}_m, \mathcal{V}_m) - \mathrm{hw}(m).$$

where $\mathcal{G}_m = \mathcal{G} \wedge m$ and $\mathcal{V}_m = \mathcal{V} \wedge m$ are the masked cache lines with which we mask away non-targeted data. Second, we derive the results of the subtraction of the Hamming weight $\mathrm{hw}(\mathcal{G})$ with the Hamming weight of the mask inverse $\mathrm{hw}(\tilde{\mathcal{G}})$. We use the following property $\mathrm{hw}(\neg x) = N - \mathrm{hw}(x)$ to derive

$$\mathrm{hw}(x) - \mathrm{hw}(\neg x) = 2 \cdot \mathrm{hw}(x) - N.$$

With the same reasoning about the active bits, we derive

$$\mathrm{hw}(\mathcal{G}) - \mathrm{hw}(\mathcal{G} \oplus m) = 2 \cdot \mathrm{hw}(\mathcal{G}_m) - \mathrm{hw}(m).$$

Resulting in the final differential power leakage model of Equation (8.4),

$$\begin{aligned}\mathcal{P}_{\mathcal{G}} - \mathcal{P}_{\tilde{\mathcal{G}}} = a_0 \cdot (2 \cdot \mathrm{hd}(\mathcal{G}_m, \mathcal{V}_m) - \mathrm{hw}(m)) \\ + w_2 \cdot (2 \cdot \mathrm{hw}(\mathcal{G}_m) - \mathrm{hw}(m)).\end{aligned}$$

```
 1 struct key *find_keyring_by_name(const char *name, bool
   -> uid_keyring) {
 2   // ...
 3   list_for_each_entry(keyring, &ns->keyring_name_list,
   -> name_link) {
 4
 5     if(!kuid_has_mapping(ns, keyring->user->uid))
 6       continue;
 7
 8     if(test_bit(KEY_FLAG_REVOKED, &keyring->flags))
 9       continue;
10
11     if (strcmp(keyring->description, name) != 0)
12       continue;
13     // ...
14   }
15   // ...
16 }
```

Listing 8.4: The Spectre-PHT prefetch gadget of the Linux kernel key management used to load arbitrary data.

10.2. Kernel Spectre-PHT Prefetch Gadget

The prefetch gadget in Listing 8.4 was discovered by Johannesmeyer et al. [13]. First, Line 3 iterates over a list of keyrings `ns->keyring_name_list` of the running process's namespace. During this iteration, the CPU's branch predictor is mistrained and speculatively accesses one additional element at the end of the loop. Due to the given memory layout [13, 18, 5, 6], during speculative execution, a type confusion occurs, where a `struct -> user_namespace` is interpreted as a `struct key`. Therefore, the speculative access of `keyring->user->uid` in Line 5 actually accesses `ns->projid_map->entry[3]`, which is controllable by an attacker, resulting in the desired prefetch gadget [13]. The array `ns->projid_map->entry[3]` can be filled by the attacker by writing to `/proc/self/projid_map`.

Statutory Declaration

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 marked all material which has been quoted either literally or by content from the used sources.