Jonas Juffinger

Attacking and Securing Leaky Systems at the Hardware-Software Boundary

PhD Thesis

Assessors: Daniel Gruss, Onur Mutlu

July 2025

Institute of Information Security
Graz University of Technology

Abstract

The interactions at the boundary between hardware and software components in modern computers are often a source of leakage. Side-channel attacks leak private data such as passwords or web browsing behavior, e.g., via timing differences. Software-triggered hardware faults let attackers elevate their privileges and fully subvert systems. Continuously smaller and faster hardware increases the attack surface by undermining mitigation efforts and introducing new vulnerabilities that can go unnoticed for years.

In this thesis, we significantly advance the understanding of the attack surface at the hardware-software boundary and how defenses can constrain it. In the direction of fault attacks, we show that DRAM disturbance attacks, like Rowhammer, can exist for a long time before being fully understood by discovering a link between effects observed in recent work and our prior work from 2017. We also extend DRAM disturbance research to new hardware by investigating how SSDs can be utilized in Rowhammer attacks, finding lower bounds for SSD-based attacks. Both of these works motivate principled mitigations against DRAM disturbance attacks that are not tailored to specific attack patterns and targets. Hence, we present a novel principled mitigation against DRAM disturbance attacks based on a practical and efficient hardware-software co-design, exceeding the detection and correction capabilities of state-of-the-art solutions significantly. Furthermore, we show that with a secure hardware-software co-design against software-based fault attacks on the CPU, we can even reduce CPU power consumption while increasing performance.

In the direction of side-channel attacks, we perform the first side-channel analyses on modern commodity off-the-shelf SSDs. Even though SSDs are widely used, they have not yet been studied as a source of side channels. We close this research gap, by presenting two novel software-based timing side-channel attacks on SSDs that leak sensitive user information, achieving both, high temporal and spatial resolution.

This thesis consists of two parts. The first part contextualizes my contributions within the state of the art. The second part presents my unmodified[1] first-author publications. All of these papers were anonymously peer-reviewed and accepted at renowned international conferences.

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

Acknowledgments

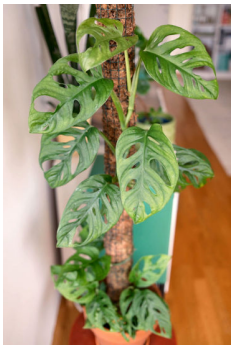
First and foremost, I want to thank my advisor, Daniel Gruss. You persuaded me to do a PhD, and I am so grateful you did. Thank you for all your support during my PhD, the interesting discussion, for giving me the freedom to experiment and pursue my own research, and for being generous with work from home, making my travels between Vienna and Graz bearable.

I also want to thank Onur Mutlu for taking the time and effort to assess my PhD thesis. Thank you also for giving me the chance to present my work to your group and the great discussions that followed.

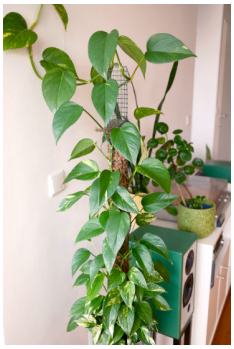
I want to thank Andreas Kogler. Your limitless perseverance keeps inspiring me, and I often think back to our travels and presentations; they were always fun and very well-prepared. I want to thank Stefan Gast. I still remember our first day at the institute and our many great discussions in the own exclusive office. You will continue to inspire me to question absolutely everything. I want to thank Lukas Giner; you taught me important life lessons and a lot about caches.

Especially, I want to thank my other colleagues, Sudheendra Raghav Neela, Fabian Rauscher, Martin Schwarzl, Claudio Canella, Moritz Lipp, Hannes Weissteiner, Carina Fiedler, Roland Czerny, Martin Heckel, Lukas Lamster, Lukas Maar, and Theresa Dachauer. I could not have dreamed of better people to work with. It was a pleasure discussing new, old and stupid ideas, complaining about reviewer B, and celebrating successes with you. All the best for finishing your PhD theses. I also want to thank all the people I met and had a great time with at conferences, Stepan Kalinin, Daniel Weber, Leon Trampert, Lukas Gerlach, Fabian Thomas, Marton Bognar, Michele Marazzi, and many more. Thanks to all the unnamed people and friends I met during this journey.

I want to thank my parents and family. Franziska and Christian, you are the best parents one could wish for. Thank you for your unmeasurable support, your drive to always strive for something bigger, and for giving me the freedom to pursue my dreams. Thank you, especially Wolfgang and all Höcks, Färbingers, as well as Rinderers, for being such great bonus families. Thank you, Thomas; I think we perfectly complemented each



(a) Monstera Adansonii's beautiful perforated leaves.



(b) Golden pothos in the morning sun.

Figure 1.: The fruits of procrastination. Golden pothos and Monsteras grow aerial roots to climb trees and absorb water. A pole filled with moss is a perfect medium for these plants' and author's support and hydration.[2]

other's hunger for success. Thank you, Julian, for igniting my interest in photography, a wonderful creative shared hobby not involving computers.

Thank you, Matthias, for all the years we lived together through school and university. I know that I can always count on you, be it for life, technical or running expertise. I also want to thank the friends I made in Graz and Vienna, Florian, Christoph, Mike, Dominik, and Silva.

A very special thanks go to Paul Smisek and Paul Schmidmayr. While you probably do not know each other, you set me up on a trajectory that led to this thesis by supplying me with my own computers at a very young age that allowed me to experiment with them freely, and introducing me to Linux and programming.

I also want to thank my plants, shown in Figure 1, that gave me great reasons to procrastinate, helping me to free my headspace and view problems from new angles.

Finally, I want to thank my wonderful partner, Sarah. You are my biggest love, inspiration, friend, and supporter. Thank you for enduring the many lonely days while I was in Graz and at conferences. Thank you for sharing the same slightly unhealthy work ethic that made it possible to work through many weekends when deadlines were approaching. Thank you for proofreading my paper drafts. Thank you for always cheering me up. I would not have been able to finish my PhD without you.

[2]The author added this figure while procrastinating.

Contents

Part I.

Attacking and Securing Leaky Systems at the Hardware-Software Boundary

leaky
adjective

Something that is leaky has a hole or crack in it that allows liquid or gas to get through [46].

1 Introduction

Users store plenty of secret data on their computers, from secret keys and passwords to private information like web browsing behavior. As users, we have to trust the programs handling this information. But for programs and the operating system to guarantee security, they themselves have to trust the hardware. Leaky hardware, however, can expose secrets to an attacker. This leakage can come in different forms, for instance, direct information leakage through side channels, as well as software-based fault attacks, like Rowhammer, introducing errors in the computer's electronics, *i.e.*, faults, because of "leaking" stray electrons.

Electronics require specific environments to run in. If requirements are violated, the electronic circuit can be disturbed, causing faults. These requirements can be direct, like the supply voltage and maximum clock frequencym or specifications on how the circuit must be operated, but also outside factors, like temperature or the maximum strength of radiation the circuit withstands. Faults in CPUs or microcontrollers can impact the software running on them. When timed correctly or if the software state is prepared properly, purposefully injected faults can predictably break the software, undermining the system's security [15, 25, 71, 229].

Until 2014, all fault attacks were performed with physical access to the target device [14, 15, 71, 236]. The discovery that they can be caused only by software has been more recent with Rowhammer [137] and overclocking or undervolting attacks [179, 208, 243].

Rowhammer is a software-based fault attack affecting the DRAM [137]. It was first identified as a potential security issue in 2014 [137], and only 9 months later, the first privilege escalation exploits were presented [229]. With the underlying effect being difficult to mitigate, seemingly every new proposed mitigation [10, 30, 41, 45, 83, 145, 168, 171, 257] was defeated by the new exploits and hammering techniques, published over the following years [39, 52, 53, 74, 105, 125, 144, 156, 169, 190, 215, 216, 244, 245,

256, 288]. Proposed and implemented mitigations were broken mainly for two reasons: a deficient implementation due to cost or optimistically misconfigured threshold values due to lack of insights into RowHammer vulnerability [39, 53, 74, 105, 144, 163, 188, 277]. For example, target row refresh (TRR) was implemented insufficiently and could, therefore, be tricked to protect the wrong rows [53, 105]. Additionally, the discovery of Half-Double Rowhammer [144] showed that TRR could be used as a confused deputy, breaking the mitigation in two different ways. Already existing memory protection mechanisms like error correcting codes (ECC) are also ineffective as they can only correct a single bit flip, which is insufficient for targeted Rowhammer attacks [39].

The other group of previously exploited software-based fault attacks includes overclocking and undervolting attacks [36, 130, 179, 208, 243]. These attacks are enabled by kernel-accessible hardware interfaces that allow software to control the CPU clock frequency and CPU supply voltage independently. These two values, clock frequency and supply voltage, are dependent on each other, and can cause faults in the CPU if misconfigured [82]. The faults manifest in a small number of CPU instructions rarely outputting wrong computation results, which can be used to corrupt cryptographic algorithms, but also program flow and array accesses. These can be used to attack trusted execution environments like ARM TrustZone [208, 243] or Intel SGX [36, 130, 179]. However, as numerous publications show, undervolting also has a great potential to save CPU power [11, 12, 68, 124, 146, 147, 172, 193, 194].

Side-channel leakage happens whenever a secret influences an unintended physical property that an attacker can measure. Safe crackers can feel the slightest manufacturing imperfections when turning locks [184], programs influence the power consumption of CPUs [31, 141, 157], or take a different amount of time based on the secret [19, 139]. Information can also leak through the state of a system, like the state of a CPU cache, influenced by the secret. The CPU cache can again be measured through a timing side channel to reconstruct the secret [191, 199]. Side-channel attacks can be divided into physical and software-based attacks. Physical side channels include electromagnetic emissions, power consumption, acoustic emissions, temperature, optical emissions, and vibrations, among others. An attacker typically needs physical access or close proximity to the device to measure the side channel signal [23, 80, 141]. Software-based side channels can be measured from software, and while the underlying reason can be manifold, they mainly cause variations in timing [19, 139, 191, 199].

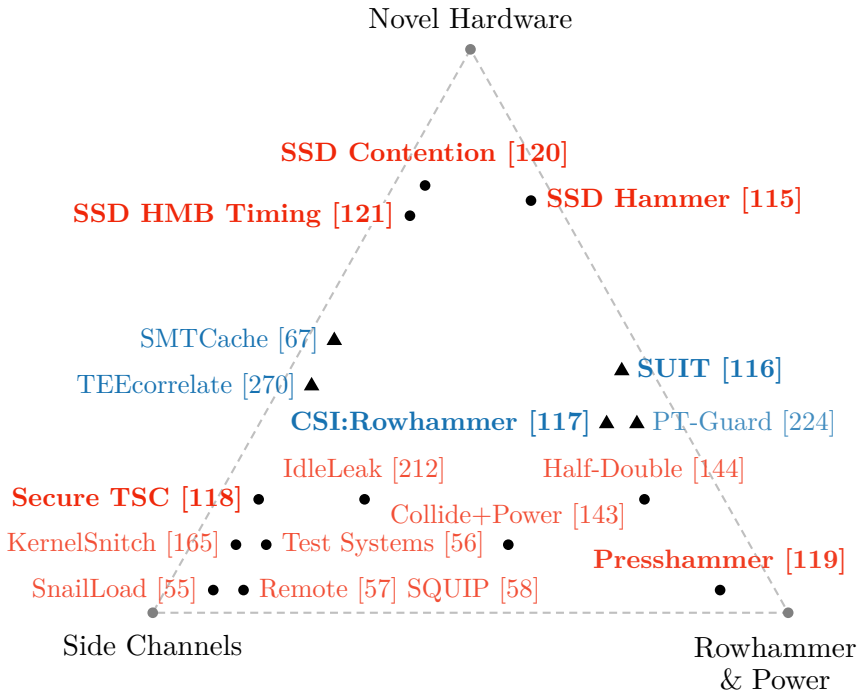


Figure 1.1.: Overview over all my contributions. My main contributions to this thesis are highlighted in bold. Circles with red text define attacks and triangles with blue text defenses or neutral work.

Due to their high spatial and temporal precision, CPU caches were the main software-based side-channel attack target for many years [76, 77, 89, 139, 176, 191, 202, 204, 211, 279, 290]. More recently, research has also looked into side channels in other CPU and computer components in general. These components include random-number generation logic [50], execution ports [4, 22], execution schedulers [57, 58], cache occupancy [233], the PCIe bus [241], idle states [212, 283], operating system data structures [111, 164, 165], device sensors [174, 186, 231, 242], software-based power measurements [143, 157, 186, 242], GPUs [3, 47, 66, 113, 183, 242, 262, 268], frequency scaling [143, 159, 263, 264], and hard-disk drives [23, 80, 128, 155]. These countless side channels were shown to leak not only cryptographic keys but also private user data like browsing behavior and user input, or secret kernel information like heap pointers and KASLR offsets.

However, one component used in almost every modern computer is absent from this list. Commercial off-the-shelf SSDs have seen only very little research on their side channel behavior. While storage systems were already identified as a means to construct covert channels in 1973 [150], only a small body of work showed potential side channel leakage in smart SSDs containing a programmable FPGA on the cloud [250, 251], and Liu et al. [161] analyzed the now-discontinued Intel Optane persistent storage.

1.1. Main Contributions

This section introduces the first-authored papers included in this PhD thesis. In total, I first-authored 7 papers, three of which were published at top-tier conferences. Figure 1.1 gives an overview of all my contributions. They range from a novel principled defense against Rowhammer [117]; a mitigation against privileged CPU undervolting attacks that does not compromise on potential energy savings [116]; a study of the recently discovered RowPress effect in light of our, in 2017 discovered, one-location Rowhammer, including the first RowPress exploit [119]; two papers on the security of the SSD host memory buffer feature, one analyzing its vulnerability to Rowhammer [115] and one showing that the HMB causes an exploitable timing side channel [121]; a comprehensive study on the vulnerability of SSDs to a contention side-channel attack [120]; and, finally, a virtual machine co-location detection method using the new secure TSC feature of AMD [118].

CSI:Rowhammer – Cryptographic Security and Integrity against Rowhammer [117]. With CSI:Rowhammer, we designed a principled defense against Rowhammer that can be implemented with minimal hardware changes and cannot be affected by an incomplete understanding of the Rowhammer problem that broke earlier mitigations. CSI:Rowhammer repurposed the additional memory used for ECCs to store a cryptographically secure message authentication code (MAC) computed over the data stored in the DRAM. This code guarantees data integrity on every read, completely independent of the cause of bit flips in the data. As MACs are, by design, not invertible, they cannot correct bit flips in the data. We show that bit flips can be corrected efficiently by search and that by including the operating system into the correction effort, more advanced corrections can be performed, e.g., by reloading corrupted data from the

disk. We evaluated the security guarantees and showed that on a system that is not under attack, silent data corruption happens on average once every 10^9 billion years. A very fortunate Rowhammer attacker has a chance of 9.75×10^{-5} % to produce a second pre-image when causing bit flips once every 128 ms for a year straight. We evaluated the performance overhead with gem5, showing that it is only 0.74 % on average. This work was published at the IEEE Symposium on Security & Privacy (S&P) in 2023 [117] in collaboration with Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss.

SUIT: Secure Undervolting with Instruction Traps [116].
The voltage requirement of a CMOS circuit cannot be defined exactly. Many unpredictable environmental factors like process variation, temperature, aging, or input voltage fluctuations do have an impact on the propagation delay of a CMOS circuit at specific voltages [64, 148, 153]. If the propagation delay is too high, undetectable faults can happen inside a CPU [167, 192]. Prior work showed that these faults can be caused by privileged software on ARM and Intel CPUs and used to attack trusted execution environments [15, 36, 130, 179, 208, 243]. However, apart from these faults, undervolting a CPU can significantly reduce power consumption [11, 12, 68, 124, 146, 147, 172, 193, 194] and even increase performance due to thermal and power throttling of CPUs [70, 197, 263]. With SUIT, we show that CPU undervolting can be made secure and reliable by trapping or slightly modifying a small subset of "faulting" instructions. With SUIT's modifications, the CPU only runs in an undervolted state with these faulting instructions disabled. If the running program executes a disabled instruction, the CPU traps, and the operating system increases the voltage to guarantee successful execution. Handling the undervolting state in software allows the operating system to optimize the CPU to the currently running workload dynamically. With SUIT we can reduce the power consumption by up to 20 % with a performance increase of over 3 %. This work was published at the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) in 2024 [116] in collaboration with Stepan Kalinin, Daniel Gruss, and Frank Mueller.

Presshammer: Rowhammer and Rowpress without Physical Address Information [119]. In 2018, we discovered one-location Rowhammer, a new hammer method that only accesses a single row in DRAM [74].

We explained the unexpected finding of one-location Rowhammer flipping bits, with the memory controller employing a closed-row policy [74]. Five years later, in 2023, Luo et al. [163] discovered a different DRAM read-disturbance phenomenon, called Rowpress. Rowpress flips bits by keeping a DRAM row open for long periods, unlike Rowhammer, which opens and closes rows as quickly as possible [137, 163]. With single-row Rowpress being almost identical to one-location Rowhammer, we revisited the latter, analyzed it again, and compared it to Rowpress. In our work, we show that one-location Rowhammer causes bit flips not only due to the memory controller's closed-row policy but also the Rowpress effect, coining the term Presshammer. This finding shows that Rowhammer attacks can exist for a long time before they are fully understood; only Luo et al. [163] were able to actually realize the new underlying phenomenon. With our new understanding of Rowpress, we built the first privilege escalation exploit that uses timing side-channels to find the correct aggressor memory locations for Rowpress. On our system, we were able to exploit Rowpress in less than 10 minutes. This work was published at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) in 2024 [119] in collaboration with Sudheendra Raghav Neela, Martin Heckel, Lukas Schwarz, Florian Adamsky, and Daniel Gruss.

An Analysis of HMB-based SSD Rowhammer [115]. Solid-state drives can use a part of the main memory, called the host memory buffer (HMB), to cache logical to physical storage address translations [185]. The HMB improves the SSD's performance, as DMA accesses to the main memory are faster than to the flash memory. In this work, we analyze whether these DMA accesses can pose a security risk by causing Rowhammer bit flips in the HMB. While we show that software-induced bit flips in the HMB can cause denial of service, data loss, and even break SSDs, the accesses from the SSD to the HMB are too infrequent to cause actual Rowhammer bit flips. This work was published at the International Conference on Applied Cryptography and Network Security (ACNS) in 2025 [115].

The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer [121]. In our third work on SSDs, we again analyze the host memory buffer, this time as a potential source of timing side-channel leakage. We analyze four SSDs that use the HMB feature and reverse engineer how they use the HMB and the cache replacement policies. We

show that the HMB induces timing differences in SSD accesses, which are clearly measurable from user space. We exploit this timing side channel in four case studies: a covert channel between processes; with up to 8.3 kbit/s; a UI redress attack that detects when the `pkexec` binary is executed; a covert channel between virtual machines; and finally, a remote covert channel exploiting a web server as a confused deputy to transmit data over the network. This work was published at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) in 2025 [121] in collaboration with Hannes Weissteiner, Thomas Steinbauer, and Daniel Gruss.

Secret Spilling Drive: Leaking User Behavior through SSD Contention [120]. Hard disk drives (HDDs) have long been known to be a source of side-channel leakage, including acoustic emanation [80], electromagnetic radiation [23], and timing variation [34, 155]. Chen et al. [34] suggest that SSDs would mitigate certain HDD timing covert channels because transfer speeds and response times of modern SSDs are magnitudes faster than HDD's. In this work, we show that user space applications can, nevertheless, mount contention side-channel attacks. We analyzed 12 different SSD models from 7 vendors, ranging from entry-level PCIe 3.0 SSDs without DRAM to the latest PCIe 4.0 SSDs with DRAM caches. First, we showed that, although different SSDs exhibit very different contention behavior, a covert channel between virtual machines can transmit data with up to 1 503 bit/s. Second, we invade user privacy by leaking websites currently visited by the victim. We exploit the fact that web browsers cache assets of websites, like images and code, and retrieve them from the disk on subsequent visits [177]. Since every website has a different number of assets and loads them at different times, they create an exact fingerprint. Using a machine learning model, we could fingerprint 100 websites in an open-world scenario with an F_1 score of up to 97.0 %. Both the covert channel and website fingerprinting are surprisingly noise resilient. This work was published at the Network and Distributed System Security (NDSS) Symposium in 2025 [120] in collaboration with Fabian Rauscher, Giuseppe La Manna, and Daniel Gruss.

Not So Secure TSC [118]. In cloud computing, programs of many different tenants are running on the same machine, each in its own virtual machine or container. This sharing of physical resources leads to countless

different side-channel attacks [20, 101, 176, 240, 274, 285, 286, 290]. However, the attacker must be co-located on the same machine to exploit these side channels [94, 289]. Distrusting the cloud provider or legal restrictions can prevent tenants from moving to the cloud [201, 239]. AMD was the first to introduce a confidential virtual machine CPU extension, SEV [127], that isolates the virtual machine from the host. Recently, AMD added the SecureTSC feature to SEV, which provides a fully trusted TSC timing source for guests [5, 42]. Until then, the host could modify every timing source inside virtual machines. In our work, we show that SecureTSC can be used for reliable and fast co-location detection. It works because every virtual machine can compute the exact uptime of the CPU it is running on. Under the assumption that it is very unlikely for two CPUs to share the exact same uptime, it can be used as a unique identifier. Using one coordination server that collects and compares all uptimes, we can detect co-location in a noisy environment in 0.13 seconds. Making use of the birthday problem, an attacker only needs 480 virtual machines to co-located at least two of them with a 90 % probability in a data center with 50 000 machines. This work was published at the International Conference on Applied Cryptography and Network Security (ACNS) in 2025 [118] in collaboration with Sudheendra Raghav Neela, and Daniel Gruss.

1.2. Other Contributions

This section briefly introduces the peer-reviewed papers I co-authored during my PhD studies. I co-authored 10 papers, 6 of which were published at top-tier conferences. These papers range from microarchitectural side-channel attacks and defenses, to web- and kernel side-channel attacks, a Rowhammer defense, and a software-based power-analysis attack.

Superscalar processors need a way to schedule ready instructions to the many execution units efficiently. AMD uses a split scheduler design with different schedulers for different execution units. In SQUIP [58], we show that attackers can intentionally fill these schedulers to specific levels. If a scheduler is full, this can cause a pipeline stall, which is measurable by the attacker. Using this timing side channel, for example, the exact executions of integer multiplications on the sibling SMT thread can be detected. We exploit this side channel to leak an RSA private key and build a covert channel. This work was published at the IEEE Symposium on Security & Privacy (S&P) in 2023 [58] in collaboration with Stefan Gast, Martin

Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss.

Virtual memory page tables are an often-exploited target of Rowhammer attacks, leading to privilege escalation [39, 53, 105, 125, 144, 229, 256, 288]. PT-Guard [224] is a defense that utilizes unused bits of page-table entries to store a cryptographic MAC for integrity protection. This work only requires hardware changes and no changes to the operating system software. The performance overhead is only 0.2 % We performed a study of page-table entries on real systems regarding their value continuity within page tables and showed that most entries can easily be corrected. This work was published at the IEEE/IFIP International Conference on Dependable Systems and Networks in 2023 [224] in collaboration with Anish Saxena, Gururaj Saileshwar, Andreas Kogler, Daniel Gruss, and Moinuddin Qureshi.

Software-based power side-channel attacks exploit secret-dependent power consumption of CPUs only from software [157, 263, 264]. This is possible because older CPUs provide direct energy measurements to software [157] and because energy consumption influences the CPU clock frequency, which is measurable from user space [263]. However, all previous works only attacked specific targets on the system, mainly cryptographic instructions or code [157, 263, 264]. With Collide+Power [143], we are the first ones to show that software-based power side-channel attacks can be used to leak arbitrary data. We do this by "colliding" the secret data with known data known to the attacker in the memory hierarchy to force Hamming weight leakage. This work was published at the USENIX Security Symposium in 2023 [143] in collaboration with Andreas Kogler, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard.

Minimizing the energy consumption of CPUs has become a very high priority in the current age of mobile computing. Intel introduced the `tpause` instruction that allows unprivileged software to enter new shallow idle states. In IdleLeak [212], we show that this instruction can be exploited to build a covert channel, an inter-keystroke timing, as well as a website and video fingerprinting attack. IdleLeak is based on the fact that the paused SMT threat is woken up by interrupts to the sibling thread, which makes it possible to get the exact timings of interrupts of the other thread. This work was published at the Network and Distributed System Security (NDSS) Symposium in 2024 [212] in collaboration with Fabian Rauscher, Andreas Kogler, and Daniel Gruss

Our work Remote Scheduler Contention Attacks [57] shows that the AMD split scheduler design can be exploited not only with handcrafted assembly but also from JavaScript. This greatly increases the attack surface as, for example, malicious advertisements could distribute the attack all over the World Wide Web. We analyzed all remaining schedulers, including FPUs absent from the original SQUIP paper [58]. Due to the lack of high-precision timers, we used a microarchitectural race condition to measure scheduler occupancy. With it, we built a high-precision inter-keystroke timing attack and a covert channel between browser windows. This work was published at the International Conference on Financial Cryptography and Data Security in 2024 [57] in collaboration with Stefan Gast, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss.

SnailLoad [55] is a novel timing side channel exploiting buffer occupancy between endpoints on the Internet. The victim connects and downloads something from the attacker's server, for example, an image in an advertisement. The attacker's server sends the image very slowly, 400 B/s. The round-trip time between each sent TCP packet and the returning acknowledgment packet gives the attacker a very detailed picture of current network utilization at the victim's side. With SnailLoad, an attacker is able to fingerprint the first 90 seconds of 10 videos with up to 98 % accuracy and fingerprint 100 websites in an open-world scenario with up to 63 % accuracy. This work was published at the USENIX Security Symposium in 2024 [55] in collaboration with Stefan Gast, Roland Czerny, Fabian Rauscher, Simone Franza, and Daniel Gruss.

KernelSnitch [165] is a novel generic timing side-channel attack exploiting various buffers in the Linux kernel. By measuring hash collisions in specific hash table buckets using timing differences of syscalls, addresses of kernel objects can be leaked because they are used as a part of the hash function input. Leaking kernel object pointers can make kernel exploitation more stable. We also showed a covert channel and a website fingerprinting attack. This work was published at the Network and Distributed System Security (NDSS) Symposium in 2025 [165] in collaboration with Lukas Maar, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard.

In our real-world study of the security of educational test systems [56], we evaluate the security of test systems from computer science university classes and identify various security issues. In three case studies, we show that security bypasses are possible. Finally, we perform a user study, asking educators responsible for the test systems about the impact of potential breaches. The breaches could compromise sensitive student

records, confidential research data, and in some cases even embargoed vulnerabilities. This work was published at the Workshop on Operating Systems and Virtualization Security in 2025 [56] in collaboration with Stefan Gast, Sebastian Daniel Felix, Alexander Steinmaurer, and Daniel Gruss.

SMTCache [67] is a new design for secure isolated L1 caches. SMTCache has multiple L1 caches per core and uses only one L1 cache per security domain. Therefore, different security domains cannot interfere with each other. Because only one L1 cache is used at a time, this design does not increase latency and increases power consumption only slightly. This work was published at the International Conference on Availability, Reliability and Security in 2025 [67] in collaboration with Lukas Giner, Roland Czerny, Simon Lammer, Aaron Giner, Paul Gollob, and Daniel Gruss.

With TEEcorrelate [270], we show how performance counter values of a confidential virtual machine guest can be made leakage-free, while still preserving relevant information to the host. TEEcorrelate decorrelates performance counter values using two components, aggregation of performance counter values within configurable length windows, and deferred and speculative performance counter increases within a configurable deviation range. We propose a window length of a few milliseconds and a range of 1024 deviation. These values are enough to mitigate performance counter attacks while the host can still perform load-balancing, accounting, and detection of unusual or malicious activity. This work was published at the USENIX Security Symposium in 2025 [270] in collaboration with Hannes Weissteiner, Fabian Rauscher, Robin Leander Schröder, Stefan Gast, Jan Wichelmann, Thomas Eisenbarth, and Daniel Gruss.

1.3. Outline

Chapter 2 provides background on digital circuits, CPUs, memory, solid state disks, and confidential computing. Chapter 3 gives an overview of the state-of-the-art Rowhammer attacks and mitigations, dynamic voltage- and frequency-scaling studies and attacks, side-channel attacks, and cloud co-location detection. 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 11.

2

Background

In this chapter, we provide background on relevant knowledge for this thesis. First, we explain how digital circuits and their main building block, MOSFET transistors, work and how supply voltage influences circuit timing. Then, we provide background on the memory subsystem of a computer, detailing virtual memory and DRAM. Finally, we give a short background on solid state drives (SSDs).

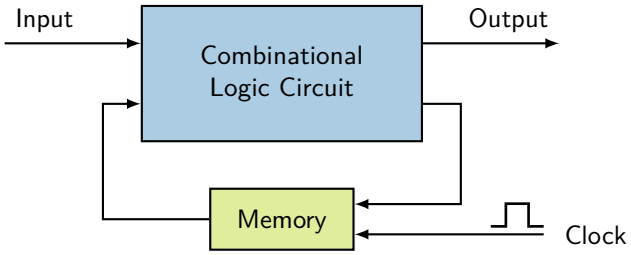


Figure 2.1.: A synchronous sequential digital circuit. The combinational circuit transforms the input and data stored in the memory into output and data to store for the next clock cycle.

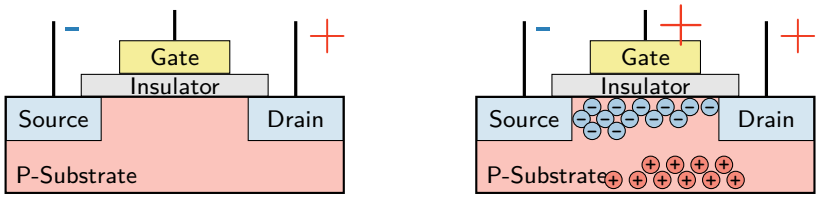
2.1. Digital Circuits

Strongly simplified, digital circuits are built using a large number of connected transistors (electronic switches) to transform input to output signals and store data [82, 148]. The transistors are connected to build different basic building blocks [82, 148]. One large set of building blocks is logical gates that perform a logical operation on their inputs and output the results; another set is flip-flops, used as a memory to store a binary value between clock cycles.

These logical gates are then connected to form combinational logical circuits to perform, for example, mathematical operations like addition or multiplication. The flip-flops are combined to build SRAM memory, storing multiple bits, like registers or caches [82, 148]. Connecting a combinational logical circuit with memory creates a sequential digital circuit, as shown in Figure 2.1. The combinational circuit transforms an input and a previous state into an output and values to store in the memory. On each clock cycle, the memory stores the values at its inputs and outputs them until the next clock cycle [82, 148].

2.1.1. Metal-Oxide-Semiconductor Field-Effect Transistor

Metal-Oxide-Semiconductor Field-Effect Transistors, short MOSFETs, are the dominantly used type of transistors for digital circuits [82, 255]. Their advantages are small static power consumption and size. Figure 2.2 shows a section view of a planar n-channel MOSFET. The substrate is slightly p-doped silicon, the drain and source contacts are n-doped. An insulator



(a) Open MOSFET, no current is flowing between source and drain.

(b) Closed MOSFET, current is flowing from the source to the drain.

Figure 2.2.: The structure of a planar N-channel MOSFET in silicon. The gate is used to "open" and "close" the transistor. To "close" the transistor, *i.e.*, make it conductive between the source and the drain, a positive charge is supplied to the gate. This attracts negative charges that conduct current.

between the substrate and the gate prevents charges from flowing between the two [82]. The substrate is not conductive.

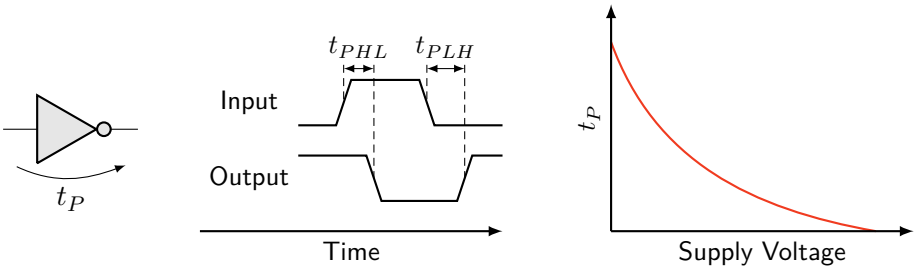
If a positive charge is supplied to the gate, it attracts electrons from the substrate, repulsing positive charges, *i.e.*, holes. Because of the insulator between the substrate and the gate, the electrons gather between the drain and source as shown in Figure 2.2b. If enough electrons gather, the MOSFET starts conducting between the drain and source[1]. When the positive charge is removed from the gate, the electrons recombine with the holes, making the substrate non-conductive again [82, 255].

P-channel MOSFETs work the same as n-channel MOSFETs, with silicon dopings and charges reversed. N- and p-channel MOSFETs are used together in pairs to build Complementary Metal-Oxide-Semiconductor (CMOS) logic used in every modern digital circuit [82, 255].

2.1.2. Circuit Timing and Voltage Requirement

For a digital circuit to work correctly, it must adhere to circuit specific timing and supply voltage limits. If these limits are not met, the circuit can experience faults [68, 82]. For example, in the case of CPUs, the integer multiplication unit can output wrong results, which can be used to attack trusted execution environments [179], see Section 3.2.

[1] Drain and source are electrically identical. Per definition, the source is the connector with a higher potential than the drain. The source is "emitting" positive charges.



(a) t_P of an inverter. (b) Impact of the t_Ps on the signal through an inverter. (c) Relationship between supply voltage and t_P.

Figure 2.3.: The impact of the propagation delay t_P on the traveling through an inverter and the relationship between supply voltage and t_P.

As discussed in the previous section, MOSFETs are switched on and off by attracting charge carriers to the gate, a process that takes time. The gate, insulator, and substrate form a capacitor that must be charged[2] to attract enough charge carriers to create a conductive channel in the substrate [68, 187, 255]. The time it takes to charge a capacitor is a function of its capacity and charging voltage. The capacity is defined by the dimensions of the transistor [235], smaller transistors switch more quickly. This is also the reason for smaller node size CPUs reaching higher frequencies with smaller supply voltages.

These switching delays of transistors in a logical gate cause a so-called propagation delay t_P as shown for an inverter[3] in Figure 2.3. The propagation delay of a circuit is defined by the addition of the propagation delays of all individual logic gates on the longest path, also called the critical path [187]. For a synchronous sequential digital circuit as shown in Figure 2.1, the clock frequency must not be higher than the inverse of the propagation delay of the critical path in the combinational logic circuit. Otherwise, the output or the values stored in the memory may not be fully computed and wrong [82].

In a real environment, the voltage frequency dependency of CMOS circuits is influenced by many internal and external factors [92, 187]. Process

[2]There are many additional parasitic capacitors in a MOSFET transistor, all influencing switching time [255].

[3]Because of the slightly different physical properties of p-channel and n-channel MOSFETs, there is a difference in the t_P from high to low t_{PHL} and low to high t_{PLH}.

variation makes every transistor slightly different. Temperature greatly influences the propagation delay of a circuit. Aging effects like hot-carrier injection or bias temperature instability cause the propagation delay to increase over time [227]. Additionally, sudden increases in current can cause voltage droops. To counter all these effects, digital circuits are supplied with a higher voltage than the absolute minimum required. This additional voltage is called the guardband [92].

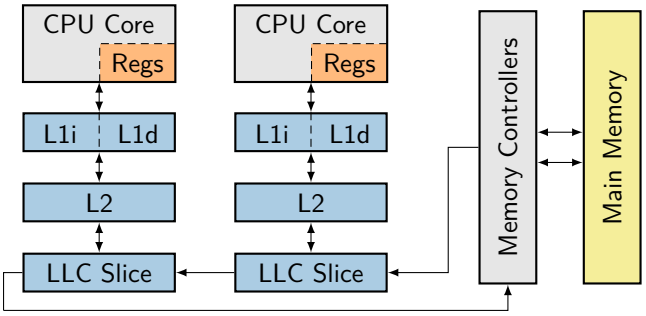


Figure 2.4.: Memory organization in a modern computer. The further away a memory is from the CPU core, the larger and slower it gets.

2.2. Computer Memory

A computer's memory stores all the data the CPU needs during execution. This includes the executed instruction stream as well as the data it processes [259]. In all modern computers, DRAM is used as the main memory because of its high density, low price, and low power consumption. However, the DRAM is too slow to feed enough instructions and data directly to the CPU, which is many magnitudes faster. Therefore, CPUs contain multiple levels of caches that make recently executed instructions and accessed data available with lower latency [237].

A logical layer, virtual memory, partitions the memory between all processes and enables access rights and execution permission management. It is configured with page tables that define to which physical address a virtual address of a specific process is mapped to. The page tables are set up by the operating system and used by the CPU to translate each memory access [98].

2.2.1. Organization

Figure 2.4 shows memory organization from the registers, the memory closest to the CPU, to the main memory, furthest away.

Registers are the memory closest to the CPU's execution units, accessible in a single clock cycle by CPU instructions. In the x86 architecture, instructions can also directly address memory [97]. This is not possible on ARM [9] or RISC-V [267], which have load and store instructions to

access the main memory. There are typically 16 to 32 general-purpose registers. Microarchitecturally, register names are not mapped to fixed locations but are dynamically allocated in the register file, which is usually a few hundred registers large. Registers use static random access memory (SRAM).

Between the CPU cores and the main memory are multiple SRAM cache levels to make data accessible with lower latency. Caches store data that has a high chance of being accessed in the near future. They work based on the principle of locality, which states that recently accessed data or data close by has a higher chance of being accessed again soon. There are multiple cache levels with increasing size and latency. Modern Intel and AMD x86 CPUs typically use three cache levels [6, 97]. The last-level cache is shared between all cores but partitioned into slices. Each core has one slice, and the slices are connected with a ring bus or a mesh [178]. Apart from a handful of exceptions, caches use SRAM [82, 232].

Apart from caches for instructions and data, CPUs also contain a cache for virtual-to physical address mappings, called the translation lookaside buffer, short TLB. It is also split into multiple levels and caches for translations of different page sizes [96].

The main memory is the largest and slowest memory of a computer that is directly accessible by CPU instructions. It uses DRAM technology for its high density and low cost. The main memory is described in more detail in Section 2.3.

2.2.2. Virtual Memory

Modern computers execute many processes simultaneously besides the operating system. For stability and security reasons, these processes must not access the memory of other processes running. All modern CPUs implement a way to isolate processes called paging. On a system with paging, a process never accesses the physical memory directly. Instead, it works in its own private virtual memory space that is translated to physical memory by the CPU.

Paging fragments the physical memory into pages, typically 4 kB large. A set of page tables, unique per process, maps the process's virtual memory pages to physical memory pages. Page tables are managed in a tree structure. The highest-level page table is unique per virtual memory space.

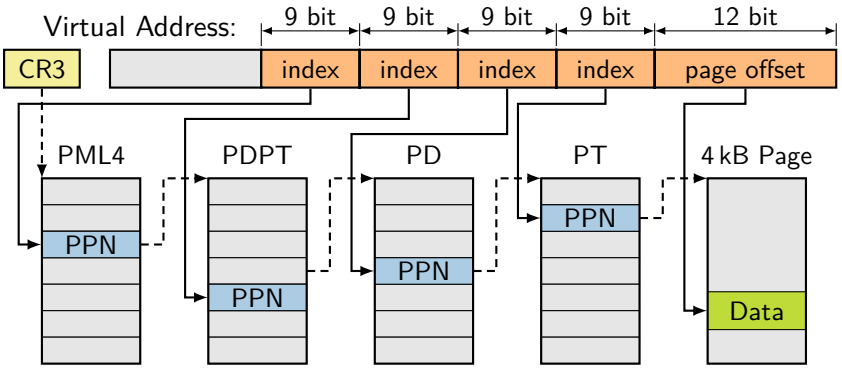
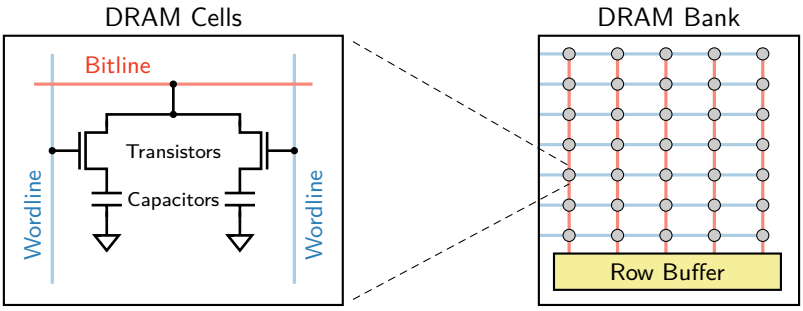


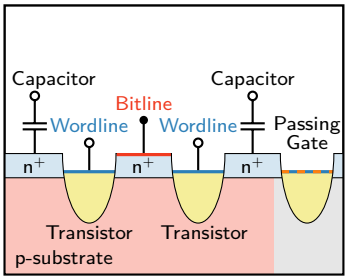
Figure 2.5.: On systems with memory paging, the virtual address is a set of indices into a tree of page tables. The lowest-level page table contains the page frame number or address of the physical memory page.

On x86, the physical address of the highest-level page table is stored in the `CR3` register. Every page table has 512 entries pointing to a lower-level page table. The lowest-level page table points to the mapped page in physical memory. As shown in Figure 2.5, the virtual addresses the process uses are actually only a set of indices into all levels of page tables and an offset within the page.

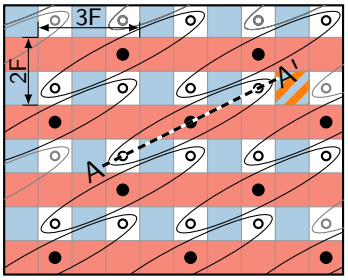
The operating system is responsible for managing the page tables for each process and itself. A process can only access the physical pages that are mapped in its own page-table structure. Apart from page-frame numbers pointing to the next lower-level page table, they also contain various bits that control, for example, access rights, *i.e.*, whether the program is allowed to read, write, and execute memory pages, or the cacheability of the mapped page. Therefore, it is crucial that only the operating system has access to page-table pages and not the user space process itself. Some Rowhammer exploits gain access to their own page tables to escalate privileges [37, 39, 52, 53, 75, 119, 125, 144, 256, 275, 288], see Section 3.1.3.



(a) The structure of DRAM. A cell consists of a transistor and a capacitor. In a DRAM bank, cells are aligned in a grid, connected by the word- and bitlines. The bitlines are connected to the row buffer.



(b1) The cross section of two cells as marked by A---A' in Figure 2.6b2.



(b2) Top view of the cell layout. A single cell is $2F \cdot 3F = 6F^2$ large. The row buffer is not shown.

(b) The physical cell layout of modern $6F^2$ DRAM [88, 170]. The cells are arranged slightly rotated to the word- and bitline grid for maximum density. Passing gates exist because of the grid, but do not connect a bitline to a capacitor.

Figure 2.6.: The schematic and physical layout of DRAM cells and banks.

2.3. DRAM

Dynamic random access memory (DRAM) is a type of memory that is characterized by its low price, low power consumption, and high density compared to SRAM. These properties make it the practically exclusive choice for the main memory of all computers, smartphones, and servers. This makes the DRAM main memory the largest memory of a computer directly accessible by CPU instructions.

2.3.1. DRAM Cell

Its high density and low power consumption in comparison to static random access memory (SRAM) is possible because every bit is stored with only two components, one transistor and one capacitor, also called a storage node. Because these capacitors slowly discharge, they require periodic refreshes not to lose any data, hence, the name *dynamic* RAM [107].

Figure 2.6 shows the schematic and the physical layout of DRAM cells. A cell is connected to two signal lines, as shown in Figure 2.6a. The wordline controls the gate of the transistor, connecting the storage capacitor to the bitline. The charge in the capacitor is read and written through the bitline. Cells are connected through the word- and bitlines in a grid. The ends of the bitlines are connected to the row buffer. By activating a wordline, all capacitors in its row are sensed by the row buffer. DRAM operation is described in more detail in Section 2.3.3.

Figure 2.6b shows the physical cell layout of modern $6F^2$ DRAM [88]. Figure 2.6b1 shows how two capacitors (storage nodes) are connected to a bitline through a MOSFET transistor each. As shown in Figure 2.6b1, the active region (p-substrate) is shared by multiple transistors and their capacitors. This shared active region, which allows electrons to travel between storage nodes, is the reason for the Rowhammer disturbance effect [137] described in detail in Section 3.1.1.

Figure 2.6b2 shows the $6F^2$ layout of the storage node grid in silicon [88]. It is optimized for maximum density by arranging the transistors and capacitors interleaved. The cross section shown in Figure 2.6b1 is marked in Figure 2.6b2 with the dotted line. In the $6F^2$ layout, not all parts of a wordline sit in an active region connecting a capacitor to a bitline. One example of this is marked by the orange-striped field in Figure 2.6b1, also marked in Figure 2.6b2. These regions are no real transistor gates and are, therefore, called passing gates. They are the reason for the passing gate disturbance effect exploited by the Rowpress attack [163], described in more detail in Section 3.1.2.

2.3.2. DRAM Structure and Addressing Functions

The individual DRAM cells are segmented into banks that work independently and allow for parallelized accesses from the CPU. The memory

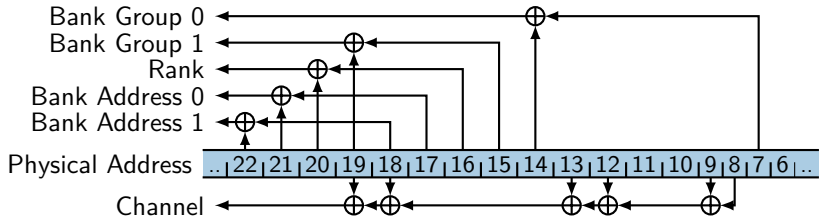


Figure 2.7.: The DRAM addressing functions of an Intel Skylake CPU with two DIMMs, one for each channel and two ranks per DIMM [200].

controller of the CPU applies functions, called the DRAM addressing functions, on the physical address to select a specific bank for each access.

The grids of transistors and capacitors storing the individual bits are split up into banks. Rows of this grid are typically 8 kB wide. There is one row buffer per bank, connected to the rows through the bitlines. On some DRAM modules, the row indexes, *i.e.*, wordlines, are scrambled. This is relevant for Rowhammer attacks, the reason for it, however, is probably due to electrical engineering [244]. A DRAM DIMM contains multiple banks, 16 on DDR4 and 32 on DDR5 per rank. Four banks each are combined into a bank group [107, 108].

A DDR4 DRAM DIMM is connected to the memory controller over a channel. Most consumer CPUs contain two memory controllers and have, therefore, two independent DRAM channels. DDR5 DRAM DIMMs have two independent channels, doubling the number of channels in most CPUs for more parallelism [108]. As the number of physical connections between CPU and DRAM are still approximately the same for DDR4 and DDR5, DDR5 has a doubled burst length to be able to transmit one cache line (64 B) in a single burst over the 32 bit wide channel [108].

Commands to the different independent banks can be interleaved, reducing the overall time the memory controller has to wait for finished commands. To make the best use of this bank-level parallelism, the memory controller interleaves the banks [96] by applying DRAM addressing functions on the physical address to compute the channel, rank, bank group, and bank select bits. DRAM addressing functions were first reverse-engineered by Pessl et al. [200]. The DRAM addressing functions of an Intel Skylake CPU are shown in Figure 2.7. New research followed up with newer tools, getting the functions for newer CPU generations [16, 53, 60, 85, 86, 106, 261]. Jung et al. [122] further reverse-engineered the physical DRAM

mapping by using faults induced by targeted heating to get the on-chip location of DRAM cells.

2.3.3. DRAM Operation

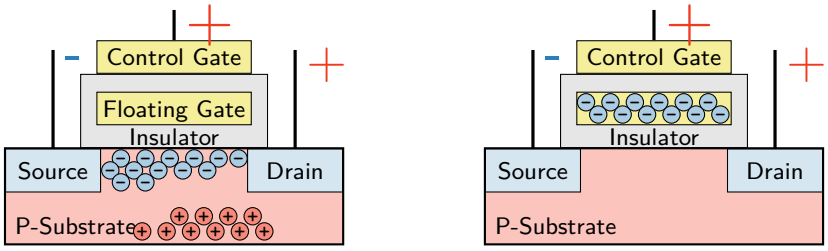
The DRAM protocol is asynchronous. This means that the CPU adheres to specific timings when sending commands, and the DRAM guarantees that every command is finished within that time.

Data Access. To access data on the DRAM, the CPU selects a rank, bank, and row and opens the row to move the data of the row into the row buffer. To do this, the bitlines of the bank must first be precharged (`PRE`). The wordline is activated, connecting the capacitors of a row to the precharged bitlines, slightly changing the voltage. The sense amplifiers detect the voltage change and store the result in the row buffer. Physically, the row buffer and sense amplifiers are one component [170]. This process is destructive, and it removes all charge from the capacitors in the row. Therefore, if there is already data in the row buffer, it must first be written into the previously opened row.

The memory controller can then access the data in the row buffer. The row buffer acts like an 8 kB large cache, decreasing the latency of accesses to the currently opened row. This creates a timing side channel, which was first reported by Pessl et al. [200].

Refresh. The memory controller is also responsible for the refreshes of all cells in the DRAM. It does so by periodically sending refresh commands to the DRAM. The DRAM itself keeps track of which rows were least recently refreshed and can refresh multiple rows within one refresh command window.

The asynchronous nature of DRAM became a problem for on-DRAM Rowhammer mitigations like TRR (see Section 3.1.4), as they only have limited time to perform mitigating refreshes additional to the normal refreshes within a refresh command window. Therefore, the DDR5 standard implements an “Alert Back-Off” (ABO) back channel from the DRAM to the CPU that allows the DRAM to request additional refreshes from the CPU [108].



(a) An erased flash memory cell, storing a logical 1. No electrons are trapped in the floating gate. The voltage at the control gate makes the transistor conductive between the source and the drain.

(b) A programmed flash memory cell. Electrons are trapped in the floating gate, raising the threshold voltage. The same voltage at the control gate does not make the transistor conductive between the source and the drain.

Figure 2.8.: A floating gate MOSFET. To program the cell, electrons are tunneled into the isolated floating gate where they can stay for many years.

2.4. Solid-State Drives (SSDs)

Solid-state drives (SSDs) are storage devices that persistently store data on integrated circuits, typically NAND flash memory. Because they do not use any moving parts, they are faster than hard disk drives (HDDs). Especially, the number of random input-output operations per second (IOPS) is magnitudes higher on an SSD than on an HDD because of the HDD's required disk arm movements for random accesses.

2.4.1. Flash Memory

NAND flash memory is an integrated circuit solid-state memory [8, 280]. The storage of individual bits happens in floating-gate MOSFETs, as shown in Figure 2.8. With a high voltage at the control gate, electrons can be trapped in the floating gate (programming). These electrons increase the threshold voltage of the transistor. The floating gate is fully isolated, and the electrons stay there for many years. To erase a flash memory cell, a voltage in the opposite direction is applied to the control gate to push the electrons back out of the floating gate.

Due to the different high voltage required, NAND flash memory cannot be written randomly, such as DRAM [8, 280]. However, data can be randomly

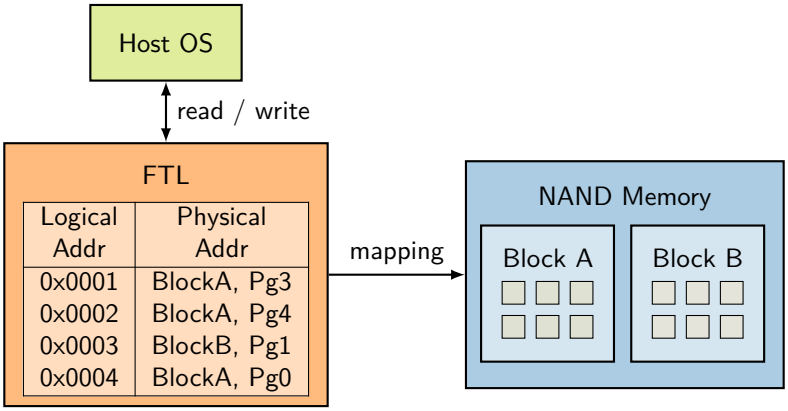


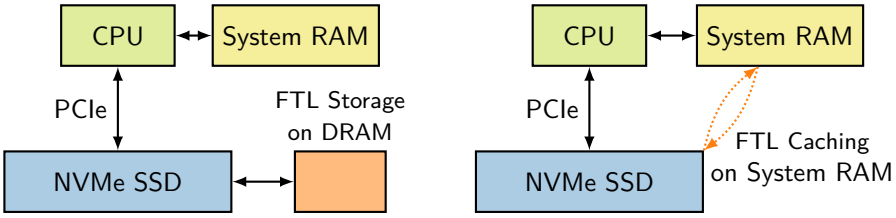
Figure 2.9.: The flash translation layer (FTL) translates the logical address used by the host operating system to the actual physical location of the page in the NAND memory on each access.

read from every location. Erased NAND flash memory stores all `1`'s, to write data, specific bits are then set (programmed) to `0`. Reprogramming is possible as long as the new data is a subset with only `1` to `0` transitions. Erasing NAND flash memory takes significantly longer than reading and programming it. Additionally, the number of bits that must be read, programmed or erased at once differs. Typically, the page size used for reads and programming is 512 B, 2 048 B, or 4 096 B. Erasures happen in blocks of 32, 64, or 128 pages.

The number of these program-erase cycles of flash memory is limited because the insulation between the substrate and floating gate is slightly damaged each time [8, 280]. To extend the lifetime of an SSD, the controller counts the number of P/E cycles each flash memory block and dynamically remaps data to wear out all blocks equally fast. This process is called wear-leveling and is automatically performed by the SSD controller.

2.4.2. Flash Translation Layer

Wear-leveling, garbage collection, and hiding the erase-before-write limitation of flash memory for performance causes the data on an SSD to be scattered [8, 13, 99, 126, 129]. This makes a persistent translation layer necessary that translates the ordered logical page addresses, accessed by the operating system, to the actual physical locations of the data on the



(a) SSD with DRAM. (b) SSD with HMB.

Figure 2.10.: NVMe SSDs with the host memory buffer (HMB) feature can use a part of the system RAM to cache FTL entries.

NAND flash chips. This flash translation layer (FTL) is also stored in the flash memory itself, as it must be persistent.

On every read and write to the SSD, the memory controller must access the FTL to either get the current translation or update it. Therefore, accesses to the FTL must be fast. If the SSD controller were to access the FTL in the flash memory for every read, each read would incur an additional read, basically halving the IOPS. Therefore, SSD controllers use caches for their FTL. Budget SSDs use a small on-chip cache in the SSD controller that caches a small subset of the FTL. High-end "pro"-level SSDs use a separate DRAM chip next to the memory controller that is large enough to hold the entire FTL while the SSD is turned on. When starting, the SSD copies the whole FTL into the DRAM and only reads it from there, greatly increasing random IOPS. Most mid-range NVMe SSDs can use a feature called Host Memory Buffer as a trade-off between cost and performance.

2.4.3. Host Memory Buffer (HMB)

With the HMB feature, NVMe SSDs can request main memory from the operating system that they then use to cache parts of the FTL [185], as shown in Figure 2.10. The operating system can then reserve memory for exclusive access from the SSD. The SSD accesses the HMB through direct memory accesses (DMA) over PCIe. Kim et al. [133] were the first to analyze the HMB usage of SSDs and found that the HMB memory is mainly used to cache parts of the FTL. We found the same to be true [115, 121], see Chapter 8 and 9. Caching parts of the FTL has a performance advantage over accessing the flash memory for each FTL entry, but is

slower than accessing an integrated DRAM. As the HMB is typically not large enough to store the whole FTL, only parts are stored, and different eviction policies and prefetching define which parts [121].

2.4.4. HMB Prevalence

In our HMB side channel paper [121], we used TechPowerUp's SSD database [54] containing 869 SSDs from 109 manufacturers to understand the prevalence of the HMB feature in SSDs. Of these 869 SSDs, 694 use the PCIe interface. PCIe is a requirement of the HMB as SATA does not support DMA accesses. 37 % or 255 of all PCIe SSDs do not have DRAM. Of these DRAM-less PCIe SSDs, 97.6 % or 249 support the HMB feature. This shows that the feature is very prevalent in DRAM-less SSDs, as it enables a performance gain at almost no cost.

3

State of the Art

This chapter discusses state-of-the-art DRAM disturbance attacks and defenses, CPU undervolting, and side-channel attacks. We detail state-of-the-art DRAM-disturbance exploit techniques and mitigations in Section 3.1. Section 3.2, shows how CPU undervolting was used to attack trusted-execution environments and its potentials. Finally, we provide an overview of side-channel attacks in Section 3.3, ranging from contention to storage side-channel attacks and cloud co-location detection.

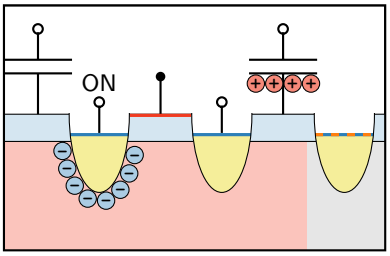
3.1. DRAM Disturbance Attacks

In this section, we discuss state-of-the-art software-based DRAM disturbance attacks and defenses. Software-based DRAM disturbance attacks allow an attacker to flip bits in the DRAM by only accessing it from software. There are two DRAM disturbance phenomena: Rowhammer [137] and Rowpress [163]. Rowhammer was identified as a potential security issue by Kim et al. [137] in 2014. Since then, countless different Rowhammer methods [53, 74, 105, 144, 206, 229], see Section 3.1.1, exploit techniques [21, 26, 37, 39, 52, 53, 75, 103, 119, 125, 144, 149, 213, 215, 249, 256, 275, 288], see Section 3.1.3, and also possible defenses [10, 18, 27, 30, 32, 33, 38, 41, 45, 51, 63, 72, 76, 83, 87, 88, 100, 102, 109, 114, 117, 123, 131, 137, 145, 151, 171, 189, 196, 198, 210, 221, 223, 224, 225, 230, 238, 253, 257, 258, 276, 281, 284], see Section 3.1.4, were published and implemented. Rowpress is another software-based DRAM disturbance attack that was only discovered recently by Luo et al. [163] in 2023, see Section 3.1.2.

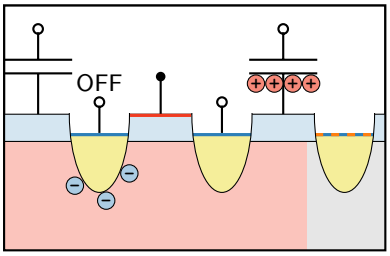
In this section, we show that although more than ten years have passed since the publication of Rowhammer, the problem still exists to this day. A countless number of defenses were repeatedly broken by novel Rowhammer patterns and exploit techniques [39, 53, 74, 105, 144]. With this thesis, we concur with others that a full mitigation of Rowhammer is only possible with full integrity protection of all data in the DRAM [51, 220]. We argue that the underlying problem is that academics, as well as DRAM manufacturers, try to mitigate Rowhammer without it being fully understood yet [74, 144, 163, 188, 277]. Our work, CSI:Rowhammer [117] (Chapter 5) shows that the performance overhead of adding integrity protection to all data in the DRAM is negligible. Integrity protection, in addition to mitigations targeting Rowhammer more directly, could solve the problem of DRAM disturbance attacks once and for all. In our works Presshammer [119] (Chapter 7) and HMB Rowhammer [115] (Chapter 8) we further the understanding of DRAM disturbance attacks.

3.1.1. Rowhammer

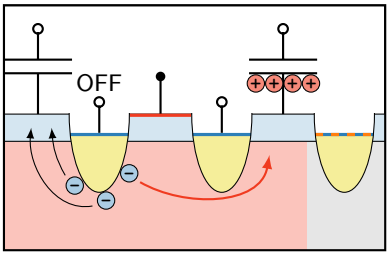
Rowhammer allows an attacker to flip bits in DRAM by frequently accessing (hammering) one or multiple rows adjacent to the victim row [137].



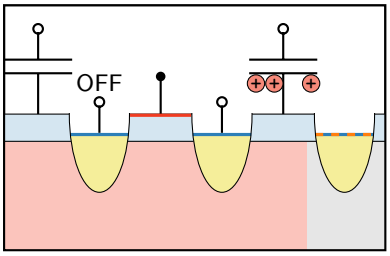
(a) Channel Inversion: When the wordline is activated, electrons collect at the channel to connect the capacitor with the bitline.



(b) Electron Spreading: After the wordline is deactivated, the electrons are free to move around.



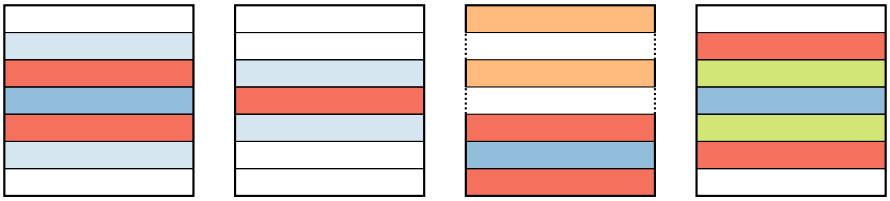
(c) Electron Injection: Most electrons move back into the closest capacitor, but some "leak" and are injected into a neighboring one.



(d) Capacitor Discharge: Every injected electron slightly discharges the neighboring capacitor until the stored bit flips.

Figure 3.1.: How the Rowhammer effect recombines charges in the victim capacitor, flipping the stored bit [88, 137].

Root Cause. The underlying cause of Rowhammer is the shared substrate in which multiple cells are embedded [88, 195, 219, 260, 278], as we explained in Section 2.3.1. Figure 3.1 illustrates the physical cause of the Rowhammer effect. When the wordline is on, the storage capacitor is connected to the bitline, and electrons from the capacitor flow along the transistor's channel. After the wordline is turned off, the electrons are no longer held to the transistor's channel and spread out. Most are injected back into the storage capacitor where they came from, as it is the closest, but some electrons "leak" and are injected into other neighboring storage capacitors. With each leaking electron, the charge in this storage capacitor is depleted slightly. If the wordline is toggled often enough, the charge depletes to a point below the threshold where the bit flips.



(a) Double-Sided (b) One-Location (c) Multi-Sided (d) Half-Double

Figure 3.2.: Four hammer patterns. Double-sided Rowhammer [229] sandwiches the victim (blue) between two attacker rows (red). One-location Rowhammer [74] accesses only a single row in a bank. Multi-sided Rowhammer patterns [53, 105] access additional dummy rows (orange) to evade TRR mitigations. Half-double Rowhammer [144] provokes mitigative TRR accesses (green) around the victim row to flip bits.

Hammer Patterns. Figure 3.2 shows four different hammer patterns. The hammering patterns describe which rows in the DRAM an attacker accesses. Initially, Kim et al. [137] only hammered by frequently accessing multiple single rows in a DRAM bank, *i.e.*, single-sided Rowhammer. In 2015, Seaborn et al. [229] showed that Rowhammer can induce more bit flips if not any two rows are hammered but two rows sandwiching the victim rows, *i.e.*, double-sided Rowhammer.

One-location Rowhammer only accesses a single row of the DRAM and still causes bit flips [74]. This is unexpected because the memory controller usually keeps the row open in the row buffer, serving the data from there. This behavior of a memory controller, keeping the row open as long as possible, is called the open-row policy. However, memory controllers can also work with a closed-row policy where they close a row after each access [74]. This can have performance benefits on highly multi-threaded systems, where subsequent accesses are unlikely to go to the same DRAM row. In Presshammer [119] (Chapter 7), we show that one-location Rowhammer also causes bit flips due to the Rowpress effect that was not yet known when one-location Rowhammer was found.

DRAM implementing the target row refresh (TRR) mitigation, see Section 3.1.4, is not vulnerable to single- and double-sided Rowhammer [53, 83]. To evade TRR, Frigo et al. [53] included decoy accesses to their multi-sided Rowhammer pattern. These decoy accesses are synchronized to refresh commands and can confuse certain TRR implementations to protect the wrong victim rows. Additionally, they can also cause too many

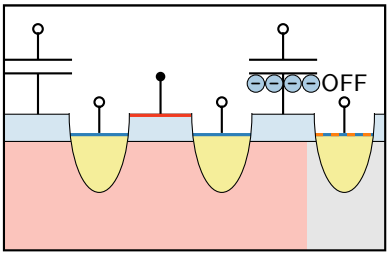
potential victim rows, so that the DRAM is overwhelmed and cannot refresh all of them in the time it has available during the periodic refreshes. Their fuzzer was able to induce bit flips in 13 of 42 tested DDR4 modules. With Blacksmith, Jattke et al. [105] describe the different decoy accesses with a frequency, phase and amplitude and their fuzzer varies them to find long and complex hammer patterns. Jattke et al. [105] were able to flip bits in all of their 40 tested DDR4 modules. Gerlach et al. [61] reproduced bit flips in 8 of 10 tested DDR4 modules using the Blacksmith fuzzer.

TRR can also be exploited as a confused deputy to aid Rowhammer attacks. With Half-Double Rowhammer [144], we present a new physical property of Rowhammer. When hammering one row further away from the victim row (far aggressor), very infrequent accesses to the row neighboring the victim row (near aggressors) are enough to "transport" the leaked stray electrons to the victim row capacitors. This hammer pattern is shown in Figure 3.2d. On DRAM that is protected with TRR, the accesses to the near aggressor are caused by TRR, which tries to protect these rows from the far aggressor. However, due to the half-double effect, these TRR accesses cause the bit flips in the actual victim row. We were able to flip bits on 5 out of 7 mobile devices with LPDDR4x DRAM [144].

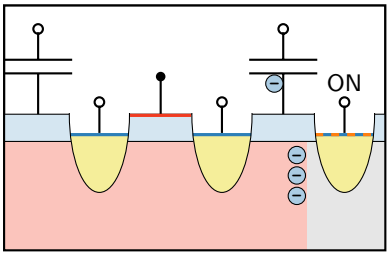
3.1.2. Rowpress

Rowpress also allows an attacker to flip bits in DRAM. Instead of opening and closing a row as quickly as possible, Rowpress keeps (presses) one or multiple rows adjacent to the victim row open for as long as possible [163]. Rowpress is caused by the passing-gate effect and affects different cells than Rowhammer [163]. Figure 3.3 shows how the passing-gate effect causes bits to flip in the DRAM. Due to the physical layout of a modern $6F^2$ DRAM cell, not all gates on a wordline connect a storage node to a bitline. This is shown in Figure 2.6b in Section 2.3. When a wordline is activated, these so-called passing gates on this wordline attract electrons from the storage node. The longer the wordline is active, the more electrons are attracted. When the wordline is turned off, most electrons return to the storage node they came from, but some "leak" further out into the substrate to another capacitor. This reduces the charge in this neighboring capacitor, flipping bits [88, 93].

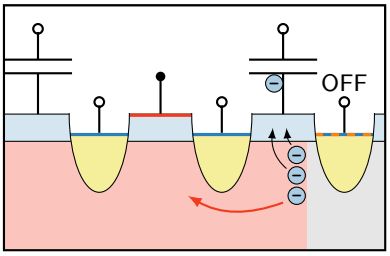
Luo et al. [163] analyzed 164 DRAM chips using an FPGA platform and showed that it is a common DRAM vulnerability across all three major



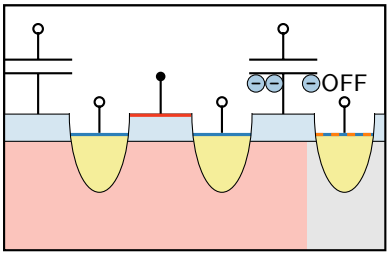
(a) Passing Gate Channel Inversion: When a wordline is activated, electrons from storage nodes are attracted to passing gates.



(b) Electron Spreading: When the wordline is deactivated again, the electrons are free to move around.



(c) Electron Injection: Most electrons move back into the closest capacitor, but some "leak" and are injected into the substrate.



(d) The electrons injected into the substrate are now missing in the capacitor, flipping the bit.

Figure 3.3.: How the Rowpress effect removes charges from the victim capacitor, flipping the stored bit [88, 163].

DRAM manufacturers. Similar to Rowhammer, Rowpress is also more effective if the victim row is sandwiched between two aggressor rows. To cause Rowpress bit flips from software, Luo et al. [163] keep a row open as long as possible by accessing up to all 128 cache lines within one row. They were able to cause Rowpress bit flips in one Samsung DRAM DIMM from 2018. The DIMM we used in the PressHammer paper [119] (Chapter 7) is also from 2017. This suggests that modern Rowhammer mitigations are well equipped to also prevent Rowpress bit flips.

Recent work by Jattke et al. [104] used a DRAM interposer and a oscilloscope to analyze Rowpress on real systems in more detail. They found that the pattern accessing all 128 cache lines within one row keeps the row open for at most 292.95 ns, much less than allowed by the standard.

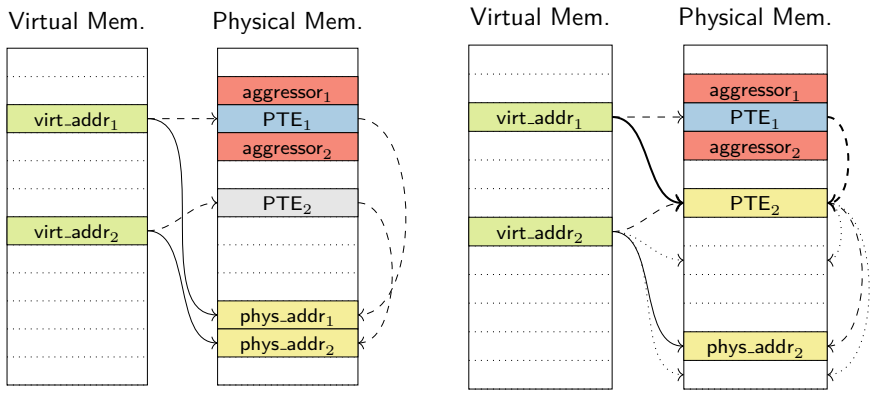
3.1.3. DRAM Disturbance Exploitation

From now on, we will use Rowhammer as a synonym for Rowhammer and Rowpress, as the following concepts apply similarly to both DRAM disturbance attacks. Rowhammer flips bits at seemingly random locations in the DRAM. Apart from the different hammering patterns to get bit flips, actually causing Rowhammer bit flips from unprivileged programs and exploiting these bit flips is well-researched [21, 26, 35, 37, 39, 52, 53, 75, 103, 119, 125, 144, 149, 213, 215, 249, 256, 275, 288].

DRAM Addressing. Programs running on a CPU cannot directly address specific rows in the DRAM. Two layers of indirection, *i.e.*, mappings are between the address a program accesses and the row and bank addressed in the DRAM [200]. The first indirection is virtual memory, as described in Section 2.2.2, that maps the seemingly contiguous and private virtual memory space to physical memory [98]. The second indirection is the DRAM addressing functions, which map the physical addresses to ranks, bank groups, banks, and rows, described in Section 2.3.2.

The virtual-to-physical address mapping is not readable by unprivileged processes anymore after it was disabled to prevent the first Rowhammer exploits [138]. Therefore, many different methods to get physically contiguous memory were proposed [75, 144, 256]. Gruss et al. [75] were the first to use 2 MB transparent huge pages. Web browsers automatically allocate them for large arrays [75], and they can also be requested from Linux using `madvise` on most systems [154, 247]. However, transparent huge pages are not available on all systems, e.g., Android, and can easily be turned off on systems where they are enabled by default [247]. Alternatively, Van der Veen et al. [256] massaged Linux's buddy allocator [69] to get predictable physical page placement. Memory allocator massaging was also used by multiple exploits afterwards [52, 149, 215]. Other exploits [52, 144] used the fact that when allocating enough memory, chunks will be contiguous, and use the bank conflict side channel to detect this contiguity.

Contiguous memory gives an attacker knowledge about additional physical address bits above the page offset. These bits can then be used with the DRAM addressing functions, which we explained in Section 2.3.2, of the attacked CPU to address rows in specific banks [52, 144].



(a) Correct Mapping (b) Successful Exploit

Figure 3.4.: The Rowhammer page table exploit. By hammering PTE_1 using $aggr_1$ and $aggr_2$ the PFN is changed to point to the physical address of PTE_2. Now the attacker has write access to PTE_2 through its virtual mapping $vaddr_1$ and can access the whole physical memory.

Exploit Targets. Rowhammer can be used to attack a large variety of systems. These attacks can be classified by their goal, e.g., privilege escalation in the operating system or the browser, stealing of cryptographic keys, or denial of service, and the target, e.g., Intel or ARM CPUs.

The first two Rowhammer exploits were presented by Seaborn et al. [229]. They flipped bits in machine code executed in the Chrome NaCl sandbox to disable the jump target sanitization, so that they could jump to unaligned targets, e.g., a `syscall` instruction hidden in a `movabs` instruction. The second exploit they developed is the most used Rowhammer privilege escalation exploit to this day [37, 39, 52, 53, 75, 119, 125, 144, 256, 288]. It uses Rowhammer to flip bits in the page-frame-number in a page-table entry to gain write access to a page table, as shown in Figure 3.4. This exploit can also be executed by hammering through implicit page-table accesses [287, 288]. A similar technique was also demonstrated against hypervisors [35, 275].

Rowhammer is also possible from JavaScript [26, 52, 75, 215]. A similar approach to the page table exploit was used in browser sandbox escapes [26, 52, 215]. For this exploit, a fake array object is created, and a pointer is hammered to point to this object. This fake array object can then be modified to point everywhere in the browser's memory. Tatar et al. [245] and Lipp et al. [156] demonstrated Rowhammer attacks over the network.

Gruss et al. [74] were the first to flip bits in a binary to elevate privileges. They identified 29 different offsets in the `sudo` binary that can be flipped to break the password verification logic. They combined this technique with one-location Rowhammer from within an SGX enclave for a stealth attack.

Kwong et al. [149] shows that Rowhammer can also be used to directly leak data, exploiting the data dependency of Rowhammer [137]. Tobah et al. [249] used Rowhammer on kernel code to reenable Spectre attacks. Rowhammer can also be used to steal cryptographic keys [21, 213], also in post-quantum schemes [7, 180]. Jang et al. [103] and Gruss et al. [74] exploit Rowhammer for denial of service attacks by halting SGX enclaves. Recently, Jattke et al. [106] demonstrated that Rowhammer is also possible from AMD CPUs, and Marazzi et al. [169] demonstrated Rowhammer on RISC-V. The first exploit on ARM was already presented in 2016 [256] using uncachable memory, followed by Rowhammer from the integrated ARM GPU [52]. Weissmann et al. [269] performed a Rowhammer attack from an FPGA through DMA. Qiao et al. [206] demonstrated Rowhammer using non-temporal instructions and Heckel et al. [84] increased the hammer effectiveness by up to 830 with multithreaded Rowhammer using the `cmpsb` and `repe` instructions.

In our work "An Analysis of HMB-based SSD Rowhammer" [115] (Chapter 8), we analyzed whether SSDs using a host memory buffer, see Section 2.4.3, can be used as a confused deputy to hammer the HMB in the main memory. However, our results show that the HMB itself is integrity protected, and bit flips are detected. Upon detection, all of our tested SSDs freeze, leading to a denial of service or a data loss in the worst case. Additionally, the SSD's accesses to the HMB are not frequent enough to induce bit flips even on highly susceptible DRAM.

3.1.4. Software-Only Defenses

Since the first publication about Rowhammer, attackers and defenders have been constantly working to outsmart each other [74, 181]. Additionally, the ever-shrinking node size exacerbates the problem with every new DRAM generation [132]. While attackers are mostly academic researchers, Rowhammer defenses were proposed by academia but also developed and implemented by DRAM and CPU manufacturers [109]. Defenses can be categorized into software-only defenses, which are easy to implement and

deploy, and defenses (partly) implemented in hardware, which are more difficult to deploy but potentially more effective.

Especially in the first years of Rowhammer research, many software-only defenses were proposed [10, 30, 41, 100, 145, 257]. As Rowhammer attacks cause suspicious CPU cache usage, similar to CPU cache side-channel attacks, cache side-channel detection methods were also proposed to detect ongoing Rowhammer attacks [38, 76, 87, 100, 198, 284].

ANVIL by Aweke et al. [10] detects ongoing Rowhammer attacks using cache hardware performance counters similarly to cache side-channel detection works [38, 76, 87, 198, 284]. ANVIL reduces false positives by only intervening if at least two rows on the same bank are hammered. Machek [41] proposed counting the number of cache misses in the kernel and halting the CPU until the next refresh if it exceeds a preconfigured threshold. Irazoqui et al. [100] use static code analysis to detect instructions often used in cache and Rowhammer attacks like `clflush`, non-temporal moves, `rdtsc`, or fences. Brasser et al. [30] add guard rows around kernel memory to increase the distance between potential aggressor and victim rows. GuardION by Van der Veen et al. [257] is a mitigation against their attack on Android ARM devices [256]. They put guard rows around DMA memory requested using Android's ION driver. ZebRAM [145] also works by adding guard rows, however, between every row. It then uses all these guard rows as an integrity-protected swap memory to not halve the memory capacity.

Di et al. [45] present "Copy-on-Flip", a defense that can be purely implemented in software if the system uses ECC DRAM. This makes it a software-only defense relying on a readily available hardware feature. ECC DRAM is not enough to prevent a Rowhammer attack, as shown by Cojocar et al. [39]. However, the exploit of Cojocar et al. [39] requires a templating phase where it collects single-bit flips that are corrected by ECC to combine these bit flips in the second step. Copy on flip [45] prevents this by remapping pages whenever correctable bit flips happen.

3.1.5. Hardware-Based Defenses

Over the years, improving attacks have shown that software-only defenses are insufficient to solve Rowhammer [74]. Therefore, many mitigations that require hardware changes have been proposed and also implemented [18, 27, 32, 33, 51, 63, 72, 83, 88, 102, 109, 114, 117, 123, 131, 137, 151, 171,

189, 196, 210, 221, 223, 224, 225, 230, 238, 253, 258, 276, 281]. These hardware-based defenses can be classified into defenses that prevent bit flips and defenses that detect bit flips to correct them. Both methods have disadvantages: While preventive defenses can become obsolete with new attack methods, correcting detected bit flips becomes inefficient quickly if bit flips are too frequent. We argue that combining both methods can most efficiently prevent current and future Rowhammer attacks.

Defenses Preventing Bit Flips

This class of defenses tries to prevent bit flips by protecting potential victim rows before their cells are discharged below the threshold [18, 27, 32, 88, 102, 110, 114, 134, 136, 137, 151, 168, 171, 189, 196, 209, 210, 221, 225, 230, 238, 253, 258, 271, 272, 276, 281]. We further classify these defenses into three different methods of identifying potential victim rows and by their actions after a potential victim row is identified.

Probabilistic Victim Row Identification. Already with the first paper on Rowhammer, Kim et al. [137] proposed a number of potential mitigations, the most prominent being probabilistic adjacent row activation (PARA), now called probabilistic target row refresh (pTRR) [123]. Intel has implemented pTRR in some of its CPUs since 2014 [123]. With pTRR, whenever a row is accessed, its neighboring rows are refreshed with a low probability. If the probability is chosen well, the performance overhead is negligible, while Rowhammer bit flips become very unlikely. Similar probabilistic mitigations were proposed by Son et al. [238], You et al. [281], and Kim et al. [136].

Qureshi et al. [210] use only a single counter to count row activations and probabilistically select which row is counted at each refresh. Jaleel et al. [102] use a small number of counters per bank and do not use heuristics to select which row is counted intentionally. Because of that, they claim to be immune to specifically crafted access patterns that manage to break policy-driven defenses [53, 105].

Counter-Based Victim Row Identification. DRAM manufacturers and JEDEC acted rather quickly and added target row refresh (TRR) to many later DDR4 DIMMs and as a required feature to the LPDDR4 standard [109]. TRR does not probabilistically refresh neighboring rows

but actually counts the accesses to rows and refreshes the neighbors if a certain DIMM-specific threshold is reached [72, 83, 109, 112, 131].

However, to save die area and power consumption, these first TRR implementations only counted a limited number of rows [53, 105]. With complex access patterns, the sampler can be tricked to count the accesses to decoy rows and not the actual aggressors [53, 105], as explained in Section 3.1.1. Another issue is that the DRAM has only limited time to perform preventive row refreshes. If more victim rows are hammered than the DRAM can refresh, bit flips still happen, even if the DRAM detected the victim rows in time [53, 168, 171, 253]. A third issue with TRR, half-double Rowhammer, was presented by Kogler et al. [144] where TRR actually aids the attacker to hammer from a greater distance, as explained in Section 3.1.1. Half-double Rowhammer can be mitigated by increasing the range of protected rows around a detected aggressor row [33, 83, 171, 189, 210, 223, 276].

A large body of works continuously improves the area-, performance- and power-overhead of counter-based mitigations [18, 27, 88, 102, 134, 168, 171, 189, 253]. These works aim to make the mitigation cheaper and at the same time more difficult to evade.

Hong et al. [88] propose a stochastic and approximate counting (DSAC) algorithm that filters out the accesses to decoy rows that evaded prior TRR implementations. However, DSAC was later broken by Qazi et al. [205] due to the internal state of the used LFSR random number generator being too small. Bostanci et al. [27] use a data structure called count-min sketch to estimate the activation count of rows, requiring significantly less area than having a counter per row. This design may overestimate but never underestimates the activation count. Olgun et al. [189] make the key observation that workloads usually access the same row ID in multiple banks due to the DRAM addressing functions aiming for bank-level parallelism, see Section 2.3.2. Therefore, they use the Misra-Gries algorithm to track aggressor rows, not per bank but shared across all banks.

Per Row Activation Counting (PRAC). Jedec recently added per-row activation counting (PRAC) to the DDR5 standard [108, 222]. A DRAM with PRAC contains a counter for every single row in each bank. Bennett et al. [18] showed that this approach is viable by using a novel

DRAM mat design that does not add any performance overhead. Bennett et al. [18] also use the already existing ALERT_n signal to pause the memory controller, called ALERT-Back-Off (ABO). This is required to perform additional preventive refreshes that did not fit into the refresh commands.

Canpolat et al. [33] are the first to perform a comprehensive study of the PRAC feature as defined in the DDR5 standard [108]. They show that due to the ABO signal, a crafted adversarial access pattern can hog up to 94 % of DRAM throughput and degrade system throughput by up to 95 %. Woo et al. [271] and Qureshi et al. [209] both show that the initial design proposed by Bennett et al. [18] is insecure, additionally to the potentially high overhead found by Canpolat et al. [33]. Both works propose secure designs with less performance overhead.

Other Victim Row Identification Methods. Apart from the designs that track potential victim rows in the DRAM, some designs use other methods inside the memory controller. Vig et al. [258] use a sliding window mechanism to detect currently attacked victim rows and add them to an integrity tree to protect the data in them. Seyedzadeh et al. [230] use adaptive trees of counters, Lee et al. [151] time window counters, Park et al. [196] content addressable memory [110], Yaglikci et al. [276] bloom-filters and Joardar et al. [114] machine learning.

Preventive Refresh. Most proposed defenses refresh victim rows after they exceed the activation threshold. To have enough time to perform all required refreshes, the DDR5 standard adds two mechanisms [108]. The first one is Refresh Management (RFM). The memory controller keeps track of the number of activations sent to the DRAM banks and gives a bank additional time for refreshes by sending RFM commands if the activations exceed a threshold. The second mechanism is the ALERT-Back-Off (ABO) signal that the DRAM can send to the memory controller to force it to send RFM commands. Jattke et al. [104] recently found that neither Intel nor AMD CPUs send RFM commands even if the DRAM requires them to mitigate Rowhammer effectively.

Recent works also looked at ways to make preventive refreshes more efficient. Rega [171] proposes a DRAM design with an additional set of buffering sense amplifiers that are only used for data transfers. These free up the other sense amplifiers to perform refreshes in parallel. Tugrul et al.

[253] study real DRAM chips and find that the refresh latency t_{RAS} can be decreased by 64 % while requiring only 0.54 % additional refreshes [253].

Row Remapping. Some defenses do not refresh potential victim rows but also remap them to another location in the DRAM bank to shield them from future attacks. Saileshwar et al. [221] detect aggressor rows similar to Park et al. [196], however, they do not refresh the victims but swap the aggressor rows with other randomly selected rows. This is to prevent attacks like Half-Double [144]. Saxena et al. [225] propose a very similar mitigation, remapping the aggressor rows to a quarantine area. Woo et al. [272] find an attack pattern that breaks the defense by Saileshwar et al. [221] in under a day and propose a more secure design themselves.

Data Integrity-Based Defenses

These defenses aim to detect data integrity violations and then correct the bit flips whenever possible. Multiple mitigations have been proposed using message authentication codes or hashes to ensure the integrity of data in the DRAM [51, 91, 220]. However, all of them have shortcomings regarding the threat model of a Rowhammer attack. Ivec [91] and Synergy [220] protect against local attackers with capabilities exceeding those of a Rowhammer attacker. Their integrity trees reduce the performance significantly. Safeguard [51] can only correct a single bit flip.

In our work PT-Guard [224], we propose a mitigation to protect page tables from bit flips because they are an easy attack target for privilege escalation [35, 39, 52, 53, 75, 119, 125, 144, 256, 275, 288]. PT-Guard [224] combines unused bits in groups of 8 page-table entries to store a 96-bit MAC for data integrity protection.

In CSI:Rowhammer [117] (Chapter 5), we protect all data against integrity violations using a MAC, albeit with a small memory overhead similar to ECC DRAM. This approach guarantees that no bit flips, whatever their reason is, go undetected and can be exploited. The only remaining possible attack on a system with CSI:Rowhammer would be a denial of service. We show that the performance overhead is small, and correction of multiple bit flips is realistic as we perform the correction in the kernel. This correction of multiple bit flips in the kernel enables techniques like reloading corrupted cached data from disk. However, CSI:Rowhammer does not have Chipkill-like correction capabilities [43] for all data, similar

to current SECDED ECC. We argue, that the possibility to correct bit flips spread over multiple DRAM chips is more important if Rowhammer is a threat. Future work could focus on further improving the trade-off between performance and Chipkill-like correction capabilities of data integrity-based defenses.

Currently, no data integrity-based defense has the correction capabilities to efficiently correct the high number of bit flips in modern high-density DRAM if they were otherwise unprotected. Data integrity-based defenses must be paired with defenses preventing bit flips. On the other hand, defenses preventing bit flips should also be paired with data integrity-based defenses to protect against novel attack methods [144], incomplete defense implementations [104], or other causes of bit flips [43].

3.2. CPU Undervolting

As described in Section 2.1.2, digital circuits are typically supplied with a slightly higher voltage than the minimum required. This so-called voltage guardband guarantees the circuit's correct functionality even when changing die temperature, aging, or supply voltage droops [70, 197]. Reducing this voltage guardband saves energy and can even increase performance on modern CPUs [70, 116]. This is due to the fact that the CPU clock rate is typically limited by the thermal design power (TDP). The TDP defines the maximum power the CPU is allowed to draw for a prolonged time, or the maximum CPU die temperature directly. With a reduced power consumption due to a lower supply voltage, the CPU can clock higher before reaching the TDP. Not surprisingly, given these benefits, the undervolting potential of CPUs and resulting power savings and performance increases were thoroughly researched [68, 70, 124, 146, 193, 194].

However, CPU undervolting can also be used to attack trusted-execution environments [15, 130, 142, 179, 208, 243]. On many CPUs, the multiplication and AES circuits are among the circuits with the highest voltage requirement. If this requirement is not met, the calculations can produce faulty results. This can be exploited by undervolting CPUs to precisely the voltage level where the CPU continues running normally, but these instructions fail [130, 179, 208, 243]. Tang et al. [243] attack ARM TrustZone by overclocking the CPU, which has the same effect as undervolting. Qiu et al. [208] manipulate the voltage to attack ARM TrustZone. Murdock et al.

[179] and Kenjar et al. [130] exploit Intel SGX enclaves by undervolting to steal cryptographic keys or induce memory safety into bug-free enclaves.

Barenghi et al. [17] analyze different countermeasures to protect cryptographic algorithms against fault attacks. Their framework can automatically add instruction duplication, instruction triplication, and parity checking for stored values to programs. Kogler et al. [142] presented a targeted countermeasure against CPU undervolting attacks against trusted execution environments. They analyzed a range of CPUs and confirmed that integer multiplication is the first faulting instruction on most CPUs at different frequencies. An enclave can therefore use multiplications where the correct results are known as trap instructions to detect whether the CPU is undervolted.

A number of works also try to enable stable and secure undervolting [11, 12, 172], including our work SUIT [116] (Chapter 6). Bacha et al. [11, 12] utilize on-chip ECC of Intel Itanium CPUs to guide their undervolting. On Intel Itanium CPUs, the cache and register file are the first components that become erroneous when undervolting. Intel Itanium CPUs can also report corrected ECC errors in the cache and register file to the operating system. This allows Bacha et al. [11, 12] to keep the voltage at exactly a level where no error occurs. Unfortunately, errors in multiplication circuits cannot easily be detected. Koutsovasilis et al. [147] undervolt the CPU depending on the running workload based on the performance monitor counter changes it causes. Therefore, Maroudas et al. [172] additionally also differentiate between kernel and user space, based on their observation that user space code can be undervolted more than kernel code. They do, however, require voltage changes on every kernel entry and exit. Ernst et al. [49] proposed a hardware solution to enable secure undervolting. Their design Razor, uses slightly skewed clock edges and shadow circuitry to detect when critical paths are close to violating their timing constraints. Razor would add considerable complexity to modern chip designs and is not used in practice. In our work SUIT [116] (Chapter 6), we trap potentially faulting instructions when undervolting to only execute them while the CPU is supplied with a high enough voltage. The performance overhead of our design is smaller than the potential performance gain from undervolting, enabling a higher performance at a lower CPU power consumption.

3.3. Side-Channel Attacks

In this thesis, we will only focus on software-based side channels. Software-based side channels do not require physical access to the device that is attacked. The first software-based side-channel attack by Kocher [139] leaked keys from Diffie-Hellman, RSA, and DSS via timing. Percival [199] was the first to perform a Prime+Probe attack on CPU caches, attacking an RSA algorithm. Osvik et al. [191] performed the first Prime+Probe attack on AES T-tables and coined the term Prime+Probe. Bernstein [19] timed the execution of AES T-tables over the network with many different messages to leak the key. Since then, the very high spatial and temporal resolution of CPU cache side channels has been exploited for a variety of attacks that became more generic compared to attacks on specific cryptographic implementations [76, 77, 89, 176, 202, 204, 211, 279, 290].

Recently, with CPU caches being very well researched, more research has focused on other parts of a computer system like, execution units and schedulers [1, 4, 22, 57, 58, 218, 265], the memory bus [90, 273, 274], operating system data structures [111, 164, 165], DRAM row conflicts [200, 228], the page cache [73], software-based power and frequency scaling [143, 157, 159, 207, 263, 264], device sensors [174, 186, 231], CPU fans [79], performance counters [59, 78], the PCIe bus [65, 234, 241, 248], in-memory computing architectures [48, 266], FGPAs [65, 203, 250, 251], interrupt detection [40, 212, 283], GPUs [3, 47, 66, 113, 183, 242, 262, 268], and random number generation logic [50]. However, while storage has been identified as a potential source for side- and covert channels long before the first cache side channels [90, 128, 150, 162, 226, 252], the research on commercial off-the-shelf SSDs is sparse.

Covert Channels. Side channels typically have a specific target, e.g., cryptographic keys, with an attack on this target evaluated in the work. However, comparing their capabilities became a non-trivial task given the wide range of side channels. One way to compare side channels is by their channel capacity, the rate at which information can be transmitted over the channel. To measure the channel capacity, a covert channel is well suited as the sender and receiver are cooperating. Most side-channel papers implement and benchmark a covert channel [4, 22, 50, 57, 58, 65, 73, 76, 81, 90, 111, 165, 175, 176, 200, 204, 211, 212, 228, 234, 250, 251, 265, 273, 274, 283]. Covert channels are also used in transient-execution attacks to transmit the transiently leaked data to the outside world [140, 158].

Covert channels were also already researched long before the first software-based side-channel attacks. Lampson [150] was the first to recognize the difficulty of confining a program on a system with a shared operating system and shared hardware. This was followed by further research on covert channels and their mitigation [90, 128, 162, 226, 252]. However, these works focused on special "secure systems" as, for example, defined by the Trusted Computer System Evaluation Criteria [44] and not commercial off-the-shelf systems, which are the target of most recent side-channel research.

Contention Side Channels. The first covert channels by Lampson and others [90, 128, 150, 162, 226, 252] exploited contention on a shared resource, e.g., the hard drive. Apart from these secure systems [44], contention was mainly a concern for the system's performance. With the emergence of multiprocessor systems, contention on the shared memory or last-level cache became a concern [173, 214, 254]. Contention was also researched in databases [2, 24].

If contention can be focused on a small subset of a system, e.g., execution ports or schedulers, powerful attacks are possible from the leaking of encryption keys [1, 4, 22, 58] and remote attacks from JavaScript [57, 218]. Contention on the network has also been shown to leak private user information [55, 62, 246]. Memory bus contention [273, 274], as well as HDD contention [155], has been used for covert communication. The additional contention caused by RowHammer mitigation-induced memory latency differences has been used for covert communication and website fingerprinting [29]. PCIe contention has been shown to enable a variety of attacks [65, 234, 241, 248] if PCIe switches or PCIe platform controller hubs are used to share a link, which is not generally the case for SSDs. As a part of this thesis [120] (Chapter 10), we show that contention on SSDs can be used for covert communication and allows to fingerprint websites visited by the victim with very high accuracy.

Storage Side Channels. Shared storage has been identified very early as a potential source for side channels [150]. Karger et al. [128] exploit the optimizations of hard disks' arm movements to build a covert channel. Hard disks were again attacked more recently, first, by Lipinski et al. [155], who presented a contention-based covert channel with up to 0.1 bit/s. Biedermann et al. [23] measured the electromagnetic emissions of a hard

drive using a smartphone to perform operating system and application fingerprinting. Finally, Guri et al. [80] measured the acoustic emissions of a hard drive to build a covert channel. Guri et al. [80] highlight that SSDs mitigate this covert channel as they do not emit noises. Chen et al. [34] also suggest that SSDs would mitigate certain timing side channels that are present in HDDs.

Trochatos et al. [250, 251] focused on smart SSDs that include a user-programmable FPGA to enable near-storage compute [152]. They use them to build a covert channel between users using the smart SSD consecutively and between those using it simultaneously. Lui et al. [161] analyze the (now discontinued) Intel Optane persistent memory for side channels and reverse engineer the internal cache to develop four new attacks: a covert channel, a keystroke-timing attack, a fully remote covert channel, and a note board attack. Gruss et al. [73] exploit the operating system's page cache that caches recently accessed storage pages to build a 7 kB/s to 273 kB/s covert channel, an ASLR break on Windows 10, a UI redressing attack, and a keystroke-timing attack. Jiang et al. [111] exploit the `fsync` syscall on the file system that writes dirty data back to storage to build a 20 kbit/s covert channel.

However, even though the storage subsystem has long been known to be a source of side-channel leakage and research exists on HDDs [23, 80, 128, 155], smart SSDs [250, 251], and Intel Optane persistent memory [161], this thesis is the first to analyze the side-channel leakage of regular off-the-shelf SSDs. In our work "Secret Spilling Drive" [120] (Chapter 10), we show that contention on SSDs can be used for covert communication and allows to fingerprint websites with very high accuracy.

Cloud Co-Location. In cloud environments, multiple tenants are running on the same shared hardware. Many of them store confidential data of themselves or their customers. Therefore, cloud environments have long been a target of side-channel research. A large body of works studied attacks [20, 176, 240, 274, 285, 286] but also their mitigation [135, 160, 284]. For most of these attacks to work, the attacker must be co-located with the victim on the same physical hardware. This motivated a specialized branch of research on cloud co-location [94, 95, 166, 217, 289], with cloud providers constantly eliminating known channels for co-location detection [95]. Ristenpart et al. [217] were the first to show that targeted co-location is feasible in the AWS cloud. Inci et al. [95] use the last-level

cache as a covert channel to detect co-location and mount an attack on RSA. Makrani et al. [166] massage the resource provisioning systems with their attack, called Cloak & Co-locate, to get targeted co-location. Zhao et al. [289] achieve co-location in Google Cloud's Function-as-a-Service environment and steal secret ECDSA nonce bits using a cache side channel [290].

In our work "Not So Secure TSC" [118] (Chapter 11), we show that the SecureTSC feature of AMD's confidential computing platform SEV SNP enables very fast and low-overhead co-location detection.

4

Conclusion

In this thesis, we presented novel Rowhammer and side-channel attacks. Motivated by these attacks, we explored how hardware-software co-designs can be effective and efficient mitigations against software-based fault attacks. We conclude this thesis with three key insights:

DRAM disturbance attacks like Rowhammer are still not fully understood, motivating the need for principled mitigations. After more than 10 years, Rowhammer is still not perfectly mitigated because of an incomplete understanding of DRAM disturbance effects [74, 144] and insufficient mitigations [105, 106, 125]. We demonstrated that phenomena like one-location Rowhammer can exist for years before being fully understood [119] (Chapter 7). We also extended Rowhammer research to new hardware and showed that SSDs using the main memory can also hammer it [115] (Chapter 8). *It is likely that future research* will uncover further previously unknown links between existing DRAM disturbance effects. Future research must also continuously assess the Rowhammer risk in new threat models that include peripheral hardware similar to SSDs. Every PCIe revision doubles the throughput, doubling the potential hammer rate of DMA accesses to the main memory. The same danger comes from novel hardware accelerators like neural processing units that access the DRAM directly. When compute-in-memory architectures become widespread, their susceptibility to Rowhammer and also their potential to be used in an attack will be a relevant research question [28, 182, 282]. With a continuously increasing understanding of DRAM disturbance attacks and novel threat models, mitigations focusing on the currently existing attacks are unlikely to guarantee sustained security. Only principled mitigations that can guarantee data integrity regardless of the specific attack and simultaneously provide strong correction capabilities can sustainedly solve the problem [117] (Chapter 5).

Principled mitigations can be inexpensive and even increase efficiency. To prevent hardware faults under all circumstances, manufacturers add guardbands to properties like timings and supply voltages. By employing principled mitigations against hardware faults, the guardbands can be reduced as they are not longer solely responsible for preventing faults. The guarantee to detect all data corruption in the DRAM of CSI:Rowhammer [117] (Chapter 5) could also be used to optimize the refresh rate of DRAM dynamically. In the future, the memory controller could reduce the refresh rate to find a sweet spot between the energy "wasted" on bit flip correction and the energy savings from the reduced refresh rate. As the reduced refresh rate also increases the performance of the DRAM, this could lead to a faster and more energy-efficient system. With increasing hardware-software system complexity, *it is likely that future research* will increase the role of software in the handling and correction of bit flips, for example, by performing bit flip correction in user space or by taking knowledge about the DRAM's structure and properties into account to reduce the search space significantly. In our work SUIT [116] (Chapter 6), we showed that the CPU voltage can be significantly decreased if the CPU guarantees that potentially faulting instructions are not executed with this lower voltage. Overall, SUIT can decrease the power consumption of the CPU while even increasing the system's performance. To improve CPU efficiency further, with more fine-grained voltage control based on more factors than just specific instructions, *we need further research* to understand the CPU's susceptibility to faults and crashes in low-voltage scenarios. Additionally, the operating system could also interact with the CPU to inform the CPU about required voltage levels in the near future. This could significantly reduce the overhead from CPU voltage change delays, similar to cache prefetchers.

For new hardware, known classes of problems, like side channels, are often not considered in the design phase, leaving the new hardware vulnerable. Even though SSDs are widely used in almost all computers, they have not yet been studied as a source of side channels. We close this research gap and perform the first side-channel analyses on modern commodity off-the-shelf SSDs, presenting two novel software-based timing side-channel attacks on SSDs. The first attack is a cache timing side channel that provides leakage with high spatial resolution in combination with a software interface [121] (Chapter 9). The second side channel leaks SSD contention with high temporal resolution [120] (Chapter 10). We performed a large study of SSDs' susceptibility to this contention side channel and showed that all tested SSDs were vulnerable. These works show that single components

like SSDs can be the source of multiple side channels, and *it is likely that future work* will uncover many more, given their complexity and wide use. Similar to potential DRAM disturbance attacks, this also extends to other hardware components where research is sparse, either because they were newly introduced or received little attention from the academic community until now. While storage contention has already been identified as a source for side channels for a very long time [150], we are the first that showed that it also impacts high performance NVMe SSDs. The high performance actually enables more fine-grained attacks, such as fingerprinting the subtle signals from websites being opened on the machine with very high accuracy. As the performance is continuously increasing, it is likely that these attacks will get worse in the future. In our work on AMD's Secure TSC [118] (Chapter 11), we showed another example of a new feature that was introduced without taking side channels into account. The Secure TSC timer allows an attacker to detect the co-location of virtual machine guests. Without a principled approach to consider known classes of problems, in particular, side-channel leakage will continue to be introduced in new hardware components and features.

References

[1] Onur Aciicmez and Jean-Pierre Seifert. Cheap Hardware Parallelism Implies Cheap Security. In: FDTC. 2007 (pp. 47, 48).

[2] Rakesh Agrawal, Michael J Carey, and Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. In: ACM Transactions on Database Systems (TODS) (1987) (p. 48).

[3] Jaeguk Ahn, Cheolgyu Jin, Jiho Kim, Minsoo Rhu, Yunsi Fei, David Kaeli, and John Kim. Trident: A Hybrid Correlation-Collision GPU Cache Timing Attack for AES Key Recovery. In: HPCA. 2021 (pp. 5, 47).

[4] 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 (pp. 5, 47, 48).

[5] AMD. AMD64 Architecture Programmer's Manual. 2023 (p. 10).

[6] AMD. AMD64 Architecture Programmer's Manual. 2024 (p. 21).

[7] Samy Amer, Yingchen Wang, Hunter Kippen, Thinh Dang, Daniel Genkin, Andrew Kwong, Alexander Nelson, and Arkady Yerukhimovich. PQ-Hammer: End-to-end Key Recovery Attacks on Post-Quantum Cryptography Using Rowhammer. In: S&P. 2025 (p. 39).

[8] Seiichi Aritome. NAND Flash Memory Technologies. John Wiley & Sons, 2015 (pp. 27, 28).

[9] ARM. ARM A64 Instruction Set Architecture. Sept. 2018 (p. 20).

[10] 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: ASPLOS (2016) (pp. 3, 32, 40).

[11] Anys Bacha and Radu Teodorescu. Dynamic Reduction of Voltage Margins by Leveraging On-chip ECC in Itanium II Processors. In: ISCA. 2013 (pp. 4, 7, 46).

[12] Anys Bacha and Radu Teodorescu. Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors. In: MICRO. 2014 (pp. 4, 7, 46).

[13] Amir Ban. Flash file system. US Patent 5,404,485. 1995 (p. 28).

[14] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. In: Proceedings of the IEEE (2006) (p. 3).

[15] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low Voltage Fault Attacks on the RSA Cryptosystem. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2009 (pp. 3, 7, 45).

[16] 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. 25).

[17] 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. 46).

[18] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A Complete In-DRAM Rowhammer Mitigation. DRAMSec. 2021 (pp. 32, 40–43).

[19] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 4, 47).

[20] Johann Betz, Dirk Westhoff, and Günter Müller. Survey on covert channels in virtual machines and cloud computing. In: Transactions on Emerging Telecommunications Technologies (2016) (pp. 10, 49).

[21] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: CHES. 2016 (pp. 32, 37, 39).

[22] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In: CCS. 2019 (pp. 5, 47, 48).

[23] Sebastian Biedermann, Stefan Katzenbeisser, and Jakub Szefer. Hard Drive Side-Channel Attacks using Smartphone Magnetic Field Sensors. In: FC. 2015 (pp. 4, 5, 9, 48, 49).

[24] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The Convoy Phenomenon. In: ACM SIGOPS Operating Systems Review (1979) (p. 48).

[25] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In: EUROCRYPT. 1997 (p. 3).

[26] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P. 2016 (pp. 32, 37, 38).

[27] F Nisa Bostanci, ISmail Emir Yüksel, Ataberk Olgun, Konstantinos Kanellopoulos, Yahya Can Tuğrul, A Giray Yağliçi, Mohammad Sadrosadati, and Onur Mutlu. CoMeT: Count-Min-Sketch-Based Row Tracking to Mitigate RowHammer at Low Cost. In: HPCA. 2024 (pp. 32, 40–42).

[28] F. Nisa Bostanci, Konstantinos Kanellopoulos, Ataberk Olgun, A. Giray Yaglikci, Ismail Emir Yuksel, Nika Mansouri Ghiasi, Zulal Bingol, Mohammad Sadrosadati, and Onur Mutlu. Revisiting Main Memory-Based Covert and Side Channel Attacks in the Context of Processing-in-Memory. In: arXiv:2404.11284 (2025) (p. 51).

[29] F Bostancı, Oğuzhan Canpolat, Ataberk Olgun, İsmail Emir Yüksel, Konstantinos Kanellopoulos, Mohammad Sadrosadati, A Giray Yağlıkçı, and Onur Mutlu. Understanding and Mitigating Side and Covert Channel Vulnerabilities Introduced by RowHammer Defenses. In: arXiv:2503.17891 (2025) (p. 48).

[30] 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 (pp. 3, 32, 40).

[31] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In: CHES. 2004 (p. 4).

[32] Oğuzhan Canpolat, A Giray Yağlıkçı, Ataberk Olgun, Ismail Emir Yuksel, Yahya Can Tuğrul, Konstantinos Kanellopoulos, Oğuz Ergin, and Onur Mutlu. BreakHammer: Enhancing RowHammer Mitigations by Carefully Throttling Suspect Threads. In: MICRO. 2024 (pp. 32, 40, 41).

[33] Oğuzhan Canpolat, A Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Oğuz Ergin, and Onur Mutlu. Understanding the Security Benefits and Overheads of Emerging Industry Solutions to DRAM Read Disturbance. In: DRAMSec (2024) (pp. 32, 40, 42, 43).

[34] Ang Chen, W Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting Covert Timing Channels with Time-Deterministic Replay. In: OSDI. 2014 (pp. 9, 49).

[35] Wei Chen, Zhi Zhang, Xin Zhang, Qingni Shen, Yuval Yarom, Daniel Genkin, Chen Yan, and Zhe Wang. HyperHammer: Breaking Free from KVM-Enforced Isolation. In: ASPLOS. 2025 (pp. 37, 38, 44).

[36] 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. 4, 7).

[37] Yueqiang Cheng, Zhi Zhang, and Surya Nepal. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. In: arXiv:1802.07060 (2018) (pp. 22, 32, 37, 38).

[38] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. In: Cryptology ePrint Archive, Report 2015/1034 (2015) (pp. 32, 40).

[39] 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. 3, 4, 11, 22, 32, 37, 38, 40, 44).

[40] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's Always a Bigger Fish: A Clarifying Analysis of a Machine-Learning-Assisted Side-Channel Attack. In: ISCA. 2022 (p. 47).

[41] Jonathan Corbet. Defending against Rowhammer in the kernel. Oct. 2016. URL: https://lwn.net/Articles/704920/ (pp. 3, 32, 40).

[42] Nikunj A Dadhania. [PATCH v7 00/16] Add Secure TSC support for SNP guests. 2023. URL: https://lore.kernel.org/all/20231220151358.2147066-1-nikunj@amd.com/ (p. 10).

[43] Timothy J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. Tech. rep. IBM Microelectronics, 1997 (pp. 44, 45).

[44] Department of Defense. Trusted Computer System Evaluation Criteria (TCSEC). 1983 (p. 48).

[45] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In: NDSS. 2023 (pp. 3, 32, 40).

[46] Cambridge Dictionary, ed. “leaky”. Cambridge University Press, 2024 (p. 3).

[47] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems. In: ISCA. 2021 (pp. 5, 47).

[48] Sina Sayyah Ensan, Karthikeyan Nagarajan, Mohammad Nasim Imtiaz Khan, and Swaroop Ghosh. SCARE: Side Channel Attack on In-Memory Computing for Reverse Engineering. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2021) (p. 47).

[49] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In: MICRO. 2003 (p. 46).

[50] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS. 2016 (pp. 5, 47).

[51] Ali Fakhrzadehgan, Yale N Patt, Prashant J Nair, and Moinuddin K Qureshi. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In: HPCA. 2022 (pp. 32, 40, 44).

[52] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (pp. 3, 22, 32, 37–39, 44).

[53] 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. 3, 4, 11, 22, 25, 32, 34, 37, 38, 41, 42, 44).

[54] Ferraz Gabriel. SSD Database. `https://www.techpowerup.com/ssd-specs/`. 2025 (p. 30).

[55] Stefan Gast, Roland Czerny, Jonas Juffinger, Fabian Rauscher, Simone Franza, and Daniel Gruss. SnailLoad: Exploiting Remote Network Latency Measurements without JavaScript. In: USENIX Security. 2024 (pp. 5, 12, 48).

[56] Stefan Gast, Sebastian Daniel Felix, Alexander Steinmaurer, **Jonas Juffinger**, and Daniel Gruss. Real-World Study of the Security of Educational Test Systems. In: Workshop on Operating Systems and Virtualization Security. 2025 (pp. 5, 12, 13).

[57] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks. In: FC. 2024 (pp. 5, 12, 47, 48).

[58] 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. 5, 10, 12, 47, 48).

[59] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In: NDSS. 2025 (p. 47).

[60] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. Efficient and generic microarchitectural hash-function recovery. In: S&P. 2024 (p. 25).

[61] Lukas Gerlach, Fabian Thomas, Robert Pietsch, and Michael Schwarz. A Rowhammer Reproduction Study Using the Blacksmith Fuzzer. In: ESORICS. 2023 (p. 35).

[62] Jim Gettys. Bufferbloat: Dark buffers in the internet. In: IEEE Internet Computing 15.3 (2011), pp. 96–96 (p. 48).

[63] 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` (pp. 32, 40).

[64] Swaroop Ghosh and Kaushik Roy. Parameter Variation Tolerance and Error Resiliency: New Design Paradigm for the Nanoscale Era. In: Proceedings of the IEEE (2010) (p. 7).

[65] Ilias Giechaskiel, Shanquan Tian, and Jakub Szefer. Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. In: ACM Transactions on Reconfigurable Technology and Systems (2022) (pp. 47, 48).

[66] 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. 5, 47).

[67] Lukas Giner, Roland Czerny, Simon Lammer, Aaron Giner, Paul Gollob, Jonas Juffiger, and Daniel Gruss. Fast and Efficient Secure L1 Caches for SMT. In: ARES. 2025 (pp. 5, 13).

[68] Dimitris Gizopoulos, George Papadimitriou, Athanasios Chatzidimitriou, Vijay Janapa Reddi, Behzad Salami, Osman S Unsal, Adrian Cristal Kestelman, and Jingwen Leng. Modern hardware margins: CPUs, GPUs, FPGAs recent system-level studies. In: International Symposium on On-Line Testing and Robust System Design (IOLTS). 2019 (pp. 4, 7, 17, 18, 45).

[69] Mel Gorman. Understanding the Linux Virtual Memory Manager. Prentice Hall Upper Saddle River, 2004 (p. 37).

[70] Corey Gough, Ian Steiner, Winston Saunders, Corey Gough, Ian Steiner, and Winston Saunders. CPU Power Management. In: Energy Efficient Servers: Blueprints for Data Center Optimization (2015) (pp. 7, 45).

[71] Sudhakar Govindavajhala and Andrew W Appel. Using Memory Errors to Attack a Virtual Machine. In: S&P. 2003 (p. 3).

[72] Zvika Greenfield and Tomer Levy. Throttling support for row-hammer counters. US Patent 9251885. 2014 (pp. 32, 40, 42).

[73] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 47, 49).

[74] 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. 3, 4, 7, 8, 32, 34, 39, 40, 51).

[75] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 22, 32, 37, 38, 44).

[76] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 5, 32, 40, 47).

[77] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 5, 47).

[78] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to violate web privacy with hardware performance events. In: ESORICS. 2017 (p. 47).

[79] Mordechai Guri. Exfiltrating data from air-gapped computers via ViBrAtIoNs. In: Future Generation Computer Systems (2021) (p. 47).

[80] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Acoustic Data Exfiltration from Speakerless Air-Gapped Computers via Covert Hard-Drive Noise ('DiskFiltration'). In: ESORICS. 2017 (pp. 4, 5, 9, 49).

[81] Jawad Haj-Yahya, Lois Orosa, Jeremie S Kim, Juan Gómez Luna, A Giray Yağlıkçı, Mohammed Alser, Ivan Puddu, and Onur Mutlu. IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors. In: ISCA. 2021 (p. 47).

[82] Sarah Harris and David Harris. Digital Design and Computer Architecture. Morgan Kaufmann, 2015 (pp. 4, 16–18, 21).

[83] 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 (pp. 3, 32, 34, 40, 42).

[84] Martin Heckel and Florian Adamsky. Flipper: Rowhammer on Steroids. In: uASC. 2025 (p. 39).

[85] Martin Heckel and Florian Adamsky. Reverse-Engineering Bank Addressing Functions on AMD CPUs. In: DRAMSec Workshop. 2023 (p. 25).

[86] 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. 25).

[87] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In: Black Hat USA. 2015 (pp. 32, 40).

[88] 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:2302.03591 (2023) (pp. 23, 24, 32, 33, 35, 36, 40–42).

[89] Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-Scope: Cache Probing at Cache Speed. In: CCS. 2024 (pp. 5, 47).

[90] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In: Journal of Computer Security (1992) (pp. 47, 48).

[91] Ruirui Huang and G. Edward Suh. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. In: ISCA. 2010 (p. 44).

[92] Vincent Huard, Souhir Mhira, A Barclais, X Lecocq, F Raugi, M Cantournet, and Alain Bravaix. Managing electrical reliability in consumer systems for improved energy efficiency. In: IEEE International Reliability Physics Symposium (IRPS). 2018 (pp. 18, 19).

[93] Jisung Im, Hansol Kim, Jinsu Kim, Seungmin Woo, Taeseong Kwon, Young Jun Yoon, Jong-Ho Bae, and Sung Yun Woo. Analysis of Row Hammer and Passing Gate Effect in DRAM Cells by BCAT Structural Design. In: IEEE Silicon Nanoelectronics Workshop (SNW). 2024 (p. 35).

[94] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-location detection on the cloud. In: COSADE. 2016 (pp. 10, 49).

[95] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. In: Cryptology ePrint Archive, Report 2015/898 (2015) (p. 49).

[96] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2023 (pp. 21, 25).

[97] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (pp. 20, 21).

[98] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (pp. 20, 37).

[99] Intel. Understanding the Flash Translation Layer (FTL) Specification. 1998 (p. 28).

[100] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. In: Cryptology ePrint Archive, Report 2016/1196 (2017) (pp. 32, 40).

[101] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Cross-VM Side Channels and Their Use to Extract Private Keys. In: Big Data and Cloud Computing. 2014 (p. 10).

[102] Aamer Jaleel, Gururaj Saileshwar, Stephen W. Keckler, and Moinuddin Qureshi. PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers. In: ISCA. 2024 (pp. 32, 40–42).

[103] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (pp. 32, 37, 39).

[104] Patrick Jattke, Michele Marazzi, Flavien Solt, Max Wipfli, and Stefan Gloor Kaveh Razavi. MCSEE: Evaluating Advanced Rowhammer Attacks and Defenses via Automated DRAM Traffic Analysis. In: Usenix Security. 2025 (pp. 36, 43, 45).

[105] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. BLACKSMITH: Rowhammering in the Frequency Domain. In: S&P. 2021 (pp. 3, 4, 11, 32, 34, 35, 41, 42, 51).

[106] 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. 25, 39, 51).

[107] JEDEC Solid State Technology Association. DDR4 SDRAM Standard. 2021. URL: https://www.jedec.org/standards-documents/docs/jesd79-4a (pp. 24, 25).

[108] JEDEC Solid State Technology Association. DDR5 SDRAM Standard. 2024. URL: https://www.jedec.org/standards-documents/docs/jesd79-5c01 (pp. 25, 26, 42, 43).

[109] JEDEC Solid State Technology Association. Low Power Double Data Rate 4. 2017. URL: http://www.jedec.org/standards-documents/docs/jesd209-4b (pp. 32, 39–42).

[110] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. In: IEEE Journal of Solid-State Circuits (2016) (pp. 41, 43).

[111] Qisheng Jiang and Chundong Wang. Sync+Sync: A Covert Channel Built on fsync with Storage. In: USENIX Security. 2024 (pp. 5, 47, 49).

[112] Yichen Jiang, Huifeng Zhu, Haoqi Shan, Xiaolong Guo, Xuan Zhang, and Yier Jin. TRRScope: Understanding Target Row Refresh Mechanism for Modern DDR Protection. In: HOST. 2021 (p. 42).

[113] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In: HPCA. 2016 (pp. 5, 47).

[114] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Machine Learning-Based Rowhammer Mitigation. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2022) (pp. 32, 40, 41, 43).

[115] Jonas Juffinger. An Analysis of HMB-based SSD Rowhammer. In: uASC. 2025 (pp. 5, 6, 8, 29, 32, 39, 51).

[116] Jonas Juffinger, Stepan Kalinin, Daniel Gruss, and Frank Mueller. SUIT: Secure Undervolting with Instruction Traps. In: ASPLOS. 2024 (pp. 5–7, 45, 46, 52).

[117] 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. 5–7, 32, 40, 44, 51, 52).

[118] Jonas Juffinger, Sudheendra Raghav Neela, and Daniel Gruss. Not So Secure TSC. In: ACNS. 2025 (pp. 5, 6, 9, 10, 50, 53).

[119] 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 (pp. 5–8, 22, 32, 34, 36–38, 44, 51).

[120] Jonas Juffinger, Fabian Rauscher, Giuseppe La Manna, and Daniel Gruss. Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS. 2025 (pp. 5, 6, 9, 48, 49, 52).

[121] Jonas Juffinger, Hannes Weissteiner, Thomas Steinbauer, and Daniel Gruss. The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer. In: DIMVA. 2025 (pp. 5, 6, 8, 9, 29, 30, 52).

[122] 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. 25).

[123] Marcin Kaczmarski. Thoughts on Intel Xeon E5-2600 v2 Product Family Performance Optimisation – component selection guidelines. Infobazy 2014. Aug. 2014. URL: https://web.archive.org/web/20240418012923/http://infobazy.gda.pl/2014/pliki/prezentacje/d2s2e4-Kaczmarski-Optymalna.pdf (pp. 32, 40, 41).

[124] Manolis Kaliorakis, Athanasios Chatzidimitriou, George Papadimitriou, and Dimitris Gizopoulos. Statistical analysis of multicore CPUs operation in scaled voltage conditions. In: IEEE Computer Architecture Letters (2018) (pp. 4, 7, 45).

[125] 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. 3, 11, 22, 32, 37, 38, 44, 51).

[126] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In: International Conference on Embedded Software. 2006 (p. 28).

[127] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. 2016 (p. 10).

[128] Paul A Karger and John C Wray. Storage Channels in Disk Arm Optimization. In: S&P. 1991 (pp. 5, 47–49).

[129] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-Memory Based File System. In: USENIX. 1995 (p. 28).

[130] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In: USENIX Security. 2020 (pp. 4, 7, 45, 46).

[131] 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) (pp. 32, 40, 42).

[132] 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 (p. 39).

[133] Kyusik Kim and Taeseok Kim. HMB in DRAM-less NVMe SSDs: Their usage and effects on performance. In: PloS one (2020) (p. 29).

[134] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. In: HPCA. 2022 (pp. 41, 42).

[135] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security. 2012 (p. 49).

[136] Woongrae Kim, Chulmoon Jung, Seongnyuh Yoo, Duckhwa Hong, Jeongjin Hwang, Jungmin Yoon, Ohyong Jung, Joonwoo Choi, Sanga Hyun, Mankeun Kang, et al. A 1.1V 16Gb DDR5 DRAM with Probabilistic-Aggressor Tracking, Refresh-Management Functionality, Per-Row Hammer Tracking, a Multi-Step Precharge, and Core-Bias Modulation for Security and Reliability Enhancement. In: International Solid-State Circuits Conference (ISSCC). IEEE. 2023 (p. 41).

[137] 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. 3, 8, 24, 32–34, 39–41).

[138] 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. 37).

[139] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (pp. 4, 5, 47).

[140] 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. 47).

[141] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (p. 4).

[142] Andreas Kogler, Daniel Gruss, and Michael Schwarz. Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In: USENIX Security. 2022 (pp. 45, 46).

[143] 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. 5, 11, 47).

[144] 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. 3–5, 11, 22, 32, 34, 35, 37, 38, 42, 44, 45, 51).

[145] 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 (pp. 3, 32, 40).

[146] Panos Koutsovasilis, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. The Impact of CPU Voltage Margins on Power-Constrained Execution. In: IEEE Transactions on Sustainable Computing (2020) (pp. 4, 7, 45).

[147] Panos Koutsovasilis, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, and Spyros Lalis. Dynamic undervolting to improve energy efficiency on multicore x86 cpus. In: IEEE Transactions on Parallel and Distributed Systems (2020) (pp. 4, 7, 46).

[148] A Anand Kumar. Fundamentals of Digital Circuits. 2016 (pp. 7, 16).

[149] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (pp. 32, 37, 39).

[150] Butler W Lampson. A note on the confinement problem. In: Communications of the ACM (1973) (pp. 6, 47, 48, 53).

[151] 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. 32, 40, 41, 43).

[152] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. In: IEEE Computer Architecture Letters (2020) (p. 49).

[153] Lanny L Lewyn, Trond Ytterdal, Carsten Wulff, and Kenneth Martin. Analog Circuit Design in Nanoscale CMOS Technologies. In: Proceedings of the IEEE (2009) (p. 7).

[154] Linux. madvise(2) — Linux manual page. https://man7.org/linux/man-pages/man2/madvise.2.html. 2025 (p. 37).

[155] Bartosz Lipinski, Wojciech Mazurczyk, and Krzysztof Szczypiorski. Improving Hard Disk Contention-based Covert Channel in Cloud Computing Environment. In: S&P Workshops. 2014 (pp. 5, 9, 48, 49).

[156] 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. 3, 38).

[157] 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, 5, 11, 47).

[158] 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 (p. 47).

[159] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency Throttling Side-Channel Attack. In: CCS. 2022 (pp. 5, 47).

[160] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In: HPCA. 2016 (p. 49).

[161] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. Side-Channel Attacks on Optane Persistent Memory. In: USENIX Security. 2023 (pp. 6, 49).

[162] Keith Loepere. Resolving covert channels within a B2 class secure system. In: ACM SIGOPS Operating Systems Review (1985) (pp. 47, 48).

[163] 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 (pp. 4, 8, 24, 32, 35, 36).

[164] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel. In: USENIX Security. 2024 (pp. 5, 47).

[165] Lukas Maar, Jonas Juffinger, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025 (pp. 5, 12, 47).

[166] Hosein Mohammadi Makrani, Hossein Sayadi, Najmeh Nazari, Khaled N Khasawneh, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Cloak & Co-locate: Adversarial Railroading of Resource Sharing-based Attacks on the Cloud. In: Secure and Private Execution Environment Design (SEED). 2021 (pp. 49, 50).

[167] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. An empirical survey of performance and energy efficiency variation on Intel processors. In: International Workshop on Energy Efficient Supercomputing. 2017 (p. 7).

[168] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. PROTRR: Principled yet Optimal In-DRAM Target Row Refresh. In: S&P. 2022 (pp. 3, 41, 42).

[169] Michele Marazzi and Kaveh Razavi. RISC-H: Rowhammer Attacks on RISC-V. In: DRAMSec Workshop. 2024 (pp. 3, 39).

[170] Michele Marazzi, Tristan Sachsenweger, Flavien Solt, Peng Zeng, Kubo Takashi, Maksym Yarema, and Kaveh Razavi. HiFi-DRAM: Enabling High-fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging. In: ISCA. 2024 (pp. 23, 26).

[171] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In: S&P. 2023 (pp. 3, 32, 40–43).

[172] Emmanouil Maroudas, Spyros Lalis, Nikolaos Bellas, and Christos D Antonopoulos. Exploring the Potential of Context-Aware Dynamic CPU Undervolting. In: ACM International Conference on Computing Frontiers. 2021 (pp. 4, 7, 46).

[173] Marsan, Balbo, Conte, and Gregoretti. Modeling Bus Contention and Memory Interference in a Multiprocessor System. In: IEEE Transactions on Computers (1983) (p. 48).

[174] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. MagneticSpy: Exploiting Magnetometer in Mobile Devices for Website and Application Fingerprinting. In: ACM Workshop on Privacy in the Electronic Society. 2019 (pp. 5, 47).

[175] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 47).

[176] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 5, 10, 47, 49).

[177] Mozilla. HTTP caching — MDN. Dec. 2024. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching` (p. 9).

[178] David Mulnix. Intel Xeon Processor Scalable Family Technical Overview. `https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html`. 2022 (p. 21).

[179] 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. 3, 4, 7, 17, 45, 46).

[180] Koksal Mus, Saad Islam, and Berk Sunar. QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme. In: CCS. 2020 (p. 39).

[181] Onur Mutlu, Ataberk Olgun, and A Giray Yağlıkcı. Fundamentally Understanding and Solving RowHammer. In: Asia and South Pacific Design Automation Conference. 2023 (p. 39).

[182] Onur Mutlu, Ataberk Olgun, and İsmail Emir Yüksel. Memory-Centric Computing: Solving Computing's Memory Problem. In: International Memory Workshop (IMW). 2025 (p. 51).

[183] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In: CCS. 2018 (pp. 5, 47).

[184] Northland Locksmith. Safe lock manipulation. `https://web.archive.org/web/20161222161226/https://northlandlocksmith.com/2016/12/09/safe-lock-manipulation/`. 2016 (p. 4).

[185] NVM Express, Inc. NVM Express, rev 1.2.1. 2016 (pp. 8, 29).

[186] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In: NDSS. 2025 (pp. 5, 47).

[187] Vojin G Oklobdzija, Vladimir M Stojanovic, Dejan M Markovic, and Nikola M Nedovic. Digital System Clocking: High-Performance and Low-Power Aspects. Wiley-IEEE Press, 2005. ISBN: 9780471274476 (p. 18).

[188] Ataberk Olgun, F. Nisa Bostanci, Ismail Emir Yuksel, Oguzhan Canpolat, Haocong Luo, Geraldo F. Oliveira, A. Giray Yaglikci, Minesh Patel, and Onur Mutlu. Variable Read Disturbance: An Experimental Analysis of Temporal Variation in DRAM Read Disturbance. In: HPCA. 2025 (pp. 4, 32).

[189] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, Abdullah Giray Yaglikci, Geraldo F Oliveira, and Onur Mutlu. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation. In: USENIX Security. 2024 (pp. 32, 41, 42).

[190] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In: MICRO. 2021 (p. 3).

[191] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 4, 5, 47).

[192] George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. Adaptive Voltage/Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs. In: HPCA. 2019 (p. 7).

[193] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs. In: MICRO. 2017 (pp. 4, 7, 45).

[194] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Charalampos Magdalinos, and Dimitris Gizopoulos. Voltage margins identification on commercial x86-64 multicore microprocessors. In: IEEE Symposium on On-Line Testing (IOLTS). 2017 (pp. 4, 7, 45).

[195] Kyungbae Park, Chulseung Lim, Donghyuk Yun, and Sanghyeon Baeg. Experiments and root cause analysis for active-precharge hammering fault in DDR3 SDRAM under 3× nm technology. In: Microelectronics Reliability (2016) (p. 33).

[196] 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. 32, 41, 43, 44).

[197] Michael K Patterson. The Effect of Data Center Temperature on Energy Efficiency. In: Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems. 2008 (pp. 7, 45).

[198] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In: ESSoS. 2016 (pp. 32, 40).

[199] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (pp. 4, 47).

[200] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security. 2016 (pp. 25, 26, 37, 47).

[201] Florian Pfarr, Thomas Buckel, and Axel Winkelmann. Cloud Computing Data Protection – A Literature Review and Analysis. In: HICSS. 2014 (p. 10).

[202] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In: AsiaCCS. 2023 (pp. 5, 47).

[203] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies. In: USENIX Security. 2022 (p. 47).

[204] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (pp. 5, 47).

[205] Salman Qazi and Daniel Moghimi. SoothSayer: Bypassing DSAC Mitigation by Predicting Counter Replacement. In: DRAMSec. 2024 (p. 42).

[206] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In: HOST. 2016 (pp. 32, 39).

[207] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In: TrustCom/BigDataSE. 2018 (p. 47).

[208] 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. 3, 4, 7, 45).

[209] Moinuddin Qureshi and Salman Qazi. MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters. In: ASPLOS. 2025 (pp. 41, 43).

[210] Moinuddin Qureshi, Salman Qazi, and Aamer Jaleel. MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker. In: MICRO. 2024 (pp. 32, 41, 42).

[211] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025 (pp. 5, 47).

[212] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (pp. 5, 11, 47).

[213] 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 (pp. 32, 37, 39).

[214] Randall Rettberg and Robert Thomas. Contention is no obstacle to shared-memory multiprocessing. In: Communications of the ACM (1986) (p. 48).

[215] 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. 3, 32, 37, 38).

[216] Finn de Ridder, Patrick Jattke, and Kaveh Razavi. Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands. In: USENIX Security. 2025 (p. 3).

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

[218] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In: AsiaCCS. 2022 (pp. 47, 48).

[219] Seong-Wan Ryu, Kyungkyu Min, Jungho Shin, Heimi Kwon, Donghoon Nam, Taekyung Oh, Tae-Su Jang, Minsoo Yoo, Yongtaik Kim, and Sungjoo Hong. Overcoming the Reliability Limitation in the Ultimately Scaled DRAM Using Silicon Migration Technique by Hydrogen Annealing. In: International Electron Devices Meeting (IEDM). 2017 (p. 33).

[220] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In: HPCA. 2018 (pp. 32, 44).

[221] 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. 32, 41, 44).

[222] Stefan Saroiu. DDR5 Spec Update Has All It Needs to End Rowhammer: Will It? Apr. 2024. URL: https://web.archive.org/web/20240628014749/https://stefan.t8k2.com/rh/PRAC/index.html (p. 42).

[223] Anish Saxena, Aamer Jaleel, and Moinuddin Qureshi. ImPress: Securing DRAM Against Data-Disturbance Errors via Implicit Row-Press Mitigation. In: MICRO. 2024 (pp. 32, 41, 42).

[224] 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. 5, 11, 32, 41, 44).

[225] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime. In: MICRO. 2022 (pp. 32, 41, 44).

[226] Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program Confinement in KVM/370. In: ACM National Conference. 1977 (pp. 47, 48).

[227] Christian Schlünder, Stefano Aresu, Georg Georgakos, Werner Kanert, Hans Reisinger, Karl Hofmann, and Wolfgang Gustin. HCI vs. BTI?-Neither one's out. In: IEEE International Reliability Physics Symposium (IRPS). 2012 (p. 19).

[228] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (p. 47).

[229] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat USA. 2015 (pp. 3, 11, 32, 34, 38).

[230] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Mitigating Wordline Crosstalk using Adaptive Trees of Counters. In: ISCA. 2018 (pp. 32, 41, 43).

[231] Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. A Side-channel Analysis of Sensor Multiplexing for Covert Channels and Application Fingerprinting on Mobile Devices. In: arXiv:2110.06363 (2021) (pp. 5, 47).

[232] Anand Shimpi. Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested. June 2013. URL: https://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested/3 (p. 21).

[233] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through The Cache Occupancy Channel. In: USENIX Security. 2019 (p. 5).

[234] Mert Side, Fan Yao, and Zhenkai Zhang. LockedDown: Exploiting Contention on Host-GPU PCIe Bus for Fun and Profit. In: Euro S&P. 2022 (pp. 47, 48).

[235] SN Singh. Basic Electrical Engineering. PHI Learning Pvt. Ltd., 2010 (p. 18).

[236] Sergei P Skorobogatov and Ross J Anderson. Optical Fault Induction Attacks. In: International Workshop on Cryptographic Hardware and Embedded Systems. 2002 (p. 3).

[237] Alan Jay Smith. Design of CPU Cache Memories. Computer Science Division, University of California, 1987 (p. 20).

[238] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM Stronger Against Row Hammering. In: Design Automation Conference (DAC). 2017 (pp. 32, 41).

[239] Mary-Jane Sule, Maozhen Li, and Gareth Taylor. Trust Modeling in Cloud Computing. In: Symposium on Service-Oriented System Engineering (SOSE). 2016 (p. 10).

[240] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multitenant Detection in IaaS Clouds. In: NDSS. 2018 (pp. 10, 49).

[241] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible Probe: Timing Attacks with PCIe Congestion Side-Channel. In: S&P. 2021 (pp. 5, 47, 48).

[242] 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 (pp. 5, 47).

[243] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security. 2017 (pp. 3, 4, 7, 45).

[244] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In: RAID. 2018 (pp. 3, 25).

[245] 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 (pp. 3, 38).

[246] Bjørn Ivar Teigen, Kai Olav Ellefsen, Tor Skeie, and Jim Torresen. Known Performance Issues Are Prevalent in Consumer WiFi Routers. In: International Conference on Network and Service Management (CNSM). 2021 (p. 48).

[247] The Linux Kernel. Transparent Hugepage Support. 2025. URL: `https://docs.kernel.org/admin-guide/mm/transhuge.html` (p. 37).

[248] Shanquan Tian, Ilias Giechaskiel, Wenjie Xiong, and Jakub Szefer. Cloud FPGA Cartography using PCIe Contention. In: International Symposium on Field-Programmable Custom Computing Machines (FCCM). 2021 (pp. 47, 48).

[249] 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 (pp. 32, 37, 39).

[250] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. Covert-channels in FPGA-enabled SmartSSDs. In: ACM Transactions on Reconfigurable Technology and Systems (2023) (pp. 6, 47, 49).

[251] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. Security Evaluation of Thermal Covert-channels on SmartSSDs. In: arXiv:2305.09115 (2023) (pp. 6, 47, 49).

[252] C-R Tsai, Virgil D. Gligor, and C. Sekar Chandersekaran. On the Identification of Covert Storage Channels in Secure Systems. In: IEEE Transactions on Software Engineering (1990) (pp. 47, 48).

[253] Yahya Can Tuğrul, A. Giray Yağlıkçı, İsmail Emir Yüksel, Ataberk Olgun, Oğuzhan Canpolat, Nisa Bostancı, Mohammad Sadrosadati, Oğuz Ergin, and Onur Mutlu. Understanding RowHammer Under Reduced Refresh Latency: Experimental Analysis of Real DRAM Chips and Implications on Future Solutions. In: 2025 (pp. 32, 41, 42, 44).

[254] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: ISCA. 1995 (p. 48).

[255] John P Uyemura. CMOS Logic Circuit Design. Springer Science & Business Media, 1999 (pp. 16–18).

[256] 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. 4, 11, 22, 32, 37–40, 44).

[257] 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 (pp. 3, 32, 40).

[258] 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 (HASP). 2018 (pp. 32, 41, 43).

[259] John Von Neumann. First Draft of a Report on the EDVAC. In: IEEE Annals of the History of Computing (1993) (p. 20).

[260] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On DRAM Rowhammer and the Physics of Insecurity. In: IEEE Transactions on Electron Devices (2021) (p. 33).

[261] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping. In: Design Automation Conference (DAC). 2020 (p. 25).

[262] Yingchen Wang, Riccardo Paccagnella, Zhao Gang, Willy R Vasquez, David Kohlbrenner, Hovav Shacham, and Christopher W Fletcher. GPU. zip: On the Side-Channel Implications of Hardware-Based Graphical Data Compression. In: S&P (2024) (pp. 5, 47).

[263] 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. 5, 7, 11, 47).

[264] 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 (pp. 5, 11, 47).

[265] Zhenghong Wang and Ruby B Lee. Covert and Side Channels due to Processor Architecture. In: ACSAC. 2006 (p. 47).

[266] Ziyu Wang, Fan-hsuan Meng, Yongmo Park, Jason K Eshraghian, and Wei D Lu. Side-Channel Attack Analysis on In-Memory Computing Architectures. In: IEEE Transactions on Emerging Topics in Computing (2023) (p. 47).

[267] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Vol. I: Unprivileged ISA, Version 20191213. 2019 (p. 20).

[268] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In: DSN. 2020 (pp. 5, 47).

[269] 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. 39).

[270] Hannes Weissteiner, Fabian Rauscher, Robin Leander Schröder, Jonas Juffinger, Stefan Gast, Jan Wichelmann, Thomas Eisenbarth, and Daniel Gruss. TEEcorrelate: An Information-Preserving Defense against Performance Counter Attacks on TEEs. In: USENIX Security. 2025 (pp. 5, 13).

[271] Jeonghyun Woo, Shaopeng Chris Lin, Prashant J Nair, Aamer Jaleel, and Gururaj Saileshwar. QPRAC: Towards Secure and Practical PRAC-based Rowhammer Mitigation using Priority Queues. In: HPCA. IEEE. 2025 (pp. 41, 43).

[272] 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 (pp. 41, 44).

[273] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: ACM Transactions on Networking (2014) (pp. 47, 48).

[274] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In: USENIX Security. 2012 (pp. 10, 47–49).

[275] 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 (pp. 22, 32, 37, 38, 44).

[276] A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In: HPCA. 2021 (pp. 32, 41–43).

[277] A. Giray Yaglikci, Yahya Can Tuğrul, Geraldo F De Oliviera, Ismail Emir Yüksel, Ataberk Olgun, Haocong Luo, and Onur Mutlu. Spatial Variation-Aware Read Disturbance Defenses: Experimental Analysis of Real DRAM Chips and Implications on Future Solutions. In: HPCA. 2024 (pp. 4, 32).

[278] Thomas Yang and Xi-Wei Lin. Trap-Assisted DRAM Row Hammer Effect. In: IEEE Electron Device Letters (2019) (p. 33).

[279] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 5, 47).

[280] Chi-Weon Yoon. The Fundamentals of NAND Flash Memory: Technology for tomorrow's fourth industrial revolution. In: IEEE Solid-State Circuits Magazine (2022) (pp. 27, 28).

[281] Jung Min You and Joon-Sung Yang. MRLoc: Mitigating Rowhammering based on memory Locality. In: Design Automation Conference (DAC). 2019 (pp. 32, 41).

[282] Ismail Emir Yuksel, Akash Sood, Ataberk Olgun, Oguzhan Canpolat, Haocong Luo, Nisa Bostanci, Mohammad Sadrosadati, A. Giray Yaglikci, and Onur Mutlu. PuDHammer: Experimental Analysis of Read Disturbance Effects of Processing-using-DRAM in Real DRAM Chips. In: ISCA. 2025 (p. 51).

[283] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In: USENIX Security. 2023 (pp. 5, 47).

[284] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: RAID. 2016 (pp. 32, 40, 49).

[285] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P. 2011 (pp. 10, 49).

[286] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (pp. 10, 49).

[287] 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. 38).

[288] 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 (pp. 4, 11, 22, 32, 37, 38, 44).

[289] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS. In: ASPLOS. 2024 (pp. 10, 49, 50).

[290] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud. In: ASPLOS. 2024 (pp. 5, 10, 47, 50).

Part II.

Publications

List of Publications

During my thesis, I contributed to 17 publications in conference proceedings, 7 of which are included in this thesis as shown below.

Publications in This Thesis

1. **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
2. **Jonas Juffinger**, Stepan Kalinin, Daniel Gruss, and Frank Mueller. SUIT: Secure Undervolting with Instruction Traps. In: ASPLOS. 2024
3. **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
4. **Jonas Juffinger**. An Analysis of HMB-based SSD Rowhammer. In: uASC. 2025
5. **Jonas Juffinger**, Hannes Weissteiner, Thomas Steinbauer, and Daniel Gruss. The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer. In: DIMVA. 2025
6. **Jonas Juffinger**, Fabian Rauscher, Giuseppe La Manna, and Daniel Gruss. Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS. 2025
7. **Jonas Juffinger**, Sudheendra Raghav Neela, and Daniel Gruss. Not So Secure TSC. In: ACNS. 2025

Other Contributions

1. Hannes Weissteiner, Fabian Rauscher, Robin Leander Schröder, **Jonas Juffinger**, Stefan Gast, Jan Wichelmann, Thomas Eisenbarth, and Daniel Gruss. TEEcorrelate: An Information-Preserving Defense against Performance Counter Attacks on TEEs. In: USENIX Security. 2025

2. Lukas Giner, Roland Czerny, Simon Lammer, Aaron Giner, Paul Gollob, **Jonas Juffiger**, and Daniel Gruss. Fast and Efficient Secure L1 Caches for SMT. In: ARES. 2025

3. Stefan Gast, Sebastian Daniel Felix, Alexander Steimaurer, **Jonas Juffinger**, and Daniel Gruss. Real-World Study of the Security of Educational Test Systems. In: Workshop on Operating Systems and Virtualization Security. 2025

4. Lukas Maar, **Jonas Juffinger**, Thomas Steinbauer, Daniel Gruss, and Stefan Mangard. KernelSnitch: Side-Channel Attacks on Kernel Data Structures. In: NDSS. 2025

5. Stefan Gast, Roland Czerny, **Jonas Juffinger**, Fabian Rauscher, Simone Franza, and Daniel Gruss. SnailLoad: Exploiting Remote Network Latency Measurements without JavaScript. In: USENIX Security. 2024

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

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

8. 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

9. 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

10. 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

5

CSI:Rowhammer Cryptographic Security and Integrity against Rowhammer

Publication Data

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

Contributions

Main author.

CSI:Rowhammer – Cryptographic Security and Integrity against Rowhammer

Jonas Juffinger[1,2], Lukas Lamster[2], Andreas Kogler[2],
Maria Eichlseder[2], Moritz Lipp[3], Daniel Gruss[1,2]

[1]Lamarr Security Research, [2]Graz University of Technology,
[3]Amazon Web Services

Abstract

In this paper, we present CSI:Rowhammer, a principled hardware-software co-design Rowhammer mitigation with cryptographic security and integrity guarantees, that does not focus on any specific properties of Rowhammer. We design a new memory error detection mechanism based on a low-latency cryptographic MAC and an exception mechanism initiating a software-level correction routine. The exception handler uses a novel instruction-set extension for the error correction and resumes execution afterward. In contrast to regular ECC-DRAM that remains exploitable if more than 2 bits are flipped, CSI:Rowhammer maintains the security level of the cryptographic MAC. We evaluate CSI:Rowhammer in a gem5 proof-of-concept implementation. Under normal conditions, we see latency overheads below 0.75 % and no memory overhead compared to off-the-shelf ECC-DRAM. While the average latency to correct a single bitflip is below 20 ns (compared to a range from a few nanoseconds to several milliseconds for state-of-the-art ECC memory), CSI:Rowhammer can detect any number of bitflips with overwhelming probability and correct at least 8 bitflips in practical time constraints.

1. Introduction

Rowhammer is a widespread DRAM issue, where cells lose charge faster upon accesses to rows in physical proximity [39]. By repeatedly accessing a row, an attacker can corrupt data in adjacent rows to undermine system security, *i.e.*, to gain kernel privilege from unprivileged applications [10, 22, 40, 60, 66, 68] and web browsers [23], escape web browser sandboxes [20, 56, 60], hammer from inside secure enclaves [22, 32], escape virtual machines [72], attack over the network [46, 67], and read memory to extract encryption keys [42]. These attacks motivated a long list of research on Rowhammer mitigations.

Rowhammer mitigations focus on *detection*, *neutralization*, or *elimination* [22] in software or hardware. *Detection*-based mitigations use static code analysis, performance counters from software, and the analysis of memory access patterns in hardware. However, they can be circumvented [22, 40]. *Neutralization*-based mitigations tolerate bitflips but restrict exploitation by physically distancing *aggressor* and *victim* rows [14, 41]. However, some hammer patterns can still succeed [39, 40]. *Elimination*-based mitigations are mainly hardware solutions, e.g., doubling the refresh rate and error correction methods like ECC DRAM and Chipkill, with limited effect on Rowhammer [16, 39]. Target Row Refresh (TRR) is designed specifically against Rowhammer. It can, however, be bypassed by exhausting the number of counters [21, 33, 56] or by using more advanced access patterns, e.g., half-double Rowhammer [40]. Thus, modern devices are still vulnerable [33, 40], highlighting the need for effective Rowhammer mitigations.

In this paper, we propose CSI:Rowhammer, a novel Rowhammer mitigation. Instead of focusing on specific properties of Rowhammer that can later turn out to be incomplete, e.g., flips happen only in directly neighboring rows [14, 40, 41], or at least two rows in a bank must be accessed [8, 22], the idea of CSI:Rowhammer is to provide more principled security guarantees. CSI:Rowhammer roots its security in a cryptographic message authentication code (MAC) for cryptographic security and integrity. It combines MAC-based error detection in hardware and flexible error correction in hard- and software. CSI:Rowhammer stores its MAC next to the data on the DRAM similarly to ECC memory and has, therefore, the same memory overhead.

With CSI:Rowhammer, the MAC is computed and compared in the memory controller upon every DRAM access. If data was corrupted and the comparison fails, the memory controller tries to correct a single bitflip. If unsuccessful, it raises a new exception which the kernel handles by trying to correct the data using different strategies. For this purpose, the MAC computation is also implemented as a new CPU instruction. A simple strategy, the universal fallback, is our parity-guided search, which outperforms a direct brute-force search that can become inefficient for higher numbers of bitflips. With this approach, we can correct 5 bitflips in a 256-bit data word in less than 300 ms. Beyond this, CSI:Rowhammer also enables more advanced strategies. It will, for instance, not correct errors in unmodified file-backed pages and reloads them from the disk instead, enabling the correction of any number of bitflips practically instantly. Another advantage of the correction in software is that CSI:Rowhammer enables OS vendors to implement an interface that allows admins to define reliability levels per process and page. For example, the maximum number of bitflips searched before killing a very important process in contrast to a process that can easily be restarted.

We analyze the detection and correction abilities of CSI:Rowhammer and show that it achieves significantly higher security and integrity than state-of-the-art error correction. In contrast to ECC memory, CSI:Rowhammer does not have a strict upper bound or constraints on the number and location of detectable bitflips. To break CSI:Rowhammer, an attacker needs to forge a cryptographic MAC, which is infeasible with the means of Rowhammer bitflips.

We evaluate the performance of CSI:Rowhammer based on a proof-of-concept implementation in a CPU and memory controller in gem5, running a modified Linux kernel. We show that our hardware-software co-design allows for low-latency, high-throughput integrity checking and, therefore, a performance impact of less than 0.75 % under normal operation. While the average latency to correct a single bitflip is less than 20 ns, CSI:Rowhammer can detect any number of bitflips and correct up to 8 bitflips in 256 bits of data within a practical time frame. CSI:Rowhammer has less than 0.01 % overhead on the CPU area. This underlines that CSI:Rowhammer is a practical mitigation that should be deployed to fully and permanently mitigate Rowhammer.

Contributions. We make the following contributions:

1. We propose CSI:Rowhammer, a hardware-software co-design to fully mitigate Rowhammer attacks using a cryptographic MAC for error detection in hardware.
2. We propose a novel software-level correction mechanism using a parity-guided bit-correction search as well as sophisticated correction strategies.
3. We evaluate CSI:Rowhammer with a proof-of-concept[1] implementation in gem5. We show that overheads are minimal, e.g., the performance overhead is below 0.75 % while the security level is raised to cryptographic levels.

Outline. First, we provide background in Section 2 and an overview of CSI:Rowhammer in Section 3. Hardware and software changes are discussed in Section 4 and Section 5. We evaluate the security and performance of CSI:Rowhammer in Section 6 and discuss related work in Section 7. In Section 8, we discuss the compatibility to other technologies and possible improvements and conclude in Section 9.

2. Background

In this section, we discuss DRAM, Rowhammer and its properties, as well as lightweight cryptography.

2.1. DRAM

The DRAM has multiple levels, allowing parallel access to the last level, the banks, to maximize throughput. Banks are divided into *rows* of *cells*, *i.e.*, the transistors and capacitors storing the data. A DRAM cell's charge depletes over time, requiring refreshes every 32 ms to 64 ms [34] to prevent data loss. These refresh commands are typically interleaved with normal data accesses by the memory controller, which is responsible to refresh each row within this time window.

Error Correction Code (ECC) and Chipkill memory have an additional memory chip to store data with correction codes [16], 8 redundancy bits per 64 bits (DDR4) or 32 bits (DDR5) of data. These are transferred

[1] Proof of Concept URL: `https://github.com/IAIK/CSIRowhammer`.

over a wider data bus of 72 bits and 40 bits, respectively. Chipkill is a DRAM error correction method, which, using the same additional memory, can correct up to 8 bitflips coming from a failure of a single DRAM chip [19]. ECC and Chipkill can correct errors from a single faulty DIMM contact. We compare ECC and Chipkill to CSI:Rowhammer in Section 7.2.

DRAM Errors. In large-scale systems, DRAM errors from various physical causes like cosmic rays [63] or random occurrences, increasing with aging cells and temperature [59] are well studied. Schroeder et al. [59] monitored the majority of Google's server fleet for 2.5 years starting in 2006. They observed failure rates of up to 70 000 per billion device hours (FIT) per Mbit of DRAM capacity. On the worst motherboard and memory configuration, 0.03 % of DIMMs saw an uncorrectable error in one year. Hwang et al. [28] studied DRAM errors on four systems and observed similar failure rates. Out of the 40 960 nodes monitoring Chipkill errors where bitflip recovery failed, 1.34 % had at least one Chipkill error in 583 days. Bautista-Gomez et al. [9] studied the frequency of multi-bit errors, focusing on independent errors, *i.e.*, they only count errors from the same word once. In total, they observed 55 000 independent memory errors in over a year, of which 85 (0.15 %) were not correctable.

2.2. Rowhammer

In 2014, Kim et al. [39] discovered the *Rowhammer effect*, *i.e.*, bitflips can be induced in DRAM through disturbance errors triggered by frequent memory accesses. For a more detailed background on Rowhammer, we refer the reader to this work or [49] for an overview. We discuss and compare mitigations in Section 7.

Data Dependency. Rowhammer causes a bit flip by discharging the capacitor in a neighboring cell. For this to happen the neighboring capacitor in the aggressor row must be discharged [39, 69]. Therefore, there is a correlation between the data in the aggressor rows and the flips, usually the victim adopts the values of the aggressor row. Kwong et al. [42] used this dependency to leak encryption keys with RAMBleed. If the orientation of the cells (true-cell / anticell) is inverted between the aggressors and victim row, the inverted values of the aggressor row are adopted [69].

Rowhammer Bitflips. Kim et al. [39] examined the number of multi-bit errors caused by Rowhammer per single 64-bit word. The worst tested module had 1.9 % uncorrectable errors, with 0.002 % being undetectable by

ECC, and Rowhammer is getting worse with every new DRAM generation and increasing density. LPDDR4 requires less than a 10th of the activations than DDR3 memory to hammer [35, 52].

Cojocar et al. [16] used a public Rowhammer bitflip database with 14 DIMMs [65] to craft the first Rowhammer attack on ECC memory. Their AMD CPU raises a machine check exception when detecting an uncorrectable bitflip, while the Intel CPU *ignores* it. On AMD, on average, 7.42 % of bitflips were undetectable; on Intel, 10.89 % were uncorrectable and, therefore, usable for an attack. They also found that many ECC implementations use more complex symbol-based correction methods [16], capable of *detecting* up to 4 bitflips if they are in different symbols but not able correct multi-bitflips. Hassan et al. [25] flipped up to 7 bits in a single 64-bit data word. This is beyond the detection and correction capability of every ECC implementation.

2.3. Lightweight Cryptography

Modern cryptography is often not designed for constrained devices that face various limitations but for high-performance devices. Lightweight cryptography offers algorithms designed to provide security and privacy guarantees while enabling their implementation on devices constrained in their energy consumption, die area, or latency [51].

Lightweight Message Authentication Codes (MAC). A message authentication code (MAC) is an authentication tag computed from a message input and a secret key [50]. The key protects the integrity and authenticity of the message: Only the owner of the key can compute and verify the MAC of a message. The attacker cannot forge a valid MAC of a new or modified message. Most widely-deployed MACs are based either on hash functions (like HMAC [50]) or on block cipher chaining (like CMAC), both of which are inherently sequential and thus have a high latency. A lightweight design for short inputs is SipHash [5], but its ARX design is more suited for software and not optimized for latency in hardware. More recently, tweakable block ciphers (TBCs) [47] have emerged as a promising lightweight primitive. TBCs are similar to block ciphers, but have an additional public tweak input that selects the encryption permutation together with the key. QARMA is a TBC with low latency [6].

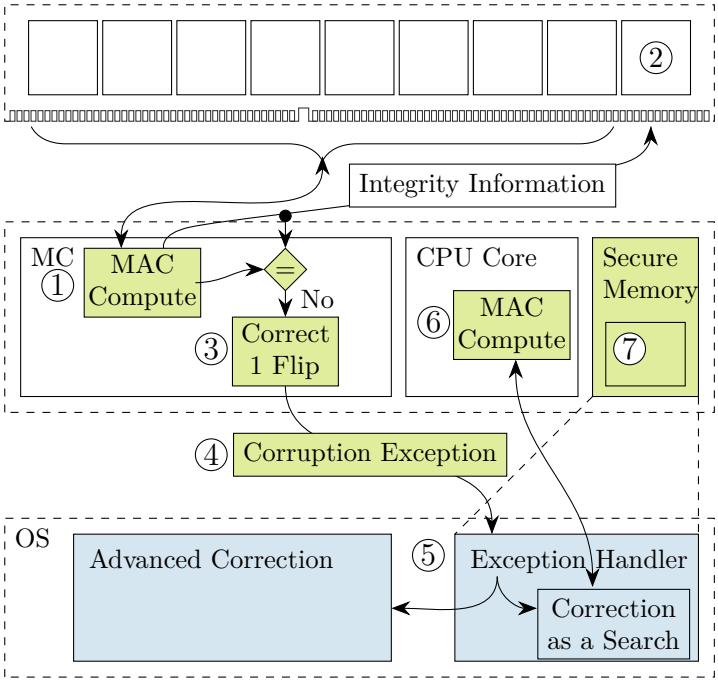


Figure 5.1.: High-level overview of the hardware and software components of CSI:Rowhammer.

3. CSI:Rowhammer

The idea of CSI:Rowhammer is to protect the integrity of all data in the DRAM with a cryptographic MAC to mitigate software-based DRAM fault attacks like Rowhammer but also randomly occurring bitflips. It is designed to work on a large variety of systems, from smartphones to large-scale virtualized server environments. As illustrated in Figure 5.1, CSI:Rowhammer is a hardware-software co-design.

On every access to main memory, CSI:Rowhammer uses a cryptographic MAC to detect if data has been corrupted. The computation and comparison of these MACs are performed in the memory controller ① and stored with the data ② to enable low-latency accesses. The MACs are stored on the same memory chip next to each other to make hammering them difficult. MACs cannot directly correct data. The memory controller tries to correct a single flip directly in hardware ③ by correction as a search. If this fails, this task is handed over to the operating system ⑤ by raising a data corruption exception ④. The exception handler uses

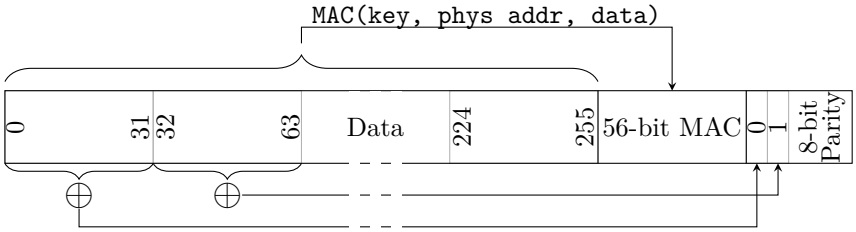


Figure 5.2.: Partitioning of the data, MAC, and parity bits. The data is 256 bits (DDR5) or 512 bits (DDR4).

different correction procedures to reconstruct the correct data. Using a new dedicated instruction set extension ⑥, the data correction is performed efficiently. However, corruption could also occur in the exception handler. Hence, it is stored in a dedicated secure on-die memory ⑦ that is not vulnerable to Rowhammer, similar to a CPU cache.

Detecting Bitflips. CSI:Rowhammer uses ECC memory for its additional space to store the integrity information (MAC and parity), as shown in Figure 5.2. The usage of a MAC allows CSI:Rowhammer to detect all memory corruptions, unaffected by their number, location, or distribution, which is enough to mitigate all Rowhammer attacks other than denial of service. We use a MAC based on the QARMA block cipher [6], designed for low latency, power, and area and used on ARM for pointer authentication [4]. With this basis, CSI:Rowhammer can be integrated into CPUs from smartphones to servers with negligible performance impact.

Typical ECC DDR4 memory stores 8 redundancy bits for 64 bits of data, ECC DDR5 for 32-bits of data. CSI:Rowhammer uses a 56-bit MAC and 8 parity bits, summing up to 64 bits in total. Thus, we protect 512 bits of data on DDR4 or 256 bits on DDR5. ECC data words can also be larger than 64 bits [16], making efficient correction and secure detection impossible. A trade-off can be a reduced correction capability in favor of an equally strong detection.

The MAC's key is randomly generated on each boot by the CPU internally. The memory controller and CPU cores have access to it, but it is not exposed to the software level. The memory controller computes the integrity information for every memory access. On a write operation, it computes the new MAC and parity and stores them in the DRAM. On a read, it computes and compares the MAC and parity with the stored one. The MAC computation is also implemented in every CPU core as a new

instruction to enable quick and efficient multi-bit error correction from software.

Correcting Bitflips. The memory controller corrects single bitflip errors caused by corrupted data, MAC, or parity and permanent errors caused by a faulty contact as-a-search.

If it fails because there is more than one bitflip, it raises an exception for the operating system. For instance, the advanced error correction algorithm can reload disk-backed data directly. Other data errors with no additional information are corrected by searching for the bitflips in the data. Our instruction set extension allows to access raw data and compute the MAC efficiently in hardware without software access to the key. Our search algorithm is guided by parity bits and brute forces possible corrections within each parity block. Thus, the search time increases exponentially with the number of bitflips. Depending on the importance of the process with the corrupted data or instruction, the operating system can decide whether to continue the search or kill and restart the process, or even halt the system. More generally, these decisions are up to the system vendor or administrator. OS vendors can decide on correction steps and to which extent they are tweakable by administrators, e.g., the maximum number of flip corrections for the kernel or user space. In case of a successful correction, the operating system can remember the necessary information for future corrections of the same physical location. If the correction is impossible, CSI:Rowhammer still mitigates Rowhammer exploits, as the detection is sufficient to stop an attack.

Protecting the Correction Mechanism. The exception handler for the data corruption exception must not be corrupted itself. Otherwise, a correction is not possible anymore, and the system must halt. For CSI:Rowhammer, we propose a small secure on-die memory, similar to a cache but mapped to a physical address range. The operating system puts all code required for correction into this secure memory.

Limitations. The goal of CSI:Rowhammer is to provide a negligible performance impact while fully mitigating Rowhammer. However, it does not protect against *all* data integrity attacks. In our threat model, the attacker can flip bits with Rowhammer, which is difficult to do precisely and allows only a limited number of flips. Replay, relocation, or substitution attacks, which require physical access to the memory bus, are out of scope. CSI:Rowhammer does not give any correction guarantees for a high number of bitflips, it can correct typical numbers of bitflips caused naturally or

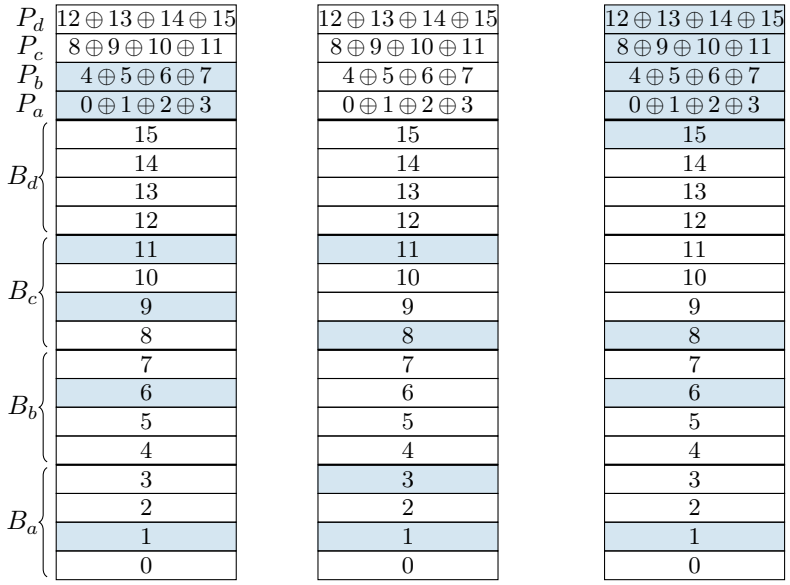


Figure 5.3.: Bitflip locations and parity bits. Gray data cells show a bitflip, and gray parity bits a corresponding parity mismatch. In the worst case, blocks (B_a - B_d) contain only an even number of flips. In the best case, every parity bit corresponds to one or zero flips.

by Rowhammer. The primary goal is the detection of memory corruption. CSI:Rowhammer detects *all* bitflips except the negligible portion resulting in MAC collisions due to the cryptographic design (see Section 6.7).

3.1. Error Correction as a Search

ECC memory uses symbol-based error-correction codes to detect a certain number of bitflips and usually correct one bitflip [16]. A MAC can only detect bitflips in the data but cannot correct them or give information about their location. To find the correction where no other strategy can be used, we have to flip bits and compute and compare the MAC successively. We find it when the MACs match. We start with flipping one bit and if the correction is not found, continue with all permutations with two flips and so on. Huang et al. [27] defined this search for bitflips in corrupted data as *Error Correction as a Search*. While our approach follows the same principle, the implementation differs.

Parity Bits. To improve the correction time, we use parity bits to find locations of flips. The 8 parity bits are computed by XORing 8 blocks of the data, as shown in Figure 5.2. The chosen partitioning allows for an efficient correction of permanent errors, e.g., on a transmission line or contact (see Section 4.4). We use Figure 5.3 to show how to get bitflip locations from the parity bits. In Figure 5.3a, the non-matching parity bits indicate an odd number of flips in B_a and B_b. The matching parities indicate an even number or no bitflips in B_c and B_d. In 5.3b, there is a mismatching MAC. The parity bits indicate an even number of flips in at least one block. In 5.3c, there is an odd number of bitflips in all blocks.

Correction Algorithm. We first consider only flips in the data and explain bitflips in the MAC and parity next. The correction algorithm pseudocode is attached in Section 11.1.

In Figure 5.3a, there are at least 2 bitflips, one in B_a and B_b. We try all permutations of these two flips. Because of the bitflips in 9 and 11, the MACs never match, and the correction continues. We add a double flip into our permutations that can be in any of the 4 blocks. While testing these permutations, we have find the correct data.

MAC Corruption. Error correction as a search in the data assumes that the MAC is uncorrupted. However, as the MAC is stored next to the data on the DRAM, it is also vulnerable.

In summary, there are four possible scenarios:

- No corruption, the MACs are equal.
- MAC corruption, the resulting MAC and the one in the DRAM only differ by a few bits.
- Data corruption, the resulting MAC differs significantly.
- Data and MAC corruption, the resulting MAC and the one in the DRAM differ by few bits when data is corrected.

CSI:Rowhammer performs error correction as a search and does not check the MAC for equality but approximate equality. If the data is correct, bitflips in the MAC can be found quickly, by checking for approximately equality. The limitations and security of approximate MAC equality are evaluated in Section 6.6. The single MAC computation for this check is done prior to the bitflip search on the data.

Parity Corruption. CSI:Rowhammer can correct data with a maximum of one flip in the parity bits. This is a limit we choose for a good compromise between correction time and capability. The parity bits make up only 2.5 %

(DRR5) or 1.4 % (DRR4) of the data and integrity information, and 2 flips are, therefore, a reasonable limit. Error detection is not compromised by the chosen limit of one parity flip.

To take a parity bitflip into account, we add one additional step to the correction algorithm. We again take Figure 5.3a and assume P_c to be flipped. In the first step, the algorithm will try the correction with 3 bitflips corresponding to the mismatching parity bits. This attempt fails, and the additional step is performed. Every parity bit is flipped successively, and the new expected number of bitflips is calculated. If P_d is flipped, the new expected number of bitflips is 4. If P_c is flipped, it is 2. The first step is repeated with all parity flips that decrease the number of expected bitflips. The correction with this additional step fails, so a double flip is assumed without flipping parity bits, which also fails. The additional step is added again. During this step, P_c is flipped, and the correction is attempted with the 2 single bitflips from P_a and P_b and one double flip. This correction succeeds, similarly to the first example.

Limitations In theory, it is possible to correct any number of bitflips this way. However, there are two upper bounds: First, the maximum number of flip permutations must not exceed the security boundaries of the MAC (see Section 6.6). Second, the number of permutations and computation time increases exponentially with the number of flips, limiting this search method to lower numbers of flips.

3.2. Cryptographic Design of the MAC

For a low-overhead mitigation, we require a MAC with very low latency, low power, and low hardware area requirements. QARMA is a lightweight tweakable block cipher design [6] fulfilling these requirements, and is notably already used in practice by ARM for pointer authentication [4]. QARMA allows pipeline stage boundaries between any two rounds [6] and supports block sizes of 64 or 128 bits.

CSI:Rowhammer MAC. We need to map 256-bit or 512-bit inputs to 56-bit tags. We use a parallel MAC (PMAC) construction [13, 48, 57] with 64-bit message blocks, where the physical address of each block is used as tweak for QARMA (see Figure 5.4). The resulting 64-bit tag is truncated to 56 bits to compute the final MAC. The security of the MAC is evaluated in Section 6.6. Table 5.1 shows latencies of the QARMA cipher variants [6], different MAC computation methods, and CPU memory

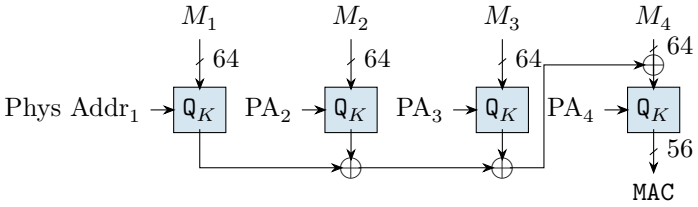


Figure 5.4.: CSI:Rowhammer PMAC for 256-bit data [57]. The output of the fourth QARMA block is truncated to 56 bit. The physical addresses used as tweaks are shifted right by $\log_2(64) = 6$ bit. They are unique for every message block.

Table 5.1.: Latency comparison of a single QARMA invocation, simulated for a 7 nm node similar to modern CPUs [6], the PMAC construction using pipelined QARMA$_5$-64-σ_0, and CPU memory accesses [18]. The throughput defines how many calls fit in the pipeline of a single QARMA block.

Operation	**Latency**	**Throughput**
QARMA$_5$-64-σ_0	2.20 ns	6
$\text{MAC}_{\text{PMAC}-256}$	5.13 ns	$\frac{\text{6 pipeline stages}}{\text{4 invocations}} = 1.5$
$\text{MAC}_{\text{PMAC}-512}$	6.60 ns	0.75
L2 cache access	3.86 ns	–
L3 cache access	10.27 ns	–
DRAM access	113 ns	–

accesses for comparison. The CSI:Rowhammer MAC is built using one QARMA cipher with 6 pipeline stages, resulting in the following latency for a 256-bit input (see Figure 5.4 and Table 5.1):

$$2.20\,\text{ns} + 2.20\,\text{ns} \cdot \tfrac{1}{6} \cdot 2 + 2.20\,\text{ns} = 5.13\,\text{ns}$$

4. Hardware Modifications

The generic design of CSI:Rowhammer can be integrated into different architectures. However, for our PoC (see Section 6.1) and this section, we focus on the changes for x86.

CSI:Rowhammer implements the MAC computation in hardware to minimize the overhead for DRAM accesses and allow the OS to compute it quickly during a correction. The MAC key is randomly generated. The memory controller computes and compares the MAC, and corrects single-bit errors and errors from faulty contacts. It provides a small secure memory to store the code required to handle a correction exception. Finally, the CPU cores must provide instructions to access corrupted data and compute the MAC.

4.1. MAC Key Generation

CSI:Rowhammer uses a MAC with a CPU-internal key generated using a cryptographically secure random number generator at boot [1, 3, 30]. At runtime, this key is available to the memory controller and all CPU cores in a software-inaccessible register. Multi-processor systems share the DRAM and, therefore, the key, over the processor interconnect. Attacks with physical access are out of scope.

4.2. Integrity Check

The memory controller is responsible for detecting data integrity failures in DRAM. It computes the integrity information for all memory accesses. On writes, the memory controller computes and stores the integrity information on the DRAM. On reads, it retrieves the integrity information and compares it with the computed one. A mismatch indicates a corruption of the data or integrity information.

4.3. Single-Bit Correction

If a corruption is detected, the memory controller starts the correction by performing a correction as a search for a single bitflip in hardware. A corruption in the MAC is found during the integrity check verification by

checking for approximate equality. If the corruption is in the data, one parity bit provides the location of the affected data block. In the worst case, the correction requires 32 or 64 MAC computations for 256 and 512 bits of data, respectively, half of that on average. If the integrity check reveals errors in the parity bits, the parity bits are recalculated.

During the search, only one of the 4 or 8 input blocks changes. The other blocks can be computed once and reused. Thus, the subsequent MAC computations are two QARMA calls with a total latency of 4.40 ns and 12 pipeline stages. This results in an average single bitflip correction time of:

$$5.13\,\text{ns} + 16 \cdot \frac{4.40\,\text{ns}}{12} = 11\,\text{ns}$$

For DDR4 memory, it is 18.3 ns. If the parity bits indicate two or more flips, the correction for a faulty contact is tried.

4.4. Permanent Errors from a Faulty Contact

Permanent faults from, e.g., a faulty DIMM contacts or transmission lines make up a considerable part of DRAM errors [62]. They must be corrected in hardware, as they cannot be fixed by writing back correct data [63]. Similar to ECC and Chipkill, CSI:Rowhammer is able to correct the errors caused by a singly faulty contact. The correction is performed before raising a corruption exception, if the MAC and more than one parity bit mismatch. For this explanation, we assume that a faulty contact causes bits to *stick-at-0*.

The memory controller first computes the logical `OR` over blocks with the size of the DRAM data bus width. The `OR` result shows the stuck contact offset as these bits are always '0'. The parity bits mismatch for the blocks where the stuck contact had an effect. Together, both block and bit offset are identified. With this, the correction requires as many MAC computations as there are '0's in the `OR` result. With random data, there is a probability of 11.8 % (32-bit bus) or 22.2 % (64-bit bus) that there is an additional '0', not caused by a faulty contact, resulting in a correction time of 5.74 ns and 8.05 ns, respectively. This the CSI:Rowhammer performance impact, but keeps the system usable.

For stuck integrity information contacts, it works similarly. It affects 7 bits in the MAC and 1 in the parity bits. Correct data can be recognized if all bits at an offset are stuck and the rest of the MAC is correct. This

reduces the effective MAC size to $56 - \frac{7}{2} = 52.5$ bit. We leave the design and analysis of a faulty contact correction with bitflips in the DRAM for future work. CSI:Rowhammer cannot correct both at the same time, similar to ECC and Chipkill.

4.5. Data Corruption Exception

CSI:Rowhammer adds one new exception to inform the operating system about a data corruption not correctable in hardware. We call it *data corruption exception* and use one of the reserved vectors 21-31 [29]. The data corruption exception is designed as a fault, allowing to continue at the faulted instruction after the correction. The handler receives the physical address of the corrupted data or instruction from an architecture-dependent register. We propose the same register used for page faults, the `CR2` on x86 [29]. It contains the physical address to not rely on page tables. In virtualized environments, the host receives the exception, translates the host physical address (HPA) to the guest physical address (GPA), and forwards it to the guest. This mechanism allows transparent correction for guests not supporting CSI:Rowhammer via the host (see Section 8.1).

4.6. Secure On-Die Memory

A corruption must not happen in any parts of the OS required to correct corruptions, otherwise, the system must halt. To guarantee no corruption for the exception handler code, IDT, and GDT, the CPU provides a small on-die random access memory.[2] It is part of the physical address range and the OS can map it into virtual memory with non-evictable TLB entries. For systems without an MMU, the memory is at a fixed physical address. It is 4 pages large to fit our prototype implementation of the correction as a search algorithm (0x1500 bytes), the IDT, and GDT.

In virtualized environments, the secure memory is simulated for the guests in the DRAM. In case of a corruption in the simulated secure memory, the host transparently corrects it. This guarantees the incorruptibility of the secure memory for all guests without requiring a larger SRAM.

[2] On-die area is not the primary limitation today anymore. Some Intel processors utilized unused die area already with the integration of, e.g., 128 MB DRAM chips, the Apple M1 quadrupled the L1 size with marginal thermal and efficiency changes, as they are using 4 times larger pages.

Table 5.2.: The three instructions of CSI:Rowhammer used for efficient and secure correction in software.

Instruction	OPs	Description

`csi_mac`	`ZMM`
	`rx`
	`rx`
		Calculates the MAC of the data (OP1) for the physical address (OP2) and writes it into OP3.

`csi_load`	`rx`
	`ZMM`
	`rx`
		Reads from physical address (OP1) the raw corrupted data (OP2) and integrity information (OP3).

`csi_xchg`	`rx`
	`ZMM`
	`rx`
	`ZMM`
		Writes new data (OP4) into the physical address (OP1) if the old data (OP2) and MAC (OP3) are unchanged at physical address (Listing 5.4 in Section 11.2).

4.7. CPU Cores

The operating system corrects data with more than one bitflip. The CSI:Rowhammer hardware modifications support the operating system with instructions to compute the MAC and access the raw corrupted data and integrity information.

Instruction Set Extension CSI:Rowhammer adds three privileged instructions, shown in Table 5.2, to allow for correction in software. `csi_mac` takes the 256-bit or 512-bit data in an AVX register and the physical address, *i.e.*, the tweak, and writes the computed MAC back to a general-purpose 64-bit register. Implementing the MAC computation in hardware allows the MAC key to stay secret.

Additionally, two new instructions access the raw data and integrity information by its physical address without raising an exception. The `csi_load` instruction loads the corrupted data into a ZMM register and the integrity information into a 64-bit register. The `csi_xchg` instruction writes the corrected data from a ZMM register back into the memory if it did not change in the memory while the correction was running. Section 8.4 describes the purpose of `csi_xchg`, *i.e.*, prevent correction race conditions. Listing 5.4 in Section 11.2 shows its pseudo-code.

Within a virtualized guest, the instructions transparently translate the given GPA internally to an HPA to match the MAC's tweak and the

actual physical location. The last translation is cached for quick succeeding executions.

Nesting Bit We need to detect nesting of the corruption exception handler (see Section 5.2). For this, we need one bit in a register that is unique for every thread. We propose using the highest bit of the CR3 register, as it is unused, and its value is already unique for every task. The only hardware change is to allow writing this bit without flushing the TLB.

Non-evictable TLB Entries To ensure compatibility with the IDT and the CPU's frontend, the secure memory is addressable with virtual addresses. The paging structures to map the secure memory to a virtual address also need to be protected against Rowhammer. We propose reusing existing mechanisms and allowing the locking of secure memory pages in the TLB [2]. This allows the CPU to resolve addresses directly from the TLB. We propose using an MSR interface to configure these virtual to physical mappings.

5. Software Modifications

The memory controller detects data corruption and tries to correct single bitflips and errors from faulty contacts. If it fails, the operating system takes over to try the correction and limit the impact of an uncorrectable corruption.

5.1. Exception Handling

The operating system must register a new handler for the data corruption exception. The handler and the correction-as-a-search code are secured against corruption by residing in the secure on-die memory provided by the CPU. The advanced correction procedure uses various kernel code and structures and is, therefore, not protected against bitflips. If kernel code or data is corrupted and accessed during a correction, another exception is raised and handled in a nested handler. The nested handler detects the nesting (see Section 5.2) and attempts a correction as a search without accessing any data outside the secure memory.

The kernel stack, used by the exception, is unique for every task, and can, therefore, not be in the secure memory, and is vulnerable to bitflips.

```
 1 corruption_exception_handler() {
 2   if (has_nested_bit_set(CR3)) {
 3     enable_interrupts();
 4     error_correction_as_a_search(corruption_address);
 5   } else {
 6     set_nested_bit(CR3);
 7     enable_interrupts();
 8     advanced_error_correction(corruption_address);
 9     clear_nested_bit(CR3);
10   }
11 }
```

Listing 5.1: Pseudocode of the corruption exception handler with nesting detection. The advanced error correction is guarded by the nested flip being one. If a second exception is raised the error correction as a search is executed.

To deal with corruption in the kernel stack, we forward the kernel stack pointer 64 bytes before accessing it. In case of corruption, the following exception handler moves the kernel stack pointer out of the corrupted data block. It can then correct the kernel stack and continue with the previous exception handler.

5.2. Nesting Detection

Detecting the nesting of our exception handler is important as it means that there was a corruption during an advanced correction attempt. If that happens, the exception handler must not run the advanced correction and can only correct by search. Listing 5.1 shows how the nesting detection is implemented using the highest bit in the CR3 register. To detect nesting, we use one bit, called the *nesting bit*. If the current Linux task is in the corruption exception handler it is 1 and if not, it is 0. It must fulfill two conditions:

- When the corruption exception happens, the nesting bit must already be stored in a CPU register as retrieving it from memory could cause another corruption exception.
- The bit is either set or not set for each task, *i.e.*, process or thread. Scheduling is enabled during correction, but two independent tasks can have two (independent) exceptions.

```
1 error_correction_as_a_search(corruption_address) {
2   // get the raw corrupted data
3   data, integrity = csi_load(corruption_address);
4   data_copy = data;
5   mac = integrity & 0x0FFFFFFFFFFFFFF;
6   parity = (integrity & 0xF00000000000000) >> 56;
7
8   correction_as_a_search(parity, [&](flip_mask) {
9     data ^= flip_mask;
10    // compute the mac of the data
11    computed_mac = csi_mac(data, corruption address);
12    if (popcount(mac ^ computed_mac) < 4) {
13      goto found_correction;
14    }
15    data ^= flip_mask;
16  });
17  // no correction found, kill the process or panic
18  panic();
19 found_correction:
20  // write the correct data back if it did not change
21  csi_xchg(corruption_address, data_copy, mac, data);
22  reti;
23 }
```

Listing 5.2: Pseudocode showing the usage of the `csi_load`, `csi_mac` and `csi_xchg` instructions.

5.3. Locking

We use a lock in the advanced correction preventing two tasks from correcting the same data. If a task accesses data already being corrected, it waits, and the CPU core can run other tasks. This lock does not exist in a nested handler as it cannot access kernel resources. In that case, the `csi_xchg` instruction prevents the unintended overwriting of new data with old corrected data (see Section 8.4). The locking of shared kernel resources allows for a theoretically unlimited number of simultaneously running advanced corrections.

5.4. Correction Procedure

The correction procedure uses the great flexibility that can be achieved with the knowledge available to the operating system. File-backed pages like memory-mapped files, page cache pages, or the kernel itself can be

reloaded from the disk, correcting any number of bitflips. Taking into account that Rowhammer bitflips are reproducible [39], the kernel can remember the locations of previously corrected bitflips and use them for future corrections. If the page cannot be reloaded from the disk and is not in the cache, the correction is performed as a search. If successful, the page is reallocated to minimize the chance of a corruption in the near future. If the correction as a search was unsuccessful, the operating system can behave differently. OS vendors can enable to define different reliability levels, for example, the maximum duration of the search based on the importance of a process or the impact of a restart of a process, e.g., a database server vs. a service logging system stats. If the uncorrectable corruption is in the kernel, it can only try to shutdown and restart as gracefully as possible. Figure 5.8 in Section 11.5 shows the correction in a flow chart.

Listing 5.2 shows the usage of the `csi_load`, `csi_mac` and `csi_xchg` instructions. First, `csi_load` is used to get the data and integrity bits from the physical `corruption_address`. The data is duplicated for the `csi_xchg`. During the correction as a search, the `csi_mac` instruction is used to compute the MAC. If data was found with an approximately equal MAC, it is written back into the DRAM with the `csi_xchg` instruction. It checks if the data was modified in the meantime and only writes it if it is unmodified.

6. Evaluation

To test the performance, correct functionality, and security of CSI:Rowhammer, we evaluate it using different techniques. We build a prototype with gem5 running a modified Linux, simulate corrections to test our algorithm, and calculate the security boundaries of our MAC under different scenarios.

6.1. Prototype Implementation with gem5

We implement a prototype of CSI:Rowhammer's hardware modifications in gem5 [12] and the required software changes in the Linux kernel. We use gem5 in the full-system emulation mode running an up-to-date Debian buster with our modified Linux kernel 4.19, as shown in Table 5.3.

Memory. There is no ECC memory in gem5. Therefore, we first modify the memory controller to extend its memory bus to 72-bit to simulate ECC DIMMs. We add the integrity information computation and check to the memory controller with the latency for the MAC computation and the official QARMA software implementation from the NIST submission [7]. The memory model of gem5 does not simulate any memory flaws like spontaneous bitflips or Rowhammer. To test CSI:Rowhammer we add an interface to cause corruptions in the DRAM. Similar to related work [15, 58] we do not use a cycle accurate DRAM model like dramsim3 [45] or dramsys4 [64], because CSI:Rowhammer does not change the number, frequency or ordering of DRAM accesses.

CPU. We add a data corruption CPU exception that is raised by the memory controller if a MAC mismatch is detected, and the three new instructions for the correction. As gem5 does not support AVX, we use code by Wang et al. [70] that adds basic support, sufficient for our use.

Linux. A modified Linux kernel is required to support the new exception and perform the correction. We add the exception handler that performs the correction as a search. As we introduce the new `csi_` CPU instructions, there is no compiler support. Hence, we call them using inline assembly to insert raw bytes into the machine code directly; for an example, see Listing 5.5 in Section 11.3.

Limitations. As the secure memory has no impact on the performance or correction, we do not implement it. The secure memory is only required

Table 5.3.: The gem5 system used for our evaluation.

Architecture	x86-64
CPU	Single Core, 3 GHz, O3 (Out-Of-Order) CPU
DRAM	2 Channel, 4 GB DDR4_2400_8x8
Cache	64 kB L1I, 32 kB L1D, 2 MB LLC
gem5 Mode	Full System
Operating System	Debian Buster with Linux 4.19

to prevent corruption in the correction code which cannot occur in our simulation.

6.2. Performance Evaluation

To evaluate the performance of CSI:Rowhammer, we first measure the performance impact without memory corruptions using the PARSEC [11] and SPLASH2x [55] benchmarks. Second, we estimate the duration of the data correction in software, depending on the number of bitflips.

Runtime Performance Overhead We evaluate the performance impact of CSI:Rowhammer during normal operation without memory errors for the PARSEC [11] and SPLASH-2x [55] benchmarks, which cover a wide variety of memory intensive and non-intensive workloads. We ran the entire benchmarks and compare the overall time they took with or without the delay added by CSI:Rowhammer. The performance degradation is caused by the MAC computation performed on every memory access. We use the MAC computation durations for 256-bit data (5.13 ns) and 512-bit data (6.60 ns) from Table 5.1. The overhead for the different applications is shown in Figure 5.5. The geometric mean overhead over all benchmarks is 0.67 % ($n = 45$, $\sigma_{\bar{x}} = 0.12$) on the system with a 5.13 ns delay and 0.74 % ($n = 25$, $\sigma_{\bar{x}} = 0.17$) with a 6.6 ns delay. The worst overhead is 3.28 % ($n = 55$, $\sigma_{\bar{x}} = 0.013$) and 4.24 % ($n = 30$, $\sigma_{\bar{x}} = 0.017$), respectively. This overhead is small in comparison to the security that CSI:Rowhammer can provide. Figure 5.7 in Section 11.4 shows the memory accesses of every benchmark. Benchmarks with a high number of memory accesses also experience a worse performance impact, confirming that the slowdown comes from the MAC computation overhead.

Simulated Correction Overhead Evaluation We evaluate the duration of the correction as a search. For data that can be corrected through

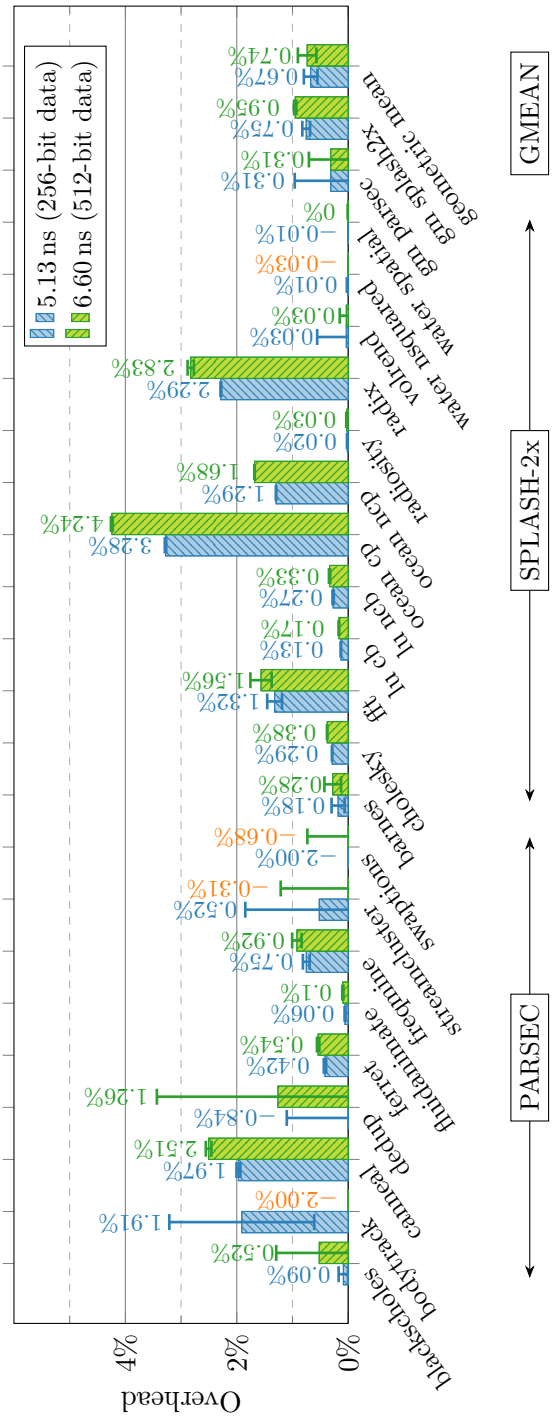


Figure 5.5.: PARSEC and SPLASH-2x benchmarks run in our modified gem5 with two different integrity check delays. The worst overhead of CSI:Rowhammer is 4.24% ($n = 30$, $\sigma_{\bar{x}} = 0.017$), the geometric mean is 0.67% ($n = 45$, $\sigma_{\bar{x}} = 0.12$) for 256-bit data and 0.74% ($n = 25$, $\sigma_{\bar{x}} = 0.17$) for 512-bit data.

Table 5.4.: The number of MAC computations in the best, average, and worst case, based on the flip distribution, see Figure 5.3. Followed by the theoretical MAC duration, and the permutation loop durations for three CPUs. The higher MAC or loop duration is a good estimate for the respective CPU. 256 Data Bits, BC = Broken Connection, *Correction in Hardware

Errors	**# MAC Computations**			**MAC Duration**	**Simulated Bit Permutation Loop Duration**		
	Best Case	Average Case	Worst Case	Average Case	Apple M1	Intel i7-1165G7	AMD 5800X
1*	17	17	17	11 ns	-	-	-
2	528	711	1 985	2.43 µs	952 ns	3.68 µs	3.85 µs
3	17 440	33 800	63 520	115 µs	40.2 µs	124 µs	126 µs
4	576 608	1.51×10^6	3.94×10^6	5.17 ms	1.74 ms	6.65 ms	7.49 ms
5	1.91×10^7	6.91×10^7	1.26×10^8	236 ms	78.9 ms	261 ms	293 ms
6	6.32×10^8	3.07×10^9	5.87×10^9	10.5 s	3.51 s	12.8 s	14.0 s
7	2.10×10^{10}	1.21×10^{11}	1.88×10^{11}	6.87 min	2.30 min	9.11 min	10.0 min
8	6.97×10^{11}	5.72×10^{12}	5.82×10^{12}	5.44 h	1.70 h	6.11 h	6.74 h
BC*	1	1.125	32	5.77 ns	-	-	-

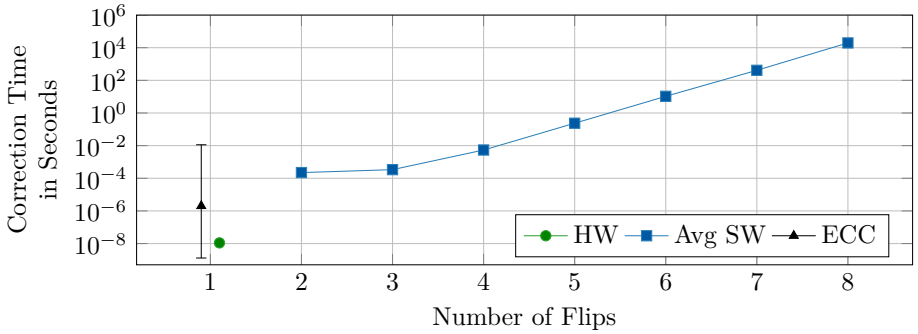


Figure 5.6.: Correction time from Table 5.4 - MAC Duration. One bitflip is corrected in hardware, the exception overhead for the software correction is 223 µs. The ECC correction time is dependent on the CPU and varies greatly. MAC and parity bitflips are corrected with one MAC computation and therefore not shown.

Table 5.5.: MAC strength with approximate equality for 256 bit. p is the number of bit permutations tried to correct the data. d is the number of flips allowed to be different in the MAC for the approximate equality. The MAC strength is a result of d. CSI:Rowhammer is secure because the strength is always higher than $\log_2(p)$.

Data Flips	$\log_2(p)$	d	MAC Strength
5	26.0	3	41.2
6	31.5	2	45.4
7	38.8	1	50.2
8	42.4	0	56.0

other means, e.g., reloading from disk, any number of bitflips can be corrected in a very short time. The gem5 simulator is well suited for relative performance comparisons but not for absolute run time numbers. To give realistic numbers for the average correction duration on a modern CPU that implements CSI:Rowhammer, we simulate the correction algorithm on three physical CPUs. First, we implement our correction as a search algorithm with a `memcmp` instead of the MAC comparison; we call it *bit permutation loop*. Then we run a Monte Carlo simulation 10 000 times to get the average number of MAC computations per number of bitflips and the duration of the *bit permutation loop* on all three CPUs.

Table 5.4 shows the results. The MAC computation duration is calculated by multiplying the number of MAC computations with the duration of a single MAC computation divided by the throughput (see Table 5.1). Under the assumption that the MAC computation is performed on an AVX floating-point execution unit, the MAC computation can run in parallel to the *bit permutation loop* that uses different execution units, and their times do not add up.

If the simulated *bit permutation loop* without the MAC computation is faster than the combined MAC computation, we use the MAC computation duration as a estimate for the correction time, otherwise, *bit permutation loop* duration. The Apple M1 could, because of its large number of ALUs, keep more than one QARMA cipher block busy, as it is limited by the MAC computation duration. The other CPUs can almost keep the QARMA cipher block busy. For 512-bit data, correcting 7 bitflips requires 1.41×10^{13} MAC computations and takes 35 hours and for 6 bitflips 28 minutes.

We use our gem5 prototype to measure the overhead caused by the exception, which is 223 µs. In Figure 5.6, we add this overhead to the durations from Table 5.4. The correction time of ECC memory is highly dependent on the CPU. Kwong et al. [42] measured access times 5 orders of magnitude larger on an error. Cojocar et al. [16] measured accesses 500 times as well as only 1.01 times larger. To compare these numbers with our durations, we multiplied them with a typical DRAM access time of 113 ns [18].

6.3. DRAM Organization & Energy Cost

Unlike other memory integrity protection methods like IVEC [27] or Synergy [58], CSI:Rowhammer requires only a single memory access to

get the data and integrity information. Memory organization in the form of DRAM channels, therefore, has no impact on the performance. The QARMA authors did not analyze the energy cost, reasoning that the single memory access dominates the energy consumption and that a few thousand gates are nearly negligible in modern CPUs [6]. Similar to Qualcomm's pointer authentication, we also use the QARMA cipher in combination with a memory access and can apply the same argument. The added SRAM has a fraction of the cache capacity on a typical modern CPU and, therefore, negligible energy cost as well.

6.4. CPU and DRAM Area Estimation

In this section, we evaluate the area requirements in the CPU and DRAM to implement CSI:Rowhammer.

CPU Die Area. The components required are a MAC circuit in the memory controller, one per core, and a secure SRAM of 4 pages (16 kB). The SRAM is not a hard requirement, and it can be omitted with a slight decrease in system reliability without compromising security.

We estimate the SRAM size for TSMC's 7 nm process [71], because the area of QARMA is also provided for a 7 nm process [6]. One TSMC 7nm SRAM cell has $0.027\,\mu m^2$ [71], resulting in an area for 4 pages of $0.027\,\mu m^2 \cdot 8 \cdot 4096 \cdot 4 = 3\,539\,\mu m^2$. With the area of one latency optimized $QARMA_5$-64-σ_0 [6] block of $1\,238.1\,\mu m^2$, the overall area requirement for a 4-core CPU is $3\,539\,\mu m^2 + 5 \cdot 1\,238\,\mu m^2 = 9\,729\,\mu m^2$. This is approximately 0.006 7 % of the Tiger Lake CPU die with comparable feature size and an area of $146.1\,mm^2$ [17], a lot less than the 0.5 % and 0.6 % of Blockhammer [73] and Graphene [54].

DRAM Area. On systems already equipped with ECC memory, CSI:Rowhammer does not add any DRAM area. For systems without ECC memory, CSI:Rowhammer adds 12.5 % (DDR4) or 25 % (DDR5) area.

6.5. Cost vs. Device Vulnerability

Previous works report that similar devices show high variance in Rowhammer susceptibility [25, 33, 39, 43]. Kim et al. [39] used DDR3 memory and observed a very low number of multi-bit errors (1.9 %), CSI:Rowhammer corrections would have a minor performance impact. More recently,

Kim et al. [35] found that first-generation DDR4 DRAM is four times as vulnerable. The higher rate of multi-bit errors resulting from it can cause longer correction times, but the chance of uncorrectable errors is still low.

Recent generations of (LP)DDR4 memory show a significantly higher susceptibility to Rowhammer. Hassan et al. [25] found up to 7 flips in a single 64-bit data word. The worst cells Kim et al. [35] found, fail after less than 5000 hammers; this memory requires TRR, see Section 8.1.

6.6. MAC Security Evaluation

We now evaluate the security and correctness of the MAC.

Construction. The PAM construction with tweakable block ciphers is proven to be secure [57]. The usage of the physical address as the tweak for the MAC makes targeted hammering of the data and MAC infeasible (see Section 6.7).

Approximate Equality. We can precisely determine the security loss due to the approximate equality check. If we consider an n-bit MAC and allow approximate equality with up to d errors, then the success probability of a single forgery attempt increases from 2^{-n} to $2^{-n} \sum_{k=0}^{d} \binom{n}{k}$, corresponding to a security loss of $\log_2(\sum_{k=0}^{d} \binom{n}{k})$ bits. As an example, consider $n = 56$ and $d = 3$, where the security degrades from 56 to $56 - 14.8 = 41.2$ bits. d is reduced with a higher number of data bitflips, as shown in Table 5.5, and is never higher than $d = 3$. For 512-bit data, $d = 3$ for 4 flips in the data and reduced to $d = 0$ for 7 flips, the maximum.

MAC Leakage. Bitflips in the MAC are corrected with approximate equality, so there is no relation between the correction time and flipped MAC bits. A timing side channel can leak the location of a flipped bit in the data. A RAMBleed-like attack could, in theory, be used to leak the MAC and virtual machines can compute the MAC. However, the dependency of the MAC on the physical address makes an attacke, even then, infeasible, see Section 6.7.

6.7. Detection and Correction Security Evaluation

In this section, we evaluate the security of CSI:Rowhammer in regards to data corruption, protection against a Rowhammer attack, erroneous corrections, and uncorrectable errors.

Bitflip Location. The location of Rowhammer bitflips and memory errors within the data or integrity information has no impact on the error detection capabilities of CSI:Rowhammer. Therefore, CSI:Rowhammer can guarantee the detection of all Rowhammer attacks and memory errors with a very high chance and mitigate all attacks, as shown in the following paragraphs. If no advanced correction method is applicable, the correction as a search can correct up to 8 bitflips in 256-bit data if there are no bitflips in the integrity information and up to 5 bitflips in the data with 3 bitflips in the MAC and 1 flip in the parity bits (see Section 6.6).

Single Bit Data Corruption. We use the MAC and the parity bits to verify the integrity. Therefore, 1 bitflip can never lead to a silent data corruption, if it creates a second preimage with the same MAC, the parity bit check still fails.

Silent Data Corruption. Every system is exposed to bitflips. We evaluate the frequency of silent data corruptions caused by naturally occurring bitflips that cause a second preimage. There must be at least two flips in 256 or 512 bits of data for the possibility of a silent corruption. With a secure MAC, the probability of producing a second preimage is independent of the number of bitflips. Previous studies only report multi-bit errors per 64-bit ECC data word. However, multi-bit errors are relatively rare, so there is a low chance of them piling up in a 256 or 512-bit data block. Therefore, we can take the frequency of multi-bit errors observed in other studies also for this evaluation. Additionally, we take the worst reported FIT by Schroeder et al. [59] of 70 000 and distribute them uniformly over the memory to get the probability of additional multi-bit errors in the data. On a double error, 3969 MAC computations are performed in the worst case. The overall Silent Data Corruption rate of CSI:Rowhammer is less than once per 10^9 billion years.

Random Rowhammer Attack. We evaluate the probability of an attacker finding a second preimage by randomly hammering data. As a single bitflip is always detected, an attacker must flip at least two bits in 256 or 512 bits. If flips did not produce a second preimage, the correction is run by the OS, which takes time exponentially corresponding to the number of flips. An attacker flipping two bits is the best case for finding a second preimage as quickly as possible. In a very optimistic scenario for the attacker, two bits always flip after hammering a page for two refresh intervals or 128 ms. The attacker can either hammer a new location containing different data or change the victim data after every

hammer. In this scenario, the probability of finding a second preimage after hammering for one year straight is $9.75 \cdot 10^{-5}\%$.

Targeted Rowhammer Attack from a Virtual Machine. We evaluate if it is possible for an attacker to create valid data and integrity information in a victim row with Row-hammer. For an attacker VM it would be possible to compute the MAC for victim data, free the row and hammer it after a victim VM or the host allocated it. The usage of the physical address as a tweak makes this attack infeasible:

In short, an attacker cannot construct an aggressor row that has both, matching data and checksum to the intended victim row data, which is a requirement stemming from the data dependency of Rowhammer (see Section 2.2).

In more detail, there are two options for an attacker:

1. Flip the victim data to a second preimage without changing the MAC: The attacker must find an exploitable second preimage of the victim row data and a second preimage of the aggressor row data that does change the victim data but not the victim MAC.
2. Flip both data and MAC: The attacker has to find aggressor data that changes the victim data to be exploitable and has a MAC that changes the MAC in the victim row to correspond to the new victim data.

Finding the required second preimages for the victim and/or aggressor rows is computationally infeasible. Additionally, most modern DRAM requires double-sided hammering to flip bits, doubling the number of data and MAC combinations for the aggressor rows an attacker must find [21, 33]. For half-double Rowhammer the near and far aggressors must contain the correct data for it to succeed, quadrupling the computational work [40].

Finally, for a second preimage to exist, on average 41.2 bits (see Table 5.5) must be flippable in the victim data. This makes the attack rather hypothetical, as it is way beyond the amount of bit flips we observe in DRAM today and would render the DRAM module likely unusable for most tasks. Additionally, these on average 41.2 changed bits must still be valid data that is useful for the exploit, so that the attacked application or kernel does not crash and behaves in the way anticipated by the attacker.

Highly Vulnerable DRAM. The number of bitflips does not change the detection guarantees of CSI:Rowhammer. A high number of flips can cause a denial of service, but cannot be exploited. The same is true if the

memory chip that stores the integrity information is very vulnerable to Rowhammer. It increases the chance of uncorrectable errors and cases where many flips in the MAC look like uncorrectable data errors, but the error detection is always guaranteed.

Erroneous Correction. During the correction, the operating system computes many MACs with different data, each of which could be a second preimage. In the worst possible case, 8 bits flip in the last tested locations of the same last parity bit. This leads to the maximum number of flip permutations: the worst-case number in Table 5.4 times 2, *i.e.*, 1.164×10^{13}, and $\log_2(1.164 \times 10^{13}) = 43.4$ is lower than the width of our MAC $n = 56$, see Table 5.5. The probability p of finding a second preimage is 0.016 1 %. This correction takes approximately 6 hours. To get a chance of 50 %, the attack takes $\frac{\log(0.5)}{\log(1-0.000161)} \cdot 6\,\text{h} = 1\,076\,\text{days}$. The outcome of this correction is impossible to predict for the attacker, and thus of only very limited use. With 512-bit data, only 6 flips are correctable in reasonable time; 7 bits take approximately 92 hours in the worst case. The number of permutations for 7 flips, in this case, is 5.035×10^{13}. The probability p of finding a second preimage is 0.069 9 %.

Uncorrectable DRAM Errors. In Section 2.1, we summarize the findings of three studies on DRAM errors in large-scale systems. From these numbers, it is possible to make predictions about the correction capabilities. Bautista-Gomez et al. [9] give insight into the number of flips in multi-bit errors. Less than 0.002 % of multi-bit errors have more than 8 bits flipped and are therefore not correctable as a search. According to Hwang et al. [28], 1.34 % of nodes in one system they observed had at least one Chipkill error in 583 days. These errors are not correctable as a search but, e.g., by reloading pages from disk and always detected.

7. Related Work

There are many proposals for Rowhammer mitigations and data integrity protection but with little overlap. All presented Rowhammer mitigations were already broken by follow-up work. Data integrity methods are reliable at detecting bitflips, but either have a significant performance impact or cannot correct bitflips caused by Rowhammer.

7.1. Rowhammer Mitigations

Rowhammer mitigations hinder specifically the flipping of bits or the exploitation of Rowhammer bitflips. Gruss et al. [22] described Rowhammer mitigations in three categories:

Elimination. TRR is a Rowhammer mitigation in recent DRAM modules. It counts the accesses to rows and refresh adjacent rows if the accesses exceed a pre-configured value. TRR can be bypassed in two ways: by multi-sided hammering [21, 33] and by half-double hammering [40]. It is also possible to detect Rowhammer attacks with frequent element analysis in the DRAM data streams [38, 54]. They have a chip area overhead that is dependent on the number of DRAM ranks and is higher than that of CSI:Rowhammer.

Neutralization. While not preventing bitflips, some mitigations prevent Rowhammer attacks by physically distancing rows. ZebRAM [41] isolates all rows by making only every second row accessible to programs. The rows in between are used as integrity-checked swap space. ZebRAM can be circumvented with half-double hammering [40]. Brasser et al. [14] physically distance kernel and userspace. CSI:Rowhammer neither introduces such physical distancing nor can it be bypassed by known or future hammering patterns.

Detection. Irazoqui et al. [31] presented MASCAT, a static code analysis tool to detect Rowhammer attack code. Others used performance counters to detect microarchitectural attacks (e.g., Rowhammer) [24, 26, 75]. However, they are limited to specific assumptions about Rowhammer and have detection rates far below those of CSI:Rowhammer.

7.2. Data Integrity Protection

Data integrity protection is a very long-studied topic. However, no solution specifically focuses on Rowhammer [15, 27, 36, 37, 53, 58, 74]. Instead, they focus primarily on sophisticated attacks requiring physical access to a device like cold-boot attacks or attacks on the memory bus, enabling replay, relocation, and data substitution and often include encryption. These protections are required, e.g., in secure enclaves like Intel SGX or ARM TrustZone, where security is the primary aspect. They come, however, with a significant performance impact on memory-intensive workloads [58]. CSI:Rowhammer manages the balance between security and performance better than any other DRAM integrity verification method by focusing on data modifications caused by single-event upsets and Rowhammer.

ECC and Chipkill. ECC and Chipkill can only detect a certain number of flips, making them unsuitable as a Rowhammer countermeasure. Chipkill significantly impacts DRAM performance or memory overhead depending on the used protection scheme [19]. All of the schemes described by Dell et al. [19] are additionally not able to correct multiple flips if they come from different chips, which is common for Rowhammer. CSI:Rowhammer can correct errors from different chips, but not from a faulty chip. All three can correct errors from a faulty contact.

The correction time of typical single-bit errors and faulty contact errors is similar between CSI:Rowhammer, ECC, and Chipkill. For systems that currently have ECC or Chipkill deployed, performance does not change with CSI:Rowhammer when these errors happen.

IVEC (Integrity Verification with Error Correction). Huang et al. [27] proposed IVEC, a memory integrity verification method that includes error correction as a search similar to CSI:Rowhammer. IVEC uses an integrity tree and a GMAC with split counters to protect against data modification and replay, relocation, or substitution attacks. The integrity tree requires multiple memory accesses for every integrity verification, which has a significant performance impact. IVEC does not use ECC DRAM to store the MAC.

Synergy. Synergy by Saileshwar et al. [58] is similar to IVEC but uses ECC DRAM to reduce memory accesses. Parity information is stored at another location for error correction but is only retrieved on corruption. Encrypting the data also protects against replay, relocation, or substitution attacks. The overall performance overhead is significantly higher than

CSI:Rowhammer's. Synergy can correct 8 bits in 64-bit data if they come from the same memory chip but only detect them if spread over multiple chips.

MemGuard. With MemGuard, Chen et al. [15] verify the data integrity by computing a write-log hash and a read-log hash with all accesses. The operating system checks the integrity of the whole DRAM periodically by comparing the write-log hash with the read-log hash. If they do not match, the OS resets the memory state to a prior checkpoint causing a delay of seconds. Frequently induced bitflips would halt the system as it could not progress to the next checkpoint.

8. Discussion

In this section, we discuss additional performance improvements, compatibility with existing technologies, and possible race conditions that can occur during the data correction.

8.1. Compatibility with other Technologies

CSI:Rowhammer and TRR TRR cannot provide any security guarantees and was already broken in multiple ways [21, 40]. Most recently, Jattke et al. [33] were able to flip bits in all 40 of their recently-purchased DDR4 DIMMs with TRR. TRR can therefore not be considered even remotely a secure Rowhammer mitigation. When used in combination with CSI:Rowhammer, it provides all secure guarantees, while TRR can provide a performance gain and reduce the risk of DoS by reducing the number of bitflips.

Compatibility with Memory Encryption. If used together with memory encryption, the `csi_load` instruction returns the encrypted data. The correction as a search is then performed on the encrypted data and written back with the `csi_xchg` instruction. If the corrupted data could be reconstructed by, e.g., reading it from disk, the operating system writes the data back with a fourth instruction similarly to `csi_xchg` that encrypts the data in the memory controller.

Backward Compatibility and Upgrade Path. An additional benefit of CSI:Rowhammer is its backward compatibility. It could be controlled by an MSR, with a fall-back to ECC, facilitating hardware upgrades, as

software can follow slower and is not required to support CSI:Rowhammer at all.

Compatibility with Virtualized Environments. CSI:Rowhammer can be used in virtualized environments as described in Sections 4 and 5. Virtual machines do not have to support CSI:Rowhammer. The MSR to activate it is simulated by the host that, therefore, knows which VM wants to use or supports CSI:Rowhammer. For VMs that did not activate CSI:Rowhammer or where security is of uttermost importance, the host corrects all flips as a search. For VMs that did activate it, the host forwards the exception to the VM, allowing the VM's OS to perform all possible operations in the advanced correction, like reloading data from disk. Not requiring every VM to support CSI:Rowhammer greatly eases the upgrade path of virtual environments, because only the host must be upgraded in the first step.

8.2. Possible Improvements

Memory Scrubbing. During phases with low memory bus load, the memory controller performs memory scrubbing by checking the MACs of data. Memory Scrubbing enables the detection and correction of data corruptions prior to the access, increasing the overall system performance. It also prevents the build-up of bitflips that exceed the maximum number of correctable flips, increasing the system reliability.

Speculative Data Forwarding. When accessing data from the DRAM, the memory controller first reads the bytes required by the CPU, and afterward, the missing bytes to fill the cache line. However, the integrity check requires the entire cache line to compute the MAC. The CPU could continue execution after receiving the first word transiently until the integrity is checked. If the integrity check passes, CSI:Rowhammer has no impact on performance. Transiently executing unverified code or data could impact the security of the system. Shi et al. [61] and Lehman et al. [44] already suggested solutions for safe speculative execution.

8.3. CSI:Rowhammer and DRAM Scaling

We believe that because of the continuously increasing susceptibility of DRAM to Rowhammer [35, 52], it is crucial to have a solution like CSI:Rowhammer that translates DRAM error rates to performance without

sacrificing security. DRAM is optimized for the optimal ratio between density and performance. If a solution like CSI:Rowhammer was widely used today, increasing the density too much would degrade the performance substantially, because of very frequent bitflips, before the density becomes so high that the bitflips can be exploited. DRAM cannot be optimized anymore without taking security into consideration.

8.4. Race Conditions during Data Correction

CSI:Rowhammer allows other tasks and multiple corrections to run simultaneously. We discuss the possible scenarios that could arise and how we prevent any race condition.

Read While Correction. If one task is correcting a corruption and another is accessing the same memory location, a lock in the exception handler prevents the concurrent correction of the same flip. In the rare case that both handlers resort to the nested exception handler, concurrent correction cannot be prevented. If the first handler finishes the correction and modifies the data, the `csi_xchg` instruction prevents the second handler from overwriting the new data with old data.

Write While Correction. If a task uses streaming or non-temporal stores, an uncached memory location can be written without loading into the cache. This can overwrite corrupted data in the DRAM that is currently corrected. When the correction is finished, `csi_xchg` detects that the data is already modified and does not overwrite it.

Corruption While Correction. Data may become more corrupted while a correction of this data is running. In that case, the correction finishes but does not write the corrected data back because the `csi_xchg` detects a change. After returning from the exception, another corruption exception is raised. This is a disadvantage but outweighed by the advantages the `csi_xchg` has in the other cases. A corruption can also happen in the advanced correction code while it is de-scheduled. This causes a corruption exception when it is scheduled again. The error is corrected as a search from the secure memory, and the initial correction is continued.

NMI During Corruption Exception. The exception mechanism follows a very similar semantic to the page fault mechanism and can be paused and unscheduled without a problem. The NMI can cause another corruption exception due to flips in the NMI handler. The nesting of

corruption handlers is detected with the nesting bit, and a correction as a search is performed. Frequently recurring NMIs, where the corrupted handler causes a corruption exception, could hang the system, similar to combinations of PFs and NMIs.

9. Conclusion

CSI:Rowhammer is a principled hardware-software co-design against Rowhammer. Our low-latency lightweight-cryptographic MAC brings cryptography-grade integrity protection, detecting any number of bitflips up to the strength of the cryptographic MAC. Our software-level correction routine takes less than 1 second to correct 5 bitflips and adds great flexibility to the correction. We implemented CSI:Rowhammer as a proof-of-concept in gem5 and Linux. We observe a 0.74 % latency overhead and no memory overhead over off-the-shelf ECC-DRAM, showing the practicality of our approach under normal conditions. Even under attack, CSI:Rowhammer maintains a low latency, with the ability to correct a single bitflip in less than 20 ns, compared to up to 63 µs for regular ECC-DRAM as well as errors from faulty DIMM contacts. This shows that CSI:Rowhammer should be deployed in future processors.

10. Acknowledgements

We thank the anonymous reviewers, especially our shepherd, for their guidance, comments and suggestions. We also thank Yoongu Kim, Martin Schwarzl, Stefan Gast and Sarah Rinderer for their valuable input. Part of the funding was provided by a generous gift from Google. 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] AMD. AMD Random Number Generator. 2017 (p. 106).

[2] ARM. ARM Cortex-A9 Technical Reference Manual. 2012 (p. 110).

[3] ARM. Arm® True Random Number Generator (TRNG) Technical Reference Manual. 2020 (p. 106).

[4] ARM Connected blog. Armv8-A architecture: 2016 additions. 2016. URL: https://www.community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions (pp. 100, 104).

[5] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In: INDOCRYPT. 2012 (p. 98).

[6] Roberto Avanzi. The QARMA Block Cipher Family: Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. In: IACR Transactions on Symmetric Cryptology 2017.1 (2017), pp. 4–44 (pp. 98, 100, 104, 105, 120).

[7] Roberto Avanzi, Subhadeep Banik, Andrey Bogdanov, Orr Dunkelman, Senyang Huang, and Francesco Regazzoni. Qameleon v.1.0 - A Submission to the NIST Lightweight Cryptography Standardization Process. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/submissions/qameleon.zip (visited on 10/2021) (p. 114).

[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 (2016) (p. 94).

[9] Leonardo Bautista-Gomez, Ferad Zyulkyarov, Osman Unsal, and Simon McIntosh-Smith. Unprotected Computing: A Large-Scale Study of DRAM Raw Error Rate on a Supercomputer. In: International Conference for High Performance Computing, Networking, Storage and Analysis. 2016 (pp. 97, 124).

[10] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: CHES. 2016 (p. 94).

[11] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis. Princeton University, Jan. 2011 (p. 115).

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. In: ACM SIGARCH computer architecture news (2011) (p. 114).

[13] John Black and Phillip Rogaway. A Block-Cipher Mode of Operation for Parallelizable Message Authentication. In: EUROCRYPT. 2002 (p. 104).

[14] 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 (pp. 94, 125).

[15] Long Chen and Zhao Zhang. MemGuard: A low cost and energy efficient design to support and enhance memory system reliability. In: ISCA. 2014 (pp. 114, 126, 127).

[16] 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. 94, 96, 98, 100, 102, 119).

[17] Ian Cutress. I Ran Off with Intel's Tiger Lake Wafer. Who Wants a Die Shot? Jan. 2020. URL: https://www.anandtech.com/show/15380/i-ran-off-with-intels-tiger-lake-wafer-who-wants-a-die-shot (p. 120).

[18] Johan De Gelas. AMD Rome Second Generation EPYC Review: 2x 64-core Benchmarked. 2019. URL: https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/ (pp. 105, 119).

[19] Timothy J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. Tech. rep. IBM Microelectronics, 1997 (pp. 97, 126).

[20] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (p. 94).

[21] 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. 94, 123, 125, 127).

[22] 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. 94, 125).

[23] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (p. 94).

[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 125).

[25] 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 (pp. 98, 120, 121).

[26] 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. 125).

[27] Ruirui Huang and G. Edward Suh. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. In: ISCA. 2010 (pp. 102, 119, 126).

[28] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In: ASPLOS. 2012 (pp. 97, 124).

[29] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (p. 108).

[30] Intel. Intel® Digital Random Number Generator (DRNG) - Software Implementation Guide. 2012 (p. 106).

[31] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Preventing microarchitectural attacks before distribution. In: CODASPY. 2018 (p. 125).

[32] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (p. 94).

[33] 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. 94, 120, 123, 125, 127).

[34] Jedec Solid State Technology Association. Low Power Double Data Rate 3. 2013. URL: http://www.jedec.org/standards-documents/docs/jesd209-4a (p. 96).

[35] 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. 98, 121, 128).

[36] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In: HPCA. 2015 (p. 126).

[37] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. Frugal ECC: efficient and versatile memory error protection through fine-grained compression. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2015 (p. 126).

[38] Kwangrae Kim, Jeonghyun Woo, Junsu Kim, and Ki-Seok Chung. HammerFilter: Robust Protection and Low Hardware Overhead for RowHammer. In: International Conference on Computer Design (ICCD). 2021 (p. 125).

[39] 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. 94, 97, 113, 120).

[40] 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 Symposium. 2022 (pp. 94, 123, 125, 127).

[41] 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 (pp. 94, 125).

[42] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (pp. 94, 97, 119).

[43] Mark Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. 2016. URL: `http://www.thirdio.com/rowhammer.pdf` (p. 120).

[44] Tamara Silbergleit Lehman, Andrew D. Hilton, and Benjamin C. Lee. PoisonIvy: Safe speculation for secure memory. In: MICRO. 2016 (p. 128).

[45] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. In: IEEE Computer Architecture Letters (2020) (p. 114).

[46] 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. 94).

[47] Moses D. Liskov, Ronald L. Rivest, and David A. Wagner. Tweakable Block Ciphers. In: CRYPTO. 2002 (p. 98).

[48] Eik List and Mridul Nandi. Revisiting Full-PRF-Secure PMAC and Using It for Beyond-Birthday Authenticated Encryption. In: CT-RSA. 2017 (p. 104).

[49] Onur Mutlu and Jeremie S. Kim. RowHammer: A Retrospective. In: IEEE TCAD (2020) (p. 97).

[50] National Institute of Standards and Technology. FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC). 2008. URL: `http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf` (p. 98).

[51] National Institute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. 2018. URL: `https://csrc.nist.gov/projects/lightweight-cryptography` (p. 98).

[52] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications

on Future Attacks and Defenses. In: Proceedings of the Annual International Symposium on Microarchitecture. 2021 (pp. 98, 128).

[53] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. COP: To compress and protect main memory. In: ISCA. 2015 (p. 126).

[54] 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. 120, 125).

[55] PARSEC Group. A Memo on Exploration of SPLASH-2 Input Sets. June 2011. URL: `http://parsec.cs.princeton.edu` (p. 115).

[56] 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 (p. 94).

[57] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In: ASIACRYPT. 2004 (pp. 104, 105, 121).

[58] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. Synergy: Rethinking secure-memory design for error-correcting memories. In: HPCA. 2018 (pp. 114, 119, 126).

[59] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In: Commun. ACM (2011) (pp. 97, 122).

[60] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat Briefings. 2015 (p. 94).

[61] Weidong Shi and Hsien-Hsin S. Lee. Authentication Control Point and Its Implications For Secure Processor Design. In: MICRO. 2006 (p. 128).

[62] Taniya Siddiqua, Athanasios E. Papathanasiou, Arijit Biswas, Sudhanva Gurumurthi, Intel Corp, and Teradata Aster. Analysis and Modeling of Memory Errors from Large-scale Field Data Collection. In: SELSE. 2013 (p. 107).

[63] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In: International Conference on High Performance Computing, Networking, Storage and Analysis. 2012 (pp. 97, 107).

[64] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In: Springer LNCS International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS). 2020 (p. 114).

[65] Andrei Tatar. Hammertime: a software suite for testing, profiling and simulating the Rowhammer DRAM defect. 2018. URL: `https://github.com/vusec/hammertime` (p. 98).

[66] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In: RAID. 2018 (p. 94).

[67] 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. 94).

[68] 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. 94).

[69] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On DRAM Rowhammer and the Physics of Insecurity. In: IEEE Transactions on Electron Devices (2021) (p. 97).

[70] Zhengrong Wang. Bring AVX Support to Gem5. 2020. URL: `https://seanzw.github.io/posts/gem5-avx/` (visited on 12/2020) (p. 114).

[71] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S. M. Jang. A 7nm CMOS platform technology featuring 4^{th} generation FinFET transistors with a 0.027um^2 high density 6-T SRAM cell for mobile SoC applications. In: IEEE International Electron Devices Meeting. 2016 (p. 120).

[72] 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 (p. 94).

[73] A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In: HPCA. 2021 (p. 120).

[74] Doe Hyun Yoon and Mattan Erez. Virtualized ECC: Flexible Reliability in Main Memory. In: IEEE Micro (2011) (p. 126).

[75] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: RAID. 2016 (p. 125).

11. Appendix

11.1. Correction as a Search Pseudocode

Pseudocode of parts of the correction as a search algorithm required to correct the flips in Figure 5.3a. The comments are based on this correction attempt.

```
1  bool correction_as_a_search(data, paddr, MAC, parity)
2  {
3    // maybe the error was transient
4    computed_mac = csi_mac(data, paddr);
5    if (popcount(MAC ^ computed_mac) < 4) {
6      return 1;
7    }
8
9    // we have two mismatching parity bits
10   mmpb =
11     compute_mismatching_parity_bits(data, parity);
12   mmpb_count = count(mmpb);
13   min_flips = mpb_count;
14
15   if (min_flips == 0) {
16     min_flips += 2; // one double flip
17   }
18
19   if (min_flips == 1) {
20     ...
21   }
22
23   if (min_flips == 2) {
24     if (mpb_count == 0) {
25       // one double flip
26       ...
27     }
28
29     if (mpb_count == 2) {
30       // our first guess are two single flips
31       if (perm_two_single_flip(data, paddr, mpb))
32         return 1;
33     }
34
35     // We do not find the correction
36     // Add a double flip to our guess
37     min_flips += 2;
38   }
39
40   if (min_flips == 3) {
```

```
41      ...
42    }
43
44    if (min_flips == 4) {
45      if (mpb_count == 0) {
46        // two double flips
47        ...
48      }
49
50      if (mpb_count == 2) {
51        // two single flips, one double flip
52
53        // try all possible double flip permutations
54        // for all four parity bit blocks
55        for (p = 0; p < 4; p++) {
56          for (i = p * 4; i < (p + 1) * 4; ++i) {
57            flip_bit(data, i);
58
59            for (j = p * 4; j < i; ++j) {
60              flip_bit(data, j);
61
62              // include all single flip permutations
63              // from the first guess
64              if (
65                perm_two_single_flip(data, paddr, mpb))
66                return 1;
67
68              flip_bit(data, j);
69            }
70            flip_bit(data, i);
71          }
72        }
73      }
74
75      if (mpb_count == 4) {
76        // four single flips
77        ...
78      }
79    }
80
81    return 0;
82  }
83
84  bool perm_two_single_flip(data, paddr, mpb) {
85    for (i = mpb[0] * 4; i < (mpb[0] + 1) * 4; ++i)
86    {
87      // flip bit at index i in data
88      flip_bit(data, i);
89
```

```
90      for (j = mpb[1] * 4; j < (mpb[1] + 1) * 4; ++j)
91      {
92        flip_bit(data, j);
93
94        // Check if we found the correction
95        computed_mac = csi_mac(data, paddr);
96        if (popcount(MAC ^ computed_mac) < 4) {
97          return 1;
98        }
99
100       flip_bit(data, j);
101     }
102     flip_bit(data, i);
103   }
104
105   return 0;
106 }
```

Listing 5.3: Correction as a search pseudocode.

11.2. `csi_xchg` Pseudocode

The `csi_xchg` instruction is used to write the corrected data back into the memory without overwriting changed or already corrected data (see Section 4.7).

```
1 int csi_xchg(rx phys_addr, ZMM old_data, rx old_mac,
2               ZMM new_data) {
3
4   ZMM curr_data, rx curr_mac = csi_load(phys_addr);
5
6   if (curr_data == old_data && curr_mac == old_mac) {
7     *phys_addr = new_data;
8     return 1;
9   }
10  return 0;
11 }
```

Listing 5.4: Pseudecode of the `csi_xchg` instruction.

11.3. Inserting Raw Bytes into Code

Listing 5.5 shows how to invoke our instruction set extension without the need for a compiler change by directly placing the byte sequence in the binary.

```
1  asm volatile(
2    "vmovdqu64 %1,%%zmm0                          \n\t"
3    "mov %2,%%rax                                 \n\t"
4    ".byte 0x62, 0xf1, 0xfe, 0x48, 0x6e, 0xc1 \n\t"
5    : "=m"(mac)
6    : "m"(*data), "m"(address)
7    : );
```

Listing 5.5: Calling the custom `csi_mac` instruction. It takes the data and the physical address as an argument and returns the computed MAC.

11.4. Benchmark DRAM Accesses

Figure 5.7 shows the number of DRAM accesses for the gem5 benchmarks of Figure 5.5. We observe a direct correlation between the performance overhead and the number of DRAM accesses due to the added latency of the MAC.

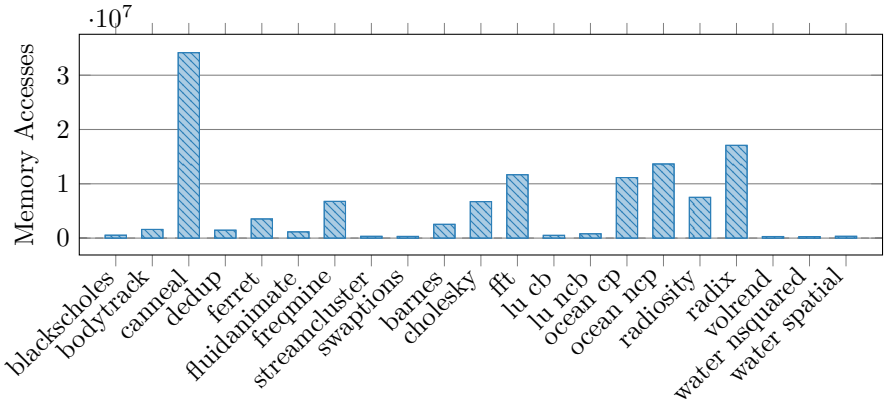


Figure 5.7.: The number of memory accesses that miss the caches. The performance impact is dependent on this number of memory accesses, but also on how well out-of-order execution prevents stalling while waiting for a value or instruction from memory for the different instruction streams of the benchmarks.

11.5. Correction Procedure Flowchart

Figure 5.8 shows the correction procedure from Section 5.4 as a flow chart.

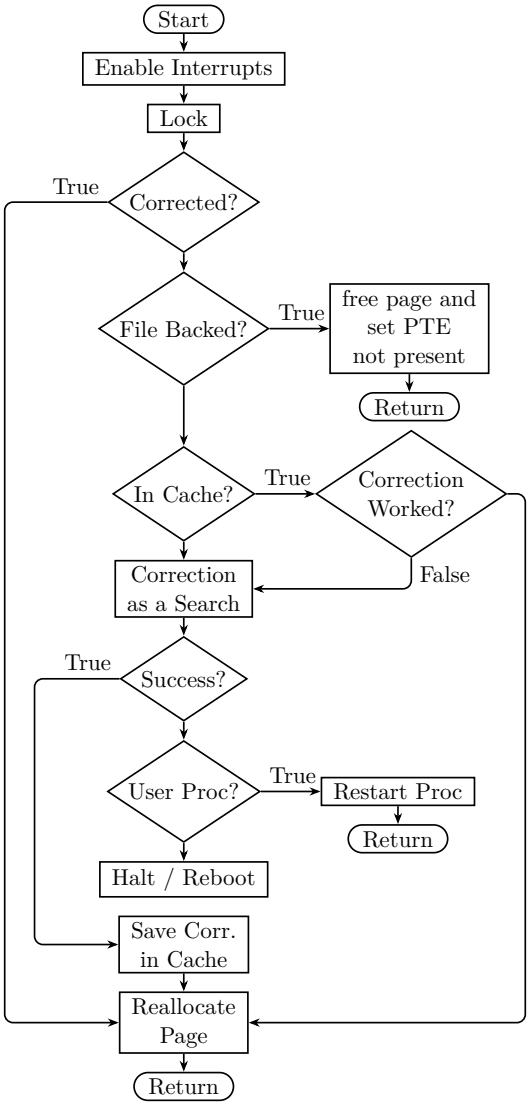


Figure 5.8.: A flow chart depicting the advanced correction procedure that is run if no nesting of the corruption exception handler is detected. It is not in the secure memory because it accesses several other kernel resources.

6

SUIT: Secure Undervolting with Instruction Traps

Publication Data

Jonas Juffinger, Stepan Kalinin, Daniel Gruss, and Frank Mueller
SUIT: Secure Undervolting with Instruction Traps
In: ASPLOS 2024

Contributions

Main author.

SUIT: Secure Undervolting with Instruction Traps

Jonas Juffinger[1], Stepan Kalinin[2],
Daniel Gruss[1], Frank Mueller[2]

[1]Graz University of Technology,
[2]North Carolina State University

Abstract

Modern CPUs dynamically scale voltage and frequency for efficiency. However, too low voltages can result in security-critical errors. Hence, vendors use a generous safety margin to avoid errors at the cost of higher energy overheads.

In this work, we present SUIT, a novel hardware-software co-design to reduce the safety margin substantially without compromising reliability or security. We observe that not all instructions are equally affected by undervolting faults and that most faultable instructions are infrequent in practice. Hence, SUIT addresses infrequent faultable instructions via two separate DVFS curves, a conservative and an efficient one. For frequent faultable instructions, SUIT statically relaxes the critical path in hardware. Consequently, the instruction is not faultable anymore on the efficient DVFS curve at the cost of performance overheads for this specific instruction. For infrequent faultable instructions, SUIT introduces a trap mechanism preventing execution on the efficient curve. With this trap mechanism, SUIT temporarily switches to the conservative DVFS curve and switches back if no faultable instruction was executed within a certain time frame. We evaluate all building blocks of SUIT, using both measurements on real hardware and simulations, showing a performance overhead of 3.79 %, and a CPU efficiency gain of 20.8 % on average on SPEC CPU2017.

1. Introduction

With increasing energy consumption of information technology [29], energy efficiency has become a crucial design goal for modern computers. However, manufacturers use generous voltage guardbands in their CPUs causing a higher energy consumption than necessary. To counteract this, CPUs optimize their efficiency with Dynamic Voltage and Frequency Scaling (DVFS) [23, 28, 59]. DVFS allows the CPU to switch between power states with different power consumption and performance. While vendors define different voltage-frequency pairs for each power state, finding an optimal DVFS curve is non-trivial. A voltage slightly too low already makes the CPU unreliable, as a small set of instructions will produce faulty results [8, 31, 47, 56, 60]. At this point the system is vulnerable and unfit for use in production. The susceptibility to faults depends on various factors during manufacturing and operation of a chip [42, 50].

Still, reducing these voltage guardbands, i.e., undervolting a CPU, is a highly researched topic because it can greatly reduce the power consumption of CPUs [6, 7, 16, 30, 35, 36, 43, 51, 52]. While this is most relevant in mobile contexts where energy availability is limited, desktop computers and servers also benefit from lower costs for cooling and energy. Although it may seem counter-intuitive, undervolting not only increases the efficiency but it can also increase the performance of CPUs. Modern CPUs are dynamically throttled to stay within the specified thermal and power limits [17, 54, 63]. With lower energy consumption, the CPU can sustain higher clock frequencies longer while staying below these limits, increasing performance.

However, reducing these guardbands also introduces severe security issues as shown by prior work [8, 31, 47, 56, 60]. These software-based fault attacks undermine all security guarantees of a CPU, even breaking trusted execution environments, which motivates the basic questions of our work:

Can system-level mechanisms guarantee security and reliability against undervolting-induced faults? How much energy do such mechanisms cost? Can these mechanisms be lightweight enough to enable substantial energy savings?

In this work, we present **SUIT**, a novel hardware-software co-design enabling a substantial increase in CPU efficiency without compromising reliability or security. SUIT is the first work on efficiency improvement by undervolting that builds on the observation that not all instructions

are equally susceptible to undervolting faults [32, 47]. In contrast to prior work [6, 7, 16, 30, 35, 36, 43, 51, 52], not explicitly protecting the aging and temperature guardband when undervolting, SUIT keeps these essential voltage guardbands by using this difference in the voltages required by different instructions.

Besides the single DVFS curve current CPUs already have, SUIT adds a second, more efficient DVFS curve that the vendor determines by *excluding* a set of instructions faulting first when the voltage is lowered. These instructions are then disabled while SUIT uses the efficient DVFS curve. We introduce a trap mechanism to handle the execution of disabled instructions, similar to existing traps, e.g., for invalid opcodes. When the instruction stream runs into a disabled instruction, SUIT transfers control to the operating system that can either emulate the instruction in software or temporarily switch to the conservative DVFS curve where the instruction can be executed with a sufficiently high voltage. SUIT uses a deadline mechanism to determine when to switch back to the efficient DVFS curve: If no disabled instruction is executed until the deadline is reached, SUIT switches back to the efficient DVFS curve. The second DVFS curve, disabling of specific instructions, trap mechanism and deadline mechanism require hardware changes of the CPU.

Our analysis, based on execution traces of applications on x86 systems, shows that faultable instructions typically occur infrequently. On average over all SPEC CPU2017 benchmarks, one such faultable instruction is encountered every 5 billion instructions. SUIT uses dynamic DVFS curve switching or emulation for these kinds of instructions. Other faultable instructions, such as the integer multiply instruction (`IMUL`) occurring as frequently as every 560 instructions, are the exception. SUIT relaxes their critical path in hardware to lower their voltage requirement. Thus, SUIT can handle *any* frequency of faultable instructions.

The first building block of SUIT addresses the *infrequently faultable instructions*. If a disabled instruction is executed, SUIT can either dynamically switch DVFS curves or emulate the instruction. Which method is more efficient depends on the delays of DVFS curve adjustments, the number of independent DVFS domains and the distribution and frequency of disabled instructions in the workload. Due to SUIT's software component, it can dynamically switch between DVFS curve adjustments and emulation. We also evaluate the impact of SIMD instructions on performance and efficiency. With SUIT, compiling applications without SIMD instructions can increase efficiency in about half of the tested workloads.

The second central building block of SUIT addresses *frequent faultable instructions*. As frequent exceptions would hurt system performance and efficiency, we instead relax their critical path in the hardware design. This results in a small performance overhead but has the benefit that the instruction is stable at lower voltages. We evaluate different faultable instructions and find our second technique to be only relevant for the x86 integer multiplication instructions `IMUL` and `MUL`, for short `IMUL`. The performance overhead of a SUIT-adjusted, one clock cycle longer, x86 `IMUL` instruction in the SPEC CPU2017 benchmarks is only 0.03 % on average with a maximum of 1.60 % for the 525.x264 benchmark. While the concept of prolonging the critical path is investigated for `IMUL` in this work, the principle can be applied to another opcode if it were to be a frequent faultable instruction on a given architecture. This provides a path forward for SUIT, even though we did not observe any such case beyond `IMUL`.

We evaluate SUIT with measurements on different CPUs from AMD and Intel. The short delay to change the core frequency, the per-core voltage domains and the positive performance impact of undervolting make Intel Xeon CPUs preferred for SUIT. SUIT uses an optimized strategy to change frequency and voltage to increase the efficiency as much as possible while causing only a negligible performance degradation in some cases. With this strategy, SUIT increases the energy efficiency by over 12 % on average over all SPEC CPU2017 benchmarks on Intel CPUs. When compiling programs without faultable SIMD instructions they can always run on the efficient DVFS curve but have the overhead of not using SIMD. With this, SUIT increases the efficiency by 19 % but requires the recompilation of applications. In our security analysis, we show that SUIT prevents the execution of instructions in the faultable set on the efficient DVFS curve. Based on this design, we use a reductionist argument to show that SUIT's security is equivalent to that of systems today. Hence, SUIT is a viable approach for substantial energy savings in a secure manner.

Contributions. Our contributions are as follows:

- We present SUIT, a novel software-hardware co-design enabling substantial energy savings through tailored DVFS adjustments while maintaining security.
- We analyze the undervolting potential due to faultable instructions and analyze their frequency in real-world software, revealing that they are often used in short bursts, e.g., for encryption, which is ideal for SUIT.

- We measure DVFS adjustment delays on real hardware showing large differences in CPUs of different vendors.
- By evaluating SUIT, we observe no performance loss and a reduction in power consumption by 14 %, resulting in an energy efficiency gain of up to 20 %.
- We show that the security of SUIT is equivalent to the security of today's CPUs in a reductionist argument.

Outline. In Section 2, we provide background. In Section 3, we present the design of SUIT. In Section 4, we detail the hardware and software changes for SUIT. In Section 5, we determine parameters for SUIT on real hardware-software setups. In Section 6, we evaluate SUIT's performance, efficiency and security. We discuss related work in Section 7, our results in Section 8 and conclude in Section 9.

2. Background

We provide background on CMOS circuits, voltage guardbands, voltage-related faults and DVFS.

2.1. Power Consumption of CMOS Circuits

The dynamic power consumption of a CMOS circuit, $P_{dyn} = C_L \cdot (V_{DD})^2 \cdot f_{CLK}$, depends on the load capacitance C_L, supply voltage V_{DD} and clock frequency f_{CLK} [44]. CMOS circuits mainly consume power when switching [44], as on every switch a CMOS gate must charge or discharge its C_L. The switching energy depends on the supply voltage V_{DD} squared [44]. This relation is also the basis for many power consumption side-channel attacks [33, 39, 41, 57, 63].

Supply voltage V_{DD} and clock frequency f_{CLK} also depend on each other [17]. In a CMOS circuit many transistors implement combinatorial logic with a propagation delay t_P, depending on V_{DD} [16, 49]. The block with the longest t_P is the *critical path* of the circuit and determines the maximum clock frequency $f_{CLK_m} = t_{crit}(V_{DD})^{-1}$. If the clock frequency is too high, data arrives late, manifesting in malfunctioning of the circuit, e.g., erroneous computed values [16, 49].

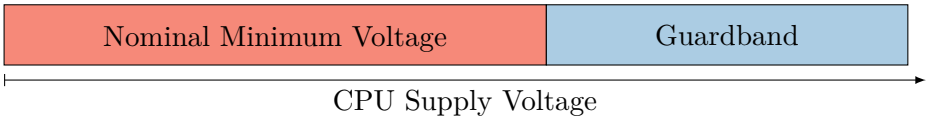


Figure 6.1.: Typical supply voltage of a CPU. It must be higher than the nominal minimum voltage to account for all uncertainties of the CMOS circuit, aging, temperature and its surroundings.

2.2. Voltage Guardbands

The dependency between voltage and frequency in CMOS circuits is influenced by many factors. Process variation influences every single transistor's performance and voltage requirement. Over time, voltage requirements increase due to high temperature, age, and voltage spikes delivered by the power supply unit [24]. To counter these effects, manufacturers use a voltage guardband. The circuit is supplied with a higher voltage than the nominal minimal voltage to take these uncertainties into account as shown in Figure 6.1.

Aging effects like bias temperature instability and hot-carrier injection cause the threshold voltage and consequently the propagation delay to increase over time [58]. The degree of aging must be predicted accurately to design a guardband. If it is too low the device can break within its expected lifetime, if it is too large more power than necessary is used.

The propagation delay of modern sub 20 nm FinFET transistors degrades by approximately 15 % over a time span of 10 years at ¿100 °C [19, 46, 53, 61, 66, 67]. After ten years, the CPU must either run with a 13 % lower frequency at the same supply voltage or with a supply voltage that supports a 15 % higher frequency at age zero. Based on these numbers and real hardware measurements, we evaluate the aging and temperate guardband in Section 5.6 and 5.7. On current CPUs the aging guardband is approximately 12 % and the temperature guardband 3.5 % of the CPU supply voltage.

2.3. Variation in Voltage Requirements

Modern CPUs are highly complex systems where vendors have multiple dials to fine-tune for the efficiency or performance optimum. In particular, they can increase the number of cycles allocated to an operation or adjust

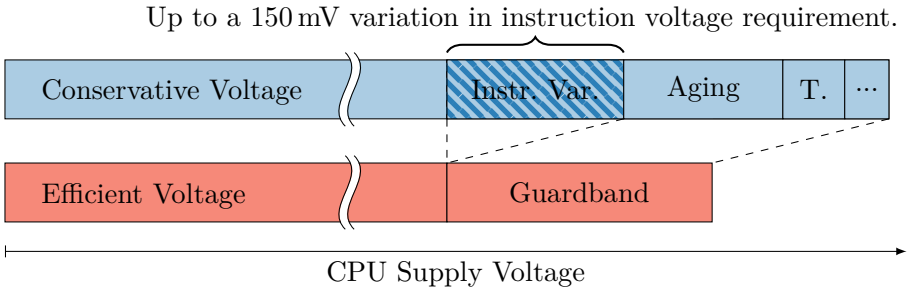


Figure 6.2.: SUIT does not eliminate the aging or temperature (T) guardband. Instead, it utilizes the variation in voltage requirements of different instructions, especially `IMUL` and SIMD instructions.

Table 6.1.: Undervolting-induced instruction faults observed by Kogler et al. [32]. If an instruction faults on a specific CPU core at a specific frequency and voltage offset it counts as one fault for the number of faults. The rarely faulting instructions (right) occur on average at lower voltages than the more frequently faulting ones.

Instruction	IMUL	VOR*	AESENC	VXOR*	VANDN*	VAND*	VSQRTPD	VPCLMULQDQ	VPSRAD	VPCMP*	VPMAX*	VPADDQ
Number of Faults	79	47	40	40	30	28	24	16	9	5	3	1

the clock frequency to account for the propagation delay within a specific part of the circuit [49]. Instructions with a long propagation delay t_P can be split up into pipeline stages, dividing t_P by the number of stages and enabling higher clock frequencies [49]. This increases the instruction's latency but keeps the throughput unchanged. An example for this is the `IMUL` instruction. On Intel and AMD CPUs, `IMUL` takes 3 clock cycles with a throughput of 1 instruction per clock cycle [15].

Even with very good optimizations some instructions can still have a significantly longer t_P than most other combinatorial acyclic logic blocks inside the CPU. This was first discovered by Murdoch et al. [47] for `IMUL` on Intel CPUs. In their experiments they measured that `IMUL` starts to produce faulty results very infrequently on one of their tested CPUs at a -100 mV undervolt level. However, apart from `IMUL` (they did not test SIMD instructions), the CPU was running stable down to a -250 mV level, resulting in a variation in voltage requirements of 150 mV as shown in Figure 6.2.

Kogler et al. [32] performed a more detailed study on this phenomenon by building a framework that automatically tests instructions for their undervolting behavior. They confirmed that across several CPUs data errors occur earlier than control-logic errors, indicating that the critical paths in modern CPUs are more often on the data paths within instructions. The highest variation in instruction voltage they measured was more than 60 mV. Following `IMUL` in sensitivity to voltage levels are `VOR` and other SIMD instructions, as well as `AESENC`, as shown in Table 6.1.

2.4. Dynamic Voltage and Frequency Scaling (DVFS)

Based on the workload, the supply voltage and clock frequency of modern CPUs is adjusted to trade off performance and power consumption. For DVFS, the vendor defines p-states, pairs of clock frequencies and voltages including a guardband that, in combination, form a *DVFS curve*. Although the p-state is hardware-controlled, the OS can still intervene in the frequency voltage pairs on some Intel CPUs with the undocumented MSR `0x150` [45].

3. Design Overview and Implementation

In this section, we provide an overview of SUIT's design (see Figure 6.3) showing, on a high level, how it enables efficiency gains without affecting the system's security. With its generic design, SUIT can be integrated into different CPU architectures. For our proof-of-concept implementation and evaluation, we focus on the changes for x86-64.

SUIT provides software interfaces to disable instructions and to switch between DVFS curves. The OS disables all instructions that may fault on the efficient DVFS curve. Hence, the OS can switch to the efficient DVFS curve without affecting the system's security with faultable instructions.

3.1. SUIT's Undervolting Potential

SUIT's undervolting potential comes from two sources. First, the variation in the required voltage by different instructions, i.e., faultable instructions. Deactivating instructions that produce faulty results undetectably while the CPU continues running eliminates a security risk exploited by various

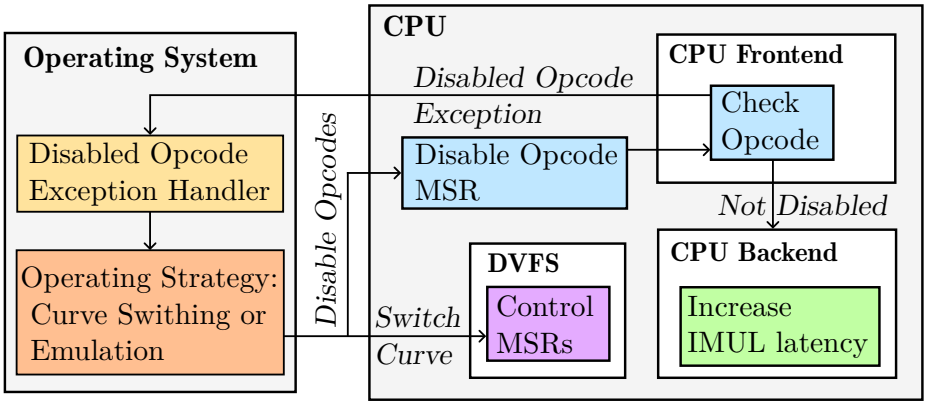


Figure 6.3.: The overall SUIT design changes several parts of the system (colored boxes). We disable certain instructions while switching to an efficient DVFS curve and increase the latency of others. Executing disabled instructions raises a `#DO` exception, handled by the OS through emulation or switching of the DVFS curve.

attacks [8, 31, 47, 56, 60]. Second, due to the elimination of this vulnerability, a small fraction of the aging guardband can be used for undervolting limited to a certain amount of time.

The studies of Murdoch et al. [47] and Kogler et al. [32] show large differences in instruction voltage requirements between CPUs and even CPU cores due to process variation. The average over all CPUs covered in these two studies that exhibit this variation in instruction voltage requirements is 70 mV ($n = 6$, $\sigma = 44$ mV), the maximum is 150 mV. CPUs from Intel 6th generation did not exhibit this variation [32].

Data centers and cloud providers, representing a growing share of the global electric power consumption [34], procure new CPUs after a few years. For example, AWS recently increased the life span of their servers from four to five years [14]. Xeon CPUs are typically supported for less than 10 years [26]. Additionally the aging degradation is larger at higher temperatures [40, 65]. During the limited life span and with well controlled core temperatures the full aging guardband designed for the worst case is not required. Also typical consumer devices are not running continuously although the guardband is designed for nonstop use. Exploiting some fraction of the aging guardband with SUIT in the first few years can have a large impact on data center and consumer device power consumption without impact on reliability.

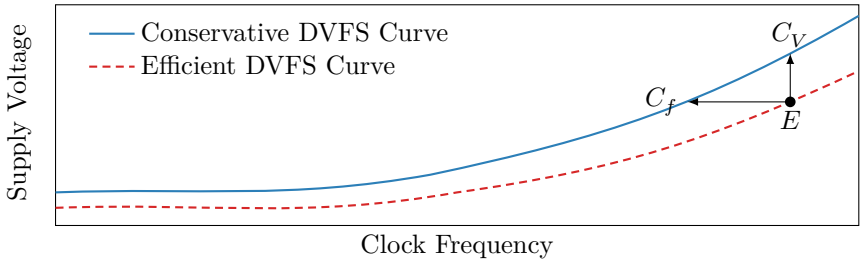


Figure 6.4.: Switching from a p-state on the efficient DVFS curve to the conservative DVFS curve can be done in two ways: by changing the voltage (C_V) or the frequency (C_f).

In Section 6, we evaluate a conservative undervolting margin of −70 mV highlighting the differences in instruction voltage requirements and with an additional 20 % of the on average 137 mV aging guardband for a combined −97 mV offset. Thus, in contrast to previous work [7, 16, 30, 35, 36, 43, 51, 52], SUIT uses either none or only a small fraction of the aging voltage guardband for undervolting.

3.2. Two DVFS Curves

A CPU with SUIT has a conservative and an efficient DVFS curve. The conservative curve is the same as on current CPUs. The efficient curve is determined by excluding the small number of instructions that fault in the "instruction voltage variation" range of the voltage supply (see Figure 6.2).

We introduce a new MSR that allows the operating system to select which DVFS curve to use. The CPU ensures that the efficient curve can only be used if the faultable instructions are disabled. Switching from the efficient to the conservative DVFS curve can be done in two ways, as seen in Figure 6.4. Section 4.3 details the different effects of the switching method on the efficiency and performance of SUIT.

3.3. Disabling Instructions

We propose a new model-specific register (`MSR`) that allows the operating system to disable all faultable instructions defined by the manufacturer per voltage or frequency domain. We use a reserved interrupt vector number [27] for a new *Disabled Opcode* (`#DO`) CPU exception. When the

CPU encounters a disabled instruction it raises a `#DO` exception. Like other CPU exceptions, `#DO` preserves the current register set so that the program can continue after the exception is handled. The OS handles this new exception by following the approaches presented in Section 4.

3.4. Instruction Emulation

Additionally to switching the DVFS curve to execute faultable instructions, a system with SUIT can also emulate instructions. SUIT emulates instructions like `VOR` or `VPCMP` with non-vectorized alternatives, and `AESENC` with a side-channel-resilient bit-sliced AES implementation. To perform this emulation, the operating system maps emulation code into the memory of the user space program. The exception handler returns to this emulation code, which is then run in user space. After the emulation, the program returns to the kernel through, e.g., a system call, where the original register contents are restored and the program continues execution after the disabled instruction. The two transitions into the kernel and back dominate the overhead.

3.5. Security

SUIT's security is established exactly as for current CPUs, namely by the vendor determining safe DVFS curves. Today, when an instruction produces erroneous results during this process, the vendor either accepts this faultable instruction as a critical path for the CPU, or increases the instruction latency to reach a slightly more efficient DVFS curve. SUIT provides a third option. Instead of increasing the latency, with SUIT, vendors determine a second DVFS curve, which excludes the small number of faultable instructions, thereby resulting in a considerably more efficient DVFS curve. As this follows the same process as before, just with fewer instructions, we obtain the exact same security level for this slightly reduced instruction set. The details of the security evaluation are given in Section 6.9.

4. SUIT Hardware-Software Interaction

SUIT uses two building blocks for low-frequency and high-frequency faultable instructions, which are generically applicable to any CPU and

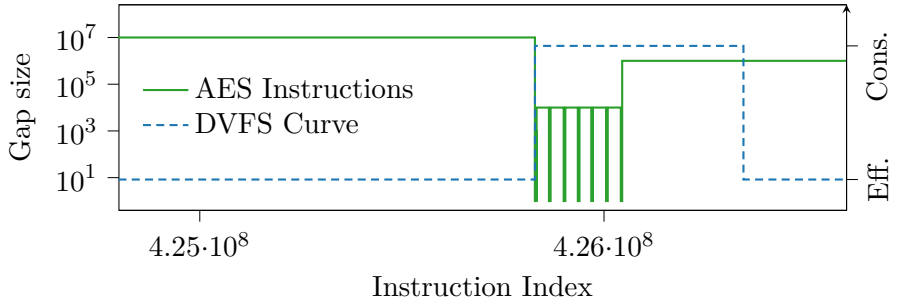


Figure 6.5.: Detailed view of an AES instruction burst and the resulting change of the DVFS curve to conservative and back. Horizontal segments correspond to the time without AES instructions executed, with the height of the segment showing the gap size, the log_{10} of the segment's length. Vertical segments show instruction bursts.

microarchitecture. While both introduce overheads, we show in Section 6.3 that, overall, SUIT not only compensates these overheads but yields a significant increase in system efficiency. We assume that the vendor has determined the two DVFS curves with and without the faultable instructions during manufacturing, one of which being the DVFS curve the vendor determines already today.

4.1. Low-Frequency Faulting Instructions

SUIT's central idea is that the CPU generally runs on the efficient DVFS curve. SUIT handles the corner case of disabled instructions by triggering an OS handler that temporarily switches to the conservative and less efficient curve, which is secure for all instructions as shown in Figure 6.5. When continuing on the conservative DVFS curve after the switch, the CPU re-executes the instruction that was formerly disabled.

Our analysis shows that programs often use faultable instructions in short bursts. Based on this observation, SUIT uses a deadline mechanism to determine when to switch back to the efficient DVFS curve. Initialized with the deadline it counts down with constant speed. At zero, an interrupt is triggered to switch back to the efficient DVFS curve. Whenever the CPU executes an instruction that would be disabled on the efficient DVFS curve, the timer is reset and starts counting down again. Therefore, SUIT automatically adjusts to many frequencies of faultable instructions and

avoids thrashing-like effects in many cases, where SUIT would frequently switch between the DVFS curves.

Instruction Emulation. For single instructions, emulation is faster than switching DVFS curves (see Section 6.6). For 65 % of our tested applications emulation is beneficial.

Overhead. The dynamic building block of SUIT incurs overheads from two sources. The first is due to the `#DO` exception and corresponding OS handler. We measured this delay to be 0.34 µs in Section 5.2. The second overhead is due to the DVFS curve switching delay or instruction emulation. SUIT only has to delay execution when switching from the efficient to the conservative curve; in the other direction, it disables the instructions and does not need to wait until the efficient curve is reached. We measured this delay on current Intel and AMD CPUs in Section 5.2. On an Intel Xeon Silver 4208, it takes on average 31 µs to change the core frequency and 335 µs to change the core voltage. On an AMD Ryzen 7 7700X it takes on average 668 µs to change the frequency.

4.2. High-Frequency Faulting Instructions

The latency of an instruction is a direct result of the propagation delay of the instruction's critical path. If the critical path of an instruction is larger than the inverse of the required clock frequency, the instruction's critical path has to be split up into multiple stages, forming a pipeline for the instruction. The number of stages corresponds to the latency of the instruction. By increasing the number of pipeline stages and, therefore latency, the shorter critical path of the individual stages allow for lower voltages or higher frequencies [49].

The `IMUL` instruction family, which multiplies two integers, is quite commonly used and considerably more complex than most other ALU instructions. It is typically split up into 3 stages on various modern CPUs [3, 4, 15]. These 3 clock cycles are still very tight, limiting the vendor in further optimizing the DVFS curve due to the critical path within the `IMUL` instruction. Kogler et al. [32] practically confirmed this by undervolting and finding that the `IMUL` instruction faulted first in 91.2 % of cases. Our analysis of the instruction usage of applications in Section 5.1 shows that `IMUL` is the only instruction used so frequently that SUIT would permanently run on the conservative DVFS curve, preventing any

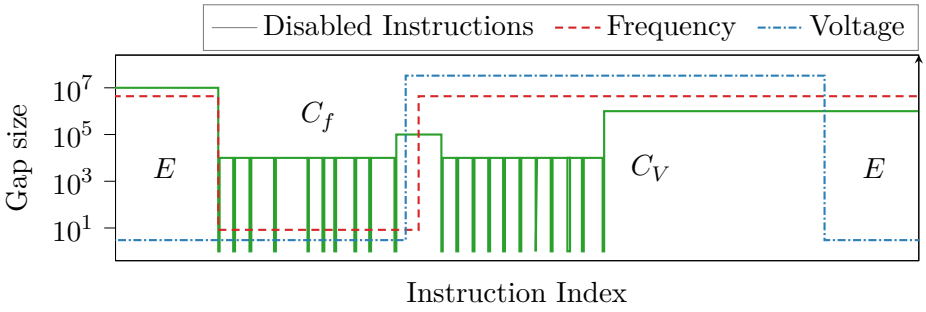


Figure 6.6.: Detailed view of a long burst of faultable instructions and how frequency and voltage change with the fV operating strategy.

potential efficiency gain. Therefore, SUIT has a second building block for frequent instructions, e.g., `IMUL`.

The central idea is to address this problem by statically removing frequent instructions from the set of faultable instructions at the cost of performance. We analyze the sensitivity of performance to latency increases of `IMUL` by adjusting its implementation in a microarchitecture simulator. It is important to note that `IMUL` is fully pipelined. While the latency is 3 cycles, already after the first cycle, another input can be pushed into the `IMUL` pipeline. Hence, changing the latency has no effect on the throughput. Overall, increasing the latency of `IMUL` by 1 clock cycle causes a performance overhead of 0.03 % ($n = 16$, $\sigma_{\bar{x}} = 0.15$) on average and 1.60 % ($n = 9$, $\sigma_{\bar{x}} = 0.55$) in the worst case (see Section 6.1).

Static Latency Increase is not Suitable for All Instructions. While it seems tempting to just increase the latency of all faultable instructions, this decreases performances and removes the flexibility of SUIT. As our evaluation shows, SUIT increases the efficiency on average and for many workloads significantly. Additionally, it does not decrease the performance of highly specialized workloads that use many faultable instructions (e.g., 520.omnetpp, Section 6.3). For these specialized workloads, SUIT will stay on the conservative DVFS curve, as faultable instructions occur frequently. Consequently, for specialized workloads, SUIT maintains a high performance while increasing the efficiency for others.

4.3. Operating Strategy

The operating strategy describes how the operating system controls the SUIT hardware. There are four main ways based on the two DVFS curve switching methods, see Figure 6.4.

Emulation. The DVFS curve is not switched but the instruction is emulated in the `#DO` exception handler. Therefore, every disabled instruction incurs the overhead of the exception and the emulation. Emulation is not possible for applications running in trusted execution environments.

Frequency $E \leftrightarrow C_f$. Switching the DVFS curve by changing the frequency enables fast switching and is highly efficient because the CPU always stays at the lower voltage. However, the performance is lowered when running on C_f.

Voltage $E \leftrightarrow C_V$. The DVFS curve switch by voltage is approximately a magnitude slower than by frequency, resulting in a high performance impact. Nonetheless, performance is high when running on the conservative DVFS curve.

Combination (fV) $E \leftrightarrow C_f \rightarrow C_V \rightarrow E$. Changing the frequency *and* voltage can lead to an optimal balance between efficiency and performance. On a `#DO` exception, a quick switch to the conservative curve is realized by changing the frequency. At the same time, a voltage increase is requested. The program continues running at C_f. If the burst of potentially faulting instructions is short, the operating strategy returns to E and cancels to voltage change. If the burst is long enough for the voltage to change, the CPU runs at C_V with full performance. This incurs another frequency change stall delay. The whole sequence is depicted in Figure 6.6.

Trashing Prevention. If the gap between disabled instructions is bit longer than the deadline, the CPU constantly switches DVFS curves adding considerable overhead. The OS detects this by counting the number of `#DO` exceptions during a specified time span and increases the deadline. This ensures that the CPU stays in the conservative DVFS curve.

Parameters. The fV operating strategy and trashing prevention are optimized with four parameters: (1) The deadline (`p_dl`) denoting the maximum time between two potentially faulting instruction before switching back to the efficient curve; (2) the time span (`p_ts`) over which trashing prevention looks back and counts `#DO` exceptions; (3) the maximum `#DO` exceptions count (`p_ec`) in `p_ts`; and, if trashing is detected, (4) the

deadline factor (`p_df`) by which the deadline is multiplied for this stable period.

A pseudo code implementing the *fV* operating strategy and trashing prevention for the evaluation simulation that use the parameters is shown in Listing 6.1.

5. Evaluation of Building Blocks on Real Hardware and Software

For each building block of SUIT, we perform measurements of real software and hardware to obtain realistic data for the evaluation. In Section 5.1, we analyze applications for their usage of faultable instructions. In Section 5.2, we measure the delays incurred by exceptions and changing the core voltage and frequency. To determine the effects of SUIT on efficiency, we measure how CPUs behave when running them undervolted in Section 5.4. In Section 5.5, we measure the DVFS curve of a contemporary CPU for our security evaluation. In Section 5.6, we measure the aging guardband and in Section 5.7 the temperature guardband. In Section 5.8, we measure the impact of disabling SIMD instructions on SPEC CPU2017. Subsequently, in Section 6, we use these results to evaluate the entire SUIT system.

5.1. Frequency of Disabled Instructions in Real Code

We analyze 25 applications for their usage of faultable instructions. The data collection was done using QEMU [9].

QEMU Plugin. We implement a QEMU plugin allowing us to log in detail when specific instructions are executed. We use Linux's `isolcpus` parameter to isolate one core for our application in QEMU, and use canary instructions to make sure QEMU doesn't record instructions executed outside of the workload being tested. The plugin records the trace based on instruction count. Due to modern CPUs being superscalar and instructions taking a variable number of clock cycles, we use the `INSTRUCTIONS_RETIRED` performance counter to estimate the number of instructions executed per clock cycle. We use it to convert between instruction count and CPU clock cycles in the evaluation simulation to obtain correct timings, e.g., for the deadline mechanism.

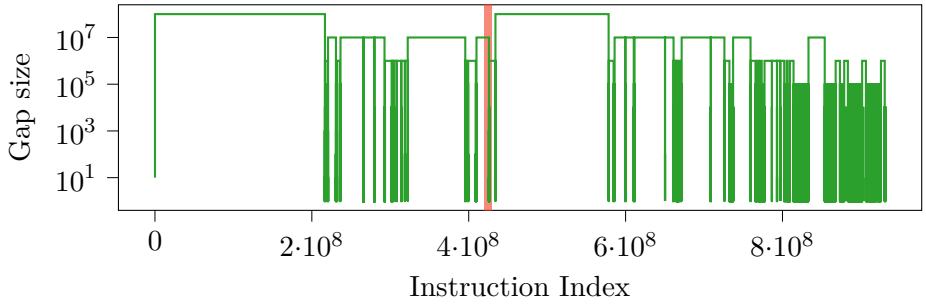


Figure 6.7.: Timeline of AES instruction execution while VLC is streaming a 1080p video to visualize how these instructions are often executed in bursts. The timeline is truncated to the first 0.5 %. For the evaluation we take all faultable instructions into account.

Instruction patterns. Using QEMU, we collect the data on AES and SIMD instructions from Table 6.1 of a client network application (nginx and VLC streaming), as well as all benchmarks in the SPEC CPU2017 benchmark suite.

The timeline shown in Figure 6.7 allows us to understand the distribution of AES instructions during the execution of VLC. More specifically, we can observe that AES instructions occur in bursts with gaps between them. We observed similar patterns with SIMD instructions in SPEC CPU2017 benchmarks, with larger gaps between faultable instructions accounting for larger parts of the overall execution time.

5.2. Core Voltage and Frequency Change Delays

We microbenchmark the overhead for each building block of SUIT on real systems for the simulation of SUIT in Section 6.

Voltage Change Delay. Figure 6.8 shows the delay to change an Intel Core i9-9900K's core voltage over 20 repetitions. We use MSR `0x150` to set voltage offsets and `MSR_IA32_PERF_STATUS` to read the current CPU core voltage. In a kernel module, we first decrease the voltage with a negative offset and wait for it to manifest. Afterward, we raise the voltage to its original level (time 0 in the figure) and measure the time it takes for the voltage to stabilize by polling `MSR_IA32_PERF_STATUS`. As shown in Figure 6.8, it takes 350 µs ($n = 20$, $\sigma = 22$) on average, with a maximum of 379 µs.

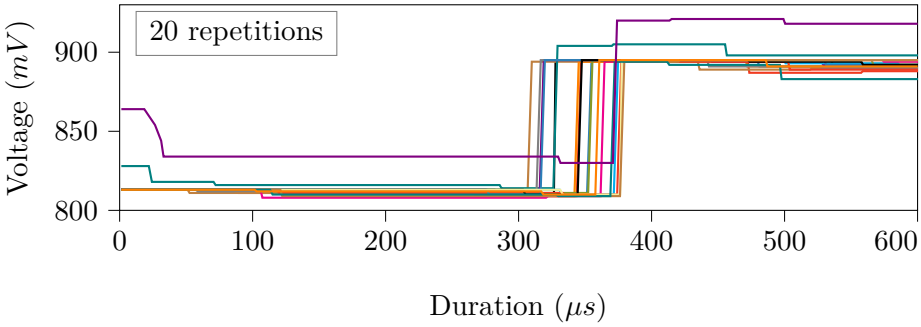


Figure 6.8.: The core voltage of an Intel Core i9-9900K after resetting the negative voltage offset to 0 mV at time 0. Faultable instructions must not be executed until the core voltage is actually increased.

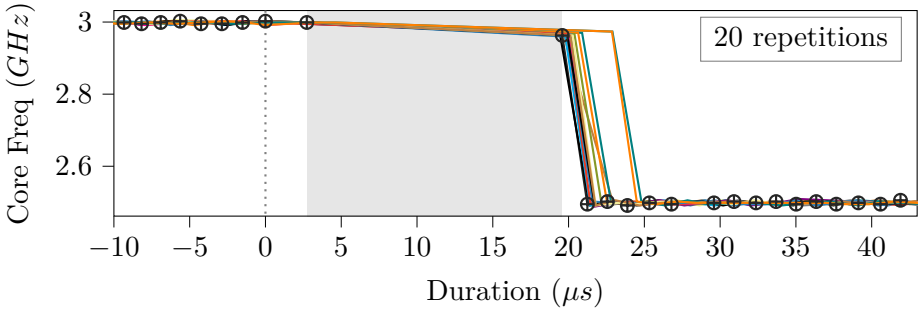


Figure 6.9.: There is a delay between requesting a frequency change and the actual change on Intel CPUs (here an i9-9900K). Measurement samples (⊕) are shown relative to time 0 (the dotted line) where the new frequency was set. The gray area marks the duration during which the CPU stalls and no samples were measured.

Frequency Change Delay. Figure 6.9 shows the delay on an Intel Core i9-9900k to change the clock frequency over 20 repetitions. We measure with a kernel module and the `APERF` and `MPERF` counters to get the current CPU frequency [28].

On the i9-9900K, we write `MSR_IA32_PERF_CTL` to change the frequency, which takes on average 22 µs ($n = 20$, $\sigma = 0.21$) with a maximum of 24.8 µs. We also confirm that there is one frequency domain. After setting the frequency, all cores of the i9-9900K stall illustrated as a gray area in Figure 6.9, where no samples were taken. The first sample after the stall still shows a high frequency, while it is actually already low, due to the `APERF` frequency being updated late during the stall. We verified this

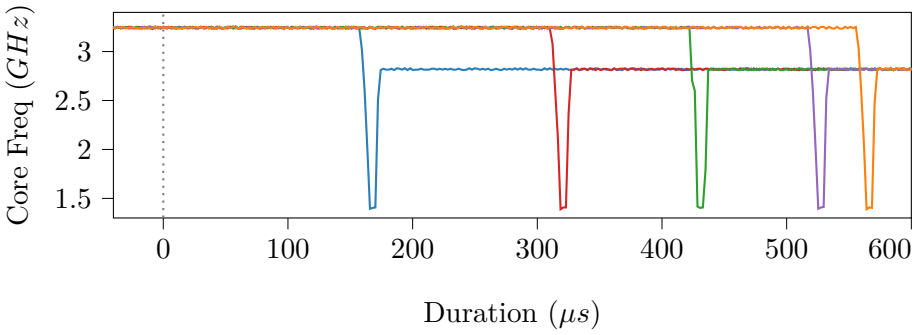


Figure 6.10.: The delay between requesting a frequency change and the change of the frequency on an AMD Ryzen 7 7700X with per-core frequency domains. The CPU core does not stall.

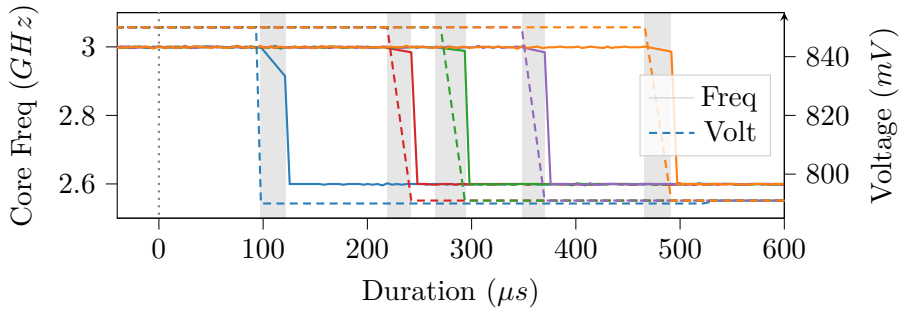


Figure 6.11.: The delay between requesting a frequency change and the change of the voltage and frequency on an Intel Xeon Silver 4208 with per-core frequency and voltage domains.

by reading the p-state value after the stall. The behavior is equal when increasing the frequency.

Figure 6.10 shows 5 frequency changes on the AMD Ryzen 7 7700X using `cpufreq_set`. The frequency change takes 668 µs ($n = 10$, $\sigma = 292$) on average. The core does not stall.

Per-Core Voltage and Frequency Change Delay. Figure 6.11 shows 5 p-state changes on an Intel Xeon Silver 4208 when changing the frequency with `MSR_IA32_PERF_CTL`. Intel Xeon CPUs since Haswell-EP have per-core voltage *and* frequency domains (PCPS) [21]. But they are coupled and always move in tandem. When changing the p-state the CPU core always first changes the voltage and then the frequency, regardless of the

direction of the change. The frequency change looks similar to the one of the i9-9900K, suggesting that the same clock source is used. The voltage change takes on average 335 µs ($n = 98$, $\sigma_{\bar{x}} = 135$) and the following frequency change 31 µs ($n = 98$, $\sigma_{\bar{x}} = 2.3$) during which the CPU core stalls for 27 µs ($n = 98$, $\sigma_{\bar{x}} = 2.5$).

5.3. Exception and Emulation Call Delay

We measure the end-to-end delay from a CPU exception to the invocation of the corresponding handler in the Linux kernel and the return back to user space.

Exception Delay. We use the *invalid opcode* exception, which closely resembles our *disabled opcode* exception. In user space, we store the current Time Stamp Counter (TSC) value in a register right before executing `UD2` to generate an invalid opcode. The exception is handled by our modified Linux kernel, which stores the current TSC value at the beginning of the exception handler. The difference of the two TSC values is the delay for entering the exception handler. This delay is 0.34 µs ($n = 10$, $\sigma_{\bar{x}} = 0.04$) on an Intel i9-9900K and 0.11 µs ($n = 10$, $\sigma_{\bar{x}} = 0.02$) on an AMD 7700X.

Emulation Call Delay. To emulate instructions in user space the kernel is entered twice: First on the *invalid opcode* exception from where it returns to the emulation code. Afterward from the emulation code back into the kernel to continue normal program execution. We measure this delay similarly to the exception delay with the `UD2` instruction. However, the exception handler returns to another function in user space that executes a second `UD2` from where the kernel jumps back to after the initial `UD2`. We read the TSC counter before and after the first UD2 instruction. This delay is 0.77 µs ($n = 10$, $\sigma_{\bar{x}} = 0.14$) on average on an Intel i9-9900K and 0.27 µs ($n = 10$, $\sigma_{\bar{x}} = 0.02$) on an AMD 7700X.

5.4. Efficiency and Performance of Undervolting

The steady state performance of most CPUs is limited by their thermal design power (TDP), which must not be exceeded for longer periods. Hence, decreasing the CPU core voltage decreases power consumption, which subsequently allows frequencies to be increased, resulting in higher performance.

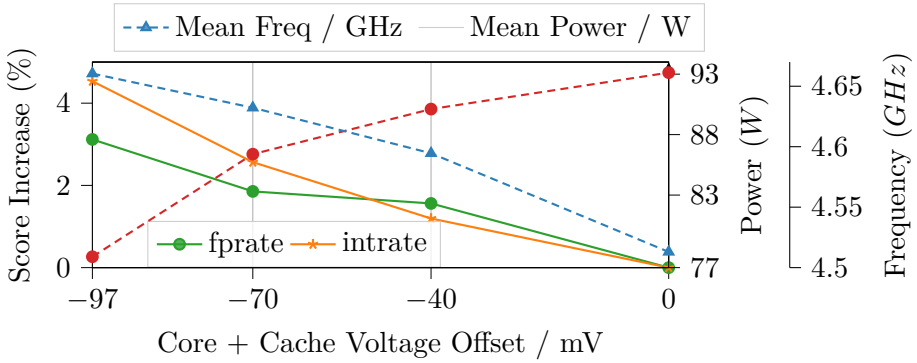


Figure 6.12.: Performance increase for multiple undervolting offsets in SPEC CPU2017 on an Intel i9-9900K. With a voltage offset of −97 mV the SPEC CPU2017 score increases by 3.8 % and the average power consumption decreases by 16 %. The second and third y-axis show the power in W and frequency in GHz.

Table 6.2.: Average performance (SPEC CPU2017 score) increase and power savings for different CPUs.
*AMD CPUs were undervolted using AMD's Curve Optimizer.

CPU	V_{off}	Score	Power	Freq.	Eff.
i5-1035G1	−70 mV	+6.0 %	−0.1 %	+8.5 %	+6.1 %
	−97 mV	+7.9 %	−0.5 %	+12 %	+8.4 %
i9-9900K	−70 mV	+2.2 %	−7.2 %	+2.6 %	+10 %
	−97 mV	+3.8 %	−16 %	+3.3 %	+23 %
7700X*	−70 mV	+1.4 %	−9.8 %	+1.8 %	+12 %
	−97 mV	+1.9 %	−15 %	+1.8 %	+20 %

We measure the CPU package power with Intel's and AMD's Running Average Power Limit (RAPL) [1, 2, 28] interfaces while undervolting the CPU to get realistic values for the power reduction. The accuracy of RAPL is sufficient for this purpose [20]. During this measurement we run the CPUs outside the vendor's defined power states down to voltage offset levels close to where the first faults happen. This is possible due to the aging and temperature guardband. Without SUIT, this is *not* recommended and can have a severe impact on system security [8, 31, 47, 56, 60]. With SUIT, undervolting is possible due to the variation in voltage required by different instructions.

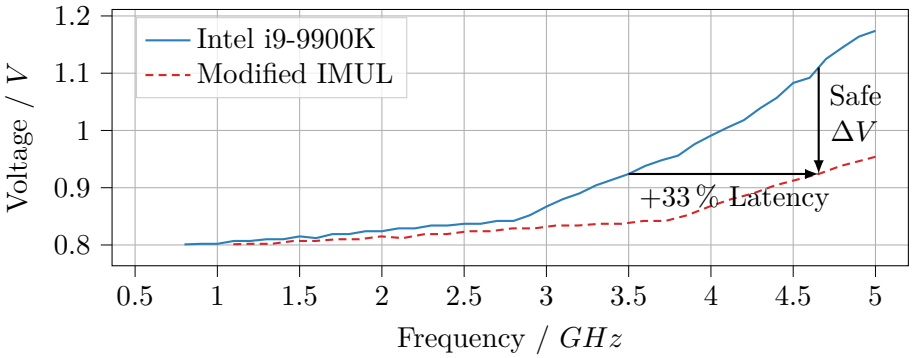


Figure 6.13.: The stable frequency-voltage pairs on an Intel Core i9-9900K measured by fixing the frequency and reading the voltage with MSR `0x198`. The modified `IMUL` plot shows the safe voltages for `IMUL` when increasing its latency from 3 to 4 clock cycles.

Figure 6.12 shows the score of the SPEC CPU2017 benchmarks, power consumption and core frequency with decreasing CPU core voltage offsets on an Intel Core i9-9900K. Table 6.2 shows the results of three CPUs. The efficiency change is computed by one over the change in benchmark duration (score) multiplied with the change in power consumption. If a CPU finishes the benchmark in half the time (score +100 %) while using half as much energy (power −50 %) the efficiency increases by $(0.5 \cdot 0.5)^{-1} = 400\,\%$.

The i5-1035G1 seems to be limited by the TDP, with the power consumption changing negligible but the frequency increasing significantly. We undervolted the AMD CPUs with the curve optimizer [11] in the BIOS as it does not have a MSR `0x150` like Intel CPUs. The curve optimizer value translates to an undervolt offset by multiplying it by 3 – 5 mV. We selected values of -18 and -24. Intel does not allow undervolting the Xeon Silver 4208 CPU at the moment.

5.5. Frequency-Voltage Pairs

For the security analysis in Section 6.9, we measure the p-states of an Intel Core i9-9900K CPU. Figure 6.13 shows the pairs for guaranteed stable operation predefined by the vendor. The modified `IMUL` plot shows the safe voltage for `IMUL` with a 4 clock cycle latency, see Section 6.9.

Table 6.3.: Clock frequency and fan RPM to run the i9-9900K at a specific temperature and the resulting maximum undervolting offset.

f_{CLK}	Fan RPM	t_{core}	V_{off}
4 GHz	1800 (max)	50 °C	−90 mV
4 GHz	300	88 °C	−55 mV

5.6. Aging Guardband

The propagation delay of modern sub 20 nm FinFET circuits degrades by approximately 15 % over a time span of 10 years [19, 46, 53, 61, 66, 67]. To counter aging without decreasing the frequency, the voltage guardband at the beginning of the life cycle must support a 15 % higher frequency than the maximum frequency of the CPU. This ensures that the voltage is high enough after the propagation delay increased by 15 % after 10 years due to aging.

Using the p-states of our Intel Core i9-9900K from Figure 6.13 we can calculate the size of the aging guardband for this CPU. At 5 GHz the CPU core voltage is 1.174 V. The gradient from 4 GHz to 5 GHz is 183 $\frac{\text{mV}}{\text{GHz}}$. This results in an aging voltage guardband of $5\,\text{GHz} \cdot 15\,\% \cdot 183\,\frac{\text{mV}}{\text{GHz}} = 137\,\text{mV}$ or 12 %. This is in line with previous work [36], where authors were able to undervolt their new CPUs by approximately 100 mV more than we are able to on our aged CPUs.

5.7. Temperature Guardband

To estimate the size of the temperature guardband we measure the maximum undervolting offset at different core temperatures on an Intel Core i9-9900K. We influence the core temperature by adjusting the CPU fan speed. With a low fan speed the CPU always clocks as highly as possible under thermal throttling constraints, which must not exceed 90 °C. On average over all SPEC CPU2017 benchmarks, we run at 88 °C since a few benchmarks never reach 90 °C. Table 6.3 shows the maximum undervolting offset at 88 °C and 50 °C. 50 °C is twice the room temperature the CPU was running in, making it a realistic lower bound temperature for a CPU under full load. The difference in voltage requirement is 35 mV, which is 3.5 % of the 991 mV core supply voltage at 4 GHz.

Table 6.4.: The performance impact of disabling SSE and AVX on SPEC CPU2017. All benchmarks exceeding 5 % impact are shown.

	fprate	intrate	508	521	538	554	525	548
i9-9900K	−4.1 %	0.5 %	−22 %	−1.4 %	−12 %	−3.3 %	7.0 %	7.7 %
7700X	−5.9 %	2.6 %	−35 %	−5.3 %	−9.0 %	−19 %	22 %	6.8 %

5.8. SPEC CPU2017 Without SIMD instructions

A program compiled without SSE and AVX support does not execute any SIMD instructions. All instruction in Table 6.1 except `IMUL` and `AESENC` are SIMD instructions. If the performance overhead is smaller then the gain of SUIT, these programs can be executed without ever causing a `#DO` exception because `IMUL` is hardened in CPUs with SUIT. We measure the influence of SIMD on SPEC CPU2017 and compare these numbers in Section 6.7. Table 6.4 shows the impact on the benchmark score for the floating point (FP) and integer suite and for selected benchmarks. The FP score decreases on both tested CPUs as expected, while the integer score *increases*. We suspect AVX throttling being the root cause for the increase [12, 37].

Table 6.5.: The gem5 system used for instruction latency evaluation.

CPU	x86-64, 2 Core, 3 GHz, O3 (Out-Of-Order) CPU
DRAM	2 Channel, 3 GB DDR4_2400_8x8
Cache	64 kB L1I, 32 kB L1D, 2 MB LLC
gem5 Mode	Full System
OS	Ubuntu 20.04.1 with Linux kernel v5.19.0

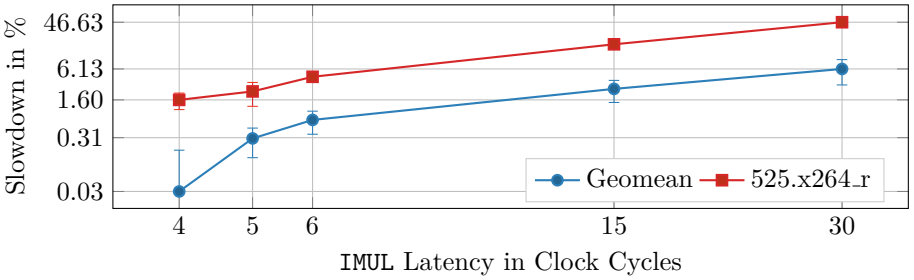


Figure 6.14.: Slowdown in % with increasing latency of `IMUL`. For small increases the relationship does not grow linearly, we suspect that the out-of-order execution hides the additional latency.

6. Evaluation

We evaluate the effects of SUIT on efficiency and performance on a variety of benchmarks and different CPUs.

6.1. Increased `IMUL` Latency

We use the gem5 simulator to evaluate the performance impact of increasing the latency of `IMUL` by 1 clock cycle to 4 clock cycles. The configuration is shown in Table 6.5. We ran the SPEC CPU2017 benchmarks in gem5 with the SPECcast tool by Prieto et al. [55]. It only runs representative parts with the out-of-order CPU model that takes longer to simulate. We also implemented the MSR to disable instructions and `#DO` exception and modified Linux v5.19 to handle the exception.

As shown in Figure 6.14, an `IMUL` latency of 4 clock cycles causes a 0.03 % ($n = 8$, $\sigma_{\bar{x}} = 0.15$) slowdown on average over all SPEC CPU2017 benchmarks. Frequent `IMUL` instructions slow down 525.x264_r most (by

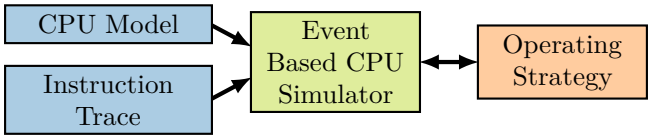


Figure 6.15.: The components of our instruction trace-based simulator.

1.60 % ($n = 9$, $\sigma_{\bar{x}} = 0.55$)). 0.99 % of all instructions executed by 525.x264_r were `IMUL` compared to 0.07 % on average over all other benchmarks.

Figure 6.14 also shows the slowdown for larger `IMUL` latencies. Small increments have only a low impact on performance because they are hidden in the out-of-order execution of CPUs. Latencies of 4 and 5 clock cycles are more easily absorbed. With higher latencies, we can see an almost linear relationship between `IMUL` latency and slowdown.

6.2. Instruction Trace-Based Evaluation

We use the data from Section 5 to evaluate the efficiency and performance impact of SUIT for a range of applications.

Methodology. Our event-based simulator models a CPU executing an instruction stream from Section 5.1 and an operating strategy from Section 4.3 as shown in Figure 6.15. The operating strategy is defined as a class that communicates with the simulated CPU as shown in Listing 6.1. The instructions in Table 6.1 are disabled on the efficient DVFS curve. The simulated CPU behaves as given by the base measurements from Section 5. The efficient DVFS curve causes changes in the power consumption and performance at different voltage offsets as shown in Section 5.4: −70 mV from the variation in instruction voltage requirements and −97 mV with an additional 20 % of the aging guardband, see Section 3.1.

Transitioning between these curves slows down the simulation by the delays given in Section 5.2. As the latency of `IMUL` is increased in all CPUs, we also factor in its negative performance impact from Section 6.1 for this evaluation.

To simulate multi-core CPUs with a single frequency domain, we record multiple instruction streams. During the simulation, one instruction stream is pinned on one core. A DVFS curve change subsequently impacts *all* cores.

```
1 class Operating_Strategy_fV:
2   def disabled_instruction_exception_handler():
3     # we wait for the frequency to change
4     cpu.change_pstate_wait(DVFS.Cf)
5     # and request the voltage change
6     cpu.change_pstate_async(DVFS.Cv)
7
8     cpu.set_instructions_disabled(False)
9
10    # trashing prevention
11    if exception_count_in_timespan(p_ts) >= p_ec:
12      cpu.set_timer_interrupt(p_dl * p_df)
13    else:
14      cpu.set_timer_interrupt(p_dl)
15
16  def timer_interrupt_handler():
17    cpu.set_instructions_disabled(True)
18    cpu.change_pstate_async(DVFS.E)
```

Listing 6.1: Implementation of the *fV* operating strategy (Section 4.3) with trashing protection. On a disabled opcode exception, the CPU switches to the *conservative* DVFS curve by changing the frequency (C_f), a voltage change (C_V) is started and all instructions are enabled. The trashing prevention sets a count-down timer that will switch the CPU to the *efficient* DVFS curve after expiration – unless a faultable instruction was executed prior to timer expiration, which would have reset the count-down timer to the deadline.

Instruction Emulation. To estimate the overhead of instruction emulation, we add the overhead from each SPEC benchmark compiled without SIMD instructions from Section 5.8. Additionally, we add the emulation call delay from Section 5.3 to any execution of disabled instructions.

Applications. We record instruction traces of a wide range of applications. The first set comprises all 23 benchmarks of SPEC CPU2017 [10]. Since they do not contain network workloads apart from the simulated network by `520.omnetpp`, we also test Nginx [25] (100 kB files over an HTTPS connection with AES encryptions and decryptions), using the `wrk` benchmarking tool [64]. The network is a virtual network between the QEMU virtual machine running Nginx and its host running wrk. To benchmark the network client side, we use VLC [62] streaming a 1080p video from Vimeo over HTTPS. Video streaming generates more traffic than web browsing.

Simulated CPUs. We evaluate SUIT on three CPU models:

$\mathcal{A}$: Intel Core i9-9900K with a single frequency and voltage domain and different core counts.
$\mathcal{B}$: AMD Ryzen 7 7700X with per-core frequency domains.
$\mathcal{C}$: Intel Xeon Silver 4208 with per-core frequency and voltage domains.

The subscript, e.g., $\mathcal{A}_1$ defines the number of utilized CPU cores for which the result is relevant. $\mathcal{X}_\infty$ denotes methods where the number of CPU cores are irrelevant on the result.

6.3. Efficiency and Performance Impact

Table 6.6 shows the results of the instruction trace-based evaluation with the CPUs $\mathcal{A}$ to $\mathcal{C}$. The following sections go into detail about the CPUs and operating strategies. Figure 6.16 shows the results for performance and efficiency of all tested applications on CPU $\mathcal{C}$ running the *fV* operating strategy.

Different Undervolting Offsets. The measurements in Table 6.2 show that the efficiency approximately doubles when decreasing the voltage offset from −70 mV to −97 mV. This can be explained with the quadratic voltage dependency in the dynamic power consumption of a CMOS circuit. The approximate doubling also translates to a system with SUIT.

Dependency on Domain Count. Besides the DVFS curve change delay, the performance of SUIT depends on two factors: whether the CPU has a single or per-core DVFS domains; and, if there is only a single DVFS domain, the number of cores. Only the emulation method is core-count independent. The impact of the core count on the efficiency and performance in discussed in Section 6.4.

6.4. *fV* Operating Strategy

The *fV* strategy changes between three p-states E, C_f and C_V (see Figure 6.4) to execute every workload as efficiently and fast as possible. It is used on CPUs with the same number of frequency and voltage domains: $\mathcal{A}$ and $\mathcal{C}$. The following numbers are from CPU $\mathcal{C}$, the results for $\mathcal{A}_1$ are analogous. Figure 6.16 shows the detailed results for the performance and efficiency impact of a −70 mV and −97 mV undervolt.

We discuss the results for −97 mV. The average efficiency gain over all SPEC CPU2017 benchmarks is +11 %, the median is +13 % while running

on the efficient DVFS curve 72.7 % of the time. Applications that execute faultable instruction sparingly stay primarily on the efficient curve. 557.xz sees the highest efficiency gain. 97.1 % of the time it is on the efficient DVFS curve, resulting in a +2.75 % performance and +16.9 % efficiency gain (not shown). Applications that execute faultable instructions frequently are primarily on the C_V p-state. 520.omnetpp is on the efficient curve 3.2 % of the time. The performance impact of −0.13 % is negligible while the power consumption is reduced by −0.60 % resulting in a +0.47 % efficiency gain (not shown). Applications in-between, switch between E, C_f and C_V repeatedly, resulting in a small performance loss but a large reduction in power consumption. 502.gcc experiences the worst impact on the performance with −2.89 %. The reduction in power consumption by −11.5 % results in an efficiency gain of +9.67 % (not shown). It is on the efficient curve 76.6 % of the time.

Core Count Influence on CPU $\mathcal{A}$. CPU $\mathcal{A}$ has a single DVFS domain, which means that every process on each core influences the others. This has a negative impact on performance and efficiency. While $\mathcal{A}_1$ sees a +12 % average efficiency gain, it is reduced to +5.8 % on $\mathcal{A}_4$ (4 cores utilized). We see the highest efficiency gain in 557.xz, which is +17.1 % vs +13.8 % (not shown). The relative difference is smaller due to the overall fewer faultable instructions. These cause fewer DVFS curve switches, influencing other cores less. For benchmarks that are primarily on the conservative curve, multiple cores have a small impact due to little curve switching. The difference for 520.omnetpp is +0.52 % on $\mathcal{A}_1$ to +0.17 % on $\mathcal{A}_4$ (not shown). Overall, SUIT with DVFS curve switching unfolds its full potential on CPUs with per-core frequency and/or voltage domains. Laptop CPUs often only have up to 4 cores that tend to be underutilized given typical office or web browsing usage. But even with 4 fully utilized cores SUIT has a small edge on the efficiency with +5.8 % on average.

Optimal Operating-Strategy Parameters. The operating strategy is configured with four parameters, see Section 4.3. We ran hundreds of simulations to find the optimal values to maximize the efficiency gain and used them for the evaluation. The result is shown in Table 6.7. We did not see a large impact of the parameters on the efficiency in a large range of values. Varying, e.g., the deadline ±10 µs changes the average efficiency by only -0.61 %. This indicates that workloads tolerate a range rather than requiring individual parameters, which simplifies SUIT as an operating system policy.

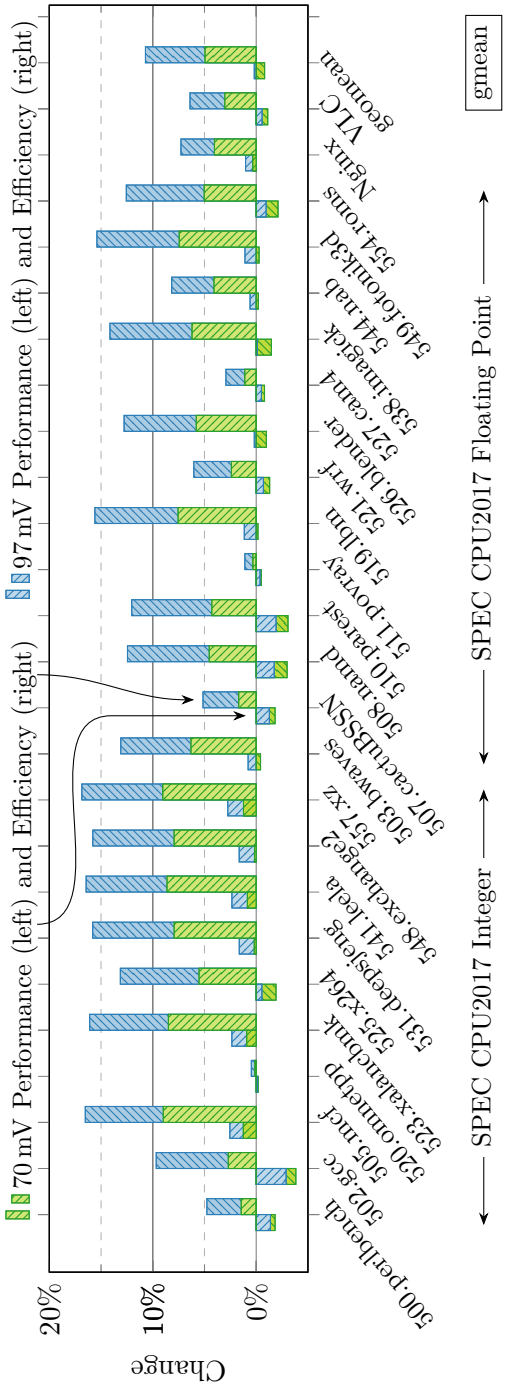


Figure 6.16.: The performance and efficiency impact of SUIT on CPU $\mathcal{C}$ running the fV operating strategy.

Table 6.6.: Power saving and performance impact of SUIT running on different CPUs with different operating strategies (OS) at a −70 mV and −96 mV undervolt. 525.x264 is shown explicitly as it is most adversely impacted by increasing the `IMUL` latency. All colored cells show an increase in efficiency or performance, where darker shades indicate the highest benefit in the respective column.

			70 mV Undervolt						97 mV Undervolt					
CPU $_{\text{cores}}$	OS		$\text{SPEC}_{\text{gmean}}$	$\text{SPEC}_{\text{median}}$	525.x264	$\text{SPEC}_{\text{noSIMD}}$	Nginx	VLC	$\text{SPEC}_{\text{gmean}}$	$\text{SPEC}_{\text{median}}$	525.x264	$\text{SPEC}_{\text{noSIMD}}$	Nginx	VLC
$\mathcal{A}_1$	fV	Pwr	−5.6 %	−7.1 %	−7.1 %	−7.1 %	−3.5 %	−3.9 %	−9.7 %	−11 %	−12 %	−15 %	−5.8 %	−6.3 %
		Perf.	−0.2 %	−1.3 %	−1.3 %	+3.0 %	+0.5 %	−0.4 %	+0.8 %	+1.3 %	0.1 %	+3.4 %	+1.2 %	+0.2 %
		Eff.	+5.7 %	+6.2 %	+6.2 %	+11 %	+4.2 %	+3.6 %	+12 %	+14 %	+14 %	+21 %	+7.4 %	+6.9 %
$\mathcal{A}_4$	fV	Pwr	−4.6 %	−0.1 %	−6.9 %	−7.4 %	−1.0 %	−1.0 %	−8.9 %	−8.7 %	−13 %	−16 %	−1.6 %	−1.6 %
		Perf.	−3.9 %	−0.0 %	−7.9 %	+1.8 %	−0.3 %	−0.6 %	−3.6 %	−3.5 %	−7.2 %	+1.8 %	−0.1 %	−0.5 %
		Eff.	+0.7 %	0.1 %	−1.0 %	+10.0 %	+0.7 %	+0.4 %	+5.8 %	+5.7 %	+6.7 %	+22 %	+1.5 %	+1.1 %
$\mathcal{A}_\infty$	e	Pwr	−7.5 %	−7.6 %	−5.4 %	−7.5 %	−7.2 %	−7.2 %	−12 %	−12 %	−10 %	−17 %	−12 %	−12 %
		Perf.	−42 %	−12 %	+6.2 %	+1.4 %	−98 %	−92 %	−42 %	−12 %	+6.1 %	+1.4 %	−98 %	−92 %
		Eff.	−37 %	−4.5 %	+12 %	+9.6 %	−98 %	−91 %	−34 %	+0.6 %	+18 %	+22 %	−98 %	−91 %
$\mathcal{B}_\infty$	f	Pwr	−8.1 %	−7.8 %	−7.8 %	−9.1 %	−4.4 %	−4.4 %	−12 %	−11 %	−11 %	−14 %	−6.7 %	−6.7 %
		Perf.	−7.8 %	−7.8 %	−9.2 %	+0.4 %	−2.5 %	−2.5 %	−10 %	−11 %	−12 %	+0.6 %	−2.3 %	−2.3 %
		Eff.	+0.3 %	−0.0 %	−1.6 %	+11 %	+2.0 %	+2.0 %	+1.4 %	0.1 %	−1.6 %	+17 %	+4.7 %	+4.7 %
	e	Pwr	−9.2 %	−8.0 %	−11 %	−9.2 %	−9.8 %	−9.8 %	−14 %	−13 %	−16 %	−14 %	−15 %	−15 %
		Perf.	−26 %	−5.1 %	+15 %	−0.5 %	−96 %	−80 %	−26 %	−5.2 %	+19 %	0.0 %	−96 %	−80 %
		Eff.	−19 %	+3.1 %	+28 %	+9.5 %	−95 %	−78 %	−14 %	+9.3 %	+41 %	+17 %	−95 %	−76 %
$\mathcal{C}_\infty$	fV	Pwr	−5.6 %	−7.1 %	−7.1 %	−6.1 %	−3.6 %	−4.0 %	−9.8 %	−11 %	−12 %	−14 %	−5.8 %	−6.6 %
		Perf.	−0.8 %	−1.9 %	−1.9 %	+3.5 %	+0.3 %	−1.1 %	+0.2 %	+0.2 %	−0.6 %	+3.8 %	+1.0 %	−0.6 %
		Eff.	+5.1 %	+5.5 %	+5.5 %	+10 %	+4.0 %	+3.0 %	+11 %	+13 %	+13 %	+21 %	+7.3 %	+6.4 %

6.5. Frequency Operating Strategy on $\mathcal{B}$

CPU $\mathcal{B}$ has per-core frequency domains but a single voltage domain. To be independent of the number of cores, it uses the frequency to change the DVFS curve ($E \leftrightarrow C_f$). Changing the frequency takes 30 times longer than on CPU $\mathcal{A}$ and $\mathcal{C}$, explaining the large impact on performance and efficiency.

6.6. Instruction Emulation

In Table 6.6, operating strategy e shows the results for instruction emulation. For brevity we did not include emulation on CPU $\mathcal{C}$ as the results are very similar to CPU $\mathcal{A}$. On $\mathcal{A}$ the efficiency impact is $-34\,\%$ on average. A few dominant negative results skew the geometric mean so that the median efficiency increase over all benchmarks is $+0.6\,\%$. With fV, all SPEC benchmarks show increased efficiency, ranging from $+0.52\,\%$ to $+17.1\,\%$, $\sigma = 5.9$. With emulation, efficiency ranges from $-91.2\,\%$ to $+27.0\,\%$, $\sigma = 36.6$.

The performance of emulation depends only on instruction frequency but not distribution. This becomes evident when comparing the results with fV on CPU $\mathcal{A}$: Nginx experiences the largest difference in efficiency: $+7.4\,\%$ with fV but $-98\,\%$ with emulation. The short bursts of many AES encryptions are good for DVFS curve switching but impose prohibitive costs when every single encryption causes a exception for emulation.

525.x264 has the overall highest efficiency gain of $+41\,\%$ on CPU $\mathcal{B}$. To simulate instruction emulation we include the benchmark results without SIMD instructions from Section 6.7, there 525.x264 is $22\,\%$ faster. This is combined with the $20\,\%$ efficiency increase at $-97\,\mathrm{mV}$ (Section 5.4) and the short exception delay of CPU $\mathcal{B}$ (Section 5.3).

Table 6.7.: The optimal parameters for the fV operating strategy and trashing prevention (see Section 4.3) used for all simulations.

CPU	p_dl	p_ts	p_ec	p_df
$\mathcal{A}$ & $\mathcal{C}$	30 µs	450 µs	3	14
$\mathcal{B}$	700 µs	14 ms	4	9

Table 6.8.: The number of SPEC CPU2017 benchmarks where compiling *without* SIMD instructions or using SIMD instructions and having the overhead of SUIT leads to higher performance at −97 mV. Emulation is always worse but does not require recompilation.

	$\mathcal{A}_1$	$\mathcal{A}_4$	$\mathcal{A}_\infty$	$\mathcal{B}_\infty$		$\mathcal{C}_\infty$
	fV	*fV*	*e*	*f*	*e*	*fV*
No SIMD	15	21	23	21	23	16
SUIT	8	2	0	2	0	7

The threshold for a positive efficiency impact is approximately one disabled instructions per 4.1×10^{10} instructions. An exact threshold is impossible to define because the distribution of the instructions and the complexity of the emulated instructions have an impact as well. This shows that the viability of instruction emulation is highly dependent on the workload. However, due the hardware-software co-design of SUIT, the operating system can dynamically choose the best operating strategy for each workload.

6.7. SPEC without SIMD Instructions

In Section 5.8, we compared SPEC CPU2017 compiled with and without SIMD instructions. We now use these numbers to evaluate if compiling an application without SIMD is more efficient than using SUIT to execute them securely.

The benchmarks using SIMD have an overhead from SUIT while the others do not but observe the overhead from not using SIMD, see Section 5.8. We compare these results and then individually conduct a sensitivity study assessing the performance before computing the mean over all benchmarks. Table 6.6 shows the results in the column $\text{SPEC}_{\text{noSIMD}}$. The average score over SPEC CPU2017 on CPU $\mathcal{C}$ is +21 %. Emulation is always worse as it incurs the same overhead as $\text{SPEC}_{\text{noSIMD}}$ plus the emulation call overhead but does not require recompilation. Thus, in the emulation $\text{SPEC}_{\text{noSIMD}}$ column no instructions are emulated, i.e., the column shows the average if *all* benchmarks are compiled without SIMD.

Table 6.8 shows how often the performance benefits from the compile-time decision to disable SIMD instructions. On CPU $\mathcal{B}$ this is almost always the case due to its long frequency change delay. However, compiling without

SIMD instructions is very unfavorable for some SPEC CPU2017 benchmarks. The worst case on CPU $\mathcal{C}$ is 508.namd. The efficiency increases by +12.4 % with SUIT but decreases by −20.5 % when compiling without SIMD instructions.

It is crucial to reiterate that these results do *not* translate to CPUs without SUIT. Only with SUIT is the security and correctness of the running applications guaranteed when undervolting. Without such guarantees, any comparison is deemed incorrect as it compromises security. Nonetheless, these results show that having less `#DO` exceptions and DVFS curve switches can be favorable for many benchmarks.

6.8. Performance and Efficiency Impact – Summary

On CPUs with one DVFS domain SUIT does not perform well if all cores are fully utilized. However, it still achieves a slight efficiency gain. On CPUs like this, emulation is better for some benchmarks, but it has a very high variance in respect to different workloads. The variance of the fV operating strategy is lower. It is a good "one fits all" approach because it stays on the C_V p-state for workloads with a high number of potentially faulting instructions, inducing negligible performance overheads. SUIT could dynamically switch between C_V and e for highest efficiency. The frequency change delay of CPU $\mathcal{B}$ is too high to work properly with SUIT. Emulation is more efficient than on $\mathcal{A}$ and $\mathcal{C}$ due to the shorter exception delay. A strategy that uses emulation for some workloads and deactivates SUIT for other workloads is the best compromise. For many benchmarks compiling them without SIMD instructions is most efficient. But this requires recompilation for CPUs with SUIT. On CPUs without SUIT this can be a disadvantage requiring different builds.

Due to the quadratic voltage dependency of the power consumption the undervolting offset from 20 % of the aging guardband has a large impact. But even without it SUIT overall increases the efficiency by 5 % to 10 %.

6.9. Security Analysis

The security of SUIT rests on the security of changing the CPU's DVFS curve and increasing the `IMUL` latency. An absolute argument or proof on the security of a specific approach to DVFS may be infeasible in the presence of process variation. It would also, security-wise, go beyond

the guarantees vendors provide for current CPUs. Instead, we present reductionist arguments for each strategy showing that the security with SUIT is equivalent to the security of current CPUs.

SUIT. With SUIT, we disable specific instructions, so that they cannot fault. This removes these instructions from the optimization vendors perform to determine the optimal number of cycles per instruction and clock frequency, yielding a more efficient DVFS curve. For the remaining instructions, the security level is exactly the same as for current CPUs, as the vendor optimizes these instructions as usual and provides a DVFS curve for them. When a disabled instruction has to be executed, we resort to a voltage-frequency level the vendor determined to be stable in an optimization including these instructions. SUIT implements this by lowering the frequency (or increasing the voltage). Thus, in summary, the security of SUIT is based on the processes that are already established by vendors but now performed separately for conservative and efficient curves, i.e., without and with considering the set of faultable instructions.

Increased `IMUL` Latency. We only found the `IMUL` instructions to be too frequent to trap efficiently. Increasing the latency of an instruction increases the timing slack proportionally. Increasing the latency for `IMUL` from 3 to 4 leads to 33 % of the latency being slack time [49]. Increasing the clock frequency, on the other hand, reduces the instruction slack proportionally, e.g., increasing it by 25 % reduces the slack space for each instruction by 20 % [49]. We can also reduce the voltage accordingly but the relation with the voltage is not linear. Figure 6.13 shows the DVFS frequency-voltage pairs on an i9-9900K. We can see that, in the best case at 5 GHz, instead of a 33 % increase in frequency we can reduce the voltage by 220 mV. In the worst case, at very low frequencies, the voltage reduction is negligible as the power consumption of the CPU is not a concern. Reducing the voltage accordingly would lead to the same security as modern CPUs currently have. Hence, security-wise, if we do not decrease the voltage further than indicated by these real-world numbers, we do not lower the security of the system.

7. Related Work

Numerous prior works investigated undervolting [6, 7, 16, 30, 35, 36, 43, 51, 52]. What differentiates them from SUIT is that they undervolt by removing or shrinking the CPU's guardband. In most of them, the impact of aging is out of scope. Because SUIT shrinks the voltage from the variation in the voltage requirements of instructions, the CPU keeps the voltage guardband required for aging fully or most of it.

xDVS. Koutsovasilis et al. [36] observed that the minimum CPU voltage depends on the workload. Their extended dynamic voltage scaling (xDVS) governor scales the voltage according to the workload by taking performance monitor counter readings into account. They observe over 40 % energy savings by undervolting the CPU by over 200 mV.

CADU++. Maroudas et al. [43] additionally differentiate between kernel and user space for their workload dependent undervolting, CADU++. They observed that the undervolting potential of user space code is consistently higher than that of kernel code. On average they achieve over 240 mV undervolting resulting in up to 30 % power savings. However, CADU++ requires a voltage change on every kernel entry and exit. According to our measurements, they ignore parts of the voltage change delay by only considering the delay to write the MSR, and not until the voltage actually changes.

ECC Cache Errors. Bacha et al. [7] base their work on cache lines faulting first when undervolting Itanium CPUs. We did not observe this on x86. The faults are corrected with ECCs. A calibration phase finds the weakest cache line and the faulting voltage level, which can be repeated periodically to counter aging. They achieved a 33 % power reduction.

Razor. Ernst et al. [13] proposed Razor. It follows the elegant idea of using slightly skewed clock edges and shadow circuitry to detect when critical paths in circuits are about to violate their timing constraints. This allows Razor to dynamically find the optimal voltage and frequency for every chip. While Razor promises substantial energy savings, the special Razor circuits increase the complexity significantly. Razor has still found no adoption in commercial systems today. In comparison, SUIT comprises a rather simple set of changes without any complexity increase on the circuit level.

Efficient Scheduling. Lawall et al. [38] group cores in a *nest* to keep their clock frequencies high and schedule tasks that share resources in proximity. This minimizes clock frequency changes and sharing of data between sockets. 2- and 4-socket multicore machines gain up to 20 % performance and efficiency. Gouicem et al. [18] design a scheduler that minimizes frequency changes. They gain up to 56 % performance on an 80-core Intel Xeon CPU. Similar scheduling methods could also be used in conjunction with SUIT to minimize DVFS curve changes.

Heterogeneous CPUs house sets of cores of different types and capabilities. Examples are recent Intel CPUs with power and efficiency cores [48] and ARMs big.LITTLE [5] design. With large differences in power consumption between power and efficiency cores, by design, they lack support for dynamic adjustment of the number of cores for each type. SUIT dynamically adapts to workloads by running any number of cores with the conservative or efficient DVFS curves.

8. Discussion

Protection against Undervolting Attacks. This work is primarily aimed at providing a benign OS with a method to securely reduce the power consumption of a computer. We did not investigate if SUIT can also protect against undervolting attacks [31, 47, 56, 60] by only allowing DVFS curve switching if all faultable instructions are disabled.

Speculative Execution. Speculative out-of-order execution of disabled instructions must not happen. This could cause exploits like meltdown where the exception is only thrown at retirement, which is too late. The 4 clock cycle IMUL is not a faultable instruction and can be speculatively executed.

Side-Channel Leakage. An attacker could learn when disabled instructions are executed to build a covert channel [22].

9. Conclusion

In this paper we proposed SUIT, a technique enabling substantial efficiency gains on modern CPUs. It is based on the fact that only a small subset of instructions requires the conservative power margins observed on current

CPUs. With SUIT we disable the execution of faultable instructions and run the CPU on a more efficient DVFS curve 72.7 % of the time, increasing the efficiency by 11.0 % with no performance impact over SPEC CPU2017, without reducing the system's reliability or security. Together with compile-time optimizations for SUIT the CPU efficiency increases by 20.8 % while the performance increases by 3.79 %. Hence, we conclude that SUIT is a practical solution to increase energy efficiency.

Acknowledgements

We thank Andreas Kogler, Sarah Rinderer, the reviewers and especially our shepherd, Guilherme Cox, for their feedback and help. This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe F8504), NSF grants 1813004, 2217020, 2316201 and a research grant from CISCO. This work has benefitted from Dagstuhl Seminar 22341 (PEACHES). Additional funding was provided by generous gifts from Red Hat, Google and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] AMD. Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors. 2017 (p. 167).

[2] AMD. Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors. 2021 (p. 167).

[3] AMD. Software Optimization Guide for AMD EPYC 7003 Processors. Nov. 2020 (p. 159).

[4] AMD. Software Optimization Guide for AMD Family 17h Processors. June 2017 (p. 159).

[5] ARM. big.LITTLE Technology: The Future of Mobile. 2013. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf (p. 183).

[6] Anys Bacha and Radu Teodorescu. Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors. In: International Symposium on Computer Architecture (ISCA). 2013. DOI: 10.1145/2485922.2485948 (pp. 148, 149, 182).

[7] Anys Bacha and Radu Teodorescu. Using ECC feedback to guide voltage speculation in low-voltage processors. In: International Symposium on Microarchitecture (MICRO). 2014. DOI: 10.1109/MICRO.2014.54 (pp. 148, 149, 156, 182).

[8] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low Voltage Fault Attacks on the RSA Cryptosystem. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2009 (pp. 148, 155, 167).

[9] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In: USENIX ATC. 2005 (p. 162).

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In: International Conference on Performance Engineering. 2018. DOI: 10.1145/3185768.3185771 (p. 173).

[11] Ian Cutress. AMD Precision Boost Overdrive 2: Adaptive Undervolting For Ryzen 5000 Coming Soon. Nov. 2020. URL: https://www.anandtech.com/show/16267/amd-precision-boost-overdrive-2-adaptive-undervolting-for-ryzen-5000-coming-soon (p. 168).

[12] Dell. Intel i9 Processor Throttling Under AVX (Advanced Vector eXtensions). 2021. URL: https://www.dell.com/support/kbdoc/en-us/000184687/intel-i9-processor-throttling-under-avx-advanced-vector-extensions (p. 170).

[13] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In: MICRO. 2003. DOI: 10.1145/1150343.1150348 (p. 182).

[14] Nicholas Fearn. Will AWS pledge to extend life of servers inspire other cloud firms to follow suit? May 2022. URL: https://www.computerweekly.com/feature/Will-AWS-pledge-to-extend-life-of-servers-inspire-other-cloud-firms-to-follow-suit (p. 155).

[15] 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. 153, 159).

[16] Dimitris Gizopoulos, George Papadimitriou, Athanasios Chatzidimitriou, Vijay Janapa Reddi, Behzad Salami, Osman S Unsal, Adrian Cristal Kestelman, and Jingwen Leng. Modern hardware margins: CPUs, GPUs, FPGAs recent system-level studies. In: International Symposium on On-Line Testing and Robust System Design (IOLTS). 2019. DOI: 10.1109/IOLTS.2019.8854386 (pp. 148, 149, 151, 156, 182).

[17] Corey Gough, Ian Steiner, Winston Saunders, Corey Gough, Ian Steiner, and Winston Saunders. CPU Power Management. In: Energy Efficient Servers: Blueprints for Data Center Optimization (2015). DOI: 10.1007/978-1-4302-6638-9_2 (pp. 148, 151).

[18] Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance. In: USENIX Annual Technical Conference (USENIX ATC). 2020 (p. 183).

[19] Xinfei Guo, Vaibhav Verma, Patricia Gonzalez-Guerrero, and Mircea R Stan. When "things" Get Older: Exploring Circuit Aging in IoT Applications. In: International Symposium on Quality Elec-

tronic Design (ISQED). 2018. DOI: 10.1109/ISQED.2018.8357304 (pp. 152, 169).

[20] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In: IEEE ISPASS. 2013. DOI: 10.1109/ISPASS.2013.6557170 (p. 167).

[21] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In: International Parallel and Distributed Processing Symposium Workshop (IPDPSW). 2015. DOI: 10.1109/IPDPSW.2015.70 (p. 165).

[22] Jawad Haj-Yahya, Lois Orosa, Jeremie S Kim, Juan Gómez Luna, A Giray Yağlikçi, Mohammed Alser, Ivan Puddu, and Onur Mutlu. IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors. In: ISCA. 2021. DOI: 10.1109/ISCA52012.2021.00081 (p. 183).

[23] Sebastian Herbert and Diana Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In: International Symposium on Low Power Electronics and Design (ISLPED). 2007. DOI: 10.1145/1283780.1283790 (p. 148).

[24] Vincent Huard, Souhir Mhira, A Barclais, X Lecocq, F Raugi, M Cantournet, and Alain Bravaix. Managing electrical reliability in consumer systems for improved energy efficiency. In: IEEE International Reliability Physics Symposium (IRPS). 2018. DOI: 10.1109/IRPS.2018.8353561 (p. 152).

[25] Igor Sysoev. Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX. 2004. URL: https://www.nginx.com/ (p. 173).

[26] Intel. Changes in Customer Support and Servicing Updates for Select Intel® Processors. 2023. URL: https://www.intel.com/content/www/us/en/support/articles/000022396/processors.html (p. 155).

[27] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (p. 156).

[28] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 148, 164, 167).

[29] Nicola Jones. How to stop data centres from gobbling up the world's electricity. In: Nature (2018). DOI: 10.1038/d41586-018-06610-y (p. 148).

[30] Manolis Kaliorakis, Athanasios Chatzidimitriou, George Papadimitriou, and Dimitris Gizopoulos. Statistical analysis of multicore CPUs operation in scaled voltage conditions. In: IEEE Computer Architecture Letters (2018). DOI: 10.1109/LCA.2018.2798604 (pp. 148, 149, 156, 182).

[31] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In: USENIX Security. 2020 (pp. 148, 155, 167, 183).

[32] Andreas Kogler, Daniel Gruss, and Michael Schwarz. Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks. In: USENIX Security. 2022 (pp. 149, 153–155, 159).

[33] 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 (p. 151).

[34] Martijn Koot and Fons Wijnhoven. Usage impact on data center electricity needs: A system dynamic forecasting model. In: Applied Energy (2021). DOI: 10.1016/j.apenergy.2021.116798 (p. 155).

[35] Panos Koutsovasilis, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. The Impact of CPU Voltage Margins on Power-Constrained Execution. In: IEEE Transactions on Sustainable Computing (2020). DOI: 10.1109/TSUSC.2020.3045195 (pp. 148, 149, 156, 182).

[36] Panos Koutsovasilis, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, and Spyros Lalis. Dynamic undervolting to improve energy efficiency on multicore x86 cpus. In: IEEE Transactions on Parallel and Distributed Systems (2020). DOI: 10.1109/TPDS.2020.3004383 (pp. 148, 149, 156, 169, 182).

[37] Vlad Krasnov. On the dangers of Intel's frequency scaling). 2017. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/ (p. 170).

[38] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores. In: Proceedings of the Seventeenth European Conference on Computer Systems. 2022. DOI: 10.1145/3492321.3519585 (p. 183).

[39] 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. DOI: 10.1109/SP40001.2021.00063 (p. 151).

[40] S Mahapatra, N Goel, S Desai, S Gupta, B Jose, S Mukhopadhyay, K Joshi, A Jain, AE Islam, and MA Alam. A Comparative Study of Different Physics-Based NBTI Models. In: IEEE transactions on electron devices (2013). DOI: 10.1109/TED.2013.2238237 (p. 155).

[41] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer Science & Business Media, 2008. ISBN: 0387381627 (p. 151).

[42] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. An empirical survey of performance and energy efficiency variation on intel processors. In: International Workshop on Energy Efficient Supercomputing. 2017. DOI: 10.1145/3149412.3149421 (p. 148).

[43] Emmanouil Maroudas, Spyros Lalis, Nikolaos Bellas, and Christos D Antonopoulos. Exploring the potential of context-aware dynamic CPU undervolting. In: ACM International Conference on Computing Frontiers. 2021. DOI: 10.1145/3457388.3458658 (pp. 148, 149, 156, 182).

[44] Ken Martin. Digital integrated circuit design. 2000. ISBN: 0195125843 (p. 151).

[45] Eleršič Miha. Guide to linux undervolting for Haswell and never Intel CPUs. 2018. URL: https://github.com/mihic/linux-intel-undervolt (p. 154).

[46] Evelyn Mintarno, Joëlle Skaf, Rui Zheng, Jyothi Bhaskar Velamala, Yu Cao, Stephen Boyd, Robert W Dutton, and Subhasish Mitra. Self-Tuning for Maximized Lifetime Energy-Efficiency in the Presence of Circuit Aging. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2011). DOI: 10.1109/TCAD.2010.2100531 (pp. 152, 169).

[47] 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. DOI: 10.1109/SP40000.2020.00057 (pp. 148, 149, 153, 155, 167, 183).

[48] Rukmabhatla Nikhil, Chabukswar Rajshree, Gohad Sneha, and Chynoweth Michael. Introduction to Performance Hybrid Architecture for 12th Generation Intel Core Processors. 2013. URL: https://www.intel.com/content/www/us/en/content-details/685861/introduction-to-performance-hybrid-architecture-for-12th-generation-intel-core-processors.html (p. 183).

[49] Vojin G Oklobdzija, Vladimir M Stojanovic, Dejan M Markovic, and Nikola M Nedovic. Digital System Clocking: High-Performance and Low-Power Aspects. Wiley-IEEE Press, 2005. ISBN: 9780471274476 (pp. 151, 153, 159, 181).

[50] George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore CPUs. In: IEEE international symposium on high performance computer architecture (HPCA). IEEE. 2019. DOI: 10.1109/HPCA.2019.00033 (p. 148).

[51] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. Harnessing voltage margins for energy efficiency in multicore CPUs. In: IEEE/ACM International Symposium on Microarchitecture. 2017. DOI: 10.1145/3123939.3124537 (pp. 148, 149, 156, 182).

[52] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Charalampos Magdalinos, and Dimitris Gizopoulos. Voltage margins identification on commercial x86-64 multicore microprocessors. In: IEEE Symposium on On-Line Testing (IOLTS). 2017. DOI: 10.1109/IOLTS.2017.8046198 (pp. 148, 149, 156, 182).

[53] Devyani Patra, Jiayang Zhang, Runsheng Wang, Mehdi Katoozi, Ethan H Cannon, Ru Huang, and Yu Cao. Compact Modeling and Simulation of Accelerated Circuit Aging. In: IEEE Custom Integrated Circuits Conference (CICC). 2018. DOI: 10.1109/CICC.2018.8357063 (pp. 152, 169).

[54] Michael K Patterson. The Effect of Data Center Temperature on Energy Efficiency. In: Intersociety Conference on Thermal and

Thermomechanical Phenomena in Electronic Systems. 2008. DOI: 10.1109/ITHERM.2008.4544393 (p. 148).

[55] Pablo Prieto, Pablo Abad, Jose Angel Herrero, Jose Angel Gregorio, and Valentin Puente. SPECcast: A Methodology for Fast Performance Evaluation with SPEC CPU 2017 Multiprogrammed Workloads. In: ICPP. 2020. DOI: 10.1145/3404397.3404424 (p. 171).

[56] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: CCS. 2019. DOI: 10.1145/3319535.3354201 (pp. 148, 155, 167, 183).

[57] Josyula R Rao and Pankaj Rohatgi. EMpowering Side-Channel Attacks. In: Cryptology ePrint Archive, Report 2006/037 (2001) (p. 151).

[58] Christian Schlünder, Stefano Aresu, Georg Georgakos, Werner Kanert, Hans Reisinger, Karl Hofmann, and Wolfgang Gustin. HCI vs. BTI?-Neither one's out. In: IEEE International Reliability Physics Symposium (IRPS). 2012. DOI: 10.1109/IRPS.2012.6241797 (p. 152).

[59] D Suleiman, M Ibrahim, and I Hamarash. Dynamic voltage frequency scaling (DVFS) for microprocessors power and energy reduction. In: International Conference on Electrical and Electronics Engineering. 2005 (p. 148).

[60] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security. 2017 (pp. 148, 155, 167, 183).

[61] Victor M Van Santen, Hussam Amrouch, Narendra Parihar, Souvik Mahapatra, and Jörg Henkel. Aging-Aware Voltage Scaling. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). 2016. ISBN: 978-3-9815-3707-9 (pp. 152, 169).

[62] VideoLan. VLC media player. 2006. URL: https://www.videolan.org/vlc/index.html (p. 173).

[63] 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. 148, 151).

[64] Will Glozer. wrk - a HTTP benchmarking tool. 2015. URL: https://github.com/wg/wrk (p. 173).

[65] MF Zainudin, H Hussin, AK Halim, and J Karim. Aging analysis of high performance FinFET flip-flop under Dynamic NBTI simulation configuration. In: IOP Conference Series: Materials Science and Engineering. 2018. DOI: 10.1088/1757-899X/341/1/012012 (p. 155).

[66] Zuodong Zhang, Zizheng Guo, Yibo Lin, Meng Li, Runsheng Wang, and Ru Huang. AVATAR: An Aging-and Variation-Aware Dynamic Timing Analyzer for Error-Efficient Computing. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023). DOI: 10.1109/TCAD.2023.3255167 (pp. 152, 169).

[67] Zuodong Zhang, Runsheng Wang, Xuguang Shen, Dehuang Wu, Jiayang Zhang, Zhe Zhang, Joddy Wang, and Ru Huang. Aging-Aware Gate-Level Modeling for Circuit Reliability Analysis. In: IEEE Transactions on Electron Devices (2021). DOI: 10.1109/TED.2021.3096171 (pp. 152, 169).

10. Appendix

10.1. Artifact Appendix

Abstract

We provide artifacts to measure key results to perform the trace-based simulation resulting in the numbers in Table 6. While we provide artifacts to reproduce all results as well as all our measurement and simulation results, only a subset of the experiments is described in this document and part of the artifact reproducibility evaluation.

Artifact check-list (meta-information)

- **Program: SPEC-CPU2017 (proprietary, not included)**
- **Compilation: gcc / g++ version $\geq$ 9**
- **Run-time environment: debian-based Linux, root access**
- **Hardware: Intel CPU up until 9th generation**
- **Execution: Sole user, process pinning**
- **Metrics: Power savings, performance impact**
- **Experiments: CPU Microbenchmarks and Simulation**
- **How much disk space required (approximately)?: $\approx$ 200 GB**
- **How much time is needed to prepare workflow (approximately)?: 4 h**
- **How much time is needed to complete experiments (approximately)?: 24 h**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPLv3 License**
- **Data licenses (if publicly available)?: MIT License**
- **Archived?: 10.5281/zenodo.10479442**

Description

How to access All code and data is available at 10.5281/zenodo.10479442. Due to license restrictions, the user has to provide an iso image of the SPEC CPU2017 benchmarks

Hardware dependencies An Intel CPU where undervolting with the 0x150 MSR still works ($\leq$ 9th gen if SGX is disabled in the BIOS).

Software dependencies Linux distribution with the possibility to install kernel modules, we used Ubuntu 20.04. SPEC CPU 2017 iso image.

Data sets We provide some data that takes too long to record for the artifact evaluationluation. EHowever, verythingall code and instructions to record this data isare available.

Installation

Every experiment directory contains a Makefile. There are some build dependencies: `coreutils, build-essential, libarchive-dev, make,linux-tools-generic`

Experiment workflow

We provide artifacts to verify the following claims. Chained together they result in similar numbers to the numbers of Table 6. Due to process variations every CPU is affected differently by undervolting, therefore, the results will not exactly, i.e., they depend on the specific variations.

C1 Voltage Change Delay
C2 Frequency Change Delay
C3 Efficiency and Performance Impact of Undervolting
C4 Final Simulation

The gem5 simulations of the increased IMUL latency and the recording of the instruction traces of the SPEC CPU2017 benchmarks take hundreds of hours to run. We publish the results but also all artifacts to run these experiments.

All experiments come with a `README.md` file containing additional information that does not fit this document.

Disclaimer

CPU undervolting can cause instabilities! This can break file systems and cause data loss or corruption. Never run this software on a system without backups. We are not responsible for any damage caused by this software.

Evaluation and expected results

For the first three claims **C1** to **C3** we do not expect specific results. The simulation results of **C4** show that SUIT has an overall positive impact on CPU energy efficiency.

Voltage Change Delay
Measure the voltage change delay from Section 5.2.

Path: `5_microbenchmarks/1_voltage_change_delay`

Run: `./run.sh`
The run script first builds everything and then loads the kernel module. If successful, the measurement is run, measuring 20 voltage offset changes of −70 mV. Finally, it calls the plot script with the `result.csv` file to plot the data.

Plot: `python3 user/plot.py result.csv`
The plot script also prints the average time it takes to change the voltage to stdout. This time is required later to define the CPU for the simulation.

Result: Time it takes to change the core voltage.

Frequency Change Delay
Measure the voltage change delay from Section 5.2.

Path: `5_microbenchmarks/3_freq_change_delay`

Prepare: Turn off hardware controlled p-states (HWP) by booting the kernel with the `intel_pstate=passive` command line option.

Run: `./run.sh`
The script measures 20 frequency changes of −500 MHz from 3 GHz. Finally it calls the plot script with the
`result.csv` file to plot the data.

Plot: `python3 user/plot.py result.csv`
The plot script also prints the average time it takes to change the frequency to stdout. This time is required later to define the CPU for the simulation.

Result: Time it takes to change the frequency.

Efficiency and Performance Impact of Undervolting
Measure the efficiency and performance impact of undervolting from Section 5.4.

Path: `5.4_power_efficiency_and_performance`
The `README.md` in this directory contains additional details.

Prepare: There are multiple preparation steps:

Install SPEC CPU2017
Do not forget to `runcpu --update` to update the installation to the newest version.

Update the `SPEC_PATH` in the first line of `5.4_power_efficiency_and_performance/Makefile` to point to the SPEC CPU2017 installation directory.

Build with `make` and copy the built files with `make copy`. The later command requires root privileges to set the root sticky bit for the `measure` binary, because it must run as root but we do not want to start SPEC CPU2017 as root.

Test the SPEC CPU2017 installation with:
```
runcpu --config=power --size=test --define \
undervolting=0 specrate
```

Run: SPEC CPU2017 is run twice, once with the CPU at the default voltage and once with the CPU undervolted. The undervolt offset is defined in the `--define undervolting=X` command line argument when starting the benchmark.

Run with the default voltage:

```
runcpu --config=power --size=ref -n 2 \
--output_format config,csv --copies=$(nproc) \
--define undervolting=0 specrate
```

Run with a −70 mV offset:

```
runcpu --config=power --size=ref -n 2 \
--output_format config,csv --copies=$(nproc) \
--define undervolting=-70 specrate
```

Gather Results: The power measurement results are in `$SPEC_PATH/power_measurements` the benchmark scores in `$SPEC_PATH/results`. Combine them with:

```
mkdir -p results/spec
cp $SPEC_PATH/power_measurements/* results/
cp $SPEC_PATH/results/* results/spec
```

The files for the three CPUs from Table 2 are available at 10.5281/zenodo.10479442.

Plot: `./plot.py result`
This plots the changes in CPU frequency, voltage, power and benchmark scores. It prints the exact changes in % to stdout.

Result: In the last printed table, the relevant columns are: `pct` showing the benchmark score increase and `eff` showing the efficiency increase at −70 mV. We expect the benchmark scores and efficiency to increase.

Final Simulation
Perform the final simulation from Section 6.

Path: `6.2_instruction_trace_based_evaluation`

Prepare:

CPU Configuration File
Copy `cpus/i9-9900K-70mV.csv` to `cpus/cpu_name-70mV.csv` as a starting point. The file is then configured with the results from the previous experiments. An exact description of all fields is in the `README.md` file.

Instruction Traces
We provide the instruction traces for all benchmarks at 10.5281/zenodo.10479442 `instruction_traces.tar.gz` except for 520.omnetpp and 521.wrf, as they are too large.

Run:
```
./simulate_all.py cpu_name-70mV \
30,15,3,14 voltfreq "" path/to/spec/traces/5*
```

Postprocess: `./postprocess.py results_XXX.json`
The simulation creates an output `.json` file containing the results of each benchmark. Running the post-process script prints the results formatted as a table to stdout.

Result: The last three columns are the most relevant, the row `geomean` contains the numbers of Table 6:

- `power perc`: change in power consumption in %: "Pwr"
- `perf perc`: change in performance in %: "Perf."
- `eff perc`: change in efficiency in %: "Eff."

We expect the efficiency change to be positive to show that SUIT is viable. Additionally, all three numbers should be similar to numbers of $\mathcal{A}_1$ with the fV operating strategy and column $\text{SPEC}_{\text{gmean}}$. The performance impact is allowed to be slightly negative.

Because we cannot provide the traces for 520.omnetpp and 521.wrf the resulting mean is slightly better than with the two benchmarks included.

On our tested CPU $\mathcal{A}$ at $-70\,\text{mV}$, excluding the two benchmarks increases the efficiency gain from 5.7 % (see Table 6) to 6.1 %.

Regenerate Table 6
We are now able to generate Table 6 from the paper with the additional results from this evaluation.

Path: `6.2_instruction_trace_based_[...]/scripts`

Prepare: Update the file path at `generate_results_table.py:246` to point to the .json result file from the simulation.

Run: `./generate_results_table.py --full`

Result: `results_table.tex`
When compiled it shows Table 6 with an additional CPU $\mathcal{D}$ showing the results of this evaluation.

7

Presshammer: Rowhammer and Rowpress without Physical Address Information

Publication Data

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

Contributions

Main Author.

Presshammer: Rowhammer and Rowpress without Physical Address Information

Jonas Juffinger[1], Sudheendra Raghav Neela[1], Martin Heckel[2], Lukas Schwarz[1], Florian Adamsky[2], Daniel Gruss[1]

[1]Graz University of Technology, Graz, Austria
[2]Hof University of Applied Sciences, Hof, Germany

Abstract

Modern DRAM is susceptible to fault attacks that undermine the entire system's security. The most well-studied disturbance effect is Rowhammer, where an attacker repeatedly opens and closes (*i.e.*, hammers) different rows, which can lead to bitflips in adjacent rows. Different hammering strategies include double-sided, hammering two rows sandwiching a victim row, and one-location, hammering a single row. One-location Rowhammer requires no physical address information, as any location in memory is mapped to a DRAM row, and no relation between rows is required for hammering. The recently discovered Rowpress differs from Rowhammer by not hammering rows but keeping them open longer, evident by a disjoint set of affected memory locations.

In this paper, we examine the differences between four attack variants: one-location Rowhammer, a one-location Rowpress variant we developed, double-sided Rowhammer, and double-sided Rowpress on a set of 12 DDR4 modules. Our methodology is to hammer and press the exact same set of physical memory locations in all attack variants. Surprisingly, our results show that on 4 out of 12 DDR4 modules, we were only able to reproduce double-sided Rowhammer but none of the other attack variants. On 2 DDR4 modules, we were able to reproduce all attack variants. We find that the number of unique bitflip locations ranges from 161 to 15 612, when hammering the exact same set of physical memory locations. Our one-location Rowhammer attack induces roughly the same amount of bitflips as double-sided Rowhammer, however, only 61.8 % of bitflip locations overlap. We explain this by one-location Rowhammer inducing bitflips due to the Rowhammer as well as the Rowpress effect, making the differentiation of both methods difficult, therefore, calling it Presshammer. Based on our

observed bitflips, we develop the first end-to-end one-location Rowpress attack. One-location Rowpress requires only minimal physical address information that an attacker can acquire through a same-row same-bank side-channel attack. Our end-to-end attack escalates to kernel privileges within less than 10 minutes.

1. Introduction

A fundamental assumption for secure software systems is the correctness of the underlying hardware. A long line of work has shown that physical glitching could violate this assumption. However, for a long time [2], it was not clear whether it could also be violated in scenarios without a physical attacker. The discovery of Rowhammer as a security issue in 2014 [21] showed that this assumption can be fatally flawed. Rowhammer is a disturbance effect in the DRAM, leading to data corruption that can undermine the entire system's security. To trigger the Rowhammer effect, an attacker repeatedly opens and closes (*i.e.*, hammers) different rows, leading to row activation operations in the DRAM [21]. Different Rowhammer strategies have been discovered, such as single-sided hammering [21], double-sided decoys [9, 17] which all open and close different rows in the same bank, triggering row activations. One-location Rowhammer [10] works slightly differently, as it does not activate multiple rows but instead performs frequent accesses to one single row. This approach has the advantage of not requiring physical address information, as any location in memory maps to a DRAM row, and no relation between rows is required for hammering.

Recently Luo et al. [26] discovered an effect in DDR4, called Rowpress. Rowpress, instead of accessing a single location in a row and then another one in a different row, accesses a larger number of offsets in one row before switching to another row. The goal of this access pattern is to keep the row open as long as possible, as this can induce bitflips due to RAS clobber, *i.e.*, the passing gate effect [12, 15, 52]. Luo et al. [26] report that Rowpress and Rowhammer bitflips belong to largely disjoint sets of memory locations. Rowpress uses physical address information to identify the rows and the offsets within the rows to target.

Comparing the open-source implementations of Rowpress [26] and one-location Rowhammer [10], it is interesting to see that both implementa-

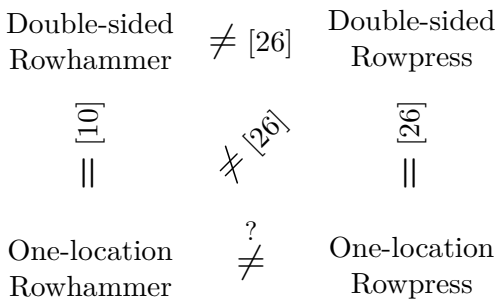


Figure 7.1.: Claims and observations from different papers. The equal sign indicates that these variants exploit the same root cause, and the unequal sign indicates that these variants have a different root cause. The reference indicates which work this claim is based on.

tions `main-algo1` and `main-algo2` by Luo et al. indeed hammer two rows[1], albeit multiple offsets within a row, in a Flush+Reload [21] loop, whereas Gruss et al. only run a Flush+Reload loop on a single address. Hence, following the definition of Rowpress, the attack loop by Gruss et al. should, similarly, actually press a row and not hammer it. However, it is not clear whether the one-location hammering and Rowpress implementations on CPUs, exploit the same or different physical root causes: Gruss et al. [10] reported a difference in bitflip rates between single-sided, double-sided and one-location hammering but no significant difference in bitflip locations, indicating that they all exploit the same root cause. Combining these reports leads to a contradiction, as illustrated in Figure 7.1. Thus, the relation between the different Rowhammer and Rowpress variants remains unclear.

In this paper, we examine the different Rowhammer and Rowpress variants and find that these effects are not easy to separate. We run Rowhammer and Rowpress tests on 12 DDR4 modules to assess the prevalence of Rowhammer and Rowpress effects on current systems. We find that double-sided Rowhammer works on only 6 out of 12 DDR4 modules. We implement a one-location Rowpress variant, which strictly presses only a single row in contrast to published proof of concepts. However, we find that Rowhammer and Rowpress variants other than double-sided Rowhammer only work on two DDR4 modules in our setup, both with the published proof of concept (PoC) as well as our own implementations. Both results pose a

[1] `https://github.com/CMU-SAFARI/RowPress/blob/main/demonstration/main-algo1.cpp#L143`

significant difference from the numbers reported by Jattke et al. [17] and Luo et al. [26].

We compare the four attack variants on all DDR4 modules with bitflips in any of the attacks. We hammer and press the exact physical memory locations on the same system in all attack variants by using a setup with fixed memory mappings so that physical addresses remain identical across different Rowhammer and Rowpress tools and runs. The number of unique bitflip locations ranges from 161 to 15 612 on the affected modules when hammering the exact same set of physical memory locations. Our one-location Rowhammer attack induces roughly the same amount of bitflips (14 264) as double-sided Rowhammer (15 612), however, only 61.8 % of bitflip locations overlap. This could be explained by the one-location Rowhammer pattern inducing bitflips due to the Rowhammer as well as the Rowpress effect, making it actually a *Presshammer* attack. We did observe significantly less bitflips with the original Rowpress pattern by Luo et al. [26] that accesses multiple offsets per row than with one-location Rowhammer, even though both keep a row open for a prolonged time.

Based on our observed Presshammer bitflips, we develop a novel one-location Presshammer approach, which requires almost no physical address information and show that an attacker can mount an end-to-end attack by using the same-bank same-row side channel. We demonstrate that our one-location Presshammer attack is practical by obtaining kernel privileges through a page-table corruption attack [44]. The attack runtime with one-location Presshammer is less than 10 minutes on average.

In summary, we make the following contributions in this work:

1. We systematically compare one-location and double-sided Rowhammer and Rowpress.

2. We study the prevalence of both effects across 12 DDR4 modules and find that 6 of these modules are affected by double-sided Rowhammer, but only 2 by all attack variants, with double-sided Rowhammer flipping the most locations and one-location Rowpress the fewest.

3. We show that our one-location Rowhammer pattern flip bits at locations only partly overlapping with the locations flipped by double-sided Rowhammer. 61.8 % of bitflip locations flipped by one-location Rowhammer also flipped with double-sided Rowhammer, while flipping roughly the same number of bits overall, 14 264 and 15 612 respectively.

4. We present the first one-location Rowpress end-to-end attack, reliably escalating to kernel privileges within less than 10 minutes of attack time.

Outline. Section 2 provides background on DRAM, Rowhammer, and Rowpress. Section 3 presents a systematic comparison between Rowhammer and Rowpress, including our prevalence study. Section 4 shows how one-location Rowpress can be mounted without physical address information. Section 5 presents a PoC attack on page tables, allowing an attacker to escalate to kernel privileges. We discuss mitigations in Section 6 before concluding in Section 7.

2. Background

This section provides background on DRAM, Rowhammer, and Rowpress.

2.1. DRAM

DRAM is the main memory used in modern computers. DRAM capacities have increased substantially with shrinking process sizes from a few kilobytes in the 1990s to multiple gigabytes in personal computers and even terabytes in larger server systems today. The DRAM has a very high latency compared to the processor, and a single DRAM chip also has a limited bandwidth. Consequently, modern DRAM has a significant amount of parallelism to improve the bandwidth, positively influencing the average latency. DDR4 has 16 DRAM banks that operate in parallel, combined in a rank. There are single- and dual-rank DDR4 modules. Finally, the DRAM modules are connected to the processor via so-called channels. Today, dual-channel is the most widely used configuration in personal computers. Server CPUs typically have four or eight channels. Banks are divided into *rows* of *cells*, *i.e.*, the transistors and capacitors storing the data. As DRAM cell charges deplete over time, frequent refreshes are required to prevent data loss [18]. Typically, each cell is refreshed every 64 ms. Rowhammer mitigations like TRR can trigger additional row refreshes if they detect Rowhammer attacks. To access data, the memory controller has to activate the row, which moves the data from the row into the row buffer from where it can be read and written.

2.2. Rowhammer

In 2014, Kim et al. [21] discovered the *Rowhammer effect*. An attacker can induce bitflips in the DRAM from software through disturbance errors triggered by frequent row activations. Rowhammer attacks have been demonstrated in sandboxed environments [34, 44], in native environments [10, 44, 45], in virtual machines [16, 37, 53], in JavaScript [3, 11, 38], on mobile devices [8, 23, 48], over the network [25, 46], to influence or steal neural networks [36, 47, 55], and to steal cryptographic keys [7, 24, 28]. Rowhammer is getting worse with every new DRAM generation and increasing density. LPDDR4 requires less than a 10th of the activations than DDR3 memory to hammer [20, 30].

The Rowhammer effect induces a bitflip by discharging the capacitor in a neighboring cell. Once a cell has lost sufficient charge, the value read during the next load will no longer be the original value [21, 24, 50]. A a correlation exists between the data in the aggressor rows and the resultant bitflips; typically, the victim row adopts the values of the aggressor row.

There are different variants of Rowhammer: double-sided Rowhammer [44], single-sided Rowhammer [21], one-location Rowhammer [10] and half-double Rowhammer [23]. Double-sided Rowhammer uses two rows called aggressors in the same DRAM bank, which sandwich another row in between, inducing more bitflips than single-sided Rowhammer. Half-double Rowhammer hammers one row further apart than double-sided Rowhammer and is dependent on the additional refreshes performed by the TRR mitigation to induce bitflips.

Gruss et al. [10] introduced one-location Rowhammer intending to defeat the mitigations published at the time. The significant difference from the other Rowhammer variants is that one-location hammering only accesses a single location in the DRAM, which should not cause row conflicts. Instead, the aggressor is kept open through repeated accesses to the exact location until it gets closed by the memory controller. When the row is closed, one-location Rowhammer reopens the row immediately. Gruss et al. [10] hypothesized that this reopening might be the underlying trigger to the disturbance effect they discovered.

Rowpress. Rowpress was discovered by Luo et al. [26]. While it is similar to Rowhammer, it induces bitflips in rows through continuous and sustained activation of the same rows. Luo et al. [26] verified that bitflips are induced by holding the rows open for a long time using a

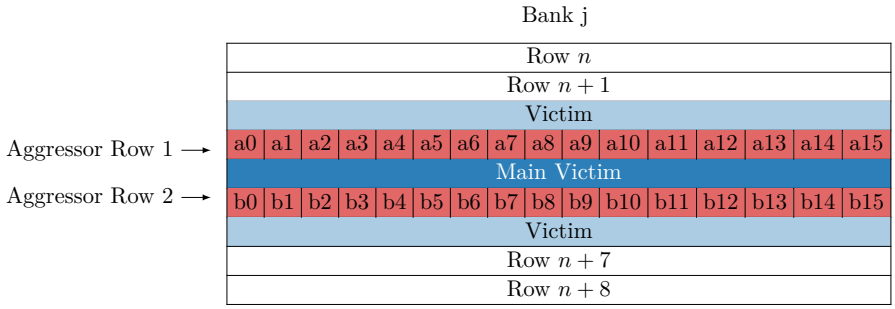


Figure 7.2.: Rowpress activates the 16 addresses in the first aggressor row (a0–a15) one after another. Then, all 16 addresses in the second aggressor row (b0–b15) are accessed. This keeps the rows open for a long period of time, causing bitflips due to the passing gate effect.

specialized FPGA platform. As the memory controller of a CPU would normally instead close the row soon after the memory access, Luo et al. [26] use a unique access pattern with different offsets in the same row. This causes the memory controller to delay precharge commands up to a maximum of $9 \times t_{REFI}$. In a Rowpress attack, the attacker chooses offsets and repetitions such that the $9 \times t_{REFI}$ delay is approached. Luo et al. [26] find that the Rowpress read disturbance phenomenon is widespread across numerous DRAM modules and manufacturers. Their paper also analyzed and compared the flipped bits caused by Rowpress and the ones caused by Rowhammer. Their data shows that the flipped bits differ for both attacks; therefore, Luo et al. [26] conclude that Rowpress is caused by another effect than Rowhammer. This effect is either called *RAS clobber* [15, 52] or the *passing gate effect* [12].

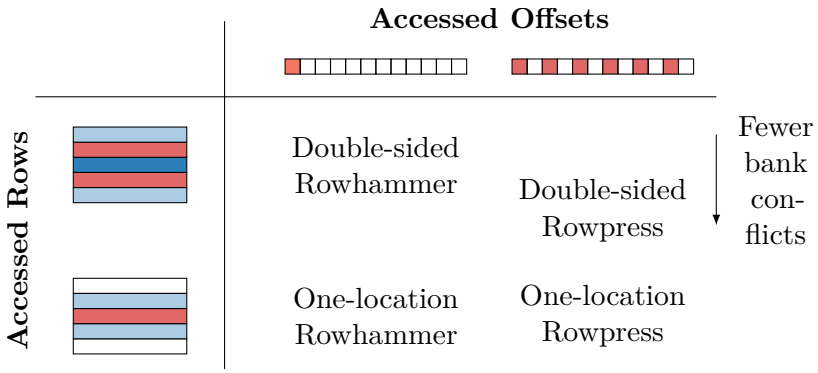


Figure 7.3.: Overview of the tested variants. Rowhammer attacks access only a single offset per DRAM row. Typically, this is offset 0. Rowpress accesses multiple offsets per DRAM row to keep the row open longer. The double-sided variants access two rows with the victim sandwiched between, however, with a lot lower frequency than double-sided Rowhammer. The one-location variants access only a single row, not explicitly causing bank conflicts.

3. Systematic Comparison of Rowhammer and Rowpress

The key difference separating one-location hammering from other Rowhammer methods is that it does not provoke bank conflicts with other rows in the same DRAM bank, *i.e.*, it is hammering only one location [10]. Similarly, Rowpress also works by keeping a row open as long as possible. With double-sided Rowpress, the open row is switched after some time, causing row conflicts but less than double-sided Rowhammer. One-location Rowpress keeps, like one-location Rowhammer, one row open. Figure 7.3 visualizes these similarities and differences.

Luo et al. [26] reported that different cells are affected by Rowpress than Rowhammer, hinting at another underlying cause. Rowpress exploits the passing gate effect [12] not the Rowhammer effect. Comparing the cells flipped by Rowpress, double- and one-location Rowhammer shows a less clear picture. While one-location Rowhammer and double-sided Rowhammer flip roughly the same number of bits, the flip locations only overlap by 61.8 %. This hints that one-location Rowhammer can induce bitflip with both effects, Rowhammer and Rowpress.

While the Rowpress paper suggested that one-location Rowpress is possible, it is essential to note that the published Rowpress PoC only contains the double-sided Rowpress variant [26]. Therefore, we used the double-sided Rowpress PoC to develop a one-location Rowpress PoC which presses in only one single location.

3.1. Differences Between One-Location Rowhammer and Rowpress

There are multiple differences between the shown one-location Rowhammer by Gruss et al. [10] and Rowpress by Luo et al. [26]. Listings 7.1, 7.2, 7.3, and 7.4 show pseudo-codes of the different Rowhammer and Rowpress methods.

Number of Accessed Rows. One-location Rowhammer accesses only a single row in one DRAM bank. While Luo et al. [26] mention Rowpress with a single row "one-location Rowpress", their PoC code performs only double-sided Rowpress. We modify this double-sided Rowpress code to support one-location pressing.

Number of Cache Block Reads. One-location Rowhammer accesses only one cache block in a row repeatedly. Typically, it is the first offset in the row. Rowpress accesses multiple cache blocks in a row.

TRR Evasion. Gruss et al. [10] did not actively evade TRR in their one-location implementation. Luo et al. [26] employ a simple TRR evasion similar to TRRespass [9]. They synchronize their hammer loop to `REF` and access dummy rows on the same bank to decrease the chance that TRR tracks the correct aggressor rows. However, as we show in Section 3.6, we were not able to reproduce Rowpress with this simple TRR evasion method.

3.2. Comparison Methodology

The goal is to record the exact cells that flip when hammering or pressing the same physical range in memory. We run the following four Rowhammer and Rowpress methods to find out whether the same or different cells flip with these methods: one-location Rowhammer (OL RH), single-sided Rowpress from now on called one-location Rowpress (OL RP), double-sided Rowhammer (DS RH), double-sided Rowpress (DS RH).

To be able to compare the flipped cells we have to ensure that we always hammer the exact same physical memory range. We use a 1 GiB huge page configured with Linux kernel command-line parameters. From this huge page we use the first 100 MiB region for all our tests. We used multiple CPUs to hammer and press all our DIMMs: Intel Core i7-6700K, Intel Core i9-9900K, Intel Core i9-10900K. Our study comprises 12 different DRAM DIMMs without ECC. Like Luo et al. [26] we disabled the cache prefetcher for all experiments[2].

Our goal is not to find the best hammer method in terms of row accesses or time passed for the attack. We want to find all cells flippable with each method to be able to measure the overlap between methods as an indication of how separated the effects are that they trigger. Therefore, we hammer our 100 MiB region multiple times with enough accesses per hammer loop to ensure we found the bulk of all flippable cells for each method. For each bitflip, we store the physical address, exact bit location, and data before and after the flip.

To find out whether one-location Rowpress flips different bits than one-location Rowhammer, we run the latter with every page offset that we also use for one-location Rowpress. We verify that the hammered offset within a row accessed does not influence the flips in the neighboring victim rows.

3.3. One-Location Rowhammer

```
1 for (iter = 0 ; iter < NUM_ITER ; iter++):
2   for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
3     // access a single cache block of the aggressor row
4     // to keep the aggressor row open longer
5     for (j = 0 ; j < NUM_READS ; j++):
6       *AGGRESSOR[x];
7       clflushopt (AGGRESSOR[x]);
```

Listing 7.1: One-location Rowhammer code.

One-location Rowhammer repeatedly accesses one offset `x` in a single row. Varying this offset does not influence the flipped bits. We were only able to reproduce one-location Rowhammer on two identical DIMMs. They are two DDR4 modules with with an early TRR implementation.

[2]Luo et al. [26] only disabled the prefetcher during initial experiments to verify that accessing multiple offsets in the same row increases the t_{AggON}, not for their bitflip experiments.

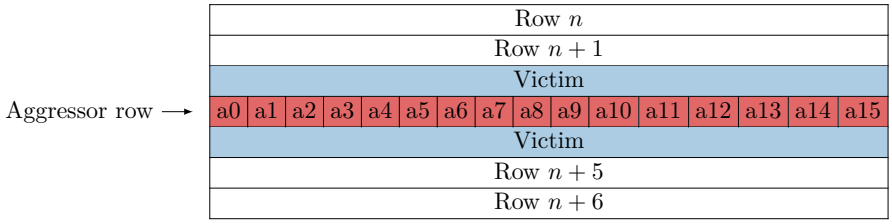


Figure 7.4.: Abstract representation of one-location Rowpress. We activate addresses a0 to a15 repeatedly to induce bitflips without provoking row conflicts.

3.4. Rowpress

We use the improved Rowpress algorithm published in the extended version [27] of Rowpress [26] in Appendix G. It interleaves the aggressor accesses and flushes. Luo et al. [27] measured four times as many induced bitflips than with the original version that first accessed all aggressors and then flushed all of them.

```
1 for (iter = 0 ; iter < NUM_ITER ; iter++):
2   for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
3     // access multiple cache blocks of the aggressor row
4     // to keep the aggressor row open longer
5     for (j = 0 ; j < NUM_READS ; j++):
6       *AGGRESSOR[j];
7       clflushopt (AGGRESSOR[j]);
```

Listing 7.2: The one-location Rowpress code used for this evaluation.

One-location Rowpress accesses only offsets within a single row, as shown in Listing 7.2 and Figure 7.4. We were only able to reproduce one-location Rowpress on two identical DIMMs, the same where we reproduced one-location Rowhammer.

```
1 for (iter = 0 ; iter < NUM_ITER ; iter++):
2   sync_with_ref();
3
4   for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
5     // access multiple cache blocks in each aggressor row
6     // to keep the aggressor row open longer
7     for (j = 0 ; j < NUM_READS ; j++):
8       *AGGRESSOR1[j];
9       clflushopt (AGGRESSOR1[j]);
10    for (j = 0 ; j < NUM_READS ; j++):
```

```
11        *AGGRESSOR2[j];
12        clflushopt (AGGRESSOR2[j]);
13
14    perform_dummy_row_accesses();
```

Listing 7.3: The double-sided Rowpress code used for this evaluation.

The double-sided Rowpress code accesses offsets in two rows in the same bank. The code by Luo et al. [26] performs simple TRR evasion. They are syncing the row accesses to `REF` and access some dummy rows after the main row accesses, as shown in Listing 7.3. We found that this evasion is not enough for all but the same two DIMMs where we were able to reproduce the one-location methods.

3.5. Double-Sided Rowhammer

```
1 for (iter = 0 ; iter < NUM_ITER ; iter++):
2   for (i = 0 ; i < NUM_AGGR_ACTS ; i++):
3     // frequently open and close each aggressor row
4     for (j = 0 ; j < NUM_READS ; j++):
5       *AGGRESSOR1[0];
6       *AGGRESSOR2[0];
7       clflushopt (AGGRESSOR1[0]);
8       clflushopt (AGGRESSOR2[0]);
```

Listing 7.4: The double-sided Rowhammer code used for this evaluation.

To hammer DRAM with the double-sided pattern, we used Blacksmith by Jattke et al. [17]. Blacksmith is the most sophisticated TRR evasion method to date. With Blacksmith we successfully hammered 6 of our 12 tested DIMMs.

3.6. Results

Table 7.1 shows an overview of all tested DIMMs and the successful hammer and press methods. We were only able to induce bitflips with Rowpress patterns on two of our tested DIMMs, D1 and D2. On DIMMs D3, D4, D6, and D9, Blacksmith [17] found double-sided Rowhammer patterns that induced bitflips. We did not try the one-location variants

Table 7.1.: The table shows whether a Rowhammer or Rowpress method was able to cause bitflips in any of our tested DIMMs. We indicate success with (✓), not causing bitflips with(✗) or not tested with (∼). We did not test the one-location (OL) variants if double-sided Rowpress (DL RP) was unsuccessful.
* On D3 to D12, we used Blacksmith to hammer double-sided with TRR evasion.

	DIMM	Size	MT/s	Double Sided		One Location	
				RH*	RP	RH	RP
DDR4	D1	8 GB	2 133	✓	✓	✓	✓
	D2	8 GB	2 133	✓	✓	✓	✓
	D3	8 GB	2 400	✓	✗	∼	∼
	D4	8 GB	2 400	✓	✗	∼	∼
	D5	8 GB	2 133	✗	✗	∼	∼
	D6	8 GB	2 666	✓	✗	∼	∼
	D7	8 GB	2 666	✗	✗	∼	∼
	D8	8 GB	2 666	✗	✗	∼	∼
	D9	8 GB	2 666	✓	✗	∼	∼
	D10	8 GB	2 666	✗	✗	∼	∼
	D11	8 GB	3 000	✗	✗	∼	∼
	D12	8 GB	2 133	✗	✗	∼	∼

of Rowhammer and Rowpress if we did not achieve bitflips with both double-sided variants.

Our theory on why we only saw bitflips induced by the Rowpress pattern on 2 DIMMs is that TRR on modern DDR4 DIMMs has evolved to be more difficult to evade. Blacksmith can only do it by accessing a very large number of aggressor rows. For example, the best pattern on D3 repeatedly accesses 60 different rows during one refresh interval to evade TRR. However, as Rowpress tries to keep a row open for a long time, fewer row activations are possible per refresh interval. Luo et al. [26] caused Rowpress bitflips on a DIMM manufactured in 2018. Our DIMMs D1 & D2 are of similar age. Whether a blacksmith-like TRR evasion with Rowpress is possible on modern DDR4 memory, is up to future work.

Table 7.2 shows the detailed results of all DIMMs where we induced bitflips. On D1, we hammered the same physical address space with all methods listed. On D3, D4, D6, and D9, we used the best pattern found by Blacksmith.

Table 7.2.: The number of bitflips per 100 MB of memory. RP-8 and RP-16 donate Rowpress patterns with 8 and 16 cache line accesses per row.
* On D3 to D11, we used Blacksmith to hammer double-sided for TRR evasion.

DIMM	Double Sided			One Location		
	RH*	RP-8	RP-16	RH	RP-8	RP-16
D1 & D2	15 612	2 174	4 264	14 264	161	0
D3	13 656	0	0	0	0	0
D4	15 786	0	0	0	0	0
D6	157	0	0	0	0	0
D9	3 040	0	0	0	0	0

On DIMM D1 and D2, we were able to flip bits with all tested methods. Most importantly, we induced bitflips in the same physical address range with all methods. This allows us to analyze precisely whether the tested Rowhammer and Rowpress methods flipped bits in the same cells on this DIMM. We also verified that the offset we access within a row does not influence the bitflips we see when hammering. Figure 7.5 shows a confusion matrix of all bitflips caused by Rowhammer and Rowpress. Because Rowpress with 8 accesses per row also causes bitflips with 16 offsets, we use its numbers for this analysis.

Based on our results it remains inconclusive, whether one-location Rowhammer [10] is, in fact, Rowpress. We measure roughly the same amount of bitflips when hammering with a double-sided and one-location pattern. However, only 61.8 % of bitflip locations found by one-location Rowhammer flipped also with double-sided Rowhammer. The comparison between double-sided and one-location Rowpress also shows a reduction in the number of bitflips and a difference in bitflip locations. Comparing Rowpress and Rowhammer variants, we find that virtually all Rowpress bitflip locations are in the sets of both Rowhammer variants. However, this is likely also influenced by the smaller number of bitflips we found with all Rowpress variants. A possible explanation for our results could be that one-location Rowhammer causes both effects simultaneously very effectively, actually making it a *Presshammer* attack. It keeps a single row open, inducing bitflips due to RAS clobber in some cells, while the opening and closing of the row that is still happening is enough to induce bitflips in other cells.

		RH		RP	
		DS	OL	DS	OL
RH	DS	100	56.5	13.7	1.0
	OL	61.8	100	14.5	1.1
RP	DS	98.1	95.3	100	1.0
	OL	96.3	98.8	81.4	100

Figure 7.5.: Detailed results of DIMM D1 & D2. Each number represents the fraction in percent of exact bitflip locations of one method (y-axis) found in the other method (x-axis). For example, 61.8 % of bitflip locations flipped by one-location Rowhammer is also in the set of locations flipped by double-sided Rowhammer.

Figure 7.6.: The offsets within a DRAM row are parts of different memory pages `A` to `D`. Each part has the size of two cache lines, 128 bytes. This example is based on the DRAM addressing functions of a Skylake CPU [33].

4. Unprivileged One-Location Presshammer Attack

As our one-location Rowpress follows the same semantics as one-location Rowhammer, we can similarly reduce the requirements. Most Rowhammer attacks require at least partial knowledge of physical address bits above the page offset. Therefore, the OS hides physical address information from user programs to respond to these Rowhammer attacks [22]. While it is possible to reverse engineer the full DRAM addressing functions [1, 33, 51] and additionally uncover row addressing and physical address bits [23, 24, 43, 45] using the bank conflict timing side-channel, these methods can be tedious and take time.

For our one-location Presshammer attack, we only need the addresses of all offsets that map to the same row in the same bank. Due to the DRAM addressing functions, one page does not map to one row. On virtually all

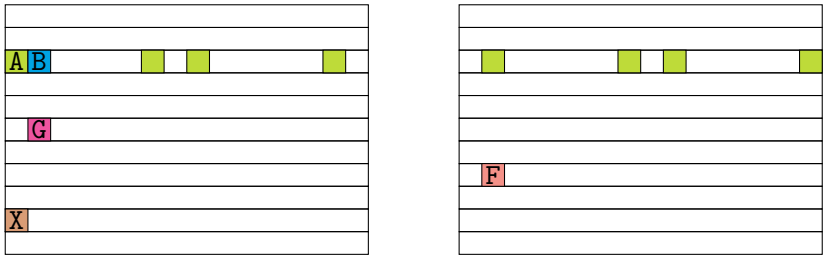


Figure 7.7.: Visualization of the two requirements to find page offsets in the same row. `F[1]` does not have a bank conflict with `X[0]`, violating **R1**. `G[1]` does have a bank conflict with `X[0]` but also with `A[0]`, violating **R2**. `B` fulfills both requirements.

recent systems, different logical pages are interleaved on one row, as shown in Figure 7.6, based on the DRAM addressing functions [33], to increase memory throughput. Thus, the missing parts of each page are simply mapped to other rows in other banks. To perform a Rowpress attack, we must access multiple offsets within one row, spread across various pages. Instead of using the physical address information to find these offsets in the correct locations, we use the bank-conflict timing side-channel in a reduced form.

Taking a random page `A`, we want to find the pages `B` to `D` that map to the same row. Because we work on virtual memory and do not have any knowledge about the underlying mapping to the physical memory, close and "far away" pages based on the virtual address are not necessarily reflected in the physical addresses. We increase this chance by allocating a large chunk of memory. After filling single-page holes in the physical memory, the buddy allocator in Linux typically returns chunks of physically contiguous memory [23]. This increases the performance of our approach but is not strictly necessary.

Two assumptions for our algorithm hold on all current x86 systems:

A1 A DRAM row is 8 KiB (2^{13} B) large.

A2 The physical address bits used for row addressing start high enough to have a single page always in the same row across banks, *i.e.*, a page is never in two rows in the same bank.

The algorithm also works with 4 KiB rows with tiny modifications.

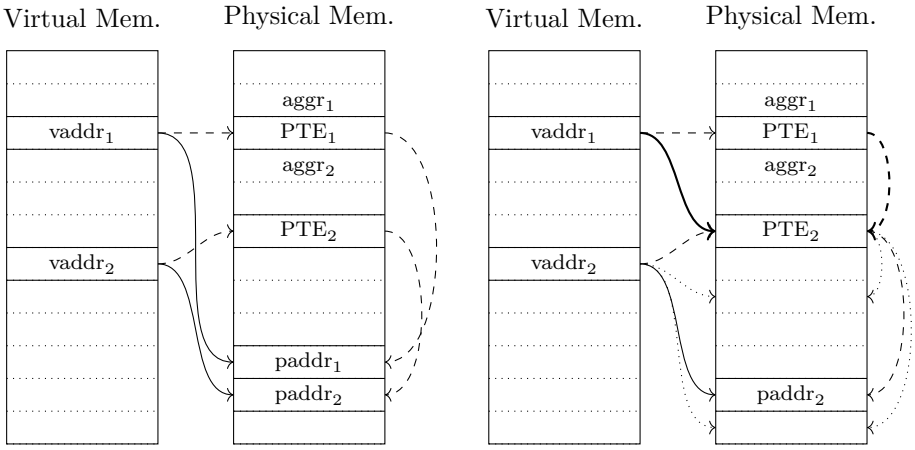
First, we have to find a page `X` that is in the same bank as `A`. Whether page `A` and `X` are in the same bank is easily detectable with the bank-conflict timing side channel. To not interfere with the following search for pages `B` to `D`, we start the search for page `X` a few hundred virtual pages away from page `A`. We denote current bank-conflict candidate page as `S`. We repeatedly flush and reload `A[0]` and `S[0]` while measuring the timing to detect a bank conflict. With page `A` and `X`, finding pages `B` to `D` is based on two requirements, also shown in Figure 7.7:

R1 Because `B` to `D` must be in the same row as page `A`, they must be in the same bank as `X`. We search for pages that have a bank conflict (same bank, but a different row) with page `X`.

R2 Because `B` to `D` are in the same row as page `A`, there must not be a bank conflict between `B` to `D` and `A`.

We iterate over chunks of 64 B to find `B` to `D`, starting with an offset of 64. We check **R1** by flushing and reloading `S[offset]` and `X[0]`. We try pages above and below `A` until we find the page conflicting with `X`. Then, we check **R2** by flushing and reloading `S[offset]` and `A[0]`. If there is no conflict, we found the page mapped to the first offset. We repeat these steps until all offsets are filled. If we found `B` to `D` once, checking the following offsets is quick because we no longer have to search for new pages. During the search, we also mark already "used" offsets to speed up the search the more rows we already found.

Because we exclusively work with virtual memory, there is a chance that any of the required pages to fill a row is not mapped in our address space. However, as shown by Luo et al. [26], Rowpress does not need all offsets to press efficiently, and the offsets we find are sufficient to mount Rowpress. However, the rate of bitflips will be reduced as we do not control the data for the entire row, which is known to influence the number of bitflips [5, 24].



(a) Correct Mapping (b) Successful Exploit

Figure 7.8.: A simplified example of page table flipping with only one level of page tables is shown. After successful exploitation (b) by hammering PTE_1, PTE_2 is writeable through $\texttt{vaddr}_1$. This makes the whole physical memory accessible by changing PTE_2 and then accessing the new page through $\texttt{vaddr}_2$.

5. Proof-of-Concept Presshammer Attack on Page Tables

Luo et al. [26] did not present an end-to-end Rowpress exploit, as their focus was on the scientific evaluation of the effect, not the security impact. Consequently, they did not discuss a threat or attacker model and simply used 1 GiB huge pages to demonstrate the effect on real systems. We bridge this gap by showing that Rowpress can be used to attack a system without using 1 GiB or 2 MiB huge pages to get additional physical address information or any knowledge of the DRAM mapping functions.

We use Presshammer to attack operating system page tables based on the exploit first shown by Seaborn et al. [44] and many following [5, 11, 17, 23, 48, 53, 57]. The proof-of-concept exploit of Seaborn et al. [44] still relied on full access to the virtual to physical address translations, while we use the algorithm from Section 4 to get all the information we need. The attack aims to use a bitflip to change a page frame number (PFN) in a page table to point to another page table. This gives the attacker write access to one of its own page tables and, through it, access to the whole

Table 7.3.: Results of our row finding algorithm using a different number of measurements for the bank-conflict side channel. There are 128 offsets per row. We search for 10 000 rows and discard the first 100. The search radius is ±50 pages.

# Measurements	Found Cache Lines	Correct Cache Lines
250	124, $\sigma = 4.9$	119, $\sigma = 14$
500	128, $\sigma = 0.8$	125, $\sigma = 16$
1000	125, $\sigma = 4.2$	123, $\sigma = 4.3$
2500	125, $\sigma = 4.9$	124, $\sigma = 5.2$

# Measurements	Duration per Row
250	40 ms, $\sigma =$ 17
500	45 ms, $\sigma =$ 15
1000	116 ms, $\sigma =$ 40
2500	260 ms, $\sigma = 109$

memory. Figure 7.8 shows the mappings before and after a successful exploitation. The exploitation technique for Presshammer does, at large, not change from the Rowhammer exploit. We ran the exploit on an Intel Core i7-6700K with Ubuntu 20.04.4 LTS using Linux version 5.4.210.

DRAM Row Finding. We use the algorithm described in Section 4 to find the aggressor rows. On our machine with 16 GiB of memory, 4 GiB were used by the operating system, and the exploit allocates the remaining 12 GiB of memory. When allocating enough memory, the buddy allocator of the Linux kernel starts to return chunks of physically contiguous memory [23, 48].

A single measurement is insufficient to get a definitive result because of system noise when using the bank conflict side-channel. Software on other cores of the CPU also accesses the memory. However, more measurements take longer, which is relevant if we want to check many rows for bitflips.

We ran the algorithm with 250 to 2500 measurements per address and analyzed the correctness of the result, which can be seen in Table 7.3. Taking 500 measurements gives the best results. We see that the standard deviation for the duration increases significantly for a higher number of measurements. This suggests more interrupts from the operating system that worsen the result.

Memory Templating. We verified that Rowpress bitflips are repeatable, like Rowhammer bitflips [21]. That means if an aggressor row flips a bit in a neighboring victim row, pressing this aggressor will flip the same bit again if the flip direction stays the same. This helps because we can template the memory in a first step, the exploit searches for aggressors that flip bits in exploitable locations. Bitflips are exploitable if they flip bits in the PFN range of a page table entry (PTE). We count bits 12 to 32 of a PTE (bits 0 to 20 of the PFN) as exploitable; this corresponds to a neighboring page up to a page 4 GiB away.

The exploit searches 10 aggressor victim pairs before continuing. This gives a higher exploit probability later because Rowpress can, like Rowhammer, only flip bits in one direction. Due to that, there is only a 50 % chance that an aggressor-victim pair works when filled with an actual page table entry.

The cache prefetcher in the CPU [14] tries to predict which addresses will be loaded next and fetches them into the cache for increased performance. Luo et al. [26] turned off the prefetcher for their experiments[3]. However, this is not possible with our threat model. We evade the prefetcher by shuffling the offsets we access. This decreases the chance that the prefetcher can detect a pattern in our accesses.

Page Table Spraying. Filling most of the memory with page tables is the easiest way to increase the chances of successful exploitation in two ways. First, it increases the chance of putting a page table in one of the previously found victim rows. Second, it increases the chance of a changed PFN pointing to another page table. Van der Veen et al. [48] used a more deterministic technique to place the page table called *Phys Feng Shui*. They place a page table in a victim page by precisely massaging the Linux page allocator. However, for this Presshammer PoC exploit, we chose the more straightforward spraying method.

Page table spraying is performed by mapping the same shared memory file repeatedly. This puts the shared memory file's content into the memory only once, while having to create new page tables for every mapping. After finding the aggressor and victim pairs, the exploit still has most of the physical memory reserved from the templating phase. Every time before mapping the shared memory file again, the exploit frees exactly as many

[3] Luo et al. [26] only disabled the prefetcher during initial experiments to verify that accessing multiple offsets in the same row increases the t_{AggON}, not for their bitflip experiments.

random pages as required for the newly to-be-created page tables. After creating around 80 % of all page tables, the exploit frees the victim rows one after another. When the page table spraying is finished, as much memory as possible is filled with page tables.

Page Table Flipping. After spraying the page tables, the exploit presses the aggressor rows one after another to induce bitflips in the victim PFNs. Because the victim rows are not mapped anymore, we cannot simply check for a bitflip. Instead, the exploit checks all mappings of the shared memory file if the content is the expected content. If not, a PFN was changed. If the content looks like a page table, the exploit changes one page-table entry of the page table and checks through all mappings again. When finding another page with changed content, the exploit found $\texttt{vaddr}_1$ and $\texttt{vaddr}_2$, as shown in Figure 7.8b.

Exploitation. With all previous steps performed successfully, the exploit now has full read and write access to the whole physical address space. The original proof-of-concept exploit code from Seaborn et al. [44] used this access to patch a victim binary and change its behavior. For a real exploit, this could be any binary owned by root with the `setuid` sticky bit set to run code with superuser privileges. Without access to the virtual to physical address translations, the exploit has to scan the memory for the victim data it wants to attack. The exploit can also search for exploitable kernel structures to, e.g., give the process or user higher privileges or directly inject attacker-controlled code into the kernel. For our proof-of-concept exploit, we just dump the physical memory into a file.

While reading the whole physical memory, it is crucial to evict the TLB entry of the PTE modified by the exploit. If the exploit does not evict the TLB entry, all reads from $\texttt{vaddr}_2$ will return the same data. To evict it, we keep half of the mappings we used during the exploit and iterate over them. We free the other half to relieve the memory pressure on the system and prevent an out-of-memory situation. Dumping the memory to disk is limited by the disk write bandwidth.

We ran the exploit 5 times on our machine. Of these 5 runs, 3 were successful. It took, on average, 513 s (n=3, σ=125) to gain full access to the physical memory. One run was unsuccessful because the operating system did not put a page table in a victim row or no bit flipped in the PFN. The other run was unsuccessful because it did not point to another of our page tables after changing a PFN by row pressing it. If unsuccessful, the exploitation can just be tried again.

6. Discussion of Mitigations and Related Work

The large number of Rowhammer attacks published [3, 5, 7, 8, 10, 11, 16, 23, 24, 25, 28, 34, 36, 37, 38, 44, 45, 46, 47, 48, 53, 55] motivated research on mitigations for DRAM disturbance errors. While many have been proposed in the academic community [4, 13, 19, 29, 31, 32, 39, 40, 41, 42, 49, 54, 56], only a few, like TRR [9], have found their way into practice. As Luo et al. [26] noted, Rowpress is also mitigated by TRR and other mitigations. However, they also note that similar to Rowhammer attacks bypassing TRR [9, 17, 23], Rowpress can also bypass it. And the same is true for many other proposed countermeasures. As CPUs cannot keep a row open longer than $9 \times t_{REFI}$, Rowpress always also causes a high number of row activations similar to Rowhammer. We show in this paper that this makes the two attacks hard to distinguish in practice, but this also means that we expect most countermeasures that are effective against current Rowhammer attacks to be also effective against Rowpress. In the worst case, with some additional tweaking to the detection threshold values. However, up to now, every deployed Rowhammer mitigation has been evaded eventually, and there is no guarantee that this will not be the case for Rowpress as well.

Our findings on the overlaps between Rowhammer and Rowpress effects indicate again [10] that mitigations must be principled not to be bypassable. That is, they cannot focus on the specific effect triggering the disturbance but must focus on preventing the effect of the disturbance. Mitigations that use this approach are, for instance, Error Correction Code (ECC) [5] and Chipkill [6]. ECC uses 8 redundancy bits per 64 bits (DDR4) or 32 bits (DDR5) of data. Chipkill can correct up to 8 bitflips, e.g., due to a chip failure [6]. However, ECC has already been bypassed with Rowhammer attacks, and Chipkill suffers from a significant overhead. As Qureshi [35] noted, it is essential to take an entirely new approach to mitigate Rowhammer and other upcoming DRAM disturbance effects. The earliest work in this direction is IVEC [13], which uses a hash instead of an error-correcting code and flips bits in corrupted memory until the hash matches again. With a similar approach, Saileshwar et al. [39] also propose a modification to ECC memory using a MAC instead of the error-correcting code to detect and correct a broader range of bitflips. Juffinger et al. [19] extended this approach into an error-correction system based on lightweight cryptography that effectively transforms DRAM disturbance errors from reliability issues into performance penalties, which,

in the worst case, lead to a denial of service. Recently, Olgun et al. [29] introduced a Rowhammer mitigation based on a single per-row hardware counter to track row activations across all banks. This minimizes the hardware overhead while solving the issue of a small number of counters in TRR.

7. Conclusion

Many works analyzed DRAM disturbance effects and reported a wide range of prevalence estimations. In our set of 12 DDR4 modules, we can reproduce double-sided Rowhammer on 6 modules. All other Rowhammer and Rowpress variants only worked on 2 modules. For a fair comparison between the attack variants, we evaluated the number of unique bitflip locations for all variants, ranging from 161 to 15 612. However, 95.3 % to 98.8 % of Rowpress unique bitflip locations are also flipped by double-sided and one-location Rowhammer. This surprising result raises questions about the difference between the effects these attack implementations trigger and indicates that they might partially exploit the same root cause. However, this result also allows us to craft the first unprivileged end-to-end one-location Rowpress attack, which uses the same-row same-bank side channel to escalate to kernel privileges within less than 10 minutes. Our work shows that further research is necessary to better understand the differences between Rowhammer and Rowpress implementations on real systems and their implications for security.

Acknowledgments

We thank Lukas Gerlach, Andreas Kogler, Stefan Gast, Patrick Rademacher, Sarah Rinderer and the reviewers for their feedback and help. This research was supported by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), the Deutsche Forschungsgemeinschaft (DFG) (503876675) and by the European Union (ROF-SG20-3066-3-2-2). Additional funding was provided by generous gifts from Red Hat and Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] 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. 217).

[2] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In: CRYPTO. 1997 (p. 204).

[3] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P. 2016 (pp. 208, 224).

[4] 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. 224).

[5] 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. 219, 220, 224).

[6] Timothy J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. Tech. rep. IBM Microelectronics, 1997 (p. 224).

[7] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. In: CCS. 2022 (pp. 208, 224).

[8] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (pp. 208, 224).

[9] 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. 204, 211, 224).

[10] 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. 204, 205, 208, 210, 211, 216, 224).

[11] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 208, 220, 224).

[12] 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. 204, 209, 210).

[13] Ruirui Huang and G. Edward Suh. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. In: ISCA. 2010 (p. 224).

[14] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (p. 222).

[15] Yutaka Ito and Yuan He. Apparatus and Methods for Refreshing Memory. U.S. Patent 11062754B2. 2019 (pp. 204, 209).

[16] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (pp. 208, 224).

[17] 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. 204, 206, 214, 220, 224).

[18] JEDEC Solid State Technology Association. Low Power Double Data Rate 4. 2017. URL: `http://www.jedec.org/standards-documents/docs/jesd209-4b` (p. 207).

[19] 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 (p. 224).

[20] 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 (p. 208).

[21] 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. 204, 205, 208, 222).

[22] 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. 217).

[23] 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. 208, 217, 218, 220, 221, 224).

[24] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (pp. 208, 217, 219, 224).

[25] 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. 208, 224).

[26] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlikçi, 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 (pp. 204–206, 208–215, 219, 220, 222, 224).

[27] Haocong Luo, Ataberk Olgun, A. 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: arXiv:2306.17061 (2024) (p. 213).

[28] Koksal Mus, Yarkin Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering TLS Signing Keys via Rowhammer Faults. In: S&P. 2023 (pp. 208, 224).

[29] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, Abdullah Giray Yaglikci, Geraldo F Oliveira, and Onur Mutlu. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation. In: USENIX Security. 2024 (pp. 224, 225).

[30] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In: MICRO. 2021 (p. 208).

[31] Jin Hyo Park, Su Yeon Kim, Dong Young Kim, Geon Kim, Je Won Park, Sunyong Yoo, Young-Woo Lee, and Myoung Jin Lee. Row Hammer Reduction Using a Buried Insulator in a Buried Channel Array Transistor. In: IEEE Transactions on Electron Devices (2022) (p. 224).

[32] 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. 224).

[33] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security. 2016 (pp. 217, 218).

[34] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In: HOST. 2016 (pp. 208, 224).

[35] Moinuddin Qureshi. Rethinking ECC in the Era of Row-Hammer. In: DRAMSec. 2021 (p. 224).

[36] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. In: S&P. 2022 (pp. 208, 224).

[37] 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 (pp. 208, 224).

[38] 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. 208, 224).

[39] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. Synergy: Rethinking secure-memory design for error-correcting memories. In: HPCA. 2018 (p. 224).

[40] 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. 224).

[41] 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 (p. 224).

[42] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime. In: MICRO). 2022 (p. 224).

[43] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 217).

[44] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. In: Black Hat USA. 2015 (pp. 206, 208, 220, 223, 224).

[45] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In: RAID. 2018 (pp. 208, 217, 224).

[46] 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 (pp. 208, 224).

[47] M Caner Tol, Saad Islam, Andrew J Adiletta, Berk Sunar, and Ziming Zhang. Don't Knock! Rowhammer at the Backdoor of DNN Models. In: DSN. 2023 (pp. 208, 224).

[48] 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. 208, 220–222, 224).

[49] 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. 224).

[50] Andrew J Walker, Sungkwon Lee, and Dafna Beery. On DRAM Rowhammer and the Physics of Insecurity. In: IEEE Transactions on Electron Devices (2021) (p. 208).

[51] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A Knowledge-assisted Tool to UncoverDRAM Address Mapping. In: Design Automation Conference (DAC). 2020 (p. 217).

[52] Gregg D. Wolff. Word line cache mode. U.S. Patent 10366733B1. 2019 (pp. 204, 209).

[53] 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 (pp. 208, 220, 224).

[54] A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In: HPCA. 2021 (p. 224).

[55] 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 (pp. 208, 224).

[56] A Giray Yauglikcci, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oğuz Ergin, and Onur Mutlu. HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips. In: MICRO. 2022 (p. 224).

[57] 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. 220).

8

An Analysis of HMB-based SSD Rowhammer

Publication Data

Jonas Juffinger
An Analysis of HMB-based SSD Rowhammer
In: uASC 2025

Contributions

Main Author.

An Analysis of HMB-based SSD Rowhammer

Jonas Juffinger

Graz University of Technology, Graz, Austria

Abstract

Rowhammer has been shown to be an extensive attack vector. In the years since its discovery, numerous exploits have been shown, attacking a wide range of targets from kernels, through web browsers to machine learning models. These attacks were not always mounted from code running on the CPU of a system. Various devices peripheral to the CPU, like GPUs or networks cards can cause Rowhammer bit flips through DMA accesses to the main memory.

In this work, we take a look at solid state drives (SSDs) and if they can be exploited as confused deputies to perform Rowhammer attacks. With the introduction of NVMe, a standardized protocol that allows SSDs to communicate directly over PCIe with the CPU, SSDs have reached performance numbers of a million input/output operations per second. PCIe also enables SSDs to use DMA for direct accesses to the main memory. This lead to the introduction of the host memory buffer (HMB) feature, that allows SSDs to use a small fraction of the host DRAM. We are the first that reverse engineer how different SSDs utilize this host memory buffer and answer the question if the accesses from the SSD to the HMB are a potential attack vector to cause Rowhammer bit flips.

Our analysis of three SSDs shows, that bit flips in the HMB cause the SSDs to lock up, which results in a denial of service or, even worse, data loss. We also show how we can cause frequent accesses from the SSD to the HMB on all three SSDs. On one SSD, we reach 5 000 DRAM accesses per refresh interval. We measure the Rowhammer impact of these accesses and show that they are effectively hammering the DRAM. However, 5 000 DRAM accesses are not enough to cause Rowhammer bit flips, even on modern, highly vulnerable DRAM.

1. Introduction

Since its discovery in 2014 [19], Rowhammer became a large research field, with new and more sophisticated exploits being published every year, escalating privileges [32], hammering from ARM, AMD x86 or RISC-V [12, 26, 35], modifying binaries [8], evading ECC [4], reading secrets [22], attacking the newest generation of mitigations [7, 11], using TRR as a confused deputy [20], finding a new RowPress attack [25] or hammering many banks simultaneously [14]. Researchers and the industry also published and implemented many Rowhammer mitigations [2, 3, 7, 13, 21, 30, 36]. Most attacks hammer the DRAM from code running on the CPU, either from native code or also from interpreted (and JIT compiled) languages like, for example, JavaScript in the web browser.

A few works have shown that Rowhammer attacks are also possible from devices attached to the CPU that can access the DRAM. Frigo et al. [6] hammered on ARM, utilizing the integrated GPU to load texture data from the DRAM to perform the hammer memory accesses. The big advantage of using the GPU instead of the CPU, were smaller, more easily evictable caches. Tatar et al. [34] attacked systems exploiting Remote Direct Memory Access (RDMA). RDMA allows DMA accesses over the network without involvement of the CPU with supported network cards. With networks speeds above 10 Gbit/s, up to 40 Gbit/s, the RDMA accesses were frequent enough to induce bit flips.

Solid state drives (SSDs) are the de facto standard storage medium in every new computer, in particular in mobile ones, like laptops. Highly increased speeds, especially on random accesses, lower power consumption and better physical durability make them superior to HDDs in pretty much every use case where storage space to price is not the number one priority [33]. With the NVMe standard, SSDs are now directly connected to the CPU over PCIe and are reaching over 1 000 000 IOPS with PCIe 4.0. Because of how flash memory storage works internally, logical addresses must be translated to physical addresses for each accesses. The data structure storing these translations, the flash translation layer (FTL), must therefore be quickly accessible for high performance. “Pro”-grade SSDs typically come with a DRAM chip directly next to the SSD controller to store the FTL to maximize performance. However, this is expensive. SSDs supporting the host memory buffer (HMB) feature, can use a part of the main memory to cache FTL translations [28]. For this, the SSD controller request memory from the operating system that it then uses

exclusively through direct memory accesses (DMA). This saves cost and achieves acceptable performance.

Zhang et al. [37] identified the FTL, stored in the DRAM, as a potential target for Rowhammer attacks. A bit flip in the correct location of the FTL can remap a page on the SSD, similar to how Seaborn et al. [32] remapped a memory page by flipping a bit in a virtual memory page table. This remapping could give an attacker access to an indirect mapping block of the file system that is inaccessible to unprivileged users. By editing this mapping block, the attacker gains read and write access to the whole file system. They use an emulated SSD from the Linux Foundation's (prior Intel's) Storage Performance Development Kit (SPDK) [24] for their evaluation of this attack.

In this work, we are the first to analyze actual SSDs and their vulnerability to Rowhammer bit flips as well as their potential as confused-deputy attack vectors. By mapping the memory reserved for the HMB and utilizing the IOMMU, we can precisely learn how different SSDs use the HMB during operation. We inject artificial bit flips into the cached FTL mappings in the HMB and observe that the SSD integrity checks the HMB. While this prevents remapping blocks with bit flips, we find that after injecting bit flips, the SSD stops responding to any commands until a complete power cycle is performed. During our experiments we even manage to break one SSD by changing HMB contents while discarding blocks. This would make HMB bit flips an effective denial of service attack.

For a real end-to-end exploit the bit flips must be caused by Rowhammer accesses from the SSD to the HMB. We show, how our understanding of the HMB helps us to achieve frequent HMB accesses by accessing the SSD. The least recently used (LRU) HMB cache replacement policy of the Samsung 990 EVO and Lexar NM790 allows to target specific pages for double sided Rowhammer. However on-chip cache eviction makes these accesses slow, only 700 per refresh interval. The Samsung 980 does not use LRU but a more rigid cache mapping. By accessing a pair of addresses spaced with a specific interval, the Samsung 980 seems to always access the HMB with every read from the SSD. With these address pairs, we achieve up to 5 000 accesses to the DRAM per refresh interval.

As this could be enough to see an impact on bit flip numbers, we evaluate the Rowhammer impact of these HMB accesses from the SSD, with a combined Rowhammer experiment. We prime the DRAM with conventional Rowhammer accesses from software to achieve a minimum number of

activations and then fill the rest of the refresh interval with HMB accesses from the SSD. In this experiment we do see a measurable impact from the SSD's accesses, equivalent to the Rowhammer accesses from software. However, with only 78 000 IOPS, the number of SSD accesses is not high enough to induce bit flip on their own.

Even with out attack not working, this work is the first one that looks into the HMB's role in Rowhammer attacks, both as a victim to bit flips and an actively hammering part. We show that bit flips in the HMB can have devastating outcomes, from lost data to broken SSDs. However, the risk of this currently happening is very low. Because SSDs manage their HMB in blocks of at least 4 kB and contain an additional on-chip translation cache, the overhead is too large to achieve memory accesses that are frequent enough to cause Rowhammer bit flips. However, a privileged attacker, could corrupt the HMB through the direct physical map to try to break hardware.

Contributions. In summary, we make the following contributions:

- We perform the first in-depth analysis of the host memory buffer of SSDs, show how it is used and what effects bit flips have on it.
- We exploit the HMB cache replacement policies to cause double-sided Rowhammer accesses and find special SSD access patterns that trigger a maximum amount of HMB accesses.
- We perform a combined Rowhammer experiment with software and SSD accesses, showing that the SSD accesses have measurable impact.
- Finally, we make a compelling argument, why the HMB is currently no Rowhammer security risk.

Outline. In Section 2, we provide background information on SSDs and the host memory buffer (HMB) feature. In Section 3, we give an overview of the attack and threat model. In Section 4, we reverse engineer how SSDs use the HMB and what happens if we inject artificial bit flips. In Section 5, we show how we can cause Rowhammer accesses from the SSDs but that these accesses are not fast enough to cause Rowhammer bit flips. We shortly talk about how the HMB can be disabled in Section 6 and conclude in Section 7.

2. Background

In this section we provide background on Rowhammer, solid state drives and flash memory, the flash translation layer flash memory requires and the host memory buffer that caches recently used translations.

2.1. Rowhammer

Rowhammer is a vulnerability of modern DRAM [19]. DRAM stores bits as charge in an array of capacitors. These capacitors are accessed through a transistor. Frequent accesses to DRAM rows injects electrons into the substrate, where they can travel to neighboring rows' transistors, opening them slightly. This increases the discharge rate of the connected capacitors. If the charge decreases too much before the next refresh happens, the stored bit flips. Rowhammer is a highly researched topic, with a large number of exploits [4, 6, 7, 8, 11, 12, 14, 20, 22, 25, 26, 29, 34, 35] as well mitigations [2, 3, 13, 21, 30, 36] shown in the past. For Rowhammer to cause bit flips, enough accesses must be made to the rows neighboring the victim row to cause enough discharge. This limit is called the maximum activation count (MAC). While this limit shrinks with every new DRAM generation, from more than 100 000 on DDR3 to only 10 000 on LPDDR4 [27], more advanced mitigations require more complex and frequent dummy accesses [7, 11].

2.2. Solid State Drives and NAND Flash Memory

Solid state drives (SSDs) are storage devices that use solid-state flash memory instead of spinning disks in hard disk drives (HDDs), to persistently store data. This has the big advantage that it is mechanically more robust and enables much higher random input/output operations per second (IOPS).

In contrast to HDDs, the NAND flash memory cannot be randomly written. The mechanisms to set bits to `1` (erasing) and setting them to `0` (programming) is different. In its empty state NAND memory is erased and stores all `1`'s. To write data to the memory, bits are programmed to `0`. Reprogramming is possible as long as the new data is a subset with only `1` to `0` transitions.

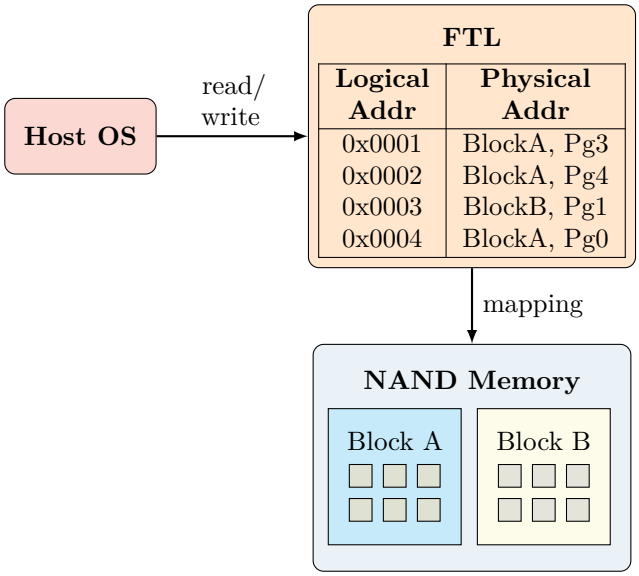


Figure 8.1.: The flash translation layer translates logical page addresses to physical page addresses. It is required because pages are scattered throughout the flash memory for performance and wear-leveling reasons.

Reading and programming typically happens in sizes of 512 B, 2 048 B or 4 096 B, called a page. However, erasures must happen in blocks of 32, 64 or 128 pages and take considerably longer. These erasures also slightly damage the NAND cells each time. Therefore, flash memory has only a limited number of so called program-erase cycles (P/E cycles). To extend their lifespan, SSDs perform wear-leveling, where the erasures are evenly distributed over all blocks.

2.3. Flash Translation Layer

To efficiently work with the different sizes for erasing and programming data and perform wear leveling, flash memory does not one-to-one map logical addresses to the actual physical addresses of the NAND memory blocks and pages. Instead, data is scattered all over the flash memory, requiring a translation layer between the logical and physical addresses, called a flash translation layer (FTL). Figure 8.1 shows the basic concept of the FTL.

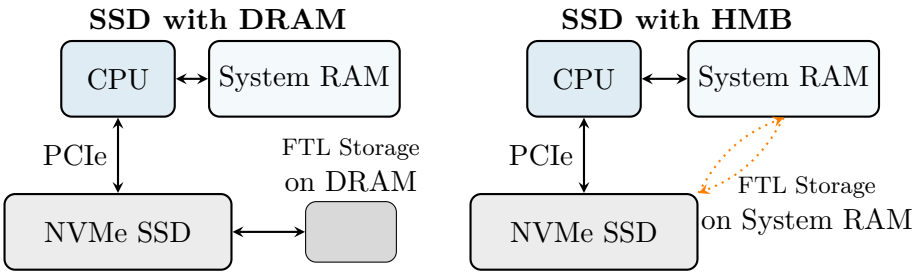


Figure 8.2.: SSDs with a host memory buffer use a small fraction of the host DRAM to cache recently used FTL translations. This slower but cheaper than a separate DRAM chip on the SSD.

Every time the data is read from or written to the SSD, the logical addresses coming from the host must be translated to the physical address with the FTL. Therefore, the latency of FTL accesses has a significant impact on the SSD's performance, especially for random accesses. Therefore, SSDs use different caches to decrease FTL access latency. "Pro"-grade SSDs come with a DRAM next to the SSD controller, large enough to contain the whole FTL. When starting, the SSD copies the whole FTL into the DRAM and only reads it from there, greatly increasing random IOPS. Because this DRAM chip adds cost, a feature called host memory buffer (HMB) was introduced with NVMe revision 1.2 [28].

2.4. Host Memory Buffer (HMB)

NVMe SSDs, being PCIe devices, can use direct memory accesses (DMA) to access the host DRAM without CPU involvement. This is used to enable the host memory buffer (HMB) feature that allows SSDs to use a small part of the main memory to cache FTL translations [28]. Access to the main memory are slower than accesses to DRAM integrated on the SSD itself but they are still faster than having to access the flash memory for every FTL translation. The HMB is a cache and typically not large enough to hold the whole FTL. HMB support was introduced into the Linux kernel in 2016 [5].

The standard does not mention what the contents of the HMB should be. This is intellectual property of the SSD controller manufacturers and

entirely undocumented. There is no research on the HMB yet, except for its performance impact [15, 16, 17, 18].

Estimating the circulation of SSDs supporting the HMB feature as difficult, as even SSD manufacturers do not always document HMB support for their SSDs. Of the approximately 20 SSDs we checked, four support the HMB feature.

2.5. Input-Output Memory Management Unit (IOMMU)

A memory management unit in modern CPUs virtually partitions and protects the main memory, that is actually physically shared between all processes. In protected and long mode, page tables are set up by the operating system, to define exactly which process is allowed to access which memory pages [9]. On x86-64 a normal page is 4 kB large, this is the smallest unit the main memory can be partitioned into. The MMU also allows the operating system to track the usage of pages by its process with the *dirty* and *accessed* bits, that are automatically updated by the MMU hardware if a page is written or accessed [9].

With the introduction of hardware assisted virtualization, Intel and AMD not only introduced a second level of pages tables to translate guest physical to host physical addresses but also mechanism to partition and protect the physical memory accessed by DMA capable devices. Intel calls this feature Intel Virtualization Technology for Directed I/O (VT-d) [10] and AMD I/O Virtualization Technology (AMD-Vi) [1]. At the heart of both is a Input-Output Memory Management Unit (IOMMU), that uses very similar page tables than the MMU with a similar feature set.

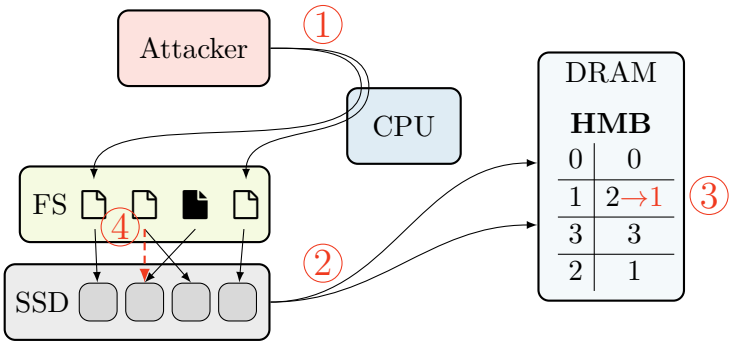


Figure 8.3.: The complete exploit chain, if every step worked. The attacker accesses specific files on the file system (FS) (1). The SSD has to access the HMB to translate the logical block addresses requested by the file system driver to the physical addresses of the NAND blocks (2). These accesses to the HMB cause enough Rowhammer accesses to flip a bit in the translation stored in the HMB, changing one mapping (3). This changed mapping allows the attacker to access the block where the secret file (black) is stored at through another file, either directly (4) or by getting access to a mapping block of the file system.

3. Attack Overview and Threat Model

In this section, we illustrate how a successful attack would look like with all components of the attack working as shown in Figure 8.3. The basic idea, especially the exploit of the ext4 file system, is based on the work by Zhang et al. [37]. However, they used an SSD simulator with simulated integrated DRAM and no HMB, while we perform our experiments on real SSDs with the HMB feature. The attacker has the permissions to create and read files from a file system on the target SSD. It has no other privileges. By frequently accessing specific files, they create an access pattern to the SSD, that forces the SSD to access the HMB frequently. These accesses to the HMB are frequent enough to cause a bit flip in the FTL. This changes one logical to physical page mapping of the HMB. With some luck, the new mapping either points to a file of interest, for example, containing a secret key or it points to a internal mapping file of the file system. This could give the attacker access to every file on the file system. In this work we show, that neither flipping bits to change an FTL mapping, nor hammering with the SSD works with real SSDs.

Table 8.1.: The SSDs used in our experiments.

SSD	PCIe	Size	HMB Size	IOPS[1]
Samsung 990 EVO	4.0	2 TB	64 MB	680 000
Lexar NM790	4.0	2 TB	40 MB	1 000 000
Samsung 980	3.0	1 TB	64 MB	500 000

[1] Read IOPS with maximum queue depth as advertised by the manufacturer.

4. Reverse Engineering the HMB

In this section, we reverse engineer how SSDs are using their HMB, how bit flips in the HMB memory area influence translations and how more complex HMB modifications can revert FTL mappings. Table 8.1 shows the SSDs used in this work. All of them use different SSD controllers supporting the HMB feature.

4.1. Tooling

The HMB specification only defines the very basic principles on how the SSD can request host memory from the operating system and how the operating system responds to these requests [28]. The operating system or any other process must not access the memory reserved for the HMB [28]. How the SSD uses the HMB is completely vendor specific and undocumented. In this section we describe the changes to the Linux kernel and our user space tools we use to gain more insight into how the HMB is used.

Linux Kernel Modifications

We modify the Linux kernel version 5.15 and utilize the IOMMU to get detailed insight on how different SSD controllers use the HMB.

HMB Mapping. When the Linux kernel reserves memory for the HMB, it does so with the `DMA_ATTR_NO_KERNEL_MAPPING` flag, not mapping the reserved memory because the specification prohibits its access anyway. As we want to know and change the HMB content in our experiments,

```
1 # cat /sys/block/nvme0n1/device/hmb_addresses
2 0 cfc00000 400000
3 1 cf800000 400000
4 ...
5 14 cc400000 400000
6 15 cc000000 400000
```

Listing 8.1: A part of the output of `/sys/block/nvmeXn1/device/hmb_addresses`. The first column is the memory chunk index, the second the I/O virtual address, and the third the length.

```
1 # cat debug/iommu/amd/iommu00/0000:01:00.0/page_tables
2 ...
3 0x00000cc000000 | 0x6000000101e7b421  0x6000000101e9f221
  -> 0x7000000113000021
4 0x00000cc001000 | 0x6000000101e7b421  0x6000000101e9f221
  -> 0x7000000113001021
5 0x00000cc002000 | 0x6000000101e7b421  0x6000000101e9f221
  -> 0x7000000113002021
6 0x00000cc003000 | 0x6000000101e7b421  0x6000000101e9f221
  -> 0x7000000113003021
7 0x00000cc004000 | 0x6000000101e7b421  0x6000000101e9f221
  -> 0x7000000113004021
8 ...
```

Listing 8.2: A part of the output of `/sys/kernel/debug/iommu/amd/iommu00/0000:01:00.0/page_tables` showing five levels of page table entries.

we add a device attribute to the NVMe driver that prints the physical addresses of the memory chunks reserved for the HMB. The attribute is accessible through the sysfs file `/sys/block/nvmeXn1/device/hmb_addresses`. Listing 8.1 shows the output with the I/O-virtual addresses because the IOMMU it active. The operating system reserves 16 memory chunks, 4 MB each for the 64 MB HMB of the Samsung 990 EVO.

HMB I/O Page Tables. With the IOMMU active, the SSD accesses I/O virtual addresses that are translated to physical addresses on each DMA access. To map the HMB in user space, we also have to translate the I/O virtual to the physical addresses. Additionally, we want access to the I/O page table entry's accessed (`A`) bits to get detailed information about which pages the SSD accessed. To do this, we need access to the

IOMMU page tables. The Intel IOMMU Linux driver already comes with a DebugFS interface to dump all page tables levels for a device at `/sys/kernel/debug/iommu/intel/{PCI_ADDRESS}/page_tables`. For the AMD IOMMU driver we wrote our own in a similar fashion. Listing 8.2 shows a part of the output of the Samsung 990 EVO.

Page Table Entry Accessed Bit. According to Intels IOMMU documentation, the accessed (`A`) and dirty (`D`) bits are always set in page tables entries [10]. However, we did not see the `A` or `D` bits being set on any of our Intel CPUs. The documentation does also not define a register where support for the `A` or `D` bit can be checked or where it can be activated.

AMDs IOMMU implementation can set the `A` and `D` bit if supported by the CPU and activated by the driver [1]. Support is specified by bits 49 (HASup) and 52 (HDSup) in the IOMMU Extended Feature Register and can be activated in bits 47:46 (HADUpdate) in the IOMMU Control Register. We added the check for support and activation to the AMD IOMMU kernel driver. We found that *only* Zen 3 CPUs support the `A` and `D` bit in their I/O page table entries. Neither Zen 2 nor Zen 4 CPUs do. Therefore, we use an AMD Ryzen 7 5800X for all of our experiments in this work.

IOMMU Page Size. The AMD IOMMU supports additional page sizes between the 4 kB, 2 MB and 1 GB of the conventional MMU. This can improve performance, because less page table entries are required to translate other buffer sizes. To get the best resolution for the `A` bit, we change the AMD IOMMU driver to only support and map 4 kB pages. With the Intel IOMMU driver this behavior can be forced with the kernel command-line parameter `intel_iommu=sp_off`.

User Space Tools

In user space, we parse the outputs of the files `/sys/block/nvmeXn1/device/hmb_addresses` and `/sys/kernel/debug/iommu/amd/iommu00/PCI_ADDRESS}/page_tables` and map the memory for the HMB and the last level page table entries with PTEditor [31]. This gives us read and write access to the HMB content, as well as the page table entries with the `A` bit.

```
1 0x3f73000: 30ae30d030ae30c  30ae30f030ae30e
  -> 30aa3100036c500  30aa312030aa311
2 0x3f73020: 30ae310030aa313  30ae312030ae311
  -> 1eeae9030ae313  30aa315030aa314
3 0x3f73040: 30aa317030aa316  30ae315030ae314
  -> 30ae317030ae316  50aa30c001e5677
4 0x3f73080: 50aa30e050aa30d  50ae30c050aa30f
  -> 50ae30e050ae30d  3d1ee8b050ae30f
5 0x3f73080: 50aa311050aa310  50aa313050aa312
  -> 50ae311050ae310  50ae313050ae312
6 0x3f730a0: 50aa31403f9c162  50aa316050aa315
  -> 50ae314050aa317  50ae316050ae315
7 0x3f730c0: 3f97dfc050ae317  70aa30d070aa30c
  -> 70aa30f070aa30e  70ae30d070ae30c
8 0x3f730e0: 70ae30f070ae30e  70aa310001ef7f2
  -> 70aa312070aa311  70ae310070aa313
```

Listing 8.3: Content at HMB offset 0x3f73000 after reading 100000 random pages from the SSD.

4.2. HMB Usage

To get more insight into how the HMB is used by the different SSDs, we performed many experiments reading and writing to the SSD while observing how the HMB contents change and which pages were accessed. We shortly summarize the most important insights for this work on our three SSDs.

Samsung 990 EVO.

The Samsung 990 EVO, like the other two SSDs, uses the HMB only to cache FTL translations. Listing 8.3 shows a small fraction of the HMB content after reading 100000 random pages from the SSD. There is definitely structure recognizable in this data, however, we keep a detailed analysis of what it means for future work. The HMB is split into four sets, 16 MB large, each set uses a last-recently used (LRU) cache replacement policy. The "cache-line" size is one page, *i.e.*, 4 kB. Every time a new translation is cached at least 4 kB are written and we also suspect that always at least 4 kB are read from the HMB. The Samsung 990 EVO also contains a small on-chip cache that caches approximately 140 pages of translations for even faster accesses. The SSD is often loading multiple

pages from the HMB, this is probably a performance optimization similar to an adjacent line prefetcher inside a CPU.

After not accessing the SSD for around 100 ms the HMB is completely reset. The content is not actually cleared, but when accessing the SSD again, the HMB is filled from the first page of every set again. This makes the HMB accesses very predicable, because they can always be reset by sleeping.

Lexar NM790.

The Lexar NM790 uses a LRU HMB eviction policy similar to the Samsung 990 EVO. However, it uses the HMB as one single set. We also only see accesses to one or, most of the time, multiple 4 kB pages. The on-chip cache has a capacity of approximately 20 pages.

Samsung 980.

The Samsung 980 also accesses the HMB in blocks of at least 4 kB. However, it does not use LRU HMB eviction. The mapping between the logical addresses and where their translations are stored in the HMB seems to be more rigid. We did not reverse engineer a set function but were able to find "cache thrashing" access patterns, where each access to the SSD, accesses the same HMB page due to this more rigid mapping. We detail and use this pattern for our hammer attempts in Section 5.1.

4.3. Simulating Rowhammer Bit Flips

In this section, we artificially flip bits in the HMB to observe how the SSD reacts. We find that every one of our tested SSDs locks up as soon as it detects the bit flip. We even broke one SSD that we could not get back working.

For this experiment, we map the memory reserved for the HMB writeable and write to it from our user space program. We want to simulate the effects a successful Rowhammer attack would have on the FTL of the SSD. As the physical location of the HMB is allocated once at boot time, it does not move while the operating system is running. We also found the location of the HMB being the same most of the time, as well as being

```
1  [  30.920489] nvme nvme2: Device not ready; aborting
   -> initialisation, CSTS=0x0
2  [  30.920517] nvme nvme2: Removing after probe failure
   -> status: -19
```

Listing 8.4: The kernel log output when the kernel tries to initilize the broken Samsung 980 during boot.

surrounded by other kernel mappings. This means that an attacker has no way to template the memory, the HMB is later stored at, for viable bit flip locations. Therefore, we randomly flip one or two bits at random locations in the HMB.

When flipping bits in the HMB, the SSD does not directly react. If it does not need the HMB page with the bit flips anymore, it just gets evicted. In this case, the bit flips are never detected and accessing the SSD page corresponding to the HMB page later, loads the unmodified data back into the HMB. An attacker would avoid this in an attack scenario.

SSD Lock Up.

By reading the SSD page, corresponding to HMB page with the bit flip, after reading some other SSD pages to evict the on-chip cache, the HMB page with the bit flips is read by the SSD. In this case, all of our tested SSDs *instantly lock up*. We suspect that a checksum of each page in the HMB is stored in the SSD controller. All requests to read or write data are never responded and simply time out. Also other NVMe commands sent with the `nvme-cli` tool are ignored. Resetting the controller or subsystem with with `nvme reset` or `nvme subsystem-reset` did not reverse the lock up. Also rebooting or turning the machine off and, after some time, on again, does *not* reset the SSDs and reverse the lock up. We found that only a real power cycle, unplugging or turning of the PSU of the computer brings the SSDs back. It seems like that even a turned off computer still supplies PCIe devices with power, preventing a reset of the SSDs.

Breaking SSDs.

During our experiments we permanently broke one Samsung 980 SSD. With a combination of `blkdiscard` commands and writes to the HMB, the

```
1 0x228f500: 0xb4691cd680300010 0x39ad3a4735a548e6
2            0xb4691cd686300000c 0x39ad3a4735a548e6
3 0x3a111e0: 0x08de003a1613149e 0x1da9144607d70994
4            0x08de003a1613149e 0x1da914460bc40994
```

Listing 8.5: Changes in the HMB after writing to one SSD page. Unchanged bytes are in gray, changed bytes are in red (previous value) and green (new value). Four bytes change in total in two different pages. Resetting these bytes to their previous value maps the previously used block.

SSD locked up and does not work anymore, even after power cycles. It does not respond to any commands and the kernel gives up initializing it during boot. Listing 8.4 shows the kernel log output. For budgetary reasons, we did not further investigate the exact course of events to reproduce the issue a second time.

Discussion.

Our experiments show that the ext4 exploit by Zhang et al. [37], that requires a remapping of flash memory blocks because of bit flips, is not possible when hammering the HMB. However, already a single bit flip in the HMB requires a power cycle of the whole machine. This can lead to a denial of service in the best case and a corrupt system or data loss in the worst case if the operating system is not able to flush dirty data to the disk. In a more advanced attack, an attacker could try to coerce the operating system to send frequent discard commands to trim the SSD. The attacker hammers the SSD at the same time to break the SSD, causing physical damage on a modern computer only from software. This would also be possible for an attacker that already gained elevated privileges and can write to the HMB through the direct physical map.

4.4. "Unwriting" Data

By carefully changing data in the HMB, we managed to "unwrite" data on the Samsung 980 SSD. To do this, we stored the HMB content before writing to one SSD page. This write changes four bytes in the HMB, as shown in Listing 8.5. Changing these bytes to any other values locks up the SSD, except for one change. If we reset the changed bytes to their previous

values, the FTL mapping is reset to the old value. Reading the previously written SSD page returns the *old* content of the page. As the HMB content is never written back to the SSD, this change is not persistent. After the eviction of the modified HMB page, reading the SSD page again returns the new content.

Being able to perform a replay attack like this, conflicts with our hypothesis that the pages cached in the HMB are simply protected with a checksum that is stored in the SSD controller. A few bits of the changed data could also be checksum bits. However, as changing data like this is far outside the reach of Rowhammer bit flips, we did not further investigate this effect.

5. HMB Rowhammer

While actual privilege escalation with the ext4 exploit [37] with bit flips in the HMB is very unlikely due to all SSDs instantly locking up, a denial of service attack or maybe even breaking SSDs is possible with flips in the HMB. However, we show that the maximum achievable accesses from the SSD to the HMB are not high enough to flip bits, even on highly vulnerable DRAM.

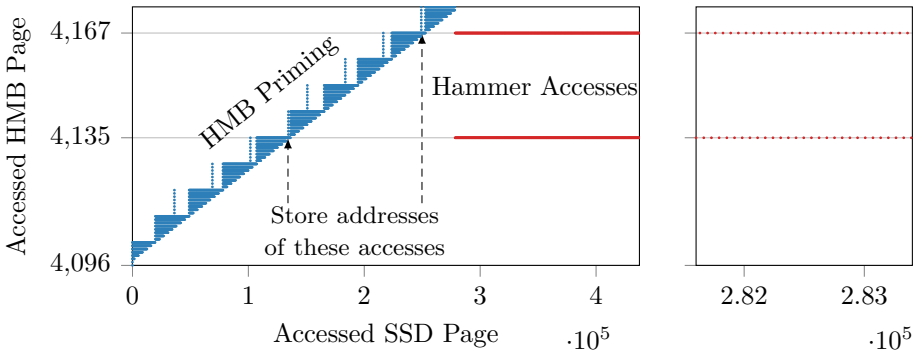
5.1. Generating Hammer Accesses

In this section, we show how we can cause targeted Rowhammer accesses from the different SSDs to the HMB. All SSDs behave differently and require different accesses patterns to frequently access the HMB. We achieve the fastest accesses on the Samsung 980, with 4 992 hammer accesses per refresh interval.

Samsung 990 EVO.

This SSD uses a LRU cache eviction policy and four 4 sets for its HMB. The LRU algorithm makes it possible to prime the cache with translations and then later accesses the pages with the cached translations to get double-sided Rowhammer accesses.

Figure 8.4a shows the cache priming and hammering. In the priming phase, we sequentially access contiguous pages on the SSD, storing where they



(a) The blue dots show the accesses to prime the HMB with translations. The stored SSD page addresses are then used for the hammer accesses.

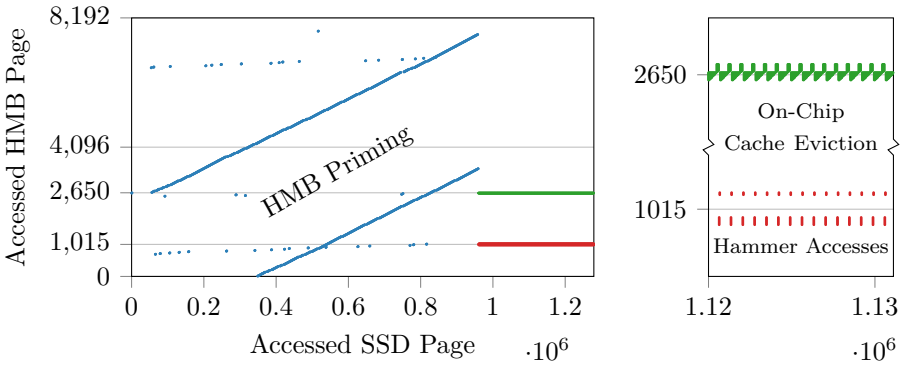
(b) Zoomed in plot on the hammering phase.

Figure 8.4.: HMB priming and hammering on the Samsung 990 EVO.

caused an HMB accesses. The SSD shows interesting behavior, visible as a stair pattern. At some accesses to the SSD, it accesses up to 8 pages in the HMB. We expected that these 8 pages are loaded into the on-chip cache and the SSD would not accesses the HMB for them again. But instead, we see that after accessing 32 pages on the SSD, the SSD accesses the HMB again, loading the *same* pages. It seems like the on-chip cache caches the data for a only 32 accesses. We could not think of a reason why this would be of an advantage. The SSD then slowly reduces the number of pages it accesses until it accesses only a single page Then it accesses the next 8 HMB pages. With this slope being constant and it always starting the same way, we compute exactly how many SSD pages we have to access to prime the HMB for double-sided Rowhammer accesses.

For this example, we assume very simple DRAM addressing functions, where two pages, 32 pages apart, hammer the page exactly in the middle. If we want to hammer HMB page n, we first prime the cache by accessing $(n+16)\cdot 3640$ pages on the SSD, e.g., from the beginning. 3640 is the slope of the priming-slope. Then we only have to access the SSD pages $(n-16)\cdot 3640$ and $(n+16)\cdot 3640$ to cause the pattern shown in Figure 8.4b. While we figured out these steps with our modified kernel to get the accessed pages with the IOMMU, they are always repeatable and can, therefore, also be executed by an unprivileged attacker.

This access pattern seems to hit some internal bottleneck of the SSD. Performing it asynchronously with maximum queue depth, reduces the



(a) The blue dots show the accesses to prime the HMB with translations, the red dots the hammer accesses and the green dots the on-chip cache eviction accesses.

(b) Zoomed in plot on the hammering phase.

Figure 8.5.: HMB priming and hammering on the Lexar NM790.

IOPS to 135 000. This is one firth of the advertised 680 000. This results in less than 300 hammer accesses to the two aggressor rows per 64 ms refresh interval.

$$\frac{135\,000\,\text{IOPS} \cdot 64\,\text{ms}}{31\,\text{Dummy Accesses}} = 280\,\text{Hammer Accesses}$$

Checking Asynchronous HMB Accesses. Figure 8.4 was measured with synchronous SSD accesses to get exactly which access to the SSD cause which HMB accesses. With asynchronous SSD accesses with large queue depths, it is impossible to check the `A` bit in the IOMMU page tables for each SSD accesses. Instead, in another thread we periodically reset all the `A` bits in all page table entries mapping the HMB, flush the IOTLB and check them again without any added delay in-between. It takes only 26 µs to check all `A` bits after flushing the IOTLB. We constantly see the same HMB pages being accessed during this small interval, giving us high confidence that the SSD is actually accessing the HMB also when we perform the SSD accesses with large queue depths.

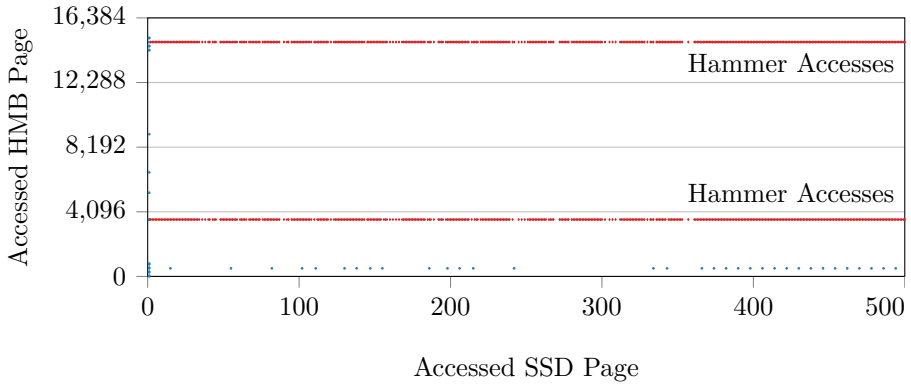


Figure 8.6.: The hammer accesses from the Samsung 980. The first access to the SSD causes many HMB accesses. Then it takes 360 accesses for the SSD to "settle in". Afterward, every single access to the SSD, causes the SSD to access pages around index 3 606 and 14 854 of the HMB.

Lexar NM790.

The Lexar SSD also uses a LRU replacement policy. However, it never accesses HMB pages that are cached on the on-chip cache. Therefore, we have to perform on-chip cache eviction. We also use sequential SSD accesses to prime the HMB and then access two specific SSD pages to hammer the HMB and in-between accesses to other SSD pages to evict the on-chip cache. Figure 8.5a shows the priming and hammer accesses. Because the Lexar SSD's HMB state cannot be "reset" by simply sleeping, the HMB priming does not start from zero. This gives us less control over what exactly is hammered and also how many pages are accesses for the hammer accesses. As shown in Figure 8.5b multiple pages are accesses for each aggressor. We find that the on-chip cache is evicted after accessing 20 other SSD pages and achieve up to 717 double-sided hammer accesses per refresh interval.

$$\frac{112000\,\text{IOPS} \cdot 64\,\text{ms}}{10\,\text{Eviction Accesses}} = 717\,\text{Hammer Accesses}$$

Samsung 980.

The Samsung 980 does not use an LRU cache eviction policy. Neither in the HMB nor in the on-chip cache. This helps us to achieve our fastest

Table 8.2.: Accessing these pages on the SSD causes these HMB accesses. $PageA$ and $PageB$ are 37 200 kB apart and the *step*-size is 1 200 kB. After $PageB + 15 \cdot step$ we accesses $PageA$ again. The accesses to $512 - 519$ every 8th time can be seen in Figure 8.6 as the blue dots at the bottom.

Accessed SSD Page	Accessed HMB Pages		
$PageA$	$512 - 519$,	$3606 - 3607$,	$14848 - 14855$
$PageB$		$3605 - 3607$,	$14854 - 14855$
$PageA + step$		$3606 - 3607$,	$14854 - 14855$
$PageB + step$		$3605 - 3607$,	$14854 - 14855$
$PageA + 2 \cdot step$		$3606 - 3607$,	$14854 - 14855$
$PageB + 2 \cdot step$		$3605 - 3607$,	$14854 - 14855$
	...		
$PageA + 4 \cdot step$	$512 - 519$,	$3606 - 3607$,	$14848 - 14855$
$PageB + 4 \cdot step$		$3605 - 3607$,	$14854 - 14855$
	...		
$PageA + 15 \cdot step$		$3606 - 3607$,	$14854 - 14855$
$PageB + 15 \cdot step$		$3605 - 3607$,	$14854 - 14855$

Rowhammer accesses of almost 5 000 hammer accesses per refresh interval. On the Samsung 980 it is possible to access specific pages of the SSD, so that each access to the SSD makes the SSD access the HMB, as shown in Figure 8.6. Each access to the SSD causes the "hammer"-accesses in red to the HMB.

To do this, we take two addresses, 37 200 kB apart, and access them alternately, adding 1 200 kB after each accesses. After 32 accesses, we start again with the initial two addresses. Table 8.2 shows the pattern of SSD accesses and the HMB accesses they cause. This pattern leads to the SSDs accessing the same pages in the HMB on each SSD page access, perfect for hammering.

However, these special patterns reduce the IOPS to around 78 000 from the advertised maximum of 500 000. But even then, if every one of these accesses to the SSD cause an access to the HMB, we still reach close to 5000 hammer accesses per refresh interval.

$$78000\,\text{IOPS} \cdot 64\,\text{ms} = 4992\,\text{Hammer Accesses}$$

While our thread that checks the accessed bits asynchronously is a good indicator of the HMB accesses from the SSD, we use a combined Row-

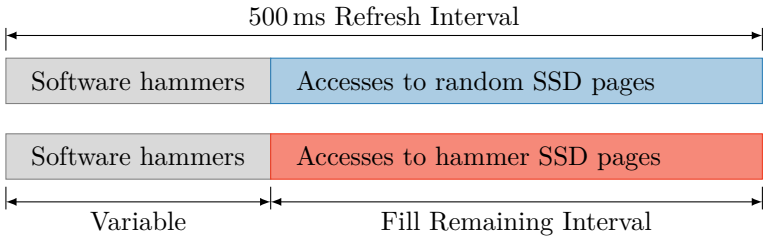


Figure 8.7.: For the combined Rowhammer experiment, two rows are first hammered from software. Within the same refresh interval, the same rows are then hammered through the SSD's HMB accesses.

hammer experiment to measure the real impact of the HMB accesses on Rowhammer bit flips.

5.2. Combined Rowhammer

To measure the real Rowhammer impact of the Samsung 980's access to the HMB, we perform a combined Rowhammer experiment. The idea is to combine Rowhammer accesses from software with accesses from the SSD, as shown in Figure 8.7. During each refresh interval, we first perform software accesses to get the DRAM victim row to an access count where it starts to flip. Then, we fill the rest of the refresh interval with accesses from the SSD to the HMB. By comparing the number of bit flips we get with the targeted hammer SSD accesses we found in Section 5.1 against the number of bit flips with random SSD accesses, we get the impact of the targeted SSD accesses on the bit flip count.

For this experiment, we use an Intel Core i9-9900K for two reasons. It is well known how to hammer on older generation Intel CPUs, and it supports changing the DRAM refresh interval t_{REFI} in the BIOS. The disadvantage of the Intel CPU is that it does not set the `A` bit in IOMMU page table entries.

Getting HMB Access Info on Intel CPUs.

There is one other, much more tedious way, to get the accessed HMB pages without the `A` bit. We just want to ensure that the SSD still accesses the same HMB pages as it did with the AMD CPU. We unmap the HMB

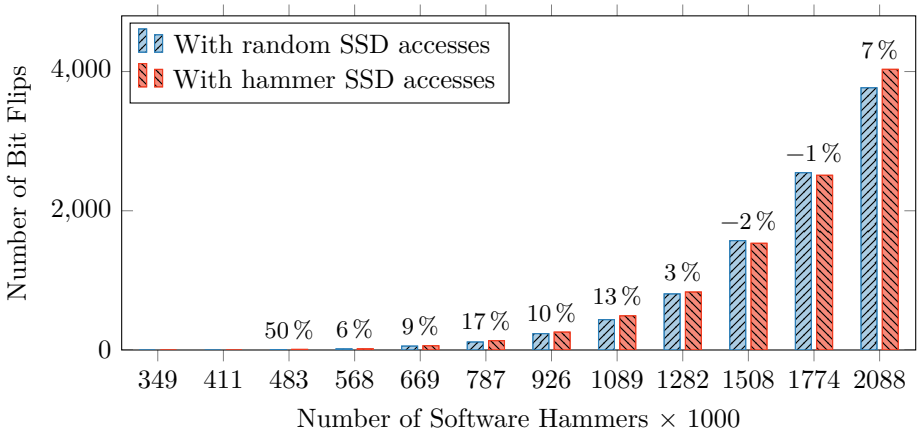


Figure 8.8.: The number of bit flips when combining Rowhammer accesses from software with accesses to the HMB. The HMB accesses have no measurable impact on the bit flip count. At high conventional hammer access counts it even reduces the bit flip count significantly.

pages in question by clearing the present bit in the corresponding IOMMU page table entries. Now, when we perform our hammer SSD accesses, the SSD tries to access the unmapped HMB page. The IOMMU raises an interrupt and the handler prints the violating access to the kernel log, where we can see that the SSD tried to access the expected HMB page. However, the SSD's DMA access fails and it locks up until a full power cycle. Nevertheless, we confirmed that the SSD accesses the HMB pages found in Section 5.1.

Experimental Setup.

We maximized the refresh interval to approximately 500 ms from the default 64 ms. This increases the number of hammer accesses achievable with the Samsung 980 to up to 38 500 per refresh interval.

We want to sweep the victim row through DRAM, to get as much bit flip data as possible. To achieve this, without having to find more SSD page patterns that cause HMB accesses at different offsets, we use the IOMMU to remap pages 3 606 and 3 607. We always map the pages to physical locations that perform double-sided Rowhammer. When remapping the

HMB pages, we also have to copy the content to the new physical location to keep the HMB consistent. This creates a short race condition, where the SSD crashes sometimes. We set up a service that automatically power cycles the computer when this happens.

After remapping the aggressor HMB pages, we start with 2 088 000 software accesses, filling the rest with SSD accesses. The 2 088 000 take up approximately half of the refresh interval. We hammer the same victim row 6 times, randomized 3 times with random SSD accesses and 3 times with our targeted SSD accesses, always resetting the victim data in-between, by writing the inverted binary data from the above aggressor row into it. Then, we reduce the software accesses by 15 % and hammer the same victim row again. We repeat this 12 times, down to 349 000 software accesses. Finally, we go to the next victim row, remap the aggressor HMB pages, and start again with 2 088 000 software accesses.

Results.

Figure 8.8 shows the results from the combined Rowhammer experiment. The bar hight shows the number of bit flips with either random accesses to the SSD or the hammer accesses to the SSD. The percentage above the bars shows the change with the hammer accesses. We measured 19 432 bit flips in total.

There is a small but clear increase in the number of bit flips with the SSD's hammer accesses. Over all numbers of software hammers, the increase in bit flips is 3.4 %, from 9 555 to 9 877. This shows that the SSD's accesses to the HMB do actually hammer the DRAM.

Hammer Effectiveness. We can compute the effectiveness of the SSD's HMB hammers. Using the bit flip counts with random SSD accesses from Figure 8.8, we can calculate that an increase of 1 000 software accesses, increases the bit flip count by 1.8 on average. Now we can calculate by how much the SSD HMB hammers increase the bit flip count. For this we look at the results with 1 089k software accesses.

1 089k software accesses take approximately 130 ms, leaving 370 ms for the SSD accesses to the HMB within one refresh interval. The SSD hammers with 78 000 IOPS for these 370 ms, causing HMB 28 860 accesses.

At 1 089k software accesses, we measured 436 bit flips with random SSD accesses and 491 with hammer SSD accesses, an increase of 55 bit flips or 13 %. Knowing that 1 000 software accesses increases the bit flip count by 1.8, we calculate the equivalence of SSD accesses to the HMB to software accesses.

$$\frac{1000 \text{ software accesses}}{1.8 \text{ bit flips}} \cdot 55 \text{ bit flips} = 30556 \text{ equivalent hammer accesses}$$

The SSD causes 28 860 HMB accesses, meaning that the SSD's HMB accesses are *equivalent* to software acesses.

Discussion. This show clearly that the SSD can hammer the DRAM through the HMB very effectively. However, the IOPS and, therefore, the access count is not high enough to induce bit flips without additional software accesses. There is currently no threat model where an unprivileged attacker could get access to the memory where the HMB is mapped to perform additional software accesses. Furthermore, we used the IOMMU to map two HMB pages to exact locations to perform double-sided Rowhammer.

6. Mitigations

In this section, we show that the HMB can easily be disabled on Linux to mitigate these attacks and discuss the impact of state-of-the-art Rowhammer mitigations on HMB Rowhammer.

Disabling the HMB. The HMB is specified as an optional feature for the SSD: "The controller shall function properly without host memory resources." [28]. The only downside is a reduced performance, as shown by previous work [15, 16, 17, 18]. On Linux the HMB can be turned off for all SSDs attached to the system with the kernel command-line parameter `max_host_mem_size_mb=0` [23].

Existing Rowhammer Mitigations. Modern DDR4 and DDR5 DRAM comes with on-die Rowhammer mitigations like TRR. To evade these mitigations, complicated access patterns are required that "confuse"

the TRR mechanism so it tracks and refresh other dummy rows instead of the actual victim row [7, 11]. To perform these access patterns, high timing accuracy as well as synchronization with refreshes is important. Additionally, Kang et al. [14] showed that modern Intel CPUs also contain a undocumented Rowhammer mitigation that can only be evaded by hammering multiple banks simultaneously. This means, that while only a few thousand accesses to the actual attacker rows are sufficient on modern DRAM, overall, the whole access pattern to actually cause bit flips on modern system still consists of many more accesses. Currently, we do not see a possibility to achieve the high number of required accesses, to the many different rows, synchronization with refreshes, and hammering multiple banks simultaneously when using HMB hammer.

7. Conclusion

Rowhammer is a long standing security vulnerability. Although, there is countless research on attacks, only very few works looked into the risk coming from devices peripheral to the CPU that can also access the DRAM. This work is the first one that analyzes the SSD as a potential victim or confused deputy attacker of a Rowhammer attack. With the introduction of the HMB feature, NVMe SSD can access the DRAM through DMA to cache FTL translations. We show that bit flips in the HMB are unlikely to lead to privilege escalation attacks but can force a system to power cycle or even break hardware. We also show how our tested SSDs can be coerced into accessing the HMB with a high frequency. On the Samsung 980 we achieve 5000 accesses to the HMB per 64 ms refresh interval. In the combined Rowhammer experiment, together with Rowhammer accesses from software, we see a clear impact of the HMB accesses from the SSD on the bit flip count. This shows that, in theory, an SSD could hammer the DRAM but it very unlikely on modern DRAM with active Rowhammer mitigations.

8. Acknowledgments

I thank Daniel Gruss, Fabian Rauscher and the anonymous reviewers for their valuable feedback on this work. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and

the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Red Hat and Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] AMD. I/O Virtualization Technology (IOMMU) Specification, rev 3.09. 2023 (pp. 242, 246).

[2] 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. 236, 239).

[3] 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 (pp. 236, 239).

[4] 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. 236, 239).

[5] Arnav Dawn. [PATCH] NVMe:Support for Host Memory Buffer(HMB). 2016. URL: https://lore.kernel.org/linux-nvme/20160615104800.GC32253@infradead.org/T/ (p. 241).

[6] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (pp. 236, 239).

[7] 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. 236, 239, 260).

[8] 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. 236, 239).

[9] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (p. 242).

[10] Intel. Virtualization Technology for Directed I/O, rev 4.0. 2022 (pp. 242, 246).

[11] 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. 236, 239, 260).

[12] 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. 236, 239).

[13] 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. 236, 239).

[14] 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. 236, 239, 260).

[15] Kyu Sik Kim and Tae Seok Kim. Performance Evaluation of HMB-Supported DRAM-Less NVMe SSDs. In: KIPS Transactions on Computer and Communication Systems (2019) (pp. 242, 259).

[16] Kyusik Kim, Seongmin Kim, and Taeseok Kim. HMB-I/O: Fast Track for Handling Urgent I/Os in Nonvolatile Memory Express Solid-State Drives. In: Applied Sciences (2020) (pp. 242, 259).

[17] Kyusik Kim and Taeseok Kim. HMB in DRAM-less NVMe SSDs: Their usage and effects on performance. In: PloS one (2020) (pp. 242, 259).

[18] Kyusik Kim, Eunji Lee, and Taeseok Kim. HMB-SSD: Framework for Efficient Exploiting of the Host Memory Buffer in the NVMe SSD. In: IEEE Access (2019) (pp. 242, 259).

[19] 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. 236, 239).

[20] 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. 236, 239).

[21] 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 (pp. 236, 239).

[22] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (pp. 236, 239).

[23] Linux. /drivers/nvme/host/pci.c. 2024. URL: `https://elixir.bootlin.com/linux/v6.11/source/drivers/nvme/host/pci.%5C#L55` (p. 259).

[24] Linux Foundation. Storage Performance Development Kit (SPDK). 2025. URL: `https://spdk.io/` (p. 237).

[25] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlikçi, 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 (pp. 236, 239).

[26] Michele Marazzi and Kaveh Razavi. RISC-H: Rowhammer Attacks on RISC-V. In: Workshop on DRAM Security (DRAMSec). 2024 (pp. 236, 239).

[27] Onur Mutlu, Ataberk Olgun, and A Giray Yağlikci. Fundamentally Understanding and Solving RowHammer. In: Asia and South Pacific Design Automation Conference. 2023 (p. 239).

[28] NVM Express, Inc. NVM Express, rev 1.2.1. 2016 (pp. 236, 241, 244, 259).

[29] 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 (p. 239).

[30] 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. 236, 239).

[31] Michael Schwarz, Moritz Lipp, and Claudio Canella. misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8. 2018. URL: https://github.com/misc0110/PTEditor (p. 246).

[32] 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. 236, 237).

[33] Anton Shilov. SSDs Outsell HDDs in Unit Sales 3:2: 99 Million Vs. 64 Million in Q1. 2021. URL: https://www.tomshardware.com/news/ssd-market-shares-q1-2021-trendfocus (p. 236).

[34] 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 (pp. 236, 239).

[35] 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. 236, 239).

[36] A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In: HPCA. 2021 (pp. 236, 239).

[37] Tao Zhang, Boris Pismenny, Donald E Porter, Dan Tsafrir, and Aviad Zuck. Rowhammering Storage Devices. In: ACM Workshop on Hot Topics in Storage and File Systems. 2021 (pp. 237, 243, 250, 251).

9

The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer

Publication Data

Jonas Juffinger, Hannes Weissteiner, Thomas Steinbauer, and Daniel Gruss
The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer
In: DIMVA 2025

Contributions

Main Author.

The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer

Jonas Juffinger, Hannes Weissteiner, Thomas Steinbauer, Daniel Gruss

Graz University of Technology, Graz, Austria

Abstract

Over the past three decades, cache side channels evolved from specialized attacks on cryptographic implementations to generic techniques (e.g., Flush+Reload and Page Cache Attacks) on general-purpose operations. During the last decade, SSDs became the de facto standard persistent storage, where capacity is not the highest priority.

In this paper, we present a novel cache side channel, targeting the host-memory buffer (HMB) used by mid-range SSDs to cache translations from logical page addresses to physical page addresses.

We demonstrate that, compared to page cache attacks, our attacks are significantly faster as we can evict reliably in only 22 ms. Consequently, we propose a hybrid attack, using the slow page cache eviction as little as possible and using the HMB side channel for our main attack. We evaluate the HMB side channel in four practical attacks: First, we evaluate the capacity of the HMB side channel in a covert channel scenario, achieving up to 8.3 kbit/s channel capacity. Second, we demonstrate a UI redress attack using the HMB side channel, where the fake UI element covers the real one within 100 ms. Third, given that multiple pages from different security contexts are translated through the same HMB entry, we demonstrate blind templating attacks, that allow to spy on accesses to arbitrary other files whose translation is co-located in the same HMB entry. We use this to demonstrate a cross-VM covert channel and a remote side channel where an unprivileged process without network access exfiltrates data to a remote system over the network, through the HMB side channel by using an nginx web server as a confused deputy.

1. Introduction

The performance of modern systems is fundamentally limited by memory latency. Caches play a central role in modern systems to alleviate this bottleneck. However, while they reduce the latency of accesses to recently or frequently accessed memory, this latency difference can also be exploited in so-called side-channel attacks, at the beginning, mainly on cryptographic implementations [1, 10]. Later, generic techniques like Prime+Probe [18] or Flush+Reload [25] enabled cache side-channel attacks on general-purpose computations [5, 6].

Despite the growing popularity and the increasing performance of SSDs, they have received relatively little attention in terms of side-channel research. Liu et al. [14] studied the now-discontinued Intel Optane, which has several caches introducing distinct timing differences. Two works focused on SSD covert channels with attacker-controllable FPGAs reaching < 1 bit/s [4, 24]. Recently, Juffinger et al. [9] demonstrated that contention on commodity SSD can be exploited to build covert channels with over 1 kbit/s transmission rate and website fingerprinting with over 90 % accuracy. While page cache attacks also leak spatial information and can be faster [6], they were partially mitigated recently [22], reducing the temporal resolution to one measurement within several seconds due to the high cost of eviction, e.g., of a multi-gigabyte page cache. In concurrent work, we discussed the negative result of performing a Rowhammer attack through the HMB and the effect on bit flips on the HMB [8].

In this paper, we present a novel cache side channel, targeting the host-memory buffer (HMB) [17] used by mid-range SSDs to cache translations from logical to physical page addresses. We investigate four different SSD models supporting the HMB feature and reverse-engineer how these SSDs use it. Our investigations show that the HMB operates either full- or set-associatively, translating contiguous blocks of multiple megabytes of memory. We demonstrate how an attacker can observe the HMB state and how they can evict the HMB through accesses to arbitrary files. As a result, we obtain a side channel with a spatial resolution of a few megabytes, low compared to the 4 kB of page cache attacks [6].

However, our HMB attack is significantly faster as we can evict the HMB in only 22 ms or less. Additionally, we demonstrate how we can use the unprivileged `FIEMAP` ioctl system call to effectively increase the spatial granularity of a few megabytes to the 4 kB of page cache side channels

by using the page cache to our advantage. Consequently, we propose a hybrid attack, using the slow page cache eviction as little as possible and the HMB side channel for our main attack.

We evaluate the HMB side channel in four practical attacks: First, we evaluate the capacity of the HMB side channel with an inter-process covert channel, achieving up to 8.3 kbit/s. Second, we demonstrate that we can monitor disk accesses to files on the SSD. We exploit this side channel in a UI redress attack using the HMB side channel within 100 ms. The goal of a UI redress attack is to detect when a program starts and then cover it with a fake UI element to, e.g., steal an entered password. Even without any sharing of files or memory, we can exploit that multiple SSD pages from different security contexts are translated through the same HMB entry in blind templating attacks. Third, we show a covert channel between virtual machines with up to 7.1 bit/s. Finally, we demonstrate a remote side channel where an unprivileged process without network access exfiltrates data to a remote system over the network with up to 8.9 bit/s, by exploiting nginx as a confused deputy.

In summary, the **contributions** are as follows:

- We are the first to analyze the host memory buffer for potential side channels and reverse engineer the behavior of different implementations.
- We show that differences in the access latency leak information about recently accessed pages that can be measured with very high accuracy.
- We present a novel blind templating attack that also works if the attacker has no read rights to the target files.

In Section 2, we provide background information on cache timing side channels and SSDs. In Section 3, we describe the HMB side channel and reverse-engineer how the SSDs use the main memory and how the HMB can be evicted. In Section 4, we evaluate the HMB side channel in attacks on accessible files, showing that yields a significantly higher performance than current page cache attacks. In Section 5, we demonstrate that even without accessible files an attacker can mount a blind templating attack using the HMB side channel. We discuss countermeasures in Section 6. We conclude in Section 7.

2. Background

In this section, we provide background on cache timing side channels, SSDs and flash memory, the flash translation layer and host memory buffer.

Cache Timing Side Channels.

Caches are used in computers to store data that is likely to be accessed soon, to decrease the delay of these accesses. They are both, faster and smaller, than the main memory they cache data from. Due to this smaller size, they cannot store all the data from the main memory but only a small subset at any given time.

Caches introduce execution timing differences based on the current cache state. This effect is exploited in cache timing side-channel attacks like Prime+Probe [13, 18] or Flush+Reload [25]. While these techniques have mainly been shown on CPU caches, they also can be applied to other caches [6, 23]. Gruss et al. [6] demonstrated the that flush- and eviction-based attacks on the OS page cache are possible. The page cache buffers data from the disk, in blocks of pages. However, as also reported by Schwarzl et al. [22], the interfaces exploited by Gruss et al. [6] are now more restricted.

Solid State Drives and NAND Flash Memory.

Solid state drives (SSDs) are persistent storage devices that use NAND flash memory to store data. They are faster than hard disk drives (HDDs), the number of random input output operations per second (IOPS) is magnitudes higher.

NAND flash memory restricts how data can be written. While data can always be read, settings bits to `1` (erasing) uses a different process than setting bits to `0` (programming). Erased memory stores all `1`'s, data is then written by setting bits to `0`. Typically, the read and programming size is 512 B, 2 048 B or 4 096 B, called a page. Erasures happen in blocks of 32, 64 or 128 pages and take much longer. Flash memory has only a limited number of program-erase cycles. To wear out all blocks equally, the SSD controller performs wear-leveling.

Flash Translation Layer. Wear-leveling and other performance optimizations cause the data on an SSD to be scattered. Therefore, a persistent translation layer is required that translates the logical page addresses, the ones seen by the operating system, to the physical page addresses of actual storage locations.

If the FTL was only stored in the flash memory, every read would incur an additional read. Therefore, SSDs employ different caches for their FTL. Budget SSDs only have a small on-chip cache in the SSD controller. "Pro"-level SSDs contain a DRAM chip that is large enough to hold the entire FTL. Mid-range SSDs can use a feature called Host Memory Buffer as a trade-off.

Host Memory Buffer (HMB). HMB is an NVMe feature that allows SSDs to request main memory from the operating system [17]. The SSD can then access the HMB through direct memory accesses (DMA) over PCIe and cache parts of the FTL there. DMA accesses to the main memory over PCIe are slower than accesses to an integrated DRAM but faster than accessing the flash memory for each FTL entry. The HMB is not large enough to store the whole FTL.

HMB Prevalence.

Overall, the HMB feature is not uncommon. We used techpowerup's SSD database [3] containing 869 SSDs from 109 manufacturers to understand the prevalence of the HMB feature in SSDs. Of these 869 SSDs, 694 use the PCIe interface. 37 % or 255 of all PCIe SSDs do not have a DRAM. Of these DRAM-less PCIe SSDs, 97.6 % or 249 support the HMB feature. This shows that the feature is very prevalent in DRAM-less SSDs, as it enables a performance gain at almost no cost.

FIEMAP Ioctl.

The file extent map (`FIEMAP`) system call allows unprivileged processes to retrieve the "physical" block addresses of all fragments of a file [12]. In this case, "physical" correspond to the logical block addresses of the SSD as seen by the operating system that are then translated to the real physical addresses using the FTL.

Table 9.1.: The SSDs used in our experiments.

SSD	Model	PCIe Revision	Size	HMB Size
$\mathcal{A}$	Samsung 990 EVO	4.0	2 TB	64 MB
$\mathcal{B}$	Lexar NM790	4.0	2 TB	40 MB
$\mathcal{C}$	Samsung 980	3.0	1 TB	64 MB
$\mathcal{D}$	Western Digital Blue SN550	3.0	1 TB	32 MB

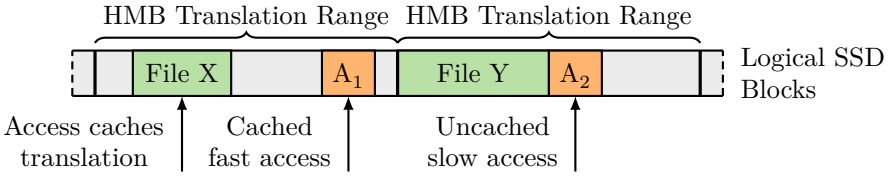


Figure 9.1.: Working principle of the HMB. An access to file X caches all logical-to-physical translations within a specific storage range in the HMB, including the translation for the page containing file fragment A_1. Therefore, a subsequent access to X or A_1 is faster than an access to Y or A_2.

3. The HMB Side Channel

The HMB caches recently used logical to physical translations of the SSD to reduce the access latency to the corresponding pages. We show that access latency differences are measurable and give an attacker valuable information about recently accessed pages on the SSD. In this section, we reverse-engineer the basic functionality of the HMB of four different SSDs and show that the HMB introduces a timing side channel that is observable from user space. Table 9.1 shows the SSDs we used for our experiments throughout the paper. Figure 9.1 shows the basic working principle of the HMB side channel. It shows an SSD's logical storage space as seen by the operating system and its file system driver. After accessing file X, the logical to physical translation to the SSD pages that contain file

X but also to the file fragment A_1 are cached in the HMB. This decreases the subsequent access time to X and A_1 while the access to the uncached pages containing file Y or fragment A_2 are slower.

3.1. Reverse Engineering HMB Usage

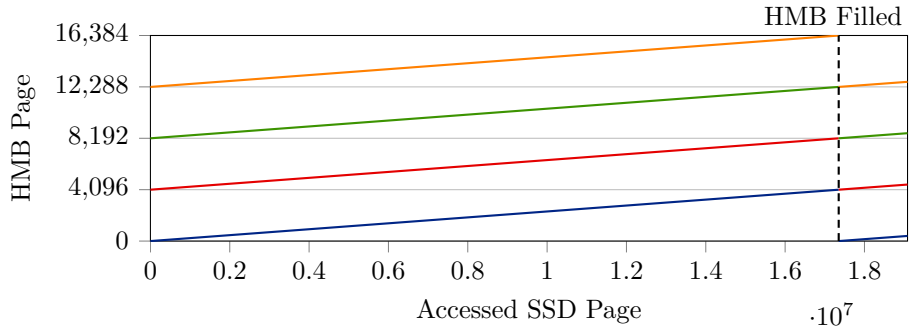
In this section, we reverse engineer how the different SSDs use the HMB to cache their translations as shown in Figure 9.2. We use the IOMMU of a AMD Zen3 Ryzen 7 5800X to observe the accesses from the SSD to the HMB with 4 kB page granularity, similar to Juffinger et al. [8].

Samsung 990 EVO. The Samsung 990 EVO splits the 64 MB HMB into four independent 16 MB cache sets, as shown in Figure 9.2a. After accessing 66 GB of data, or 17 million pages, sequentially, the HMB is filled with translations. Afterwards, the SSD controller starts replacing the least recently used (LRU) translations. Bits `14` and `15` of the logical block address select a set.

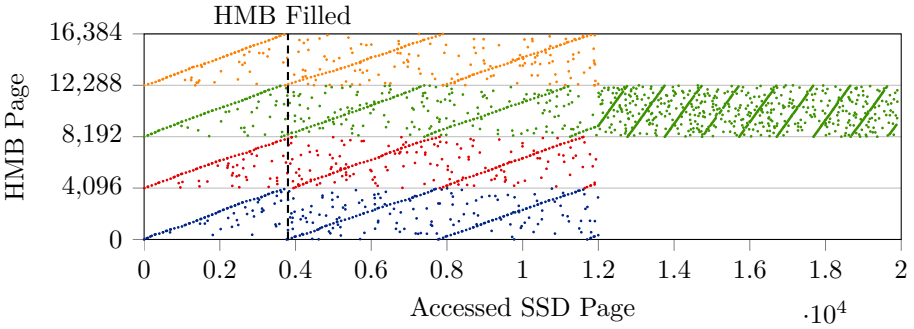
A lot less *random* accesses are required to fill the HMB, as shown in as Figure 9.2b. This is because a single access also prefetches many translations of surrounding pages, filling the HMB. There are also seemingly random HMB accesses in-between the linear patterns. These HMB accesses come from accesses to addresses within an already cached HMB translation range. To get size of these prefetches, we build an experiment where we access an addresses and then measure the cache state of a neighboring addresses with increasing distance. This shows that up to 15 360 pages, *i.e.*, translations to 60 MB of data, are cached at once. After 12 000 accesses in Figure 9.2b, we only access addresses of set 2 by fixing bits `14` and `15` of the logical block address to `10`.

When the SSD is not accessed for 100 ms, it resets the HMB state. Afterwards, each cache set is again filled from the first page and all previously cached translations are not used anymore. The SSD seems to enter a low power state after 100 ms but we cannot think of a reason why this resets the HMB state.

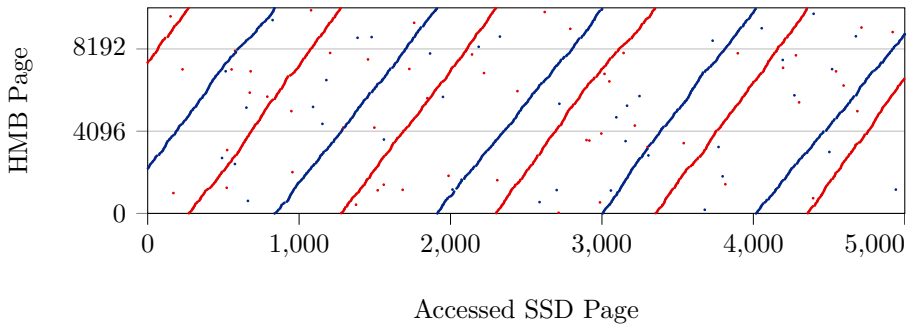
Lexar NM790. The Lexar NM790 also uses LRU cache eviction, as shown in Figure 9.2c. It also uses two cache-sets, but differently than the Samsung 990 EVO. Both cache sets can occupy every page of the HMB,



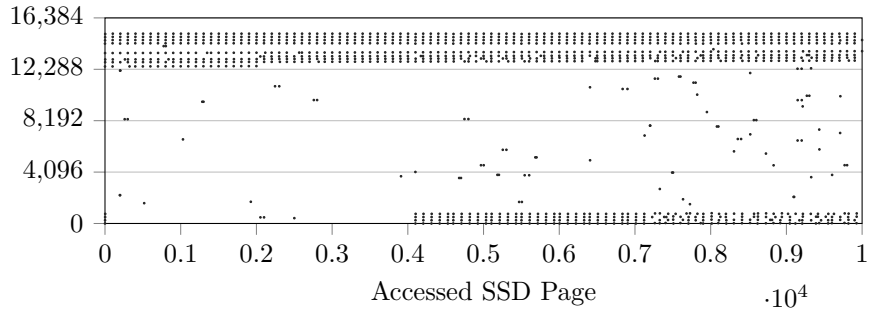
(a) $\mathcal{A}$: Sequential SSD accesses causes a linear pattern of HMB accesses. The HMB is split into four sets.



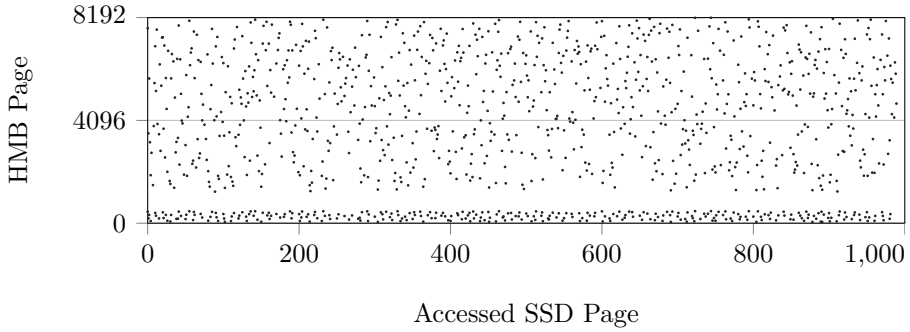
(b) $\mathcal{A}$: Random SSD accesses show the linear pattern plus "random" accesses to the HMB.



(c) $\mathcal{B}$: Sequential and random (shown) SSD accesses show two LRU patterns.



(d) $\mathcal{C}$: Sequential and random (shown) SSD accesses show a pattern to specific offsets at the start and end of the HMB range.



(e) $\mathcal{D}$: Sequential and random (shown) SSD accesses show no particular pattern.

Figure 9.2.: The accesses from the SSD to the HMB caused by accesses to the SSD. The four SSDs behave very differently.

but they seem to use an individual LRU counter. Bit `25` of the logical SSD address defines the set. We find in Section 3.3 that accessing only addresses of a single set greatly increases variance in the access timings. So it seems like these two sets enable some sort of load balancing in the SSD.

Samsung 980. The Samsung 980 does not show a clear usage pattern like the Samsung 990 EVO or Lexar NM790, neither ith sequential nor random accesses, as shown in Figure 9.2d. Most parts of the HMB were only used very sparsely. The Samsung 980 also resets the HMB state after 100 ms.

WD Blue SN550. This SSD splits the HMB into two distinct parts, as shown in Figure 9.2e. Every translation consists of two HMB accesses. One to the lower part, between HMB pages 48 and 488 and the other to the upper part above HMB page 1255. We think that the lower part contains a cache directory to store which translations are stored in the upper part of the HMB. The other SSDs seems to store this directory in the SSD controller, requiring more memory.

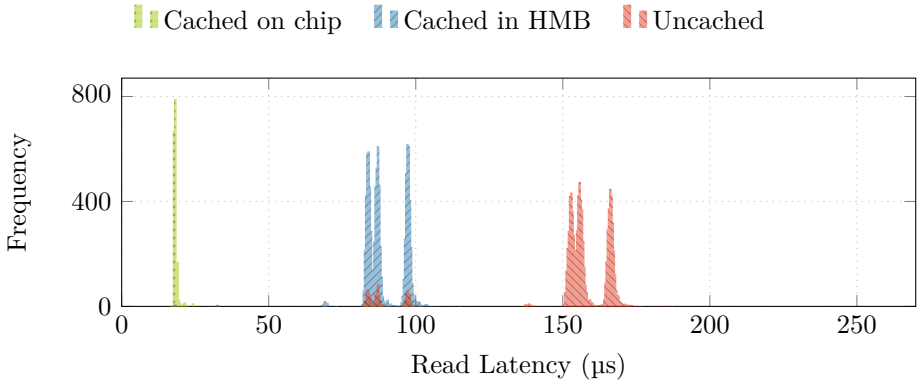
3.2. Cache Timing Differences

In this section, we show that the timing differences caused by the HMB cache are measurable and clearly distinguishable by user space applications, see Figure 9.3. We start by measuring with direct access to the block device and verify that we can also distinguish the cache levels when accessing files as an unprivileged user.

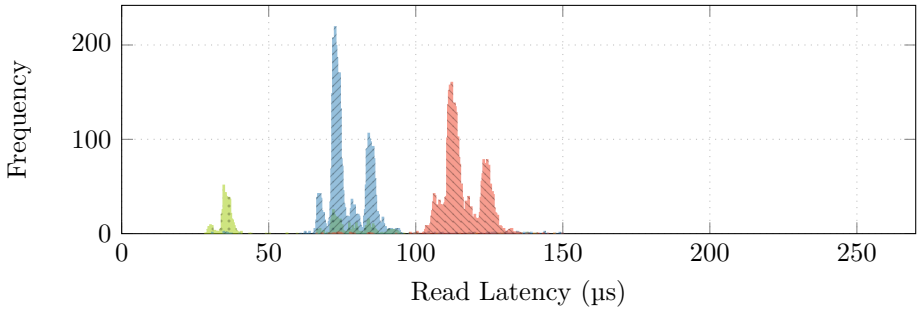
For this experiment, we first "reset" the HMB state by accessing 50 000 random pages of the SSD. Then we access 500 additional random pages and store their addresses. As the translations to these addresses are now cached in the HMB, we put them in our "HMB cached" set. The last N_C of those, we put in the "on-chip" set. With the sets initialized, we randomly choose between accessing a random, very likely uncached page, a random page from the HMB set or a random page from the "on-chip" set. We repeat these access 200 times and continuously update the sets of HMB cached translations by adding all new performed uncached accesses and the "on-chip"-set by always keeping the last N_C accesses in this set. The size N_C is only an estimate for the on-chip cache size because knowledge of the exact size is not required for our attacks.

Samsung 990 EVO. Figure 9.3a shows the timing histogram. The timings are separated by 50 µs with a threshold between uncached and HMB cached accesses of 125 µs. The on-chip cache has an approximate size of $N_C = 150$ entries.

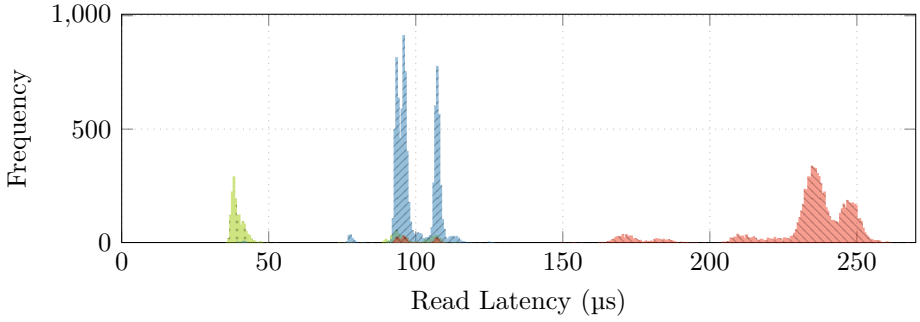
Lexar NM790. Figure 9.3b shows the timing histogram. The cached accesses are still clearly separated from the uncached accesses with a threshold of 100 µs. The on-chip cache is smaller with an approximate size of only $N_C = 40$ entries.



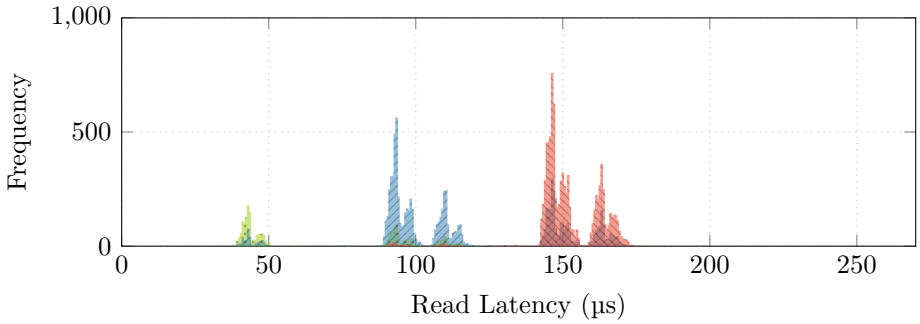
(a) SSD $\mathcal{A}$ (Samsung 990 EVO)



(b) SSD $\mathcal{B}$ (Lexar NM790)



(c) SSD $\mathcal{C}$ (Samsung 980)



(d) $\mathcal{D}$ when accessing each page twice.

Figure 9.3.: The timing differences between a translation cached on-chip, cached in the HMB and an uncached translation are clearly distinguishable on our SSDs.

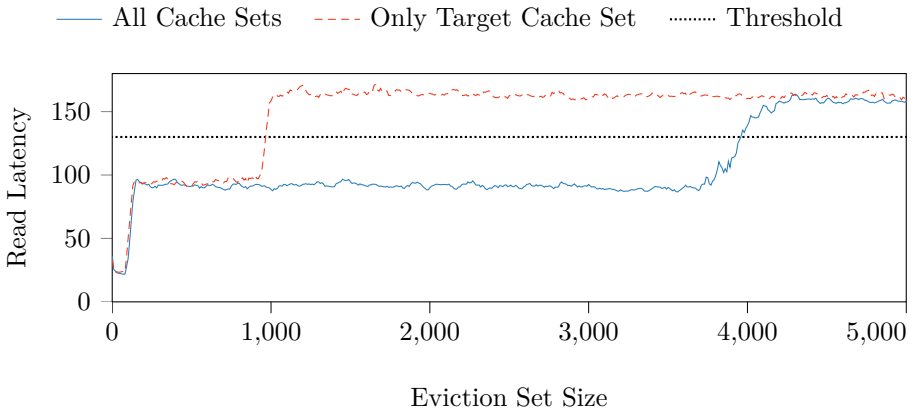
Samsung 980. Figure 9.3c shows the timing histogram. The timings are again very clearly separated with a threshold of 130 µs. The Samsung 980 has a very small on-chip cache $N_C \approx 5$.

WD Blue SN550. 9.3d show the timing histograms when accessing every page twice. Almost no translations are cached, when performing the experiment like on the other SSDs by only accessing each page once. With two accesses, translations get cached but there are still many uncached timings for recently accessed pages.

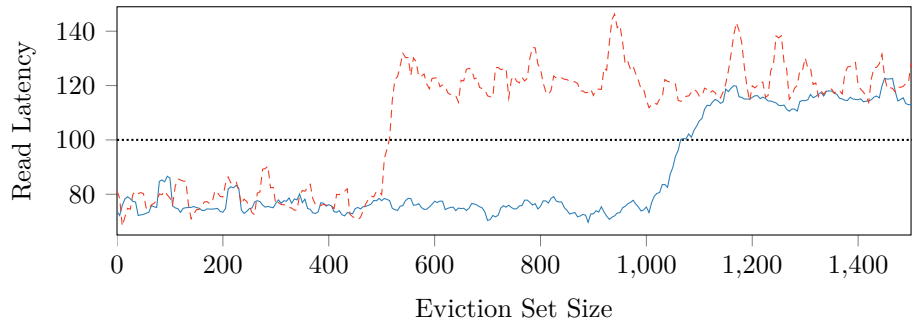
Unprivileged File Access. While we measured the histograms shown in Figure 9.3 by directly accessing the block device, we verified that we measure the same timings when accessing files as an unprivileged user. As already shown by Juffinger et al. [9], the overhead from the file system is negligible.

3.3. HMB Eviction

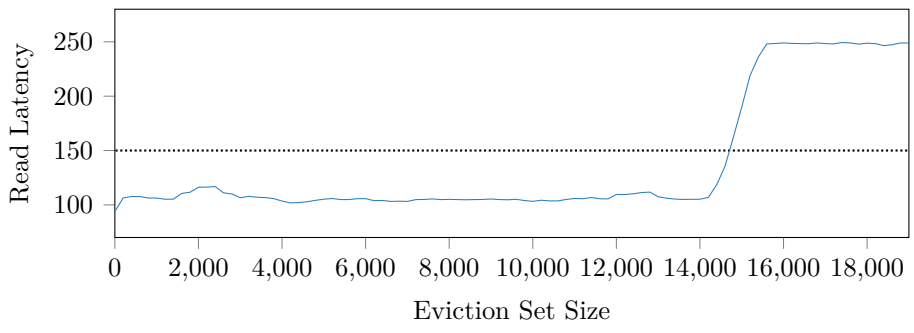
Measuring the HMB state of a target file is destructive. After the read, the translation is cached and has to be evicted again quickly. In this experiment, we find the minimum number of accesses required to evict the HMB. To do this, we read a random page from the SSD, caching the translation in



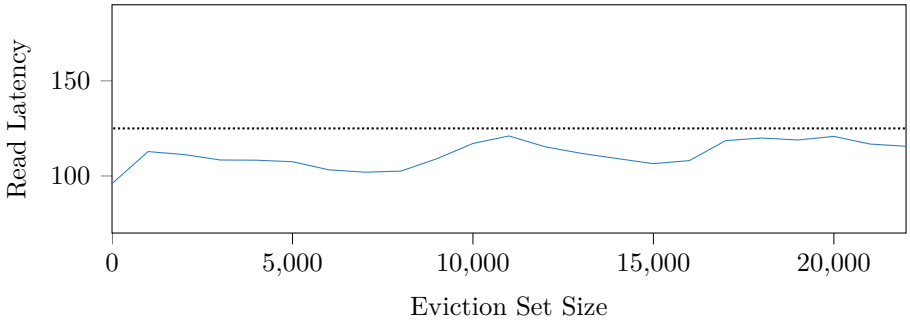
(a) SSD $\mathcal{A}$ (Samsung 990 EVO)



(b) SSD $\mathcal{B}$ (Lexar NM790)



(c) SSD $\mathcal{C}$ (Samsung 980)



(d) No eviction on SSD $\mathcal{D}$.

Figure 9.4.: Eviction with accesses to random pages of the SSD.

the HMB. Then we access an increasing number of pages, either randomly or sequentially. For the sequential accesses, we do not access directly neighboring pages but pages with a stride length of one HMB translation range. We use `io_uring` to submit the reads asynchronously to maximize IOPS. Finally, we access the randomly selected page again and measure the access time. Based on the thresholds from Section 3.2, we know if the page was evicted or not.

Samsung 990 EVO. Figure 9.4a shows eviction with random accesses to all or one cache set. When accessing random addresses, 4 000 accesses are required to evict a translation from the HMB. This takes on average 45 ms. However, due to the four sets, a more targeted eviction is possible. By only accessing addresses that fall in the same set as the target, with bits `14` and `15` matching, the eviction only requires 1 000 accesses and takes on average 22 ms.

When accessing the SSD sequentially, 17 million accesses are required to evict the HMB. Our hypothesis is, that sequential accesses are optimized to not pollute the HMB with translations only used once. Translations evicted from the on-chip cache could, e.g., be dropped instead of being moved to the HMB.

Lexar NM790. Figure 9.4b shows eviction with random accesses to all or one cache set. It takes at least 1 150 random disk accesses, which takes on average 11 ms. When only accessing the same cache set as the target is in, by fixing bit `25`, the eviction time is halved to 530 accesses (6 ms).

However, when only accessing addresses from the same cache set, there is a significantly higher variance in the access timings: $\sigma = 29.5$ vs $\sigma = 15.7$ when accessing both sets, $n = 1000$.

Samsung 980. On the Samsung 980 it takes around 15 000 random accesses (340 ms) to evict a translation from the HMB, as shown in Figure 9.4c. Sequential accesses, even for over 20 s, only very infrequently evict the target from the HMB.

WD Blue SN550. We did not manage to reliably and quickly evict a translation from the HMB on the WD Blue SN550 with neither sequential nor random accesses, as shown in Figure 9.4d. Even when accessing more than a 100 000 pages, multiples times each, for over a second, our target page translations was still fast. Our hypothesis is, that the SN550 can store more detailed translations due to a bigger directory. We show in Section 3.1 that the SN550 accesses the lower 2 MB of the HMB for every translation, hinting its usage as a directory. While other SSDs store their directory on the controller, they only store translations of large blocks of addresses in the HMB. Storing translations of only small blocks, maybe even in page granularity, makes eviction very tedious.

4. Attacks On Accessible Files

In this section, we show a covert channel and a UI redress attack. In this threat model, the attacker has read permissions for the file they want to observe. This means that the attacker can measure the translation cache state of the file they want to observe directly by reading it. We show a covert channel between processes that share access to files in `/usr` and other shared directories in Section 4.1. The covert channel reaches up to 8.3 kbit/s. In Section 4.2, we show a UI redress attack that can detect when `pkexec` is started with a high accuracy.

As we could not reliably evict the HMB of the WD Blue SN550, we are not further evaluating our attacks with it.

Cache Eviction. The big advantage of the HMB side channel over the page cache side-channel is that evicting the HMB is a magnitude faster while causing no memory pressure. Gruss et al. [6] managed to evict the page cache in 149 ms, while it took Juffinger et al. [9] 347 ms with lower memory pressure. Evicting the HMB takes only 22 ms or less on the Samsung 990 EVO and Lexar NM790. This enables a higher sampling rate for more accurate attacks and smaller blind spots. For even faster eviction on the Samsung 990 EVO or Lexar, the attacker needs to know the cache set of the victim file. For this, we use the unprivileged `FIEMAP` ioctl to get the physical mappings of the file on the disk.

If the target we want to observe is in the page cache, we have to evict it. This is once at the beginning and then only after the event we monitor, like the execution of a binary, actually happens. We perform our HMB timing measurements with `O_DIRECT` to not load the target into the page cache.

Preventing False Positives. Due to the design of the HMB, multiple translations are cached together in what we call the HMB translation range. While we exploit this in blind attacks in Section 5, they can cause false positive measurements. A false positive happens if not our target file is accessed and its translation therefore cached in the HMB, but any other file that is in the same HMB translation range. We can reduce the chance of this happening with the *help* of the page cache.

Again, with the unprivileged `FIEMAP` ioctl, we can get all files we have read permissions for that are in the same HMB translation range. On a not strongly fragmented file system, the chance that we have the permissions to read neighboring files is high. If our target is, for example, a binary or library file, they probably reside next to other binary or library files which are typically readable on Linux. After learning which files could cause fault-positive measurements, we read them to load them into the page cache. Therefore, every other program reading any of these files does not access the SSD but gets the file served from the page cache. We then add these files to our page cache eviction set, so they are not evicted when we have to evict our target file from the page cache.

Table 9.2.: All covert-channel results.

Threat Model	SSD	# Co-Locations	Transmission Rate	Bit Error Rate	Channel Capacity
Process (Section 4.1)	$\mathcal{A}$	256	4.4 kbit/s	3.9 %	3.3 kbit/s
	$\mathcal{B}$	512	8.8 kbit/s	0.7 %	8.3 kbit/s
	$\mathcal{C}$	256	594 bit/s	1.4 %	532 bit/s
Virtual Machines (Section 5.1)	$\mathcal{A}$	1	7.1 bit/s	0 %	7.1 bit/s
	$\mathcal{B}$	1	1.9 bit/s	0 %	1.9 bit/s
	$\mathcal{C}$	1	1.7 bit/s	0 %	1.7 bit/s
Remote (Section 5.2)	$\mathcal{A}$	1	10 bit/s	1.5 %	8.9 bit/s
	$\mathcal{B}$	1	4 bit/s	3.0 %	3.2 bit/s
	$\mathcal{C}$	1	2.7 bit/s	0.9 %	2.5 bit/s

4.1. Covert Channel Across Processes

In this section, we show a covert channel across processes that can transmit data with up to 8.3 kbit/s, as shown in Table 9.2.

Threat Model.

Two processes are on the same machine without any means of communication. They have access to a shared set of files, e.g., in in `/bin`, `/lib` or `/usr` and a precise clock source like `rdtsc` or `clock_gettime`. If no process has access to enough files to evict the HMB, at least one of the processes must be able to create a file for HMB eviction. No other privileges are required.

Implementation.

The basic idea is to find a set of files both processes have access to and use each file to transmit a bit of the message. This enables for the transmission of multiple bits per time slice and HMB eviction. For this to work, two requirements for these files must be met: First, each file used for the transmission must be in its own HMB translation range, so they are all independent of each other. Second, the two processes must agree on the files used for the transmission and the order of these files.

File Independence. If two files used to transmit individual bits were in the same HMB translation range, setting one bit would always also cache and, therefore, set the other bit. We use the unprivileged `FIEMAP` ioctl to get the physical block location of each file on the disk. The sender and receiver can then select files that are spaced out enough on the disk. Large or fragmented files can also be used to transmit multiple bits if they span multiple HMB translation ranges. We do not perform *false positive prevention* for our covert channel.

File Selection and Order. Both, the sender and receiver must use the same files in the same order to transmit the message. To achieve this, we only use files that are world readable, if a file is readable due to owner or group permissions, it is not used. To use the files in the correct order, we lexicographically order the full file paths and then randomize them with a seed known to the sender and receiver. The randomization ensures that we do not perform too many sequential accesses during transmission which could trigger HMB optimizations of the SSD.

Transmission. The sender and receiver open the files with the `O_DIRECT` flag to evade the page cache. With both processes sharing a precise clock source, we can use a time-sliced approach like prior work [4, 9, 13, 19, 24]. This means that the transmission is divided into time slices known to the sender and receiver.

In the first time slice, the sender accesses all files corresponding to `1`-bits, to cache their translation. We use *two* files per bit. In the next time slice, the receiver accesses the files and records slow timings from *both* files as a `0` and one or two fast timings as a `1`. The third time slice is used by the sender to evict the HMB. Sender and receiver can swap their roles for bi-directional communication.

We use `io_uring` for asynchronous accesses for sending, receiving, and eviction. While asynchronous accesses are considerably faster, issuing them too fast can cause contention in the SSD [9], slowing down the accesses and interfering with our cache timing measurements. Therefore, we find the optimal delay between the measurement (receive) accesses for all three SSDs. We also had to add a shorter delay between the sending accesses.

We record the raw transmission rate and bit-error rate to compute the true channel capacity, the maximum transmittable information over a noisy

channel. A bit-error ratio of 0 or 1 corresponds to a perfect transmission; 0.5 means no information was transmitted. We use the binary symmetric channel model to compute the true channel capacity T as $T = C \cdot (1 + (1-p) \cdot \log_2(1-p) + p \cdot \log_2(p))$ where C is the raw transmission rate and p the bit-error rate.

Results.

All results are shown in Table 9.2. On the Samsung 990 EVO we transmit 128 bits using 256 files, reaching a raw transmission rate of 4.4 kbit/s with a 3.9 % bit error rate, resulting in a channel capacity of 3.3 kbit/s. We use a 10 µs delay between the send accesses and 50 µs between the measurement accesses. We only access SSD pages in a single set to make the eviction four times faster.

On the Lexar NM790 we got an even lower error rate while doubling the number of files used for the transmission to 512. With them, we transmit 256 bits per time slice for a raw transmission rate of 8.8 kbit/s with a bit error rate of 0.7 %. This results in a channel capacity of 8.3 kbit/s. The raw transmission rate is not only increased by the larger number of files used, but also by the smaller required delays between the send and receive accesses, 3 µs and 20 µs respectively. This enables us to access all 512 files for sending and receiving, in the same time, we can send and receive 256 files on the Samsung 990 EVO. We use SSD pages from both cache sets. Using a single set increased the variance of the timing measurements and the error rate, decreasing the channel capacity.

The very slow eviction that takes over 350 ms on the Samsung 980, makes its covert channel considerably slower. Accessing the 256 files for sending and receiving takes up only 15 % of each time slice. However, the bit error rate is also very low with only 1.4 %. With a raw transmission rate of 594 bit/s, this results in a channel capacity of 532 bit/s.

4.2. Authentication UI Redress Attack

In this section, we show that we can detect when a file is accessed on the disk with high accuracy. We use this for a user-interface redress attack [2, 6, 20]. In an UI redress attack, the attacker waits for a program to be started to then draw their own fake window over the genuine one. In our example, we detect the start of `pkexec` to steal the password of the victim

user. `pkexec` displays a password promt to execute GUI applications with super user privileges. For this to work well, the latency between the genuine drawn and the fake window drawn over it must be small. The short eviction time of our HMB side channel enables this low latency detection. It takes at most 100 ms to detect the start of `pkexec`.

False Positive Prevention.

To prevent false positives, which would be strange for the victim and could make them suspicious, we load the other files in the same HMB translation range in the page cache. To test the viability of this approach, we use a privileged user to find all co-located files for both a 2 MB and a 60 MB HMB translation range. In the 2 MB surrounding `/usr/bin/pkexec` find that all 15 co-located files are in `/usr/bin`. All of these files are readable by a user without elevated privileges. In the 60 MB range, we find 1 969 files. Constantly keeping all of these files in the page cache is time consuming. We therefore exclude files that are unlikely to be read, like man files for other locales (e.g., `/usr/share/man/id/man1/`), firmware images in `/lib/firmware/`, or `/lib/x86_64-linux-gnu/fwupd-plugins-5/`. This leaves us with 793 files, overall 46.5 MB large, to keep in the page cache.

Some of these files are log files. Keeping them in the page cache does not prevent disk accesses if the file is written. To prevent false positives from writes, we check the last modification date of all of our 793 files when we measure a HMB cache hit. If one or more of the files where written in the last few seconds, we count the cache hit as a false positive.

Implementation.

We implement a page cache eviction algorithm similar to Gruss et al. [6]. However, we only have to perform page cache eviction once at the beginning of the monitoring to evict `pkexec`. We, additionally, perform page cache eviction every minute because our monitoring could miss an execution of `pkexec` due to the blind spot during HMB eviction.

In the monitoring loop, we start by sleeping for 100 ms, during this time `pkexec` could be executed. This means that it takes us at most 100 ms to detect the execution of `pkexec`. During this sleep interval the Samsung SSDs require periodic reads to keep the HMB active and preserve the HMB state, we perform one 4 kB read each millisecond. Then, we perform

our measurement by reading the first page of `pkexec`. Our attacks also works by reading a co-located file like `ping`. It could be less suspicious if a program opens `ping` instead of `pkexec`.

Performing the measurement caches the translation in the HMB. This creates a blind spot until the HMB is evicted. For the Samsung 990 EVO and Lexar NM790 we know the correct cache set, because we know the physical block address from `FIEMAP`. This enables quick HMB eviction.

Results.

On the Samsung 990 EVO SSD the eviction takes on average 23.2 ms ($n = 1000$, $\sigma = 0.18$ ms). This means that one whole measurement window takes 100 ms + 23.2 ms = 123.2 ms. In this window, we are blind for 23.2 ms or 18.8 %. This is a trade-off between the relative size of the blind spot and the time it takes at most to detect the execution of `pkexec`. On the Lexar NM790, the eviction takes only 6 ms resulting in a blind spot of only 6 %. We chose a sleep duration of 100 ms, because this means that with a 30 fps screen we are at most 3 frames too late to draw our window over the `pkexec` window.

Because our attack process sleeps a lot and also block device accesses are not CPU intensive it, does not cause easily noticeable CPU usage. The attack causes on average 3.3 % CPU usage and 32 MB/s disk usage.

The slow eviction on the Samsung 980 takes approximately as long as page cache eviction. This defeats the main advantage of the HMB side channel over the page cache side channel, making the page cache an easier and more portable attack target.

5. Blind Attacks

In this section, we show how attacks can be performed if the attacker does not have read permissions for the files it wants to observe. Because the attacker cannot read the file it cannot use the `FIEMAP` ioctl to get the exact mappings of the file on the disk. Therefore, we fall back to a templating approach, where the attacker learns which parts of the attacker file are in the same HMB translation range as the victim files. We first show a covert channel between two virtual machines in Section 5.1, reaching up to 7.1 bit/s. In Section 5.2, we show how an attacker can exploit nginx as a confused deputy to build a remote covert channel over the network. This covert channel reaches up to 8.9 bit/s. For both covert channels, we only use a single file for transmission, for simplicity, instead of the 256 and 512 for the process covert channel.

5.1. Covert Channel Across Virtual Machines

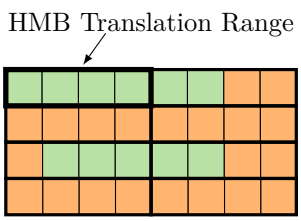
In this section, we show a covert channel between two virtual machines. We achieve up to 7.1 bit/s on the Samsung 990 EVO.

Threat Model.

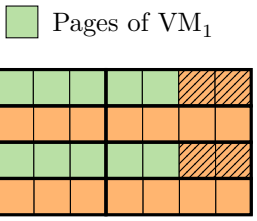
Two unprivileged processes are inside two virtual machines. They want to communicate because, e.g., one process wants to transmit a secret to the other one. The processes have no means of communication. Both processes have the permission to create files on their system. They also have a access to a shared clock. The VM disk is not cached on the host, as recommended, for example, by Red Hat [7]. To enable repeated tries to co-locate, the VM disk must also be configured with TRIM support or even better, with automatic trim when zeros are written [11]. We run our experiments on KVM with libvirt.

Implementation.

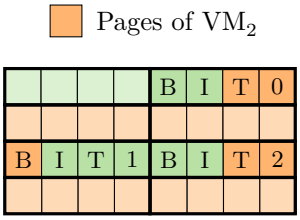
For this attack to work, we need HMB co-location between the VM disks for one VM to be able to influence the cache state of the other.



(a) Step 1: Initial disk layout of sender and receiver files.



(b) Step 2: After the sender accesses the whole file, the receiver gets all pages with cached timings: .



(c) Step 3: In the final step, the sender gets its pages with cached timings.

Figure 9.5.: The algorithm used to find all pages within the sender and receiver files that are in one HMB translation range. With five overlapping pages, five bits can be transmitted in one time slice.

Co-Location. The communicating parties previously agree on a time when they start their covert communication. At the respective time, sender and receiver create a file containing random data. We assume that the operating system in the virtual machine, periodically uses TRIM to return unused disk space to the host, this is the default behavior, e.g., in Ubuntu. Therefore, the disk files of both virtual machines are sparse and grow with the created files, causing fragmentation and interleaving of the disk files.

We find to achieve maximum fragmentation and most overlaps between the two VM disk by writing in a three step process. In each VM, we write a 16 kB block, flush it to the disk using the ‘sync‘ syscall and finally, punch a hole into this block, keeping only the first 4 kB. We repeat these three steps until we wrote between 50 MB and 200 MB. The steps are not synchronized between the VM.

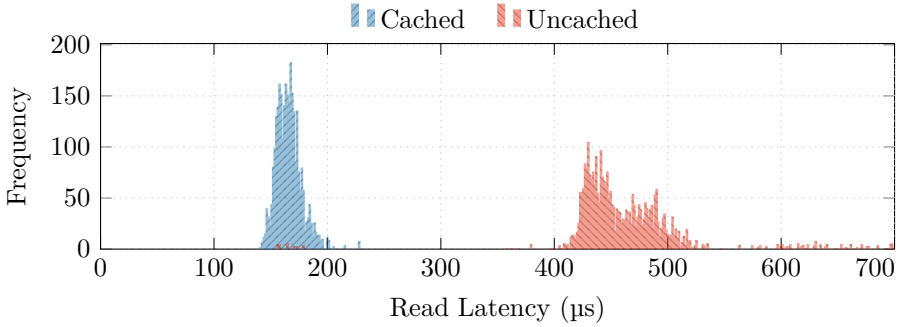


Figure 9.6.: Timing histogram measured from within a VM on SSD $\mathcal{C}$ (Samsung 990 EVO). It seems like libvirt multiplies the timing difference between the cache states, cf. Figure 9.3b.

Co-Location Detection. The virtual machine guest cannot get `FIEMAP` data of the host, therefore, it has to resort to another algorithm to detect which pages (4 kB blocks) of the files are within one HMB translation range, as shown in Figure 9.5. The sender repeatedly access a number of pages of their file to cache the translations. The receiver accesses their whole file and measures the access times. If the receiver measures a cached timing they store the offset. Then the receiver accesses the stored offsets that are potentially overlapping and the sender reads and measures the timings. After doing this for the whole file, the sender and receiver both know which offsets of their files are in the same HMB translation ranges. In a last step, the bit order must be measured. For this, the sender sends 1 bit every second and the receiver records the order in its file.

Transmission. We use the same time-sliced transmission as in the covert channel across processes, see Section 4.1. However, we only implement the transmission with a single bit per time slice for simplicity.

Results.

To measure the overhead introduced by the disk virtualization of libvirt, we transmit a toggling bit between two virtual machines and measure and plot the timings on the receiver side. Figure 9.6 shows the resulting histogram. The timings are clearly more separated than in the native scenario, cf. Figure 9.3b. On the Lexar NM790, the timings are even more separated, 569 µs ($n = 1000$, $\sigma = 267$ µs) vs 4 881 µs ($n = 1000$,

$\sigma = 2\,298\,\mu s$). We ruled out other reasons for this timing like the page cache. Our hypothesis is, that our highly fragmented disk files span many different HMB translation ranges and the read-ahead of the VM operating system, therefore, multiplies the uncached timing.

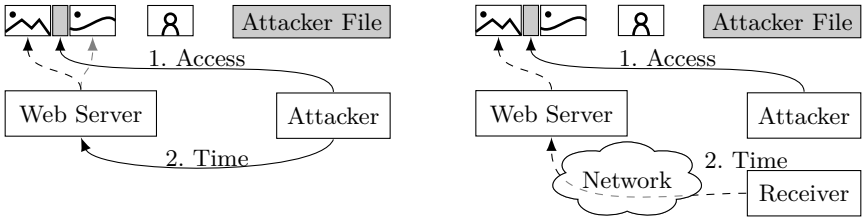
Co-Locations. For the covert channel to work, at least one fragment of the VM disk files each must be co-located in the same HMB translation range. The technique described above maximizes the chance of this happening. We run the file creation five times, always starting from a trimmed VM disk. On average we get 14091 ($n = 5$, $\sigma = 11002$) co-locations on the Lexar NM790. In one case no co-location was found. The larger HMB translation ranges of the two Samsung SSDs increase the number of co-locations respectively.

Transmission. We transmit random data to measure the transmission and error rate of the covert channel. Because we only transmit a single bit per time slice for simplicity, the transmission is considerably slower than with the process covert channel. However, the high average number of co-locations we found between the VMs would allow for multi bit permission with higher transmission rates.

On the Lexar, the large delay for an uncached accessed and the resulting slow eviction time of 500 ms results in a slow transmission of 1.9 bit/s. However, due to the large separation between cached und uncached timings, we did not measure a single bit error over 400 transmitted bits. On the Samsung 990 EVO, we achieved a raw transmission rate of 7.1 bit/s, again without any errors over 400 transmitted bits. The HMB eviction takes 120 ms. On the Samsung 980 we, we achieved a raw transmission rate of 1.7 bit/s, again without any errors

5.2. Remote Covert Channel Attack

We show that an attacker can communicate through a benign website by inducing timing differences into HTTP responses. Schwarzl et al. [21] showed that timing differences of only 60 µs are measurable over 14 hops on the internet. This matches the timing differences on the Samsung SSDs.



(a) Templating: The attacker accesses all offsets of the attacker file and learns about co-located assets by timing the web server.

(b) While the attacker transmits a known sequence the receiver probes all assets and finds the one used for the transmission.

Figure 9.7.: The remote covert channel exploiting a benign web server as a confused deputy. The translation to one of the assets is either cached in the HMB or not.

Threat Model.

We assume a threat model with two processes running on a system: First, a networked application, e.g., a web server like nginx, that is reachable by a remote attacker. Second, an unprivileged application that is isolated from the network but can access the web server through localhost, *i.e.*, the process has access to secret information but no outside network access. We assume that both application have access to a (non-overlapping) set of files on the disk, *i.e.*, they can perform disk operations but not directly communicate through any shared writeable or read-only files. For example, the attacker process has *no* access to the website assets served by the web server. This is typical, with the web server running with its own user. The web server uses direct disk I/O for large files, see Section 5.2. If the attacker had access to website assets, the templating phase could be skipped.

Communication Protocol.

Figure 9.7 shows the attack overview. The basic idea is that the sender process either caches or not caches the translations to assets of the website. For this, the sender process must co-locate a file with a websites asset. If the receiver then accesses this asset they can infer if the translation was cached or not based on the response time of the HTTP request.

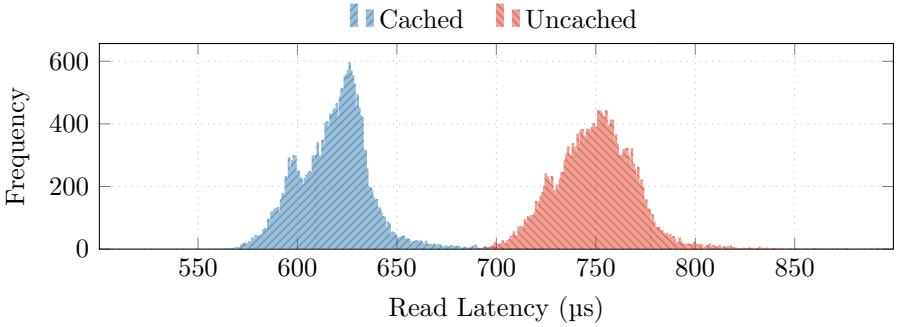


Figure 9.8.: Timing histogram of the Samsung 990 EVO measured over the network, exploiting nginx as a confused deputy. The distributions are spread out but overlap only barely, cf. Figure 9.3a.

Implementation.

For this attack to work we need HMB translation range co-location, direct I/O and low measurement overhead on the receiver.

Co-Location. To gain reliable co-location with many website assets, we create a large file, filling the whole disk for a brief moment. After creation of the file, we instantly free the first 90 % using `FALLOC_FL_PUNCH_HOLE`. The high disk pressure only persists for less than a second. In a final step, we punch more holes into the file to keep only one page per HMB translation range. Using the `FIEMAP` ioctl, we know which file offsets to free. This further reduces the size of our file, while we keep at least one page in many HMB translation ranges. If we do not yet have co-location with a usable website asset, there is a high chance that new assets will be co-located with fragments of our file.

Direct I/O. For this attack to work, we again have to evade the page cache. The nginx web server can be configured to use direct disk I/O with `O_DIRECT` on Linux when serving large files [16]. This has the advantage, that these big files then do not pollute the page cache. The nginx documentation recommends to enable it for files with 4 MB or more [16].

To minimize overhead and noise from the network transmission, we want to transmit as little data as possible. However, the asset served with `O_DIRECT` are multiple megabytes large. To circumvent this restriction, we

use the range requests feature of HTTP [15]. It allows the client to request only a specified part of a file. We find that nginx always accesses the disk, when requesting at least 32 kB. Shorter ranges are still cached somewhere as we did not see nginx access the disk when requesting a shorter range more than once.

Templating. Before the transmission can start, the sender must find website assets co-located with its file fragments. First, the sender traverses the whole website to find files larger than 4 MB. Then it transmits a known pattern through up to 512 fragments of its file using the HMB and measures the access times to the assets though localhost. If the assets are large enough to span multiple HMB translation ranges, HTTP range requests are used to check multiple file offsets.

Sender. We again split one time slice in three parts. One for the sender to evict the HMB, the next to cache the required HMB translations, and the last for the receiver to measure the access time. We synchronize the sender and receiver using `clock_gettime` with the `CLOCK_REALTIME` clock source. After evicting the HMB, the sender accesses the co-located file fragment if the bit to transmit is a `1`, caching the translation in the HMB. While waiting, the sender periodically accesses the disk to prevent it from going into a low power state.

Receiver. The receiver measures the timing of the HTTP request, using the `Range` header to limit the response to 32 kB. While the receiver is waiting for the response to the HTTP request, Linux halts the core the receiver is running on, adding large timing variations to the timing measurement of the request. To mitigate this, we keep the core awake by running a busy loop in the sibling SMT thread while waiting for the response. At the beginning of the transmission, the receiver also has to traverse the whole website to get all large assets and then measure all of them to detect the one that is transmitting a known sequence.

5.3. Results

We measured the covert channel using nginx/1.18.0 between two machines in the same network connected by one switch. Figure 9.8 shows the timing histogram of the Samsung 990 EVO. The added timing overhead from

nginx and the network is not large enough for the two distributions to overlap, except for some outliers. This allows us to transmit 10 bit/s with an error rate of 1.5 %, resulting in a true capacity of 8.9 bit/s. The faster eviction on the Lexar NM790 allows for a raw transmission rate of 20 bit/s. However, the cached and uncached timing overlap, therefore, we use 5 measurements per bit, decreasing the raw transmission rate to 4 bit/s. With an error rate of 3.0 % this results in a true capacity of 3.2 bit/s. The slower eviction of the Samsung 980 slows down the covert channel to a raw capacity of 2.7 bit/s with an error rate of 0.9 %, for a true capacity of 2.5 bit/s.

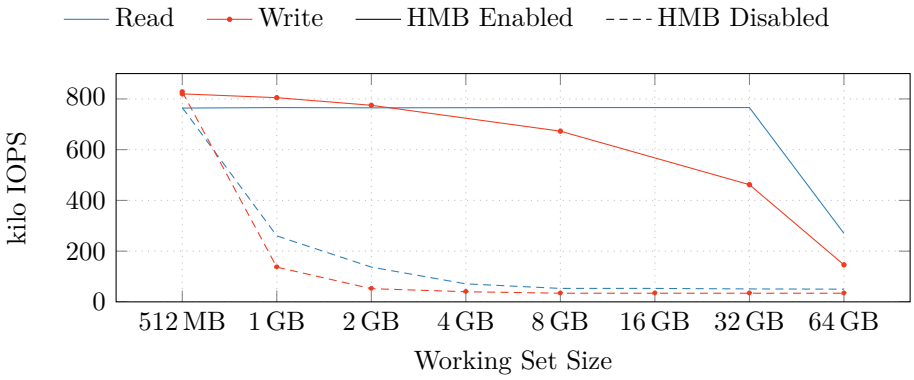
6. Countermeasures

There are multiple ways to mitigate the HMB side channel or make exploiting it more difficult.

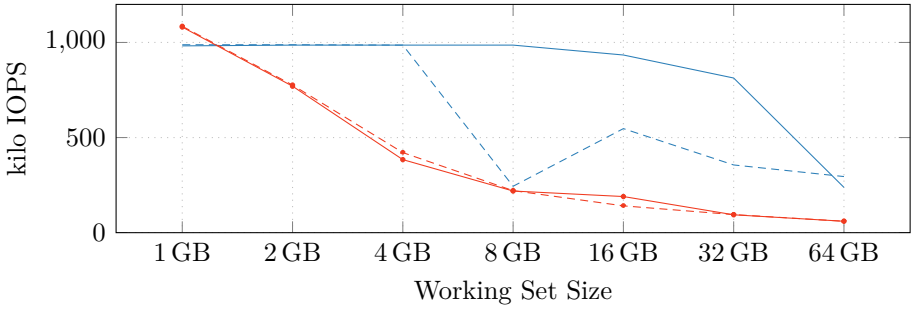
Disabling the HMB.

The root cause of our side channel is the host memory buffer. Every SSD with the HMB feature must also work without host memory resources [17], albeit with reduced performance. On Linux the HMB can be disabled for all SSDs in the system with the kernel command-line parameter `nvme.max_host_mem_size_mb=0`, on Windows it is possible by setting the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\StorPort\HMBAllocationPolicy` to 0.

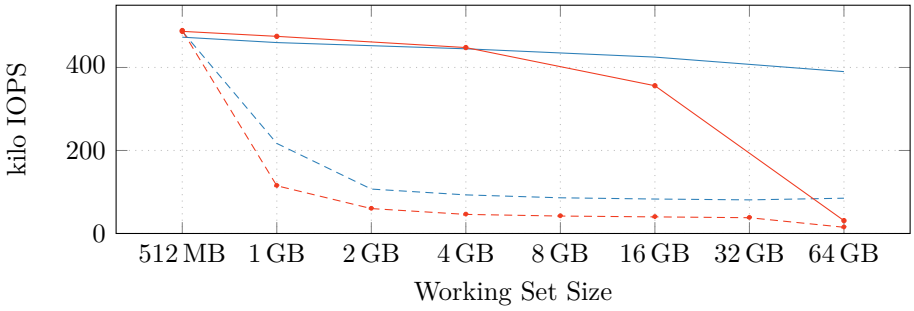
Figure 9.9 shows the performance degradation on our four SSDs when turning of the HMB. We measured the maximum achievable random IOPS when performing random 4 kB reads or writes with a queue depth of 32 and 16 threads on different working set, *i.e.*, file sizes. On SSDs $\mathcal{A}$ and $\mathcal{C}$, the IOPS of read and write accesses already decline by 80 % with a working set size of only 1 GB. They are practically unusable without the HMB. SSD $\mathcal{B}$ seemingly uses the HMB only for reads, they stay fast up until a working set size of 4 GB. We did not measure any difference in the write IOPS. On SSD $\mathcal{D}$ we measured a performance improvement when disabling the HMB over multiple measurements. However, it is also the SSDs least affected by the side channel. We did not see any differences in the sequential accesses on any of the SSDs. These results show, that



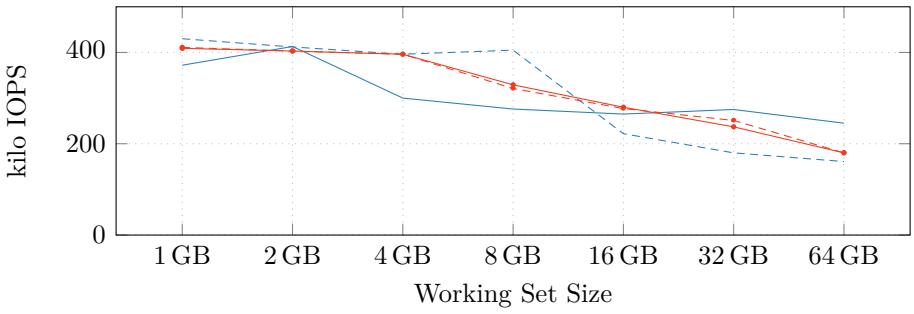
(a) SSD $\mathcal{A}$



(b) SSD $\mathcal{B}$



(c) SSD $\mathcal{C}$



(d) SSD $\mathcal{D}$

Figure 9.9.: The impact of disabling the HMB on random 4 kB IOPS at different working set sizes. The queue depth for the measurement is 32 with 16 parallel threads.

depending on the workload and SSD, disabling the HMB can have a significant impact on the performance.

Modifying HMB Usage.

We also see very different behavior of the HMB. The two newer PCIe 4.0 SSDs use LRU cache replacement which makes eviction very fast. On the Samsung 980 eviction takes more than 10 times longer and on the WD Blue SN550 we could not reliably evict the HMB. These older HMB implementations may have other disadvantages but they are superior security wise as they make attacks slower or impossible.

Making Software Interfaces Privileged.

We use `O_DIRECT` to bypass the page cache for our timing measurements. Disabling it would not prevent the attacks but make them slower by multiple magnitudes, as the page cache would have to be evicted for every measurement. However, `O_DIRECT` is also used by benign applications like databases or as shown in Section 5.2 by nginx, for performance reasons. The usage of `O_DIRECT` could be controlled through a new capability that restricts its use to select programs.

The same is true for the `FIEMAP` ioctl. Restricting untrusted applications from using it would greatly impede our attacks on accessible files.

7. Conclusion

In this work we show that the HMB feature of NVMe SSDs introduces exploitable timing differences measurable from use space. Almost all DRAM-less NVMe SSDs implement the HMB feature and we expect most of them to be exploitable. We evaluate four SSDs and reverse engineer how they use the HMB, what the timing differences are and how we can quickly evict the HMB. Then we show four attacks exploiting the HMB. In the first two attacks we have access to the files we want to observe and build a covert channel with up to 8.3 kbit/s and a UI redress attack that detects the start of an application in less than 100 ms. In the blind attacks we use templating to find co-locations in the HMB and show a covert channel across virtual machines and network covert channel exploiting nginx as a confused deputy. Finally, we show that the HMB side channel can be mitigated by disabling the HMB, albeit with a significant performance impact on some workloads and SSDs.

References

[1] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (p. 270).

[2] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In: S&P. 2017 (p. 287).

[3] Ferraz Gabriel. SSD Database. https://www.techpowerup.com/ssd-specs/. 2025 (p. 273).

[4] Ilias Giechaskiel, Shanquan Tian, and Jakub Szefer. Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. In: ACM Transactions on Reconfigurable Technology and Systems (2022) (pp. 270, 286).

[5] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (p. 270).

[6] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 270, 272, 284, 287, 288).

[7] Jiri Herrmann, Yehuda Zimmerman, Dayle Parker, and Scott Radvan. Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide. 2019 (p. 290).

[8] Jonas Juffinger. An Analysis of HMB-based SSD Rowhammer. In: uASC. 2025 (pp. 270, 275).

[9] Jonas Juffinger, Fabian Rauscher, Giuseppe La Manna, and Daniel Gruss. Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS. 2025 (pp. 270, 280, 284, 286).

[10] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 270).

[11] libvirt. Domain XML format. 2025 (p. 290).

[12] Linux. Fiemap. 2024. URL: `https://docs.kernel.org/filesystems/fiemap.html` (p. 273).

[13] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (pp. 272, 286).

[14] 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. 270).

[15] Mozilla. HTTP range requests — MDN. Feb. 2025. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests` (p. 296).

[16] nginx. directio — Docs ngx_http_core_module. https://nginx.org/en/docs/http/ngx_http_core_module.html#directio. 2025 (p. 295).

[17] NVM Express, Inc. NVM Express, rev 1.2.1. 2016 (pp. 270, 273, 297).

[18] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 270, 272).

[19] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security. 2016 (p. 286).

[20] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In: 4th USENIX Conference on Offensive Technologies. 2010 (p. 287).

[21] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical Timing Side Channel Attacks on Memory Compression. In: S&P. 2023 (p. 293).

[22] Martin Schwarzl, Erik Kraft, and Daniel Gruss. Layered Binary Templating. In: ACNS. 2023 (pp. 270, 272).

[23] Linke Song, Zixuan Pang, Wenhao Wang, Zihao Wang, XiaoFeng Wang, Hongbo Chen, Wei Song, Yier Jin, Dan Meng, and Rui Hou. The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems. In: arXiv (2024) (p. 272).

[24] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. Covert-channels in FPGA-enabled SmartSSDs. In: ACM Transactions on Reconfigurable Technology and Systems (2023) (pp. 270, 286).

[25] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 270, 272).

10

Secret Spilling Drive: Leaking User Behavior through SSD Contention

Publication Data

Jonas Juffinger, Rauscher Fabian, Giuseppe La Manna, and Daniel Gruss
Secret Spilling Drive: Leaking User Behavior through SSD Contention
In: NDSS 2025

Contributions

Main Author.

Secret Spilling Drive: Leaking User Behavior through SSD Contention

Jonas Juffinger[1], Fabian Rauscher[1], Giuseppe La Manna[2,3], Daniel Gruss[1]

[1]Graz University of Technology,
[2]Amazon

Abstract

Covert channels and side channels bypass architectural security boundaries. Numerous works have studied covert channels and side channels in software and hardware. Thus, research on covert-channel and side-channel mitigations relies on the discovery of leaky hardware and software components.

In this paper, we perform the first study of timing channels inside modern commodity off-the-shelf SSDs. We systematically analyze the behavior of NVMe PCIe SSDs with concurrent workloads. We observe that exceeding the maximum I/O operations of the SSD leads to significant latency spikes. We narrow down the number of I/O operations required to still induce latency spikes on 12 different SSDs. Our results show that a victim process needs to read at least 8 to 128 blocks to be still detectable by an attacker. Based on these experiments, we show that an attacker can build a covert channel, where the sender encodes secret bits into read accesses to unrelated blocks, inaccessible to the receiver. We demonstrate that this covert channel works across different systems and different SSDs, even from processes running inside a virtual machine. Our unprivileged SSD covert channel achieves a true capacity of up to 1 503 bit/s while it works across virtual machines (cross-VM) and is agnostic to operating system versions, as well as other hardware characteristics such as CPU or DRAM. Given the coarse granularity of the SSD timing channel, we evaluate it as a side channel in an open-world website fingerprinting attack over the top 100 websites. We achieve an F_1 score of up to 97.0 %. This shows that the leakage goes beyond covert communication and can leak highly sensitive

[3]Part of the work was done while affiliated with the Polytechnic University of Milan.

information from victim users. Finally, we discuss the root cause of the SSD timing channel and how it can be mitigated.

1. Introduction

Covert channels and side channels can bypass architectural security boundaries. The concept of covert channels in computers was first described by Lampson [38] in 1973. Computers have numerous shared resources of limited size and the contention or occupancy of each of these resources can potentially open up a channel for covert communication or even as a side channel. The properties of the shared resource thereby dictate the properties of the resulting channel: Caches have a fine spatial granularity and are fast, leading to a high temporal precision [53, 88]. Other channels, e.g., memory bus contention [83, 84], port contention [2, 6], and cache occupancy [67], only have a binary contention state. Information leakage through these channels is purely in the time domain, e.g., frequency and significance of timing variations. Still, these channels can leak substantial sensitive information [2, 6, 67], necessitating research on mitigations.

As the research landscape around high-spatial high-temporal precision channels has grown substantially over the past two decades [36, 53, 54, 88], more recent works investigate channels in other parts of the system, including random number generation logic [16], execution ports [2, 6], execution schedulers [17], cache occupancy [67], the PCIe bus [74], idle states [58, 89], operating system data structures [33], software-based power measurements [37, 41], and frequency scaling [42, 79, 80]. While some channels are not be suitable to attack cryptographic algorithms, others are not suitable to attack user input, and others cannot extract secret kernel information (e.g., KASLR offsets). While covert channels by themselves may have limited impact, it is a best practice for evaluating new side channels in an ideal scenario where the attacker has full control over both sender and receiver.

Given the properties of the various side channels published in the literature, fingerprinting applications, websites, and videos has become part of the standard evaluation methodology for side channels that primarily leak in a single domain, e.g., time, power, size. For instance, sizes of encrypted network packets can be used to fingerprint applications [64, 75], websites [5, 59], or video streams [13]. Spreitzer et al. [68] mounted a closed-world website fingerprinting attack using data-usage statistics

on Android. More recent work [58] demonstrated a video fingerprinting attack based on the detection of network interrupts on the victim machine, indirectly observing network traffic. Shepherd et al. [65] exploit sensor multiplexing across applications to build a covert channel and fingerprint applications. Matyunin et al. [45] exploit the magnetometer as a side channel to fingerprint applications and websites. Website fingerprinting is the most widely used fingerprinting attack used for side-channel evaluation, applied to e.g., memory usage [32], performance counters [21], power consumption [56], cache occupancy [67], interrupt detection [11, 58, 89].

In this paper, we perform the first study of timing channels in modern commodity off-the-shelf SSDs. We systematically analyze the behavior of NVMe PCIe SSDs with concurrent userspace workloads using the unprivileged `io_uring` interface. We observe that exceeding the maximum I/O operations leads to significant latency spikes that are visible to unprivileged userspace processes. In our analysis of 12 SSDs, we narrowed down the number of I/O operations required to still induce latency spikes. Our results show that a victim process needs to read as little as 32 blocks or write as little as a single block to be still detectable by an attacker. We analyze the root cause of the timing differences by measuring the precise time of I/O operations in the kernel. With timing differences much larger than the overall time spent in the kernel, we identify the SSD as the source of the timing channel. Using this timing side-channel we show a covert channel and website fingerprinting attack as shown in Figure 10.1.

We construct a covert channel, encoding bits into reads of unrelated blocks, inducing latency spikes. Our covert channel works across different systems and SSDs, even from processes in a virtual machine. We achieve a true capacity of up to 1 784 bit/s between processes, while also being agnostic to virtual machines (cross-VM) with an capacity of up to 1 503 bit/s and operating system versions, as well as other hardware characteristics such as processor and DRAM.

We evaluate the SSD side channel in an open-world website fingerprinting attack over the top 100 websites and an “open”-class with unknown websites. For previously visited websites, the browser loads static files from its cache on the SSD, which we measure through the contention channel. Based on the website, the amount, size, and timing of these loads vary and create a unique fingerprint.

We achieve an accuracy of up to 97.0 %, which is on-par with other state-of-the-art side channels that exploit significantly faster channels inside

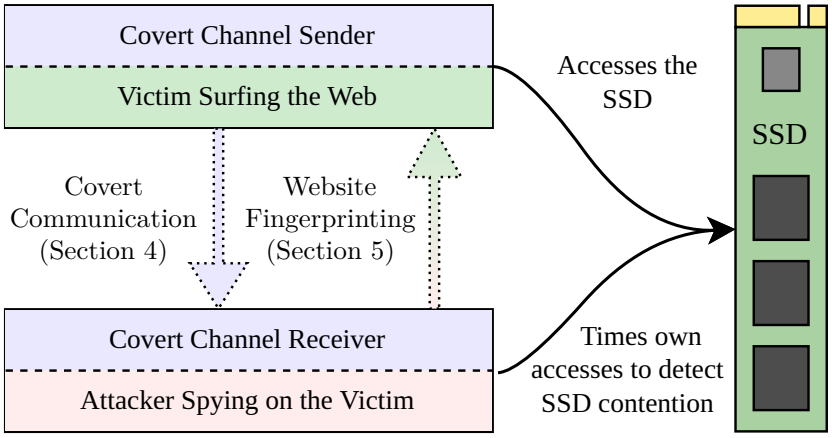


Figure 10.1.: Overview of of our attack. SSD contention is detectable by measuring the round-trip time of accesses. This makes it possible to build a covert channel and to spy on user behavior by fingerprinting visited websites.

the processor [11, 58, 89]. This also shows that the leakage of the SSD timing channel goes beyond covert communication and can leak highly sensitive information from victim users.

We also evaluate the covert channel and website fingerprinting attack under various noise scenarios and show that they can still work with other disk accesses present on the system. Finally, we discuss the root cause of the SSD timing channel and the challenges we identify to mitigate the attack or eliminate the channel.

In summary, the contributions of our work are as follows:

In Section 2, we provide background information. In Section 3, we characterize the timing behavior of commodity SSDs. In Section 4, we evaluate the SSD timing channel in native and cross-VM covert channels. In Section 5, we evaluate the SSD timing channel in a website-fingerprinting attack. We discuss the context of our work and mitigations in Section 6. We conclude in Section 7.

2. Background

In this section, we provide background information on covert and side channels, SSDs, NVMe, and asynchronous I/O.

2.1. Covert Channels

Covert channels date back to Lampson [38] in 1973. Many subsequent works studied covert channels [4, 71], initially exploiting timing variations on multi-user systems [27, 28, 82], and covert channels on inter-connected systems, e.g., using network packets, contention, and latency, later on [8, 49, 76]. Subsequently, the move to virtual machines and cloud computing motivated research on covert channels across virtual machines [51, 57, 61]. Consequently, various covert channels have been presented on modern CPUs, e.g., based on CPU load [51], various CPU caches [25, 46, 47, 48, 55, 60, 81, 86, 88], the memory bus [83, 84], and 4K-aliasing [70]. On the software level, memory deduplication [85], and on a broader system level scale, room temperature [22]. Covert channels have also been studied on GPUs [14, 15, 50], on mobile devices [44, 66], on cloud FPGAs [18, 19], on power-performance measurement and management features [24, 34].

In this work, we focus on covert channels at the outer layers of the memory hierarchy, namely persistent drive storage, *i.e.*, SSDs in our case. Hence, most closely related to our work are covert channels in the context of persistent drive storage: Lipinski et al. [39] present a 0.1 bit/s covert channel through hard-disk drive contention. Lui et al. [43] present a covert channel on (now discontinued) Intel Optane persistent memory. Tan et al. [74] exploit PCIe contention to leak behavior of other PCIe devices connected to the same PCIe link. This is only possible if PCIe switches or PCIe platform controller hubs are used to share a link, which is not generally the case for SSDs. Gruss et al. [20] present a 7 kB/s to 273 kB/s covert channel on the OS page cache. Jiang et al. [33] present a 20 kbit/s covert channel exploiting `fsync` operations on the file system (and disk). Guri et al. [23] mounted an air-gapped covert channel through audio signals emitted by hard disk drives. They highlight that SSDs do not emit such noises and, thus, mitigate this covert channel. Besides the early works on HDD covert channels, no works have studied the timing differences caused internally within commodity off-the-shelf SSDs yet. Chen et al. [9] suggest that SSDs would mitigate certain HDD timing covert channels.

Most closely related to our work are the works by Trochatos et al. who studied covert channels using FPGAs integrated into SmartSSDs [77] and achieved a bandwidth of up to 0.066 bit/s at an error rate of 25 %, as well as temperature generated by the SmartSSD which can then be sensed by the integrated FPGA [78], achieving a bandwidth of up to 0.002 bit/s. Giechaskiel et al. [19] also use an FPGA to mount a covert channel between AWS instances exploiting SSD contention. They report a bandwidth of 0.125 bit/s. All of these works rely on the use of FPGAs, which are not available in most commodity systems, in particular not to unprivileged workloads or inside virtual machines.

2.2. Website-Fingerprinting Side Channels

Building a side channel from a covert channel is not trivial: In a covert channel, sender and receiver can adjust to noise with error correction [48], coarse-grained contention, e.g., cache occupancy [67], already suffices to transmit a signal. In a side channel, the victim is a non-colluding sender that inadvertently transmits information into the channel.

Website fingerprinting often serves as a benchmark for side channels with low spatial resolution or no spatial component at all, e.g., cache occupancy [67]. For a website fingerprinting attack, the attack only needs to distinguish different patterns in the e.g., frequency of interrupts [11, 58, 89], or energy consumption [56], depending on website accesses. Many side channels have been evaluated using website fingerprinting: Spreitzer et al. [68] used data-usage statistics on Android and achieved 89 % accuracy over 100 websites (closed-world). Jana et al. [32] used browser memory statistics and achieved 30 % to 50 % accuracy over 100 000 websites (closed-world). Gulmezoglu et al. [21] used hardware performance counters and achieved 86.3 % accuracy over 40 websites. Qin and Yue [56] achieved an accuracy of 55 % using the power side channel. Shusterman et al. [67] exploited the cache occupancy channel and, in an open-world fingerprinting scenario across 100 websites, the detection accuracy of 77.3 % to 94.8 %. Three works used monitored interrupts or interrupt timings and achieved accuracies from 70 % (top 100, closed-world) [89], and 85.2 % (top 100, open-world) [58], to 95.2 % (top 100, open-world) [11].

2.3. Solid-State Drives (SSDs)

Solid state drives are typically based on persistent flash memory. SSDs have no moving parts and can, therefore, achieve significantly lower latencies. Flash memory operates in larger blocks, e.g., 512 B or 4 kB, written at once. Before overwriting a block, the flash memory needs to be erased by setting the block content to all 1s. As erasing is time-consuming, SSDs only batch erasure of many blocks, typically to multiple megabytes. How often flash memory can be written and erased depends on how many bits are stored in a flash memory cell, ranging from 100 000 for single-level cells to only 1 000 for quad-level cells. To avoid disproportional wear on heavily used locations, SSDs perform wear leveling, balancing writes across all blocks. Hence, the SSD controller keeps track of how often blocks where written and manages a block mapping table, called the flash translation layer (FTL) to map logical to physical block addresses [7]. Hence, different SSD controllers can behave slightly differently, as we also show.

2.4. Asynchronous I/O

`io_uring` is a recent and modern feature of Linux enabling asynchronous low-overhead file operations. The idea is to let the kernel use user-space buffers to copy as little data as possible. The user-space application fills a read queue with requests and then informs the kernel about the new entries. The kernel then processes the read queue, performing the file operations on the I/O-device, and placing the responses in the response queue where the user-space application can directly read them.

Table 10.1.: All SSDs used for our experiments and respective results.

SSD	Model Name	Size	DRAM Cache	PCIe	Covert Channel Process	Covert Channel VM	Website Fingerpr.
$\mathcal{A}$	Samsung 980 Pro	256 GB	512 MB LPDDR4	4.0	808 bit/s	258 bit/s	88.8 %
$\mathcal{B}$	Samsung 980	1 TB	64 MB HMB	3.0	787 bit/s	615 bit/s	93.3 %
$\mathcal{C}$	Samsung 970 Evo	1 TB	–	3.0	1 492 bit/s	303 bit/s	95.9 %
$\mathcal{D}$	Samsung MZVL21T0HCLR-00BL7	1 TB	1 GB LPDDR4	4.0	1 784 bit/s	124 bit/s	95.3 %
$\mathcal{E}$	Samsung 970 Evo Plus	2 TB	–	3.0	1 447 bit/s	1 503 bit/s	96.6 %
$\mathcal{F}$	Crucial P5	1 TB	512 MB LPDDR4	4.0	660 bit/s	404 bit/s	97.0 %
$\mathcal{G}$	Crucial P5 Plus	500 GB	1 GB LPDDR4	4.0	814 bit/s	605 bit/s	91.7 %
$\mathcal{H}$	Kingston SA2000M81000G	1 TB	1 GB DDR3L	3.0	403 bit/s	493 bit/s	80.2 %
$\mathcal{I}$	Toshiba KXG6AZNV1T02	1 TB	–	3.0	664 bit/s	258 bit/s	94.5 %
$\mathcal{J}$	ADATA XPG Gammix S50 Lite	512 GB	512 MB DDR4	4.0	647 bit/s	592 bit/s	88.7 %
$\mathcal{K}$	Gigabyte Aorus Gen4	500 GB	512 MB DDR4	4.0	755 bit/s	702 bit/s	97.0 %
$\mathcal{L}$	Western Digital Blue SN550	1 TB	64 MB HMB	3.0	1 313 bit/s	429 bit/s	96.2 %
	Average				965 bit/s	524 bit/s	92.9 %

The "Average" results show the average transmission rate of the covert channels and the geometric mean of the website fingerprinting accuracies over all SSDs. The SSDs where tested in different machines, always with all four supported PCIe lanes directly connected to the CPU. We will only refer the different SSDs by the letters $\mathcal{A}$ to $\mathcal{L}$ throughout the paper.

3. SSD Characterization

In this section, we analyze SSDs with and without DRAM caches for their behavior under certain contention scenarios. First, we show how the round-trip time changes with an increasing number of read requests. For SSDs with a DRAM cache, there is conflicting information about how the DRAM is used: for the FTL and as a write cache, or also to buffer data reads. We use a timing side channel to show that our set of SSDs with a DRAM cache do not use the DRAM to cache recently read data. Accessing an SSD from user space involves a large software stack from the PCIe NVMe driver and the filesystem driver in the kernel to the userspace library. However, we show that the overhead and timing variance of these layers is negligible in comparison to the latency and timing variations of the SSD. We measure the minimum amount of data read or written by a victim program that we can detect with our timing-based contention side channel. Finally, we show that SSDs are subject to thermal throttling.

3.1. Setup

Hardware. For our initial experiments, we use two systems, one with an AMD Ryzen 7 5800X and one with an Intel Core i7-1260P. On each system, we test a subset of the SSDs listed in Table 10.1. All SSDs are connected to the CPU directly without any PCIe switches and use all 4 supported PCIe lanes. We use a selection of SSDs with different controllers, with and without a DRAM cache and using PCIe 3.0 and PCIe 4.0. The higher number of tested SSDs by Samsung does in no way indicate a higher vulnerability but is due to Samsung being the market share leader for many years [69] and the resulting abundance of Samsung SSDs in our testing environment.

Software. The AMD system runs Ubuntu 20.04 LTS and the Intel system Ubuntu 23.04. For the experiments, until and including Section 3.4, we minimize the software overhead by performing the measurements as follows.

Measuring with Low Software Overhead. We modify the Linux kernel to measure timing on the SSD with the least amount of software overhead possible. We take the first timestamp when the NVMe command is written in the memory and the PCIe doorbell is rang in the NVMe driver of Linux and the second timestamp in the NVMe `command-complete`

interrupt handler. The tested SSD is installed as an additional drive and it does not contain the running OS nor any other data used by the system. For the experiments in Section 3.5 and 3.6, we run the OS on the tested SSD.

For the experiments in Section 3.2 to 3.3, we access the SSDs directly through their block device file at `/dev/nvmeXn0`. For all tests, we use the `io_uring` kernel interface for fast asynchronous I/O and open the file with `O_DIRECT` to bypass all caching of the OS. `O_DIRECT` requires reading a multiple of the SSD block size. Hence, we read 4 kB with every request. The modified kernel returns the request's round-trip time in the `user_data` field of the completion queue entry struct (`io_uring_cqe`).

Measuring with Realistic Software Overhead. For the experiments in Section 3.4 to 3.6, we access the SSD through a file on the `ext4` partition on the SSD, which does not require any elevated privileges. We again use `io_uring` with `O_DIRECT` but we measure the time in user space with an unmodified kernel. For our timing measurement, we store a nanosecond-accurate timestamp (`rdtsc`) right before submitting the SSD operation to the operating system with `io_uring_submit(&ring);` and again when getting the response from the operating system via `io_uring_wait_cqe(&ring, &cqe);`.

3.2. Round-Trip Time (RTT) Experiment

In the first experiment, we test how the round-trip time of individual requests change with an increasing number of accesses in quick succession, hinting at a potentially exploitable contention-based timing side channel. We read random and sequential blocks on the SSDs while continuously increasing the number of read requests we submit to the SSD, called one burst. For small burst sizes the submission of the requests happens simultaneously, for larger burst sizes in quick succession to always keep the SSD maximally busy. Our modified kernel measures the round-trip time of each individual request.

Figure 10.2 shows the results of SSD $\mathcal{A}$. The x-axis shows the number of read requests of each burst. The x-axis time resolution is scaled so that all burst have the same width in the plot. The round-trip time increases with an increasing number of read accesses on all SSDs. Similarly, the standard deviation of the round-trip times also increase. With random accesses, the round-trip time increase is approximately proportional between read sizes

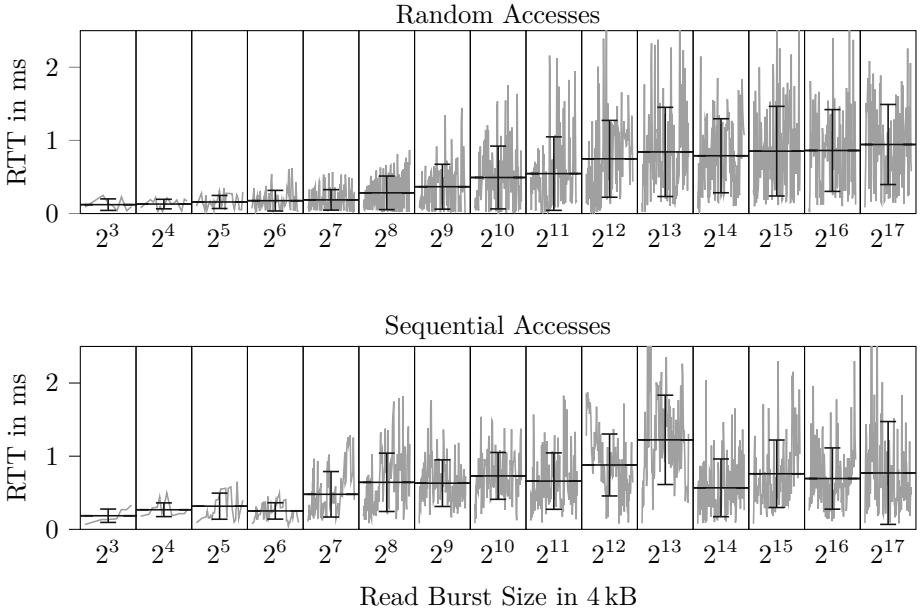


Figure 10.2.: Read access round-trip times with an increasing number of reads in quick succession to SSD $\mathcal{A}$. The x-axis time resolution is scaled so that each burst has the same width. The dark bar shows the mean value and standard deviation of each burst. The round trip time increases with an increasing number of read accesses.

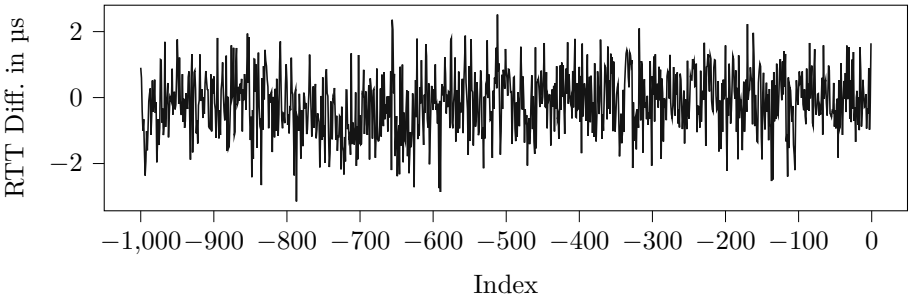


Figure 10.3.: The difference between the round-trip times in backward direction and forward direction on SSD $\mathcal{A}$ averaged over 1 000 measurements. Index 0 is the "turning point" where an address is accessed twice. If the SSD cached recently accessed data, we would see a dip around index 0.

of 2^7 and 2^{13} 4 kB blocks. Above read sizes of 2^{13} 4 kB blocks the round-trip times of individual accesses do not increase significantly anymore. The increase is less continuous when accessing the SSD sequentially. Hence, we can conclude that there is only negligible optimization for sequential reads. The mean round-trip times of sequential as well as random read accesses reach almost 1 ms when reading more than 2^{11} blocks.

3.3. SSD Cache Behavior

For the SSDs with a DRAM cache, we investigate whether it is used to cache recently accessed data. We build a set of random block addresses and access all of them successively, start to end, and then again in reverse order (e.g., $\ldots \to 9 \to 3 \to 1 \Rightarrow 1 \to 3 \to 9 \to \ldots$). If the SSD were to cache data, some accesses after the "turning point" would be faster as they can be served from the cache.

Figure 10.3 shows the difference between the RTT in the backward direction and the RTT in the forward direction on SSD $\mathcal{A}$ averaged over 1 000 measurements. If the SSD cached recently accessed data, we would see a certain amount of accesses after the "turning point" having a lower RTT, *i.e.*, a dip in the graph around index 0. However, the access times do not significantly change on any of our SSDs. This indicates that the DRAM cache is not used to cache recently accessed data but only to hold the FTL and possibly as a write cache.

3.4. Timing Measurement Software Overhead

For the previous experiments, we used a modified kernel to perform the measurements with as little software overhead as possible. However, our goal is to perform measurements on an unmodified system without root privileges. Consequently, we now investigate the overhead and variance an unprivileged software-based attacker can expect to experience. We show that the software overhead is small and adds only little noise.

Implementation. We run this experiment on SSD $\mathcal{F}$ with the operating system installed and running on it. To estimate the software overhead in a realistic unprivileged attack scenario, we compare three different RTT measurement methods: *First*, our modified kernel, *second*, accessing the block device and measuring the delay in user space, and *third*, accessing a file and measuring the delay in user space. Accessing the block device

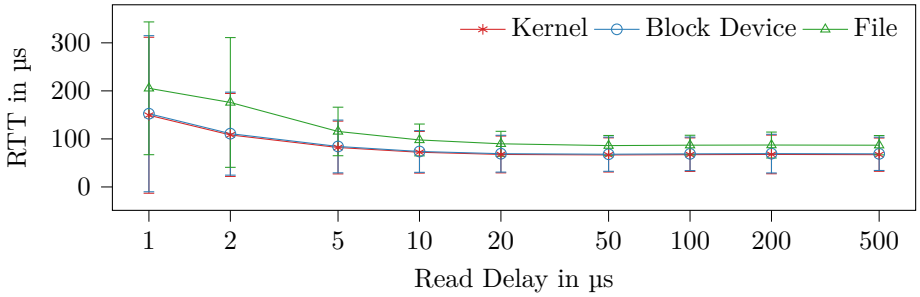


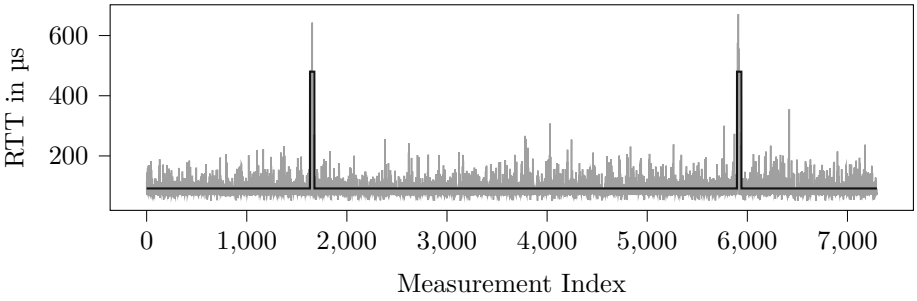
Figure 10.4.: Comparison of RTT measurements in the kernel and in user space in SSD $\mathcal{F}$. The software overhead is minimal and it does not increase the jitter.

adds software overhead from the block device driver. Accessing a file on the SSD adds additional overhead from the `ext4` file system driver that may also access the SSD for file system meta-data.

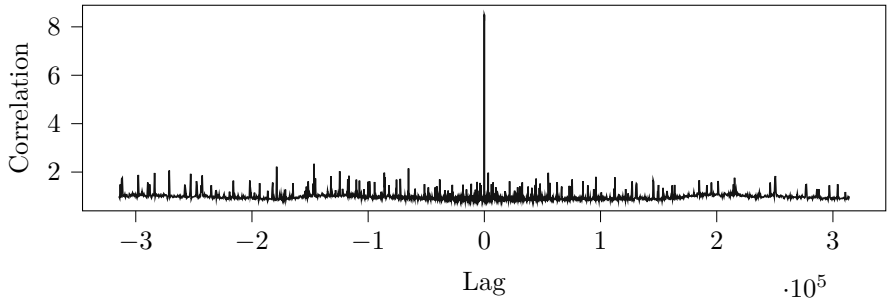
We compare the different measurements by their average access time as well as the standard deviation of multiple measurements. An increase in the average access time shows the constant software overhead but does not influence measurement accuracy. An increase in the standard deviation, on the other hand, shows the unpredictable random jitter that reduces the measurement accuracy. We perform 250 measurements, each with 10 different *read delays* causing different utilizations of the SSD. The read delay defines the sleep duration between two subsequent read requests. The file we access is 1 GB large.

Results. Figure 10.4 shows the measured RTTs and their standard deviations. There is a large difference between read delays $\geq 5\,\mu s$ and $< 5\,\mu s$. We analyze these two cases separately.

If the read delay is $\geq 5\,\mu s$, measuring in the kernel or the block device access leads very similar results, showing that the software overhead of the block device driver is negligible. On average, the kernel measurement is $70.1\,\mu s$ ($n = 250$, $\sigma = 40.3$) and the block device measurement $71.8\,\mu s$ ($n = 250$, $\sigma = 40.3$)). When accessing the SSD through a file, the average access time is higher, with $92.9\,\mu s$ ($n = 250$, $\sigma = 28.3$). The relative standard deviation is very similar, meaning that the file system does not add additional jitter to the measurement.



(a) The plot shows a small section of the round-trip times measured by the observer program. The dark gray line shows the ground-truth signal when the victim program was submitting read burst to the SSD.



(b) The plot shows the cross-correlation between round-trip times measured by the observer program and the ground-truth signal. There is a clear peak.

Figure 10.5.: Measured round-trip time by the observer program and cross-correlation between this measurement and the ground-truth signal on SSD $\mathcal{G}$ with an observer read delay of 10 µs and a victim read size of 2^7 blocks.

If the read delay is < 5 µs, measuring in the kernel or the block device access, still leads very similar results (kernel: 129 µs ($n = 250$, $\sigma = 124$), block device: 132 µs ($n = 250$, $\sigma = 124$)). However, when accessing a file, the measurement shows a higher file system overhead and standard deviation 191 µs ($n = 250$, $\sigma = 137$). This increase indicates a higher jitter, making individual measurements less exact.

We conclude that our SSD timing side channel can indeed be observed by an unprivileged process. For measurements with low jitter the read delay should be at least 5 µs.

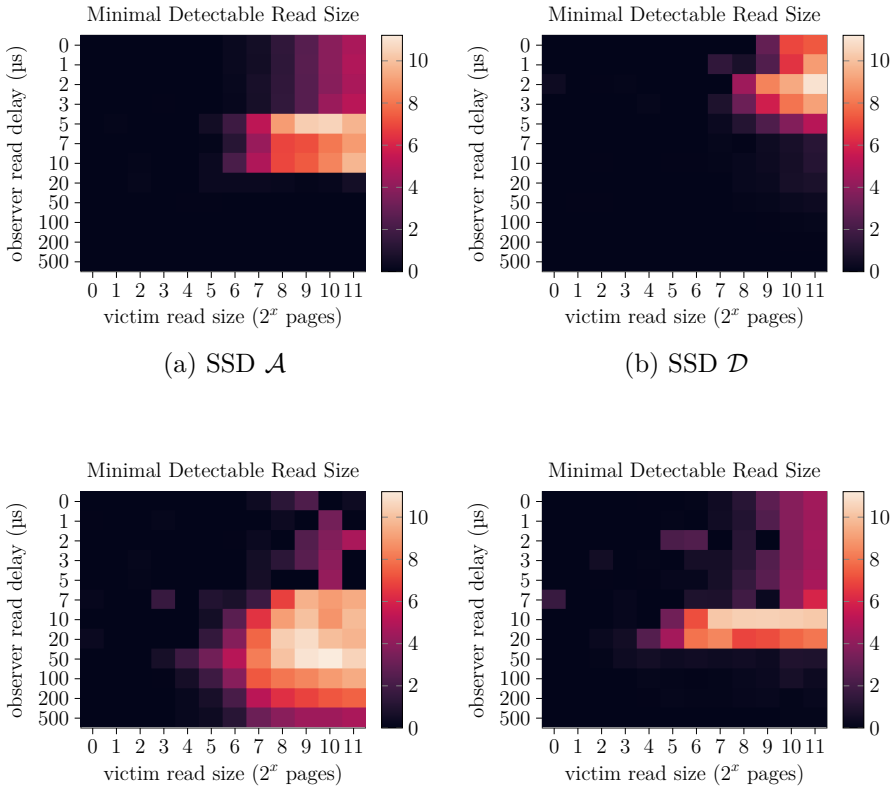
3.5. Minimal Detectable Read Size

The number of read operations influences the contention and, therefore, the round-trip times. Thus, we profile what the smallest detectable number of read operations is, considering two dimensions: First, the *victim read size*, the number of bytes the victim reads in a short succession. Second, the *observer read delay*, the read delay between two read requests submitted by the observer. We sweep both variables and compute the cross correlation between the sent and received signal.

The *observer read delay* has an impact on the detectable read size as the observer reads itself utilize the SSD. If the observer utilizes the SSD too much, it causes a higher variation in RTTs making it difficult to detect a victim signal, see Section 3.2. If the observer utilization is too low, the measurement can miss smaller reads.

Implementation. In this experiment we access two files on the SSD. The "victim" program reads from a file with an increasing number read bursts (*victim read size*). The timestamps of the read bursts are stored. The observer program reads from another file and varies the delays between consecutive reads (*observer read delay*) and measures the RTTs. We assume a threat model where the observer cannot access the victim's file, e.g., no permissions.

For each *victim read size* and *observer read delay*, we compute the cross-correlation between the sent signal and the measured round-trip times by the observer. Figure 10.5a shows the raw signal measured by the observer program and an overlay in red with the ground-truth signal where the victim program read from the disk. Figure 10.5b shows the cross-correlation between these two signals. The significant peak at the center shows a clear correlation. Finally, we compute the ratio between the correlation peak and the maximum value of the surrounding noise. Combining the sweeps of the n *victim read sizes* and m *observer read delays* gives us $n \cdot m$ correlations. A higher value means that more information was transferred, i.e., the signal was better detectable.



(a) SSD $\mathcal{A}$ (b) SSD $\mathcal{D}$

(c) SSD $\mathcal{H}$ (d) SSD $\mathcal{J}$

Figure 10.6.: These heat maps show how many 4 kB blocks a victim program must read from the SSD for the read to be detectable through our SSD timing side channel. Whether a read is detectable by an observer, additionally depends on the observer read delay, *i.e.*, the delay between two timed reads. The value of the heat map is the value of the correlation peak, divided by the maximum value of the surrounding noise, as shown in Figure 10.5b.

Results. Figure 10.6 shows the heat maps for the SSDs $\mathcal{A}$, $\mathcal{D}$, $\mathcal{H}$ and $\mathcal{J}$. Brighter values show a higher correlation between sent and received signal. SSD $\mathcal{H}$ is a PCIe 3.0 SSD, whereas the others use PCIe 4.0. We see different behavior of the SSDs.

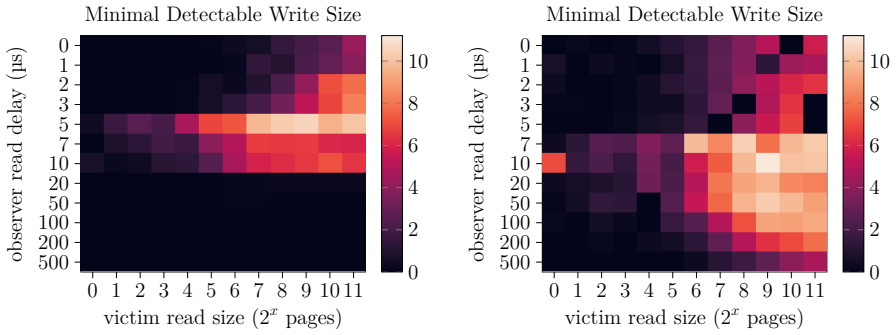
On SSD $\mathcal{H}$, the lowest detectable victim read size is 2^3 blocks or 32 kB with an observer read delay of 7 µs. From an observer read delay of 7 µs upwards the side channel shows clear correlation for a higher number of victim read sizes. Below an observer read delay of 7 µs the SSD only a small number of specific victim read sizes are detectable. A possible explanation would be that these faster accesses cause too much SSD utilization on the slower PCIe 3.0 SSD.

The PCIe 4.0 SSDs $\mathcal{A}$, $\mathcal{D}$, and $\mathcal{J}$, all have a maximum observer read delay where no contention is detectable anymore. On SSD $\mathcal{A}$, it is at 10 µs, on SSD $\mathcal{D}$ at 5 µs, and on SSD $\mathcal{J}$ at 20 µs. We suspect that the multiple command queues and the faster PCIe 4.0 interface allow for efficient scheduling that allows the SSD to respond quickly to the infrequent read requests of the observer process. SSDs $\mathcal{A}$ and $\mathcal{J}$ show a similar pattern, where a small range of observer read delays work significantly better than all others. On SSD $\mathcal{A}$, victim reads down to a size of 2^6 are detectable with observer read delays from 5 µs to 10 µs; on SSD $\mathcal{J}$, victim reads down to a size of 2^5 are detectable with observer read delays from 10 µs to 20 µs. We observe the lowest effect of contention on the observer's timings on SSD $\mathcal{D}$. With an observer read delay above 5 µs almost no reads are detectable. Even with the best observer read delay, the victim has to read at least 2^8 blocks, *i.e.*, 1 MB.

These results show that SSDs behave very differently and that there is no observer read delay that works on all SSDs. We will show that we are still able to build a covert channel by dynamically adjusting to the underlying SSD in Section 4.

3.6. Minimal Detectable Write Size

In Section 3.5, we examined the minimal detectable read size. In this section, we perform the same experiment but with a victim that performs writes to the SSD. The observer program still issues reads to measure the round-trip time.



(a) SSD $\mathcal{A}$ (b) SSD $\mathcal{H}$

Figure 10.7.: These heat maps show how many 4 kB blocks a victim program must write to the SSD for the write to be detectable through the side channel. Whether a write is detectable by an observer, additionally depends on the observer read delay, the delay between two timed reads by the observer program. The value of the heat map is the peak of the correlation, divided by the surrounding noise as shown in Figure 10.5b. Compared to the minimal detectable read size in Figure 10.6, smaller writes are detectable.

Results. Figure 10.7 shows the results for SSDs $\mathcal{A}$ and $\mathcal{D}$. On SSD $\mathcal{A}$ (Figure 10.6a), a very faint signal is detectable down to a single written block of 4 kB, with observer read delays of 5 µs and 10 µs. As this is the smallest unit that can be accessed on this SSD, we want to emphasize that also a write of only a single byte is detectable. Comparing this result to the minimal detectable *read* size of 2^5 blocks (Figure 10.6a) shows that writes cause a significantly higher SSD utilization, e.g., contention in the SSD controller. However, it is interesting to note that the impact of the observer read delay on the detectability does not change. The best observer read delay is in the range 5 µs to 10 µs, with some observability in range 0 µs to 3 µs and no signal with a read delay above 10 µs.

The comparison of read and write detectability is similar on SSD $\mathcal{H}$, smaller writes are detectable than reads and the impact of the observer read delay stays the same. The observer read delay must be at least 7 µs to properly detect writes.

SSD Write Caching. Typical modern SSDs store most of their data in triple- or quad-level cells (TLC & QLC). These cells can store multiple bits but are considerably slower to modify than single-level cells (SLC). The write time (`tPROG`) of QLCs is higher than 1 ms [72], resulting in write

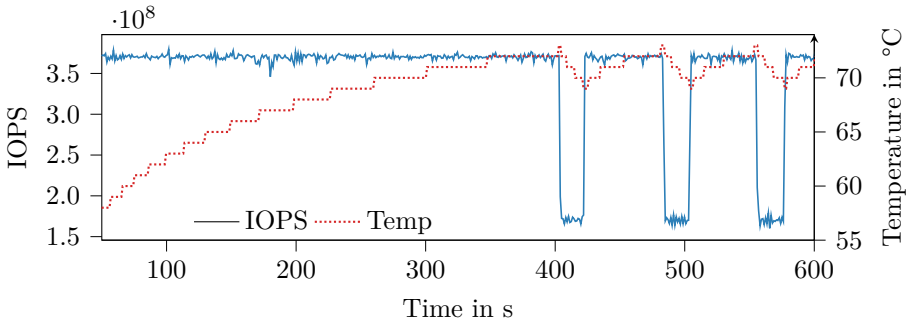


Figure 10.8.: SSD *I* throttling when exceeding 72 °C as observed by the halving IOPS. After cooling down below 70 °C the throttling is reversed.

throughput rates of QLCs in modern SSDs in the range of 10 MB/s [35]. The solution is to run a fraction of the cells as an SLC cache for fast writing. When idling, the SSD controller moves data from the SLC cache into the QLC memory. If the SLC cache is full, the write performance degrades significantly, down to the QLC write throughput rate. While this effect itself could be used to build a covert-channel we expect the transmission rate would be very slow and did not further investigate. Therefore, we only use read operations to measure and cause contention in the remainder of this paper.

3.7. Thermal Throttling

Modern SSDs have a maximum power consumptions of multiple watts [62, 73]. Therefore, to control temperature, SSDs throttle when becoming too hot, which typically does not happen under regular load. Samsung calls this feature "Dynamic Thermal Guard" [62]. To analyze thermal throttling and confirm that it is reliably reversed, we fully utilize SSDs while measuring their temperature and IOPS. As shown in Figure 10.8, SSD *I* throttles when exceeding 72 °C as observed by the halving IOPS. Halving the IOPS leads to a quick decrease in temperature, meaning that only IOPS very close to the maximum actually cause throttling on our systems. After cooling down below 70 °C the throttling is reversed. Because throttling chances the SSD's characteristics we ensured during all following experiments that the SSDs do not throttle by not exceeding certain transmission rates and properly cooling the SSDs. Only for the experiments with 100 % noise that fully utilizes the SSDs, throttling is allowed to happen.

4. Covert Channel

We show that NVMe SSD read contention can be used to build a covert channel between two parties that are isolated from each other and have no legitimate communication channel. We run our experiments mainly on an AMD Ryzen 7 5800X and Intel Core i7-1260P. The detailed system configuration for each SSD is shown in Table 10.3.

We demonstrate our covert channel in two threat models: first, between isolated processes (Section 4.3) with a channel capacity of up to 1 763 bit/s, and second, between separate virtual machines (Section 4.4) with up to 1 503 bit/s. Additionally, we investigate the impact of noise on the covert channel and show little impact at moderate noise levels. The communication protocol and implementation is the same for both variants. Table 10.2 overviews the results.

4.1. Implementation

We implemented a time-sliced covert channel, with a fixed length transmission window for each bit. The raw transmission rate is the inverse of the bit transmission window length. The receiver periodically reads from the SSD and measures the round-trip time of the reads. The sender sleeps to send a `0`-bit and sends a burst of read requests to send a `1`-bit.

To rule out the influence of software caches like the page cache of the OS, both processes open the file with the `O_DIRECT` flag. This does not require any special privileges.

The two processes synchronize their time slices with the help of a shared clock. We investigated two clocks discussed in prior work [40, 52]. First, on x86, reading the time stamp counter (TSC) is possible without privileges with the `rdtsc` instruction, returning a timestamp that is shared across all cores but could be manipulated by a hypervisor. Second, the POSIX `clock_gettime` function also returns time with nanosecond resolution, providing another shared clock.

To adjust for the different SSDs behavior, there are parameters an attacker can tune to optimize the performance. For example, as shown in Section 3.5, the minimal detectable read size as well as the frequency with which the receiver reads (observer read delay) has a big impact. Additionally, depending on the SSD, the duration in which the sender performs read

requests within one time slice to transmit a `1` has an impact on the covert channel performance.

Sender. To transmit a `1`-bit, the sender must continuously send read requests. These read requests to the SSD take some time to finish. If the sender were to submit read requests until the end of the time slice, the higher delays on the receiver side would leak into the following time slice, adding up until all requests are done. To prevent this from happening, the sender performs read requests only for a fraction f of the time slice. We test $f = 0.5$, 0.25 and 0.12.

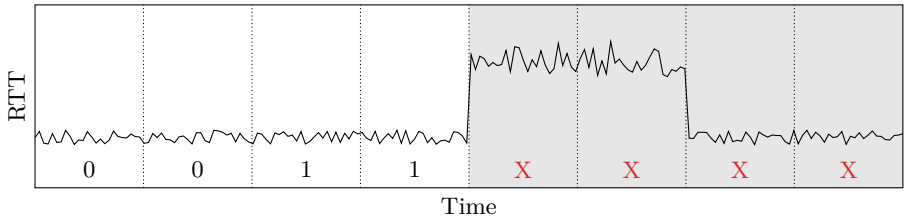
As the sender only performs read requests to induce contention with the receiver, the sender is not interested in the result of the reads. Read requests can be flagged with `IOSQE_CQE_SKIP_SUCCESS` in `io_uring`. With this flag, they skip the completion queue, lowering the CPU load and simplifying the code on the sender side.

Receiver. The receiver periodically reads from the SSD and measures the round-trip time of each access. After every time slice, it performs the threshold training and start sequence detection as described in Section 4.2. If it is able to detect the start sequence with over 80 % accuracy it starts to decode the following bits as data with the learned threshold.

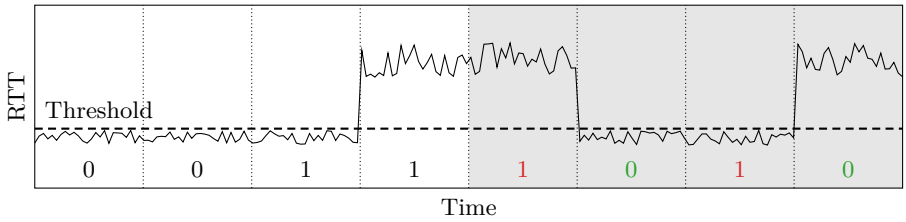
4.2. Threshold Learning & Communication Protocol

As shown in the previous section, SSDs exhibit very different behavior under access contention. The minimum number of read blocks that can be detected vary greatly and is additionally dependent on the read delay of the observer. In both of our threat models we want the covert channel to work SSD agnostic, so the communication parties do not have to know anything about the underlying SSD. This is especially relevant for the VM threat model where this information is not available or for a sandboxed processes where the operating system could choose to hide it. Therefore, for our covert channel, we developed an approach that dynamically adjusts the parameters to the underlying SSD based on the observed timings.

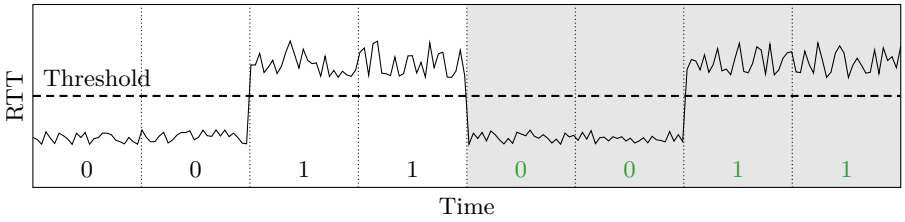
The sender repeatedly transmits two known bytes, the start sequence, in our case `J2`, before every transmission. The receiver does not know when a transmission starts and, therefore, repeatedly performs threshold learning and start start sequence detection as described in the following paragraph.



(a) In the beginning the receiver is not able to learn a threshold because the timings for the `0`-bits and `1`-bits are too similar. Hence, the receiver does not proceed with the next steps.



(b) With the next slice, the receiver is able to learn a threshold which is not yet optimal, and applies it on the test slices. The resulting sequence is not the start sequence.



(c) Finally, the receiver learns an optimal threshold and when applying it on the test slices it is the start sequence. The receiver found a threshold for this SSD and the start of the transmission.

Figure 10.9.: Visualization of the automatic threshold learning and start sequence detection. For this example, the start sequence is `0011`.

Figure 10.9 shows the threshold learning and start sequence detection with an example start sequence of `0011`. The receiver constantly measures. If it measured enough data that could contain start sequences, it splits the measured data into 70 % training set and 30 % test set. Using the training set, it assumes that the start sequence was sent and tries to find a threshold that distinguishes `0` and `1` bits. It computes the mean round-trip times of all segments and then assigns these mean round-trip times to the `0`-group or `1`-group based on the binary representation of the start sequence. The threshold is then computed by taking the mean over all `0`-segments and the mean over all `1`-segments and selecting the access time in the middle of the two. If the transmitter is not sending anything, the start sequence is misaligned or other data is sent, the receiver does not find a proper threshold. For example, the mean over all `0`-segments could then be larger than the mean over all `1`-segments. If a threshold was found, the receiver tests it by trying to classify the segments of the test set, again with the binary representation of the start sequence. Whenever a new bit is received the receiver performs the training procedure.

4.3. Covert Channel across Processes

We mount our first covert channel in a cross-process scenario. It achieves a channel capacity of up to 1 763 bit/s on SSD $\mathcal{D}$ with an average of 964 bit/s across all SSDs.

Threat Model. We assume the sender process has access to secret information the attacker wants to exfiltrate but no network access, e.g., firewall or sandbox restrictions. We assume the receiver process has no access to the secret information but network access. Jointly the two unprivileged processes aim to exfiltrate the secret information by transmitting it through a covert channel. We assume each process has read access to one file on the same SSD but not the same files, e.g., the sender runs in a sandbox. Both processes have access to a clock to synchronize the sending of individual bits. We assume the processes are isolated from each other and have no additional permissions that would help their efforts to communicate, *i.e.*, in particular there are no other communication channels between the two processes. The processes are not pinned to specific CPU cores.

Evaluation. To evaluate the covert channel, we transmit random data for 20 seconds, including the start sequence. On every SSD, we test raw

Table 10.2.: Covert channel results of all SSDs.

	Process					Virtual Machine				
	1 000 bit/s, $f = 0.12$		Fastest per SSD			500 bit/s, $f = 0.5$		Fastest per SSD		
SSD	Error	Capacity	Trans. Rate	Error	Capacity	Error	Capacity	Trans. Rate	Error	Capacity
$\mathcal{A}$	13.4 %	431 bit/s	2 000 bit/s	14.5 %	808 bit/s	17.9 %	161 bit/s	1 000 bit/s	21.0 %	258 bit/s
$\mathcal{B}$	16.3 %	359 bit/s	2 000 bit/s	14.9 %	787 bit/s	15.4 %	191 bit/s	2 000 bit/s	18.6 %	615 bit/s
$\mathcal{C}$	13.1 %	439 bit/s	5 000 bit/s	19.0 %	1 492 bit/s	15.8 %	186 bit/s	1 000 bit/s	18.8 %	303 bit/s
$\mathcal{D}$	19.6 %	285 bit/s	5 000 bit/s	16.4 %	1 784 bit/s	25.8 %	88 bit/s	1 000 bit/s	29.5 %	124 bit/s
$\mathcal{E}$	13.0 %	441 bit/s	5 000 bit/s	19.4 %	1 447 bit/s	13.3 %	217 bit/s	5 000 bit/s	18.9 %	1 503 bit/s
$\mathcal{F}$	17.4 %	333 bit/s	2 000 bit/s	17.5 %	660 bit/s	13.7 %	212 bit/s	1 000 bit/s	14.5 %	404 bit/s
$\mathcal{G}$	15.6 %	375 bit/s	2 000 bit/s	14.3 %	814 bit/s	18.2 %	158 bit/s	2 000 bit/s	18.8 %	605 bit/s
$\mathcal{H}$	25.0 %	188 bit/s	2 000 bit/s	24.2 %	403 bit/s	14.0 %	207 bit/s	2 000 bit/s	21.6 %	493 bit/s
$\mathcal{I}$	13.5 %	428 bit/s	2 000 bit/s	17.5 %	664 bit/s	19.4 %	145 bit/s	1 000 bit/s	21.0 %	258 bit/s
$\mathcal{J}$	13.2 %	437 bit/s	2 000 bit/s	17.8 %	647 bit/s	14.9 %	196 bit/s	2 000 bit/s	19.1 %	592 bit/s
$\mathcal{K}$	29.2 %	129 bit/s	2 000 bit/s	15.5 %	755 bit/s	14.6 %	200 bit/s	2 000 bit/s	16.6 %	702 bit/s
$\mathcal{L}$	13.1 %	439 bit/s	5 000 bit/s	20.8 %	1 313 bit/s	15.1 %	194 bit/s	1 000 bit/s	13.5 %	429 bit/s
Avg	16.9 %	357 bit/s	3 000 bit/s	17.7 %	964 bit/s	16.5 %	180 bit/s	1 750 bit/s	19.3 %	524 bit/s

The capacity is computed from the raw transmission rate and the error rate. The first two columns of "Process" (1 000 bit/s, $f = 0.12$) and "Virtual Machine" (500 bit/s, $f = 0.5$) show the raw transmission rate that resulted in the highest channel capacity over all SSDs. The parameter f defines the fraction of the transmission window the sender was submitting read requests. This transmission rate could be used to test and negotiate a higher transmission rate. The "Fastest per SSD" columns show the highest channel capacity we achieved on each individual SSD.

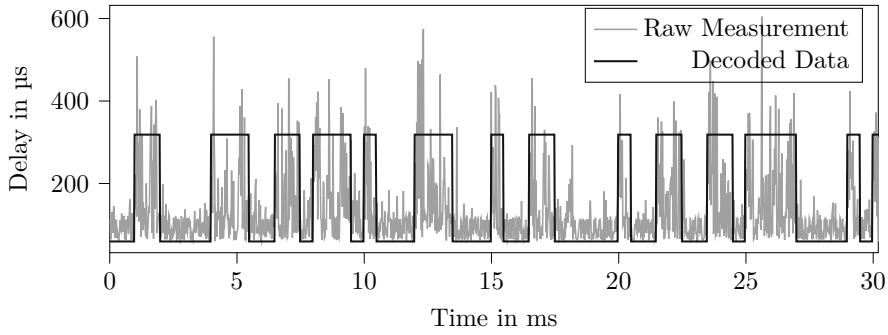


Figure 10.10.: The raw measurements of the access round trip times of the covert channel receiver and the decoded binary signal on SSD $\mathcal{G}$. The transmission rate is 2 000 bit/s, the transmission fraction f is 0.12 and the observer read delay 20 µs.

transmission rates from 5 bit/s to 10 kbit/s. This translates to transmitted data amounts of 6 B to 30.9 kB. After the transmission, we apply the threshold learning and start sequence detection in a post-processing step and then extract the data and compute the Levenshtein distance between the sent and received data, giving us the bit-error ratio. A bit-error ratio of 0 or 1 corresponds to a perfect transmission without errors, a bit-error ratio of 0.5 means that no information was transmitted. We use the binary symmetric channel model to compute the true channel capacity T as $T = C \cdot (1 + ((1-p) \cdot \log_2(1-p) + p \cdot \log_2(p)))$ where C is the raw bit-rate and p the bit-error probability. According to Shannon's noisy-channel coding theorem, T is the maximum transmittable information over a noisy channel.

Results. Figure 10.10 shows the raw measurement trace of the RTTs on the receiver side and the extracted binary data on SSD $\mathcal{G}$. The transmission rate is 2 000 bit/s, the transmission fraction f is 0.12 and the observer read delay 20 µs.

Table 10.2 shows all covert channel results of all SSDs. If the receiver cannot find a threshold or detect the start sequence with at least 70 % correctness, we cannot transmit data on this SSD through our covert channel. When excluding transmission rates and fractions where at least on one SSD no data transmission is possible, a raw transmission rate of 1 000 bit/s with a transmission fraction f of 0.12 resulted in the highest average channel capacity over all tested SSDs of 357 bit/s with an average bit-error ratio of 16.9 %. We find that independent of the underlying SSD,

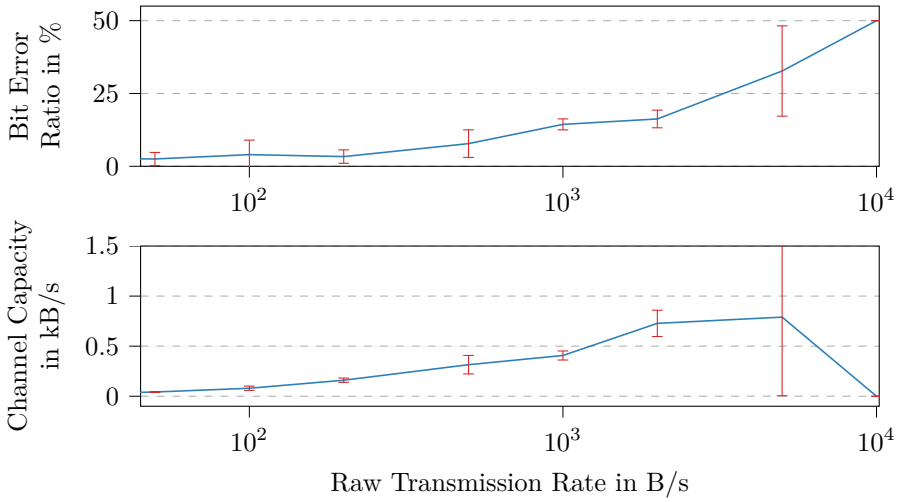


Figure 10.11.: Raw transmission rate vs bit-error ratio and resulting channel capacity across processes on average over all SSDs.

1 000 bit/s with $f = 0.12$ can be used to transmit data on all SSDs and a higher transmission rate can then be negotiated.

With a theoretical negotiation of a higher transmission rate, the next three columns in Table 10.2 "Fastest per SSD" show the highest achievable channel capacity on each SSD with the corresponding raw transmission rate and bit-error ratio. The fastest covert channel runs on SSD $\mathcal{D}$ with 1 784 bit/s with a 5 000 bit/s raw transmission rate and a bit-error ratio of 16.4 %. This relatively high bit-error ratio decreases the raw transmission rate significantly, meaning that approximately $1 - {}^{1784}/_{5000} = 65\,\%$ of the transmitted information must contain redundant error correction data. The slowest covert channel is on SSD $\mathcal{H}$, achieving a channel capacity of 403 bit/s with a raw transmission rate of 2 000 bit/s and the highest bit-error ratio (24.2 %).

There is no real correlation between PCIe version and covert-channel capacity. The average capacity over all PCIe 3.0 SSDs is 1 017 bit/s while the average capacity over all PCIe 4.0 SSDs is 911 bit/s. However, the fastest covert channel was on a PCIe 4.0 SSD, while the slowest was on a PCIe 3.0 SSD. Similarly, the SSD cache also does not appear to be a significant influence factor on the channel capacity: While the fastest

SSD $\mathcal{D}$ (1 784 bit/s) has 1 GB of LPDDR4 cache, the second fastest SSD $\mathcal{C}$ (1 492 bit/s) does not have a cache. Of the two SSDs with the slowest and second slowest covert channel, one has a cache and the other has no cache, indicating that the existence of a cache inside the SSD is not a significant influence factor for the timing leakage we observe, *i.e.*, the SSD cache is not a contention source or mitigation.

4.4. Covert Channel across Virtual Machines

As a second scenario, we mount our covert channel in a cross-VM attack setup. Our covert channel across virtual machines reaches a channel capacity of up to 1 503 bit/s on SSD $\mathcal{E}$.

Threat Model. We assume the sender process is running inside a VM, with access to secret information but without network access. With a receiver process either in another VM or directly on the host that has network access, they can use the covert channel to exfiltrate the secret data. We assume no special privileges of the processes in their respective VM. We run the VMs on a fully updated Linux Kernel-based Virtual Machine (KVM) with QEMU. We assume both processes have access to a clock to synchronize the sending of individual bits. Furthermore, we assume that the VM disk images are on the same SSD. We did not observe large differences between raw disk images and disk images in the `qcow3` format. As it is generally recommended to not use raw disk files, we present the results for `qcow3` disk images. The disk images are accessed without caching on the host, as recommended for example by Red Hat [26].

We evaluate the covert channel across VMs like the covert channel across processes (cf. Section 4.3).

Results. Table 10.2 shows all covert channel results of all SSDs. If the receiver cannot find a threshold or detect the start sequence with at least 70 % correctness, we cannot transmit data on this SSD through our covert channel. When excluding transmission rates and fractions where at least on one SSD no data transmission is possible, a raw transmission rate of 500 bit/s with a transmission fraction f of 0.5 resulted in the highest average channel capacity over all tested SSDs of 180 bit/s and an average bit-error ratio of 16.5 %.

With a theoretical negotiation of a higher transmission rate, the next three columns in Table 10.2 “Fastest per SSD” show the highest achievable

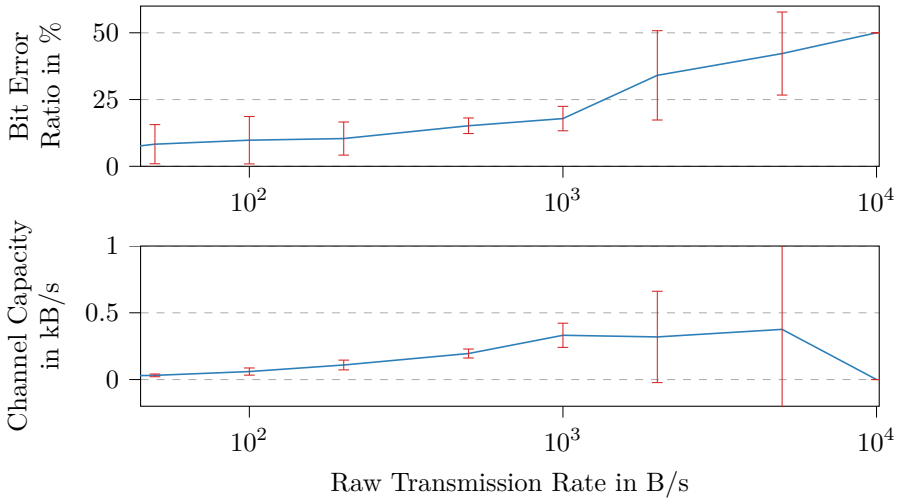


Figure 10.12.: Raw transmission rate vs bit-error ratio and resulting channel capacity across virtual machines on average over all SSDs.

channel capacity on each SSD with the corresponding raw transmission rate and bit-error ratio. The fastest covert channel runs on SSD $\mathcal{E}$ with 1 503 bit/s with a 5 000 bit/s raw transmission rate and a bit-error ratio of 18.9 %. The slowest covert channel is on SSD $\mathcal{D}$, achieving a channel capacity of 124 bit/s with a raw transmission rate of 1 000 bit/s and a bit-error ratio of 29.5 %.

With the only exception being SSDs $\mathcal{E}$ and $\mathcal{H}$, the covert channel between processes is generally faster than between VMs. On SSD $\mathcal{E}$ the error rate is lower between virtual machines, 18.9 % versus 19.4 %, resulting in a slightly higher channel capacity of 1 503 bit/s versus 1 447 bit/s. On SSD $\mathcal{H}$ the error rate is lower between virtual machines, 21.6 % versus 24.2 %, resulting in a slightly higher channel capacity of 493 bit/s versus 403 bit/s. Our hypothesis why this is the case on some SSDs has two reasons. First, while software overhead from virtualization and management of the `qcow2` disk image is higher than accessing a file directly, management of `qcow2` disk images could also induce additional disk accesses we measure. Second, we do not rule out triggering contention inside the software stack of KVM, QEMU or libvirt.

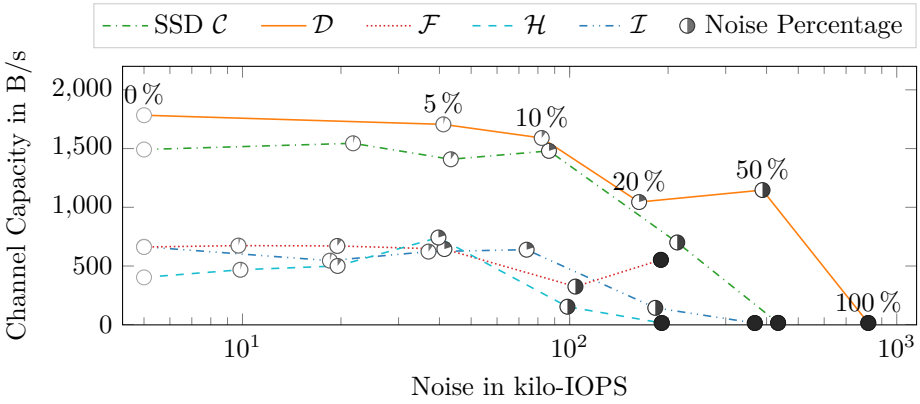


Figure 10.13.: The covert channel capacity on various SSD under different noise scenarios. The noise is always relative to the maximum achievable IOPS on each SSD. The marks mark the percentage of the noise's IOPS in relation to this maximum IOPS. For example, a 50 % noise on SSD $\mathcal{C}$ creates more IOPS than the maximum IOPS of SSDs $\mathcal{F}$ and $\mathcal{J}$ and the covert channel still achieves 700 bit/s on SSD $\mathcal{C}$.

4.5. Impact of Noise

In the previous experiments the operating system was installed on the SSD where the covert channel was performed but apart from occasional background tasks no programs were running that access the SSDs. To investigate the impact of noise on the covert channel, we run the experiments again with additional artificial noise of different strengths.

Implementation. To make the noise comparable, we measure the maximum IOPS of each SSD using up to four threads submitting random reads. Then we cause 5 %, 10 %, 20 %, 50 % and 100 % of the IOPS in a separate process while running the covert channel. For the covert channel across VMs the noise is generated on the host.

Results. Figure 10.13 shows the impact of noise on the covert channel capacity between processes on five SSDs. The results of all SSDs in the process and VM threat model are shown in Table 10.4.

On almost all SSDs 20 % noise has negligible impact on the channel capacity in the process threat model. Between VMs, 20 % noise more than halves the channel capacity on 8 SSDs. Figure 10.13 plots the absolute strength of the noise in kilo-IOPS. The absolute comparison shows that,

e.g., 50 % noise on SSD $\mathcal{C}$ has more IOPS than 100 % on SSDs $\mathcal{F}$ and $\mathcal{H}$ while only slightly impacting the channel capacity.

The noise we inject is relatively constant over the time of a transmission. Because of that, the automatic threshold learning is able to identify and ignore the noise. This makes it even possible to transmit data when the noise already causes 100 % of the maximum IOPS on SSD $\mathcal{F}$ because the additional accesses of the sender just make the SSD "even slower". Strongly fluctuating noise could directly impact the error rate or hinder the threshold learning by injecting false `1`-bits into the transmission, rendering the covert channel unusable.

4.6. Discussion

In Section 3.5, we measured that even only reading 32 blocks is clearly detectable and could, therefore, be used for a covert channel. However, we found that such reads of this size happen constantly on a typical system introducing a lot of fault positive bit detections into the transmission. Increasing the blocks read by the sender, reduces this problem.

We explain the large differences in channel capacities between the SSDs with the different behaviors we saw in Section 3.5. To enable a fast transmission, the receiver (observer) must be able to perform many measurements to always have multiple measurements per time slice, *i.e.*, bit. The sender is always sending bits with large bursts, so the minimal *victim read size* has a lower impact. As shown in Figure 10.6, SSD $\mathcal{D}$ is able to detect reads with very low observer read delays, it also the SSD where the highest transmission rate was achieved. In contrast, SSDs $\mathcal{H}$ and $\mathcal{J}$ require higher read delays and achieve only a lower transmission rate.

We assumed a synchronized clock between sender and receiver. While hypervisor of conventional VMs could modify the TSC value, secure VMs (AMD SEV [3, 12] and Intel TDX [30]) can guarantee unmodified TSC values for their guests. Under the relaxed assumption, that the clocks only have to run with the same rate, the threshold learning and start sequence detection could also learn the offset between the clocks. For this the receiver would perform them on every new received measurement, automatically learning the offset when the threshold is most optimal, *i.e.*, it would perform many intermediate runs in Figure 10.9.

5. Website Fingerprinting

The goal of website fingerprinting is to determine the websites visited by a victim. Leaking visited websites has huge implications on the victim's privacy as it can reveal sensitive information about a user and could be, for example, be used for extortion campaigns. We show that the browser cache access patterns, caused by the web browser when loading a website, are enough to fingerprint the Alexa top 100 websites in an open-world scenario with up to 97.0 % accuracy. In an open-world scenario, there is an additional "other" group containing all websites not explicitly classified, including those that were never seen during training. The goal is to have a classifier that can explicitly classify the top 100 websites, and if it encounters a website not in the top 100, it should classify it as "other". Our open-world set contains websites randomly sampled from the Alexa top 10 000 websites combined into the "other" group. As each trace is from a different website, the websites seen during training in this class do not overlap with the ones encountered during testing.

If a user opens a previously visited website, the browser serves a significant amount (*i.e.*, megabytes) of content of the website from the web cache. The browser reads the website's cache as well as data stored in cookies or `localStorage` when loading the website. The reading of this data can be detected and is unique for every website as shown by the following cache analysis. Analyzing the cache of the Alexa top 100 websites loaded in Chrome reveals that the average cache size of a website is 2.2 MB ($n = 90$, $\sigma = 3.9$ MB) and consists of 148.5 ($n = 90$, $\sigma = 390$) files. The median is 1.1 MB and 71 files. As shown in Section 3.5, this is well in the range of detectable reads, especially because reading many small files still loads entire blocks from the SSD, typically 4 kB and additionally the operating system typically reads ahead multiple blocks. The high deviation in cache size and file count per website contributes to the great performance of our attack.

Threat Model. The attacker has native code execution on a system and the possibility to access a file on the same disk where the web browser stores its data. The data is stored in the home directory of the victim that is usually on the same disk as the operating system (e.g. `~/.cache/chromium` on Linux and `%AppData%\Local\Chromium` on Windows). However, the attacker is isolated, e.g., in a sandbox or another user, and has no means to access this data, any other data of the user, or to interact with the user's processes.

The victim uses Chromium and opens websites they visited before. When the victim opens a website for the first time, it is not yet in the web cache. Hence, the disk activity from writing data to the disk cache cannot be classified by *our* model. However, if the victim navigates through the page, any subsequent click and load will be served from the web cache and, thus, is classifiable.

We run the attacks on unmodified, default-configured Ubuntu and Chromium version 126.0. The detailed system configuration for each SSD is shown in Table 10.3. The background processes and activities that are present in the default configuration are also present in our test and may access the disk. The attacker process and Chromium are not pinned to specific CPU cores.

Implementation. In the data recording phase, we record 300 traces of the web browser opening each website (Section 5.1). A trace consists of the access times of periodical read accesses to a file on the disk. To classify the traces we use a convolutional neural network (CNN). We also perform experiments with artificial noise to see how it impacts the fingerprinting accuracy.

5.1. Data Recording

We measure the access round-trip times while a website is loading 100 times for every website in the top 100. Furthermore, we collect traces of 400 randomly selected websites, one trace each, for the open-world class. The 400 randomly selected websites are **not** in the top 100. We then repeat this measurement 3 times to have 300 website traces each and 1200 open-world traces. Performing 3×100 measurements reduces the risk that we measure and classify environmental influences like the time of the day. Recording all traces takes approximately three days.

Browser Instrumentalization. We use an unmodified default Chromium installation and script the website navigation using `xdotool`. First, we open the target website once to load it into the cache. Second, we record the round-trip traces by opening the website once per trace recording.

RTT Trace Measurement. We again time disk accesses by reading from a file on the SSD. One thread periodically sends read requests and a second thread receives the responses and computes the duration. We use

the `clock_gettime()` timer with ns accuracy. We record approximately 40 000 round-trip times for the first 3 s of a website loading.

Page Cache Eviction. Because the operating system uses all available memory to cache recently used disk data, web-cache files do not cause disk accesses after the first load. To circumvent this, the attacker has to evict the page cache periodically. During page cache eviction, no traces can be recorded, leaving blind spots. Therefore, it is crucial to make this page cache eviction as fast as possible while at the same time keeping memory footprint and pressure low. The amount of available memory is known to unprivileged code.

We implement the eviction first shown by Gruss et al. [20]. It uses three sets of data. The first two sets are a file on the SSD mapped into the memory. The first set, 95 % of the page cache large, is constantly kept in the page cache by periodically reading all pages. The second set has the size of the remaining 5 % and is used for actual page cache eviction. For the eviction, both sets are read, filling the whole page cache with their data and evicting everything else. The third set limits the size of the page cache by allocating memory. We use this set to limit the page cache size to 4 GB, this is a good compromise between free memory and page cache eviction speed.

We are able to evict the page cache in 347 ms ($n = 10000$, $\sigma = 75.5$) on average. We never observed an out-of-memory problem during our experiments because 4 GB of memory are always kept available for other processes.

5.2. Classification Machine Learning Model

With all round-trip time traces recorded we use a convolutional neural network (CNN) to build a classifier.

Our CNN has a very typical structure with nine convolutional layers with max pooling and dropout layers in-between. The output is then flattened and classified with three subsequent dense layers, again with dropout layers in-between.

We train the CNN on spectrograms of our recorded traces. The spectrograms are computed using a Short-Time Fourier Transform (STFT) with a window size of 256. This is a well established technique for signal classification [10, 29, 58, 87]. Each website has a unique spectrogram, e.g.,

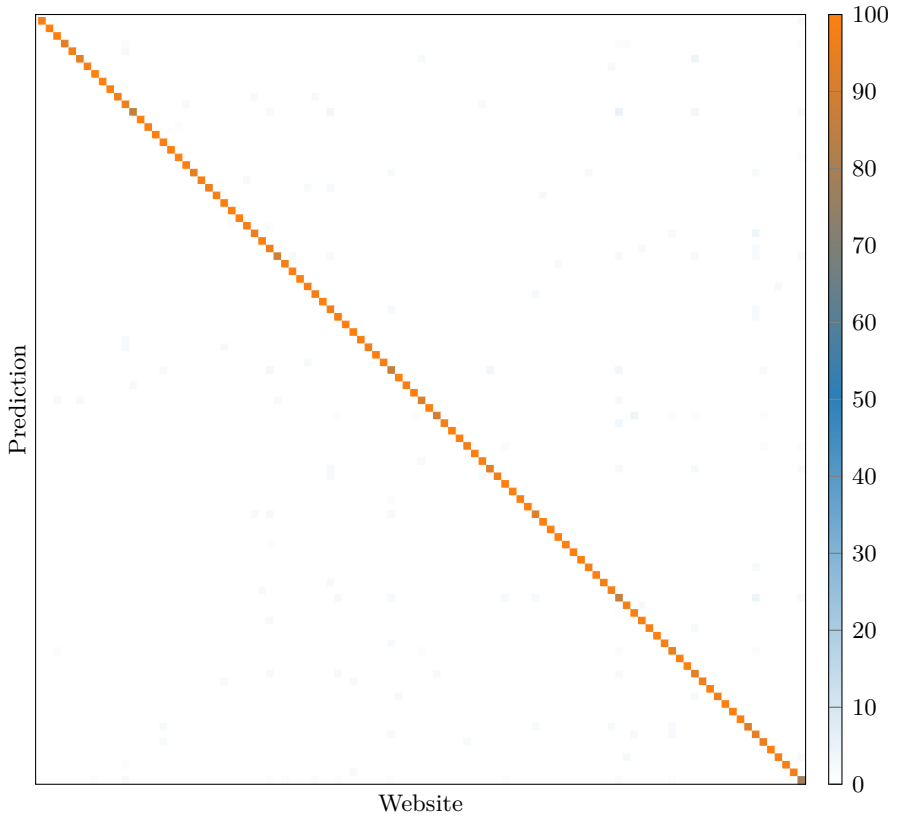


Figure 10.14.: Classification heat map of the website fingerprinting on SSD $\mathcal{K}$. The F_1 score is 97.0 %, the open-world dataset is correctly classified with an accuracy of 78 %.

see Figure 10.16. Spectrograms from the same websites have key features that are the same for each trace and only vary slightly, most likely due to timing variations and other noise.

To be able to compare different SSDs, with and without caches, with PCIe 3.0 or PCIe 4.0, we train a classifier for each SSD. We randomly split our traces into 64 % for training, 16 % for the validation set and the remaining 20 % for the final test set. Additionally, we try to classify the traces of an SSD unknown while training. For this we do not include any traces of one SSD in the training or validation set and then only use traces from this SSD for the final test. For training and validation we use all traces from all other 11 SSDs.

5.3. Results

The results for all SSDs are shown in Table 10.1. Figure 10.14 shows the classification heat map of the SSD $\mathcal{K}$ with the best results of 97.0 %, the open-world dataset is correctly classified with 78 % accuracy. The worst result is on SSD $\mathcal{H}$ with an F_1 score of 80.2 %, however the open-world dataset is correctly classified with 96 % accuracy.

Over all SSDs the geometric mean F_1 score is 92.9 %. On PCIe 3.0 SSDs we get a geometric mean F_1 score of 92.8 %. On PCIe 4.0 SSDs it is 93.1 %. Similar to the covert channel results, we do not really see a difference in the results between PCIe 3.0 and 4.0 SSDs.

When fingerprinting traces from an SSD *unknown* during training we never achieve a result above 5 % which is just slightly better than random guessing. This was independent of the tested SSD. We suspect that the very different SSD behavior that we demonstrated in Section 3.5 is the reason for it not working properly. However, this does not mean that better data preprocessing or a more refined machine learning model would not be able to improve classification in this scenario. Future work could focus on the classification and machine learning aspect in more detail.

5.4. Impact of Noise / Mitigation

To investigate the impact of noise on website fingerprinting, we run the experiments again on a few selected SSDs with additional artificial noise of different strengths like we did for the covert channels (Section 4.5). We also injected random burst, to analyze if artificial noise is a viable approach to mitigate the website fingerprinting attack. The burst are relatively small, from a few hundred kB to a few MB, around the same size at the different website's caches.

Results. Figure 10.15 shows the F_1 scores at different injected noise levels on SSDs $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{H}$. On SSDs $\mathcal{A}$ and $\mathcal{C}$ the noise has a comparable impact. While it decreases the classification accuracy, it is still 58.1 % on SSD $\mathcal{C}$ and 37.0 % on SSD $\mathcal{A}$ at a 50 % noise level, significantly higher than the 1 % chance when randomly guessing. SSD $\mathcal{H}$ showed the worst classification accuracy without noise and was also more strongly influenced by noise than the others. Already adding only a 2 % noise level decreased the accuracy by over 50 % to 27.6 %. This strong influence could also be the reason for the worst result without artificial noise, because other

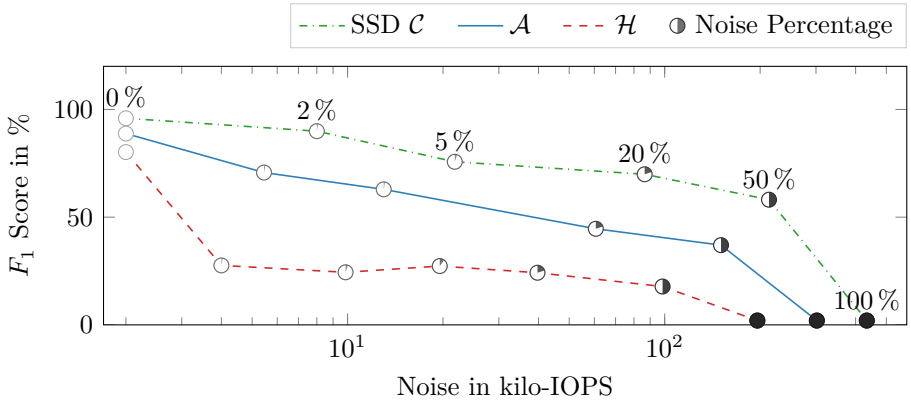


Figure 10.15.: The website fingerprinting F_1 score on various SSD under different noise scenarios. The noise is always relative to the maximum achievable IOPS on each SSD. The marks mark the percentage of the noise's IOPS in relation to this maximum IOPS.

system activities were always running during our experiments. The F_1 score than further decreases to 17.8 % at a 50 % noise level. Figure 10.17 shows spectrograms with noise. Figure 10.18 and 10.19 show heat maps at different noise levels on SSDs $\mathcal{C}$ and $\mathcal{H}$.

At a 100 % noise level, classification was not possible on any of the three SSDs. The machine learning model was not able to learn anything from the data during the training phase.

When injecting random noise with similar size as the website's caches, classification was also not possible with our model. The model was again, not able to learn from the data. We do however, not rule out, that a more advanced model or a huge amount of data could make this possible. For now, web browsers could perform additional random reads when accessing the web cache to mitigate this attack.

6. Discussion and Mitigations

We exploited the unprivileged low-overhead `io_uring` interface. Removing unprivileged access to low-overhead interfaces could hinder attacks, at a considerable performance cost, but not mitigate them. The coarse granularity of the channel limits the potential attack targets and makes the attack more susceptible to noise. Similar as it has been suggested for the interrupt keystroke side-channel attacks [63], it could be a viable approach to add noise. By letting the kernel or the web browser perform additional dummy operations, more noise could be induced to make website fingerprinting impractical on all SSDs, without significant performance costs attached.

However, even with noise or an interface with higher overheads and latencies, the covert channel cannot be closed. This problem is not easy to overcome and hardware vendors typically take the perspective that closing covert channels cannot be an acceptable goal [31]. It is, however, important to understand these channels and their properties to learn when they can be extended to side channels and leak sensitive information, e.g., sensitive browsing activity. Our work shows that the SSD timing channel is a relevant threat and further analysis is required understand whether more severe information leakage is possible as well.

Private browsing does not mitigate the website fingerprinting attack. While all browser clear the cache created during private browsing after finishing the session [1], for the time of the session, the cache is used. Therefore, after the first visit to a website it is loaded from the cache and can be fingerprinted.

We used `O_DIRECT` for direct disk access bypassing the operating system's caches. Making this flag privileged does not mitigate our attacks as accessing a file larger than the kernel caches leads to constant cache eviction and thus also to direct disk accesses. Additionally, `O_DIRECT` is used by benign applications like databases for performance reasons.

In real cloud systems, SLAs (Service Level Agreements) guarantee a certain disk bandwidth for every customer. To enforce this, the hypervisor has to throttle I/O. We did not evaluate hypervisor caused disk throttling in this work.

7. Conclusion

In this work, we showed that access contention to an SSD causes measurable differences in the round-trip time of individual requests. Different SSDs behave very differently in contention scenarios. Still, we showed that covert channel transmissions with up to 1 784 bit/s across processes and 1 503 bit/s across virtual machines are possible. The covert channel is also resilient to noise. Furthermore, we show that SSD contention can also be used to invade the privacy of internet users. In a website fingerprinting attack, we identified loading website with up to 97.0 % accuracy in a top-100 open-world scenario. While contention-based timing side channels are always difficult to mitigate, better scheduling of queues inside the SSD controller or artificial noise from the operating system or web browser could decrease the channel capacity of covert channels and mitigate fingerprinting attacks.

Acknowledgements

We thank our anonymous reviewers for their valuable feedback on this work. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85 and FWF project NeRAM I6054). Additional funding was provided by generous gifts from Red Hat, Google and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. An Analysis of Private Browsing Modes in Modern Browsers. In: USENIX Security. 2010 (p. 343).

[2] 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. 308).

[3] AMD. AMD64 Architecture Programmer's Manual. 2023 (p. 336).

[4] Johann Betz, Dirk Westhoff, and Günter Müller. Survey on covert channels in virtual machines and cloud computing. In: Transactions on Emerging Telecommunications Technologies (2016) (p. 311).

[5] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning. In: PoPETS 4 (2019), pp. 292–310 (p. 308).

[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: CCS. 2019 (p. 308).

[7] Jalil Boukhobza and Pierre Olivier. Flash Memory Integration: Performance and Energy Issues. Elsevier, 2017 (p. 313).

[8] Serdar Cabuk, Carla E Brodley, and Clay Shields. IP Covert Timing Channels: Design and Detection. In: CCS. 2004 (p. 311).

[9] Ang Chen, W Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting Covert Timing Channels with Time-Deterministic Replay. In: OSDI. 2014 (p. 311).

[10] Zhibo Chen, Yi-Qun Xu, Hongbin Wang, and Daoxing Guo. Deep STFT-CNN for spectrum sensing in cognitive radio. In: IEEE Communications Letters (2020) (p. 339).

[11] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's always a bigger fish: a clarifying analysis of a machine-learning-assisted side-channel attack. In: ISCA. 2022 (pp. 309, 310, 312).

[12] Nikunj A Dadhania. [PATCH v7 00/16] Add Secure TSC support for SNP guests. 2023. URL: `https://lore.kernel.org/all/20231220151358.2147066-1-nikunj@amd.com/` (p. 336).

[13] Ran Dubin, Amit Dvir, Ofir Pele, and Ofer Hadar. I Know What You Saw Last Minute—Encrypted HTTP Adaptive Video Streaming Title Classification. In: IEEE Transactions on Information Forensics and Security 12.12 (2017), pp. 3039–3049 (p. 308).

[14] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In: ISCA. 2021 (p. 311).

[15] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael B. Abu-Ghazaleh, Andres Marquez, and Kevin J. Barker. Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems. In: ISCA. 2022 (p. 311).

[16] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS. 2016 (p. 308).

[17] 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 (p. 308).

[18] Ilias Giechaskiel, Ken Eguro, and Kasper B Rasmussen. Leakier Wires: Exploiting FPGA Long Wires for Covert-and Side-channel Attacks. In: ACM Transactions on Reconfigurable Technology and Systems (TRETS) 12.3 (2019), p. 11 (p. 311).

[19] Ilias Giechaskiel, Shanquan Tian, and Jakub Szefer. Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. In: ACM Transactions on Reconfigurable Technology and Systems (2022) (pp. 311, 312).

[20] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 311, 339).

[21] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to violate web privacy with hardware performance events. In: ESORICS. 2017 (pp. 309, 312).

[22] Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations. In: IEEE CSF. 2015 (p. 311).

[23] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Acoustic data exfiltration from speakerless air-gapped computers via covert hard-drive noise ('DiskFiltration'). In: ESORICS. 2017 (p. 311).

[24] Jawad Haj-Yahya, Lois Orosa, Jeremie S Kim, Juan Gómez Luna, A Giray Yağlikçi, Mohammed Alser, Ivan Puddu, and Onur Mutlu. IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors. In: ISCA. 2021 (p. 311).

[25] Youngkwang Han and John Kim. A Novel Covert Channel Attack Using Memory Encryption Engine Cache. In: DAC. 2019 (p. 311).

[26] Jiri Herrmann, Yehuda Zimmerman, Dayle Parker, and Scott Radvan. Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide. 2019 (p. 333).

[27] Wei-Ming Hu. Lattice Scheduling and Covert Channels. In: S&P. 1992 (p. 311).

[28] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In: Journal of Computer Security (1992) (p. 311).

[29] Jingshan Huang, Binqiang Chen, Bin Yao, and Wangpeng He. ECG arrhythmia classification using STFT-based spectrogram and convolutional neural network. In: IEEE access (2019) (p. 339).

[30] Intel. Intel Trust Domain Extensions Module Base Architecture Specification. 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html (p. 336).

[31] Intel Corporation. Configuring Workloads for Microarchitectural and Side Channel Security. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/securing-workloads-against-side-channel-methods.html (p. 343).

[32] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In: S&P. 2012 (pp. 309, 312).

[33] Qisheng Jiang and Chundong Wang. Sync+Sync: A Covert Channel Built on fsync with Storage. In: USENIX Security. 2024 (pp. 308, 311).

[34] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R Karpuzcu. POWERT channels: A novel class of covert communicationexploiting power management vulnerabilities. In: HPCA. 2019 (p. 311).

[35] Beomjun Kim and Myungsuk Kim. LazyRS: Improving the Performance and Reliability of High-Capacity TLC/QLC Flash-Based Storage Systems Using Lazy Reprogramming. In: Electronics (2023) (p. 325).

[36] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 308).

[37] 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 (p. 308).

[38] Butler W Lampson. A note on the confinement problem. In: Communications of the ACM 16.10 (1973), pp. 613–615 (pp. 308, 311).

[39] Bartosz Lipinski, Wojciech Mazurczyk, and Krzysztof Szczypiorski. Improving Hard Disk Contention-based Covert Channel in Cloud Computing Environment. In: S&P Workshops. 2014 (p. 311).

[40] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (p. 326).

[41] 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 (p. 308).

[42] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In: CCS. 2022 (p. 308).

[43] 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. 311).

[44] Nikolay Matyunin, Jakub Szefer, Sebastian Biedermann, and Stefan Katzenbeisser. Covert channels using mobile device's magnetic field sensors. In: ASP-DAC. 2016 (p. 311).

[45] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In: ACM Workshop on Privacy in the Electronic Society. 2019, pp. 135–149 (p. 309).

[46] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015 (p. 311).

[47] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 311).

[48] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 311, 312).

[49] Steven J Murdoch and Stephen Lewis. Embedding Covert Channels into TCP/IP. In: Workshop of Information Hiding. 2005 (p. 311).

[50] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael B. Abu-Ghazaleh. Constructing and characterizing covert channels on GPGPUs. In: MICRO. 2017 (p. 311).

[51] Keisuke Okamura and Yoshihiro Oyama. Load-Based Covert Channels between Xen Virtual Machines. In: Symposium on Applied Computing (SAC). 2010 (p. 311).

[52] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015 (p. 326).

[53] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 308).

[54] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (p. 308).

[55] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. In: arXiv:1908.03383 (2019) (p. 311).

[56] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In: TrustCom/BigDataSE. 2018 (pp. 309, 312).

[57] P. Ranjith, Chandran Priya, and Kaleeswaran Shalini. On covert channels between virtual machines. In: Journal in Computer Virology 8.3 (June 2012), pp. 85–97 (p. 311).

[58] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (pp. 308–310, 312, 339).

[59] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In: NDSS. 2017 (p. 308).

[60] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS. 2021 (p. 311).

[61] Mickaël Salaün. Practical overview of a Xen covert channel. In: Journal in Computer Virology 6.4 (Aug. 2010), pp. 317–328 (p. 311).

[62] Samsung. 980 PRO NVMe M.2 SSD Specification. Sept. 2020 (p. 325).

[63] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (p. 343).

[64] Meng Shen, Jinpeng Zhang, Liehuang Zhu, Ke Xu, and Xiaojiang Du. Accurate decentralized application identification via encrypted traffic analysis using graph neural networks. In: TIFS 16 (2021), pp. 2367–2380 (p. 308).

[65] Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. A side-channel analysis of sensor multiplexing for covert channels and application fingerprinting on mobile devices. In: arXiv:2110.06363 (2021) (p. 309).

[66] Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. A Side-Channel Analysis of Sensor Multiplexing for Covert Channels and Application Profiling on Mobile Devices. In: Transactions on Dependable and Secure Computing (2023) (p. 311).

[67] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through The Cache Occupancy Channel. In: USENIX Security. 2019 (pp. 308, 309, 312).

[68] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. Exploiting data-usage statistics for website fingerprinting attacks on Android. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks. 2016 (pp. 308, 312).

[69] StorageNewsletter. Overall SSD Shipments Decreased 6% Q/Q to 82.8 Million in 1Q24. June 2024. URL: https://www.storagenewsletter.com/2024/06/20/overall-ssd-shipments-decreased-6-q-q-to-82-8-million-in-1q24/ (p. 315).

[70] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In: NDSS. 2018 (p. 311).

[71] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. In: Journal of Hardware and Systems Security 3.3 (2019), pp. 219–234 (p. 311).

[72] Billy Tallis. 2021 NAND Flash Updates from ISSCC: The Leaning Towers of TLC and QLC. Feb. 2021. URL: https://www.anandtech.com/show/16491/flash-memory-at-isscc-2021 (p. 324).

[73] Billy Tallis. The ADATA GAMMIX S50 Lite 2TB SSD Review: Mainstream PCIe Gen4. Apr. 2021. URL: https://www.anandtech.com/show/16635/the-adata-gammix-s50-lite-ssd-review-mainstream-pcie-gen4/5 (p. 325).

[74] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible probe: Timing attacks with pcie congestion side-channel. In: S&P. 2021 (pp. 308, 311).

[75] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust smartphone app identification via encrypted network traffic analysis. In: TIFS 13.1 (2017), pp. 63–78 (p. 308).

[76] Jing Tian, Gang Xiong, Zhen Li, and Gaopeng Gou. A survey of key technologies for constructing network covert channel. In: Security and Communication Networks 2020 (2020), pp. 1–20 (p. 311).

[77] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. Covert-channels in FPGA-enabled SmartSSDs. In: ACM Transactions on Reconfigurable Technology and Systems (2023) (p. 312).

[78] Theodoros Trochatos, Anthony Etim, and Jakub Szefer. Security Evaluation of Thermal Covert-channels on SmartSSDs. In: arXiv:2305.09115 (2023) (p. 312).

[79] 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 (p. 308).

[80] 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. 308).

[81] Zhenghong Wang and Ruby B Lee. Covert and Side Channels due to Processor Architecture. In: ACSAC. 2006 (p. 311).

[82] John C Wray. An Analysis of Covert Timing Channels. In: Journal of Computer Security (1992) (p. 311).

[83] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: ACM Transactions on Networking (2014) (pp. 308, 311).

[84] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In: USENIX Security. 2012 (pp. 308, 311).

[85] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. A covert channel construction in a virtualized environment. In: CCS. 2012 (p. 311).

[86] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In: CCSW. 2011 (p. 311).

[87] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, et al. Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks. In: The World Wide Web Conference. 2019 (p. 339).

[88] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 308, 311).

[89] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In: USENIX Security. 2023 (pp. 308–310, 312).

8. Appendix

Table 10.3 shows which SSD was tested with which SSD, Ubuntu version and Linux kernel version.

Table 10.4 shows the detailed results of the covert channel noise experiments. It contains the maximum covert channel transmission rates of all SSDs, between processes and virtual machines at artificial noise levels from 0 % to 100 %.

Figure 10.16 shows the spectrograms of 8 selected websites from the website fingerprinting traces on SSD $\mathcal{A}$. Figure 10.17 shows the spectrograms of the same 8 websites again with an injected 50 % noise level.

Figure 10.18 shows classification heat maps of SSD $\mathcal{C}$ at noise levels 2 %, 20 % and 50 %. Figure 10.19 shows classification heat maps of SSD $\mathcal{H}$ at noise levels 2 %, 20 % and 50 %.

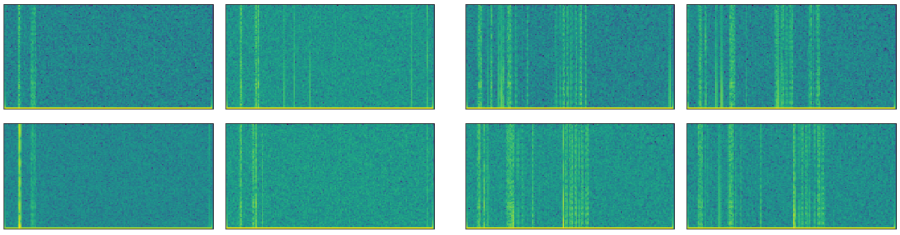
Table 10.3.: The CPU, operating system and Linux kernel version each SSD was tested on.

SSD	Model	CPU	OS	Kernel Version
$\mathcal{A}$	Samsung 980 Pro	Intel Core i7-1260P	Ubuntu 23.04	Linux 6.2.0-39-generic
$\mathcal{B}$	Samsung 980	AMD Ryzen 7 5800X	Ubuntu 23.04	Linux 6.2.0-39-generic
$\mathcal{C}$	Samsung 970 Evo	Intel Core i7-1260P	Ubuntu 23.04	Linux 6.2.0-39-generic
$\mathcal{D}$	Samsung MZVL21T0HCLR-00BL7	Intel Core i7-1260P	Ubuntu 22.04.4 LTS	6.5.0-45-generic
$\mathcal{E}$	Samsung 970 Evo Plus	Intel Core i9-13900KF	Ubuntu 22.04.4 LTS	6.5.0-44-generic
$\mathcal{F}$	Crucial P5	AMD Ryzen 7 5800X	Ubuntu 22.04.6 LTS	Linux 5.15.2-generic
$\mathcal{G}$	Crucial P5 Plus	Intel Core i7-1260P	Ubuntu 23.04	Linux 6.2.0-39-generic
$\mathcal{H}$	Kingston SA2000M81000G	Intel Core i7-5820K	Ubuntu 22.04.4 LTS	Linux 6.8.0-32-generic
$\mathcal{I}$	Toshiba KXG6AZNV1T02	Intel Core i7-1165G7	Ubuntu 22.04.4 LTS	5.15.0-58-generic
$\mathcal{J}$	ADATA XPG Gammix S50 Lite	AMD Ryzen 7 5800X	Ubuntu 23.04	Linux 6.2.0-39-generic
$\mathcal{K}$	Gigabyte Aorus Gen4	AMD Ryzen 7 5800X	Ubuntu 23.04	Linux 6.2.0-39-generic
$\mathcal{L}$	Western Digital Blue SN550	Intel Core i7-1260P	Ubuntu 23.04	Linux 6.2.0-39-generic

Table 10.4.: Detailed covert channel noise results.

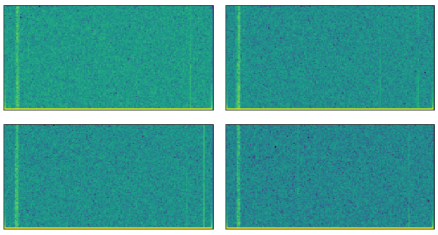
SSD	Process						Virtual Machine					
	Noise Level						Noise Level					
	0 %	5 %	10 %	20 %	50 %	100 %	0 %	5 %	10 %	20 %	50 %	100 %
$\mathcal{A}$	808 bit/s	626 bit/s	706 bit/s	674 bit/s	243 bit/s	4 bit/s	258 bit/s	247 bit/s	233 bit/s	214 bit/s	198 bit/s	264 bit/s
$\mathcal{B}$	787 bit/s	828 bit/s	826 bit/s	806 bit/s	652 bit/s	5 bit/s	615 bit/s	328 bit/s	337 bit/s	279 bit/s	15 bit/s	6 bit/s
$\mathcal{C}$	1 492 bit/s	1 545 bit/s	1 409 bit/s	1 480 bit/s	701 bit/s	16 bit/s	303 bit/s	387 bit/s	333 bit/s	319 bit/s	33 bit/s	6 bit/s
$\mathcal{D}$	1 784 bit/s	1 706 bit/s	1 590 bit/s	1 045 bit/s	1 147 bit/s	15 bit/s	124 bit/s	52 bit/s	69 bit/s	66 bit/s	48 bit/s	14 bit/s
$\mathcal{E}$	1 447 bit/s	1 420 bit/s	853 bit/s	818 bit/s	643 bit/s	16 bit/s	1 503 bit/s	1 359 bit/s	791 bit/s	1 125 bit/s	42 bit/s	15 bit/s
$\mathcal{F}$	660 bit/s	674 bit/s	671 bit/s	644 bit/s	325 bit/s	292 bit/s	404 bit/s	357 bit/s	324 bit/s	339 bit/s	170 bit/s	68 bit/s
$\mathcal{G}$	814 bit/s	755 bit/s	615 bit/s	564 bit/s	44 bit/s	23 bit/s	605 bit/s	379 bit/s	434 bit/s	176 bit/s	17 bit/s	2 bit/s
$\mathcal{H}$	403 bit/s	468 bit/s	500 bit/s	743 bit/s	154 bit/s	2 bit/s	493 bit/s	644 bit/s	348 bit/s	265 bit/s	10 bit/s	3 bit/s
$\mathcal{I}$	664 bit/s	546 bit/s	623 bit/s	639 bit/s	144 bit/s	15 bit/s	258 bit/s	117 bit/s	36 bit/s	30 bit/s	15 bit/s	13 bit/s
$\mathcal{J}$	647 bit/s	735 bit/s	691 bit/s	432 bit/s	408 bit/s	7 bit/s	592 bit/s	705 bit/s	662 bit/s	318 bit/s	2 bit/s	2 bit/s
$\mathcal{K}$	755 bit/s	555 bit/s	696 bit/s	372 bit/s	331 bit/s	6 bit/s	702 bit/s	448 bit/s	566 bit/s	351 bit/s	2 bit/s	2 bit/s
$\mathcal{L}$	1 313 bit/s	1 202 bit/s	1 253 bit/s	898 bit/s	16 bit/s	13 bit/s	429 bit/s	252 bit/s	243 bit/s	228 bit/s	43 bit/s	13 bit/s

The results of the covert channels between processes and virtual machines of all SSDs at different injected artificial noise levels.

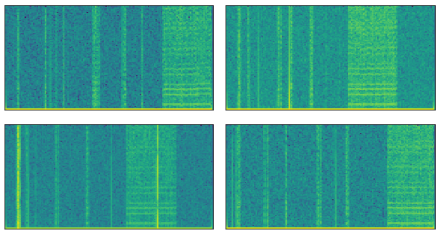


(a) google.com

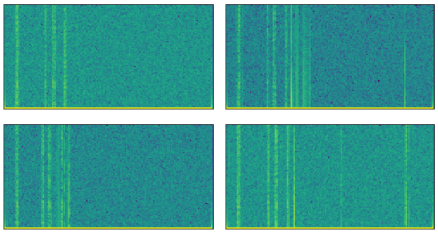
(b) youtube.com



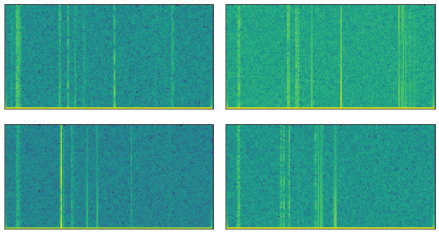
(c) facebook.com



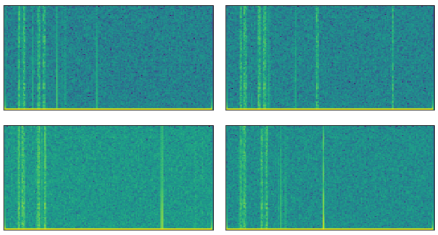
(d) qq.com



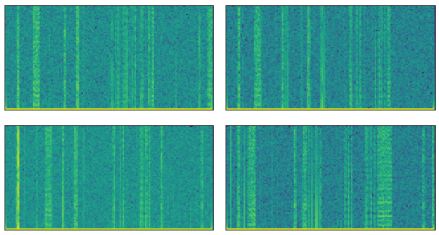
(e) xhamster.com



(f) imdb.com

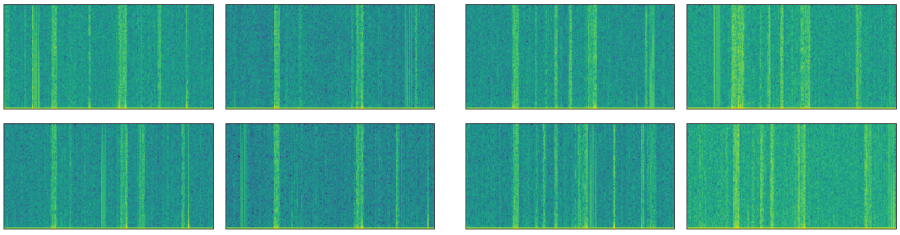


(g) force.com



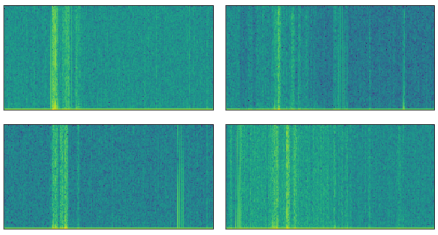
(h) dropbox.com

Figure 10.16.: Spectrograms of different websites on SSD $\mathcal{A}$ with the time on the x-axis and frequency on the y-axis. The differences between the websites as well as the similarities of the same websites are clearly visible, easily distinguishable by a convolutional neural network.

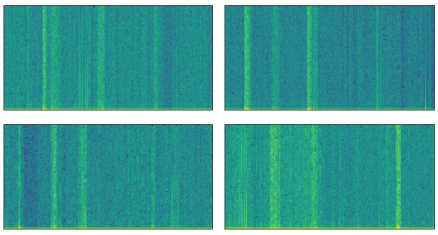


(a) google.com

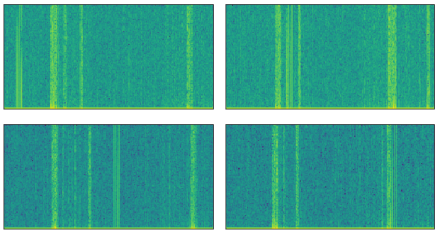
(b) youtube.com



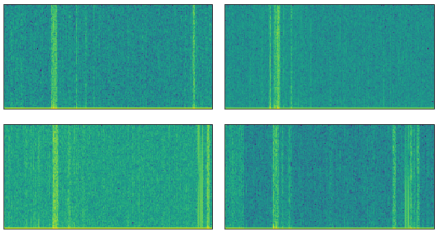
(c) facebook.com



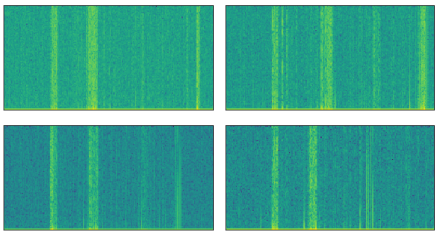
(d) qq.com



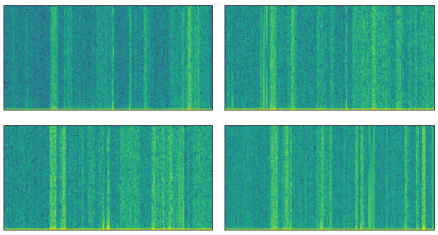
(e) xhamster.com



(f) imdb.com

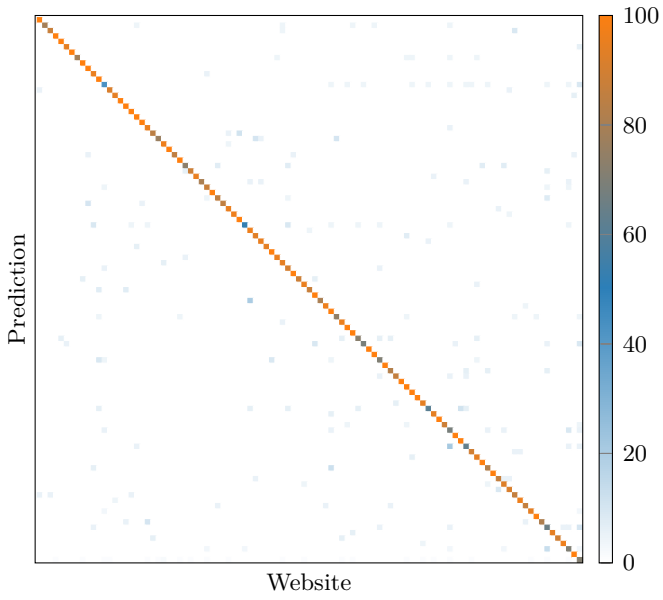


(g) force.com

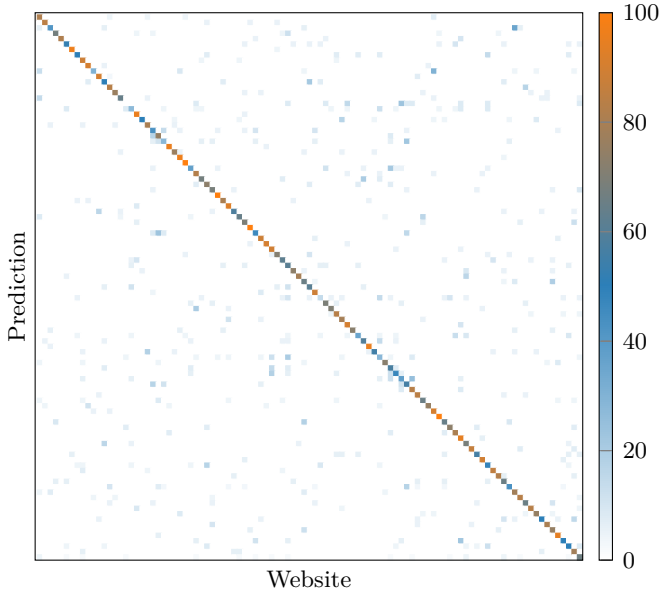


(h) dropbox.com

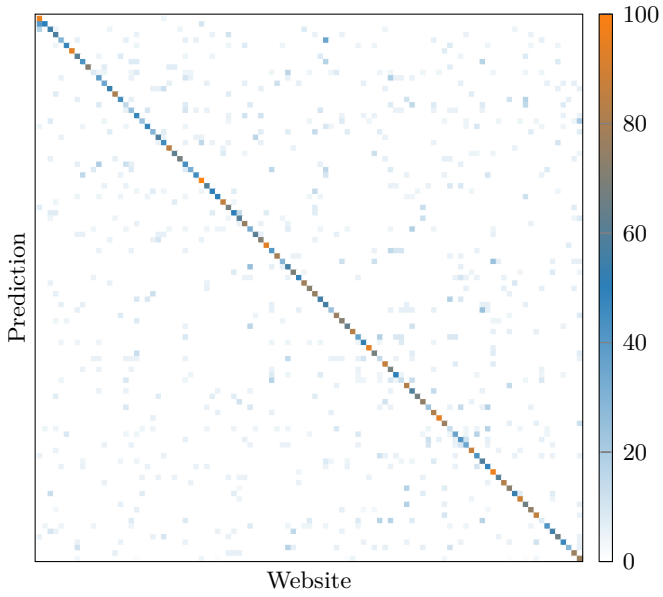
Figure 10.17.: Spectrograms of different websites on SSD $\mathcal{A}$ at an injected 50 % noise level with the time on the x-axis and frequency on the y-axis. The differences between the websites as well as the similarities of the same websites are less clearly visible. The individual visible reads from the browser are longer, probably due to the longer accesses time because of contention with the artificial noise accesses. Because the noise is so constant it is not visible in the spectrograms, making classification possible.



(a) 2 % noise: The F_1 score is 89.9 %, the open-world dataset is correctly classified with an accuracy of 71 %.

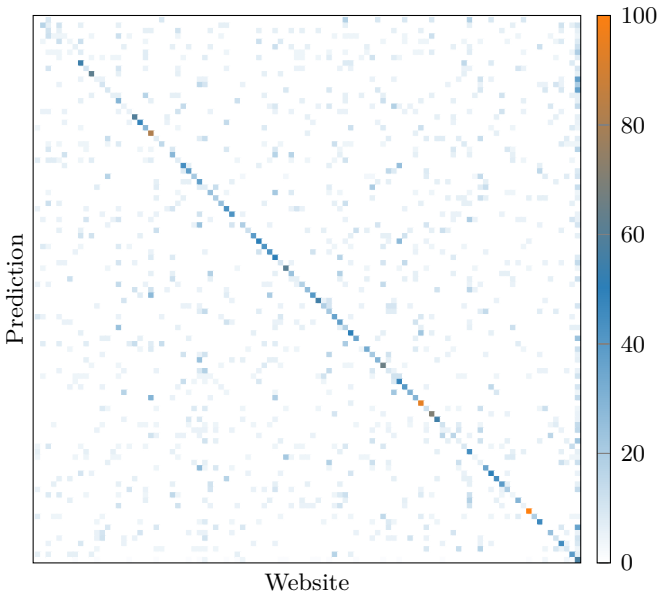


(b) 20 % noise: The accuracy is 69.9 %, the open-world dataset is correctly classified with an accuracy of 66 %.

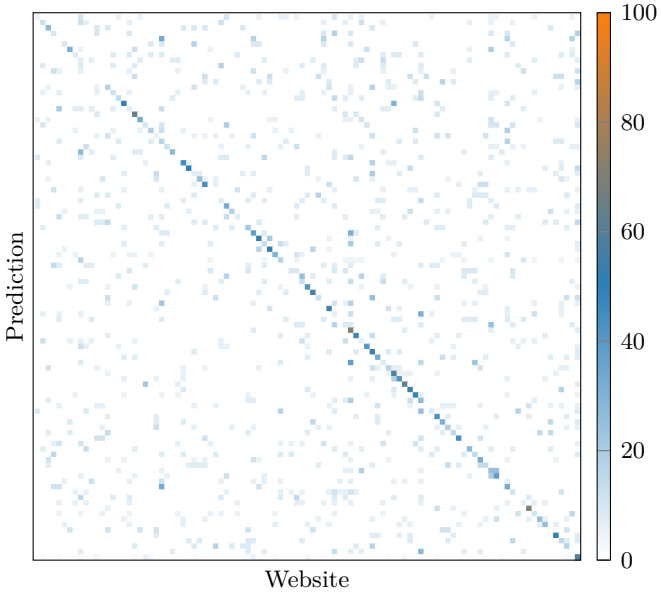


(c) 50 % noise: The accuracy is 58.1 %, the open-world dataset is correctly classified with an accuracy of 77 %.

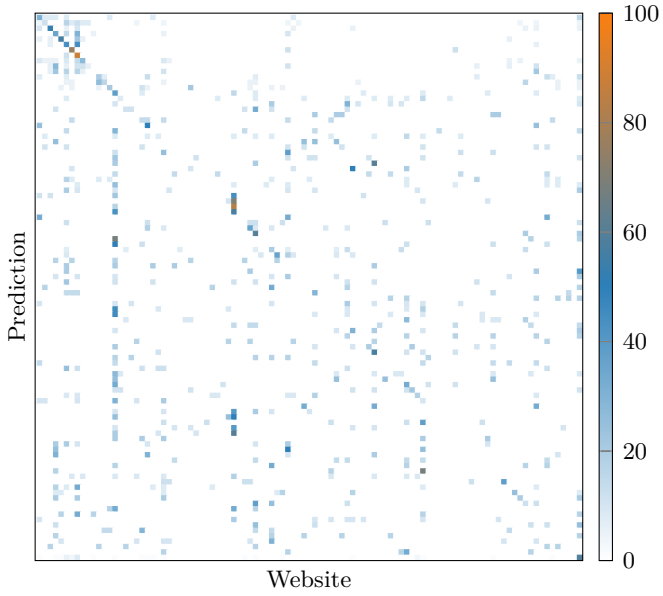
Figure 10.18.: Website fingerprinting classification heat maps of SSD $\mathcal{C}$ at different noise levels.



(a) 2 % noise: The F_1 score is 27.6 %, the open-world dataset is correctly classified with an accuracy of 57 %.



(b) 20 % noise: The accuracy is 24.2 %, the open-world dataset is correctly classified with an accuracy of 59 %.



(c) 50 % noise: The accuracy is 17.8 %, the open-world dataset is correctly classified with an accuracy of 59 %.

Figure 10.19.: Website fingerprinting classification heat maps of SSD $\mathcal{H}$ at different noise levels.

11

Not So Secure TSC

Publication Data

Jonas Juffinger, Sudheendra Raghav Neela, and Daniel Gruss
Not So Secure TSC
In: ACNS 2024

Contributions

Main Author.

Not So Secure TSC

Jonas Juffinger , Sudheendra Raghav Neela , Daniel Gruss

Graz University of Technology, Austria

Abstract

Modern cloud environments greatly facilitate efficiency as workloads of multiple tenants running on the same physical system, often grouped by functionality, e.g., function-as-a-service containers, or platform-as-a-service virtual machines. However, the sharing of physical hardware resources across mutually distrusting tenants comes with security implications. Consequently, cloud providers also offer higher security levels with trusted execution environments in the form of confidential virtual machines, e.g., Intel TDX and AMD SEV-SNP. Many works have shown that attackers can exploit microarchitectural side channels in cloud scenarios, including attacks on trusted-execution environments. However, the practical relevance of these attacks hinges on the co-location of the attacker with its victim on the same physical machine within the data center. Prior work has presented co-location detection techniques that work inside containers or on regular virtual machines. However, co-location detection techniques for confidential virtual machines, e.g., AMD SEV-SNP, have not been studied yet, as these have to be performed from within AMD SEV-SNP virtual machines to target the same physical host machine.

In this paper, we present the first co-location detection technique working on confidential virtual machine hosts. We exploit the new SecureTSC feature supported on recent AMD processors with SEV-SNP. SecureTSC is intended to provide a trusted timing source to confidential virtual machines that cannot be tampered with by the host. We systematically study the behavior of SecureTSC and show that it can be exploited to detect whether two confidential virtual machines are co-located. We demonstrate our co-location detection in a concrete scenario, where two confidential virtual machines attempt to co-locate with each other. Our attack uses a minimal network protocol, *TsCupid*, to determine whether any of the connected confidential virtual machines are co-located. We practically evaluate our attacks and show that we can detect co-location within 0.13 seconds in a fully parallelized way, minimizing the cost for

an attack. Finally, we show that our attack cannot be mitigated without modifying the SecureTSC feature and propose a concrete design change that would fully prevent our attack. Keywords: Side Channels, Timing Attack, Co-Location Attack, AMD SEV, Confidential Computing.

1. Introduction

The efficiency and economic advantages of cloud computing and virtualization are rooted in the sharing of physical resources. Today, various types of cloud computing exist, often categorized into function-as-a-service, software-as-a-service, platform-as-a-service, and infrastructure-as-a-service. The idea for all types is similar: Place workloads of the same type but from different tenants on the same physical machine and share as many physical resources as possible while maintaining confidentiality, integrity, and availability, of the customer's data and workloads. As a consequence, a long line of work studied the security properties of workloads in cloud environments, in particular in terms of side-channel attacks [10, 44, 60, 69, 73, 74] as well as their mitigation [33, 42, 72]. These attacks typically monitor the usage of a shared hardware resource and exfiltrate secret information based on the observed behavior, e.g., timing [30], or access patterns [44].

Beyond distrusting other co-located tenants, customers of cloud computing may also distrust the cloud provider [59] or have legal restrictions on how and where their data may be processed [50]. Consequently, the concept of trusted execution environments was developed [8, 18] and implemented in billions of processors, e.g., Arm TrustZone [8], Intel SGX [15], AMD SEV [32], and Intel TDX [28]. Today, commercial cloud providers already offer confidential virtual machines, e.g., using AMD SEV or Intel TDX, typically in an platform-as-a-service model. However, while these confidential virtual machines promise higher security, they are still susceptible to side-channel attacks as demonstrated in many scientific works [12, 37, 38, 39, 62]. Moreover, the lack of a trusted timing source in trusted execution environments limits use cases that require a trusted timing source, especially those that rely on time for security-critical decisions [1, 7], including defenses against privileged attackers that resort to busy-looping timing threads instead [13, 49]. This changed with AMD's introduction of the SecureTSC feature[1], providing a trusted, non-malleable timing source to

[1]Intel introduced an equivalent feature dubbed "Virtual TSC" [29].

confidential virtual machines, allowing to detect manipulations from the untrusted outside.

While the side-channel attacks themselves may be practical, their practical relevance often hinges on the attack scenario. In particular, attacks on virtual machines of other tenants in modern public cloud environments, require co-location, which is challenging to achieve for multiple reasons: First, given the massive expansion rate of cloud computing, and growing sizes of data centers, the a priori probability of co-location is shrinking. Consequently, an attacker has to spawn more containers or virtual machines to possibly achieve co-location. Second, cloud providers have systematically eliminated prior channels for co-location detection [26], requiring attackers to resort to new and less reliable side-channel-based co-location detection [25]. Third, the resource provisioning of cloud providers limits the possible co-location targets: Cloud vendors typically dedicate entire servers to one type of workload, e.g., function-as-a-service or platform-as-a-service. Furthermore, resource provisioning systems monitor workloads and dynamically move similar workloads to the same physical machine to optimize the utilization of hardware resources [43]. Consequently, to perform side-channel attacks on co-located confidential virtual machines, the attacker needs to detect co-location and mount the attack from inside a confidential virtual machine. However, co-location detection techniques for confidential virtual machines, e.g., AMD SEV-SNP, have not been studied yet.

In this paper, we present the first co-location detection technique working on confidential virtual machine hosts. We systematically study the behavior of the new SecureTSC feature supported on recent AMD processors with SEV-SNP support. Furthermore, we show that SecureTSC can be utilized as a reliable mechanism for co-location detection. We devise a minimal network protocol, `TsCupid`, which solves the challenges to this co-location detection, using a central coordination server, adjusting for various factors influencing the timestamp counter values and by threshold-comparing normalized timestamps across all connected confidential virtual machines.

We evaluate our co-location detection technique in a realistic scenario on an AMD Epyc 7313P (Zen 3, Milan microarchitecture), as public cloud vendors are still in the process of adopting SecureTSC and enabling it for customers. We focus on a scenario where two confidential virtual machines attempt to co-locate with each other. We demonstrate that, based on the SecureTSC values, we can correctly and reliably detect co-location on the same physical system within 0.13 seconds.

Finally, we show that our attack cannot be mitigated without modifying the SecureTSC feature. As long as the attacker has access to the unperturbed `rdtsc` value as a timing source, the scaling register, and the offset field, an attack remains possible. We propose a concrete design change that would fully prevent our attack while maintaining `rdtsc` as a trusted and non-malleable timing source.

In summary, this paper makes the following contributions:

- We present the first co-location detection technique on confidential virtual machines, exploiting the novel SecureTSC feature on AMD SEV-SNP processors.
- We develop a scalable methodology for co-location using our detection technique and analyze its costs on the AWS cloud and the Google Cloud Platform.
- We practically evaluate our co-location detection on an AMD Epyc 7313P system with SEV-SNP and SecureTSC enabled. We correctly detect co-location within 0.25 seconds with a success rate above 99.56 % with a 0.13 seconds detection threshold.
- We propose a design change for SecureTSC that fully mitigates our attack while maintaining the functionality of SecureTSC, by moving certain values and operations to AMD's security processor (AMD-SP).[2]

Outline. Section 2 provides background on trusted-execution environments, side channels, co-location detection, and SecureTSC. Section 3 presents the high-level idea of our co-location detection. Section 4 presents the design and implementation. Section 5 evaluates our co-location detection in a realistic scenario on an AMD Epyc 7313P machine with SEV-SNP and SecureTSC enabled. Section 6 discusses potential mitigations to prevent co-location detection. We conclude our work in Section 7.

Responsible Disclosure to AMD. We reported using SecureTSC for co-location detection to AMD on March 14th, 2024. We further showed our findings using the TsCupid protocol on April 29th, 2024. AMD acknowledged our findings and concluded that this was outside their current threat model for SecureTSC.

[2] AMD-SP and PSP are sometimes used interchangeably in the literature and official documentation.

2. Background

In this section, we provide background on trusted execution environments, side-channel attacks and co-location detection, as well as the novel SecureTSC feature.

2.1. Security of Trusted Execution Environments

The trusted computing base is the minimal set of hardware and software components that a user or workload has to trust. If the trusted computing base is compromised, the workload or the user's data may be compromised as well. Traditionally, the operating system was part of the trusted computing base. Trusted execution environments aim to minimize the trusted computing base by restricting the capabilities of the operating system and even the system administrator. Arm TrustZone [8, 18] has been deployed for more than a decade in billions of mobile devices. TrustZone has an additional, fully isolated execution realm, running a separate trusted operating system and trusted applications (called trustlets). While architectural manipulation of TrustZone is only possible with the exploitation of bugs inside the TrustZone [20], microarchitectural attacks remain unmitigated [40, 51, 53, 71]. Intel introduced a trusted execution environment, SGX [15, 27], which splits applications into an untrusted part and a trusted part (called enclave). Architectural access is mitigated, and the memory is encrypted against physical attacks. However, again bugs inside an SGX enclave [35, 65], or side channels [11, 57] can still lead to exploitation. Furthermore, transient-execution attacks, e.g., Foreshadow [61] and ZombieLoad [55], as well as power side channels [34, 41] can still leak precise information from enclaves.

The current generation of trusted execution environments follows a different approach, focusing on virtual machines. AMD's Secure Encrypted Virtualization (SEV) [4, 32] has been available in commercial processors since 2020. More recently, Intel also introduced their alternative, Trusted Domain Extension (TDX) [28]. The scientific community has studied the security of AMD SEV thoroughly. Early works exploited unencrypted virtual machine states [23, 66], control over nested page tables [23, 47, 48], and insufficient encryption [19, 67]. More recent works demonstrated that side channels on the ciphertext can deterministically leak data [37, 39] as well as the possibility of power side channel attacks [64].

Side-channel attacks on trusted execution environments generally assume co-location on the same physical host. Since in a personal computer setting this is realistic, only few works studied the security implications of malicious workloads inside trusted execution environments [54, 56]. With confidential virtual machines, this also has implications on the threat model: Cloud vendors typically dedicate entire servers to one type of workload [43], e.g., a host running many AMD SEV virtual machines. Furthermore, resource provisioning systems monitor workloads and dynamically move similar workloads to the same physical machine to optimize the utilization of hardware resources [43], *i.e.*, again many similar virtual machines are grouped onto the same server. Consequently, to perform side-channel attacks on co-located confidential virtual machines, the attacker needs to detect co-location and mount the attack from inside a confidential virtual machine.

2.2. Co-location Detection Attacks

As Zhao et al. [75] point out, co-location detection is a pivotal step to side-channel attacks in the cloud. Consequently, a long line of research studied techniques to detect co-location in commercial clouds [9, 25, 26, 52, 58, 73, 74], for instance by analyzing IP addresses, hard disk performance, network latency, and cache covert channels on the L1 cache and the last-level cache. Inci et al. [25] report that most of the aforementioned techniques have been addressed with mitigations by public cloud providers. In particular, cloud providers are aware of the concerns around SMT side channels and have in some instances mitigated them in the past by disabling hyperthreading [60] or strictly schedule only workloads of the same virtual machine on two hyperthreads [3]. Inci et al. [25] also show and argue that some side channels remain possible, e.g., Prime+Probe on the last-level cache. However, as inclusive last-level caches do not scale with high core counts, more recent processor designs often have non-inclusive L3 caches, e.g., recent Intel server processors. Consequently, even if an attacker manages to craft a Prime+Probe eviction set evict data from the L3 cache, repeatedly iterating over Prime+Probe eviction sets may reach the L3 cache but will not evict data from other co-located workload's private L1 and L2 caches. AMD processors also split the last-level cache into isolated parts per core complex. Hence, while Flush+Reload via shared memory may still be possible within one core complex, cache attacks across core complexes are currently infeasible. Varadarajan et al. [63] showed that

the effects of memory bus contention [68] can influence the performance on shared cloud systems in a way that allows for co-location detection. Zhao et al. [75] study the practicality of co-location attacks in function-as-a-service clouds and demonstrate that an attacker can reliably co-locate with a victim workload. Being specific to function-as-a-service clouds, their approach motivates further research into co-location techniques for other types of contemporary cloud systems.

2.3. AMD SecureTSC

Several works proposed to detect ongoing attacks on trusted execution environments requiring a trusted timing source [13, 49]. Focusing on Intel SGX, these works note that they have to rely on a timing thread as the `rdtsc` instruction is not available in SGX. Similarly, `rdtsc` can also be manipulated by the hypervisor in (confidential) virtual machines and, hence, even when available in a trusted execution environment cannot inherently be considered a trusted timing source. This situation changed with the novel SecureTSC feature introduced for AMD SEV-SNP [5, 16] and the Intel Virtual TSC feature [29] respectively. SecureTSC makes the `rdtsc` instruction a trusted and non-malleable timing source. A confidential virtual machine can enable SecureTSC via a bit in the `SEV_FEATURES` field in the VM Save Area. When enabled, all accesses to the timestamp counter, e.g., using `rdtsc` or `rdtscp`, or by reading the TSC Model-Specific Register (MSR), will be resolved directly by the hardware, without any way for the hypervisor to manipulate the value.[3] The SecureTSC timestamp counter is not affected by the P-State 0 frequency or by writes to the TSC MSR, which we confirm in our experiments.

Hypervisors can set the TSC frequency of virtual machines (guests). Even with SEV-SNP, the hypervisor still has control over the guest's TSC frequency, via the `DESIRED_TSC_FREQ` field in the `SNP_LAUNCH_START` request, sent to the AMD-SP while launching a guest. This value is stored in the Guest Context, a structure containing information, keys, and metadata associated with the guest. After being launched, the guest queries the AMD-SP with the `TSC_INFO` request. The response to this request contains the Guest TSC Scale, Guest TSC Offset, and TSC Factor. The Guest TSC Scale is calculated as the ratio of `DESIRED_TSC_FREQ` to the mean native frequency.

[3] This may involve the AMD Secure Processor (AMD-SP).

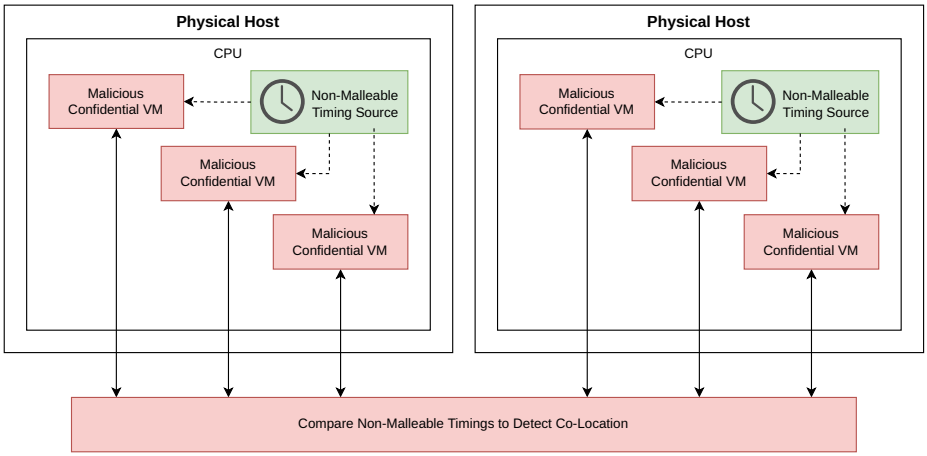


Figure 11.1.: Overview of our attack. The confidential virtual machines exchange the non-malleable timing information with a remote coordination server that evaluates the data and detects which of the virtual machines are co-located.

The Guest TSC Offset is an offset added to the guest TSC value on every read. According to official documentation [6], this value is decided by the AMD-SP. In our experiments, we did not receive a Guest TSC Offset other than 0 and we think that this is due to AMD-SP firmware not being ready. When a guest invokes an instruction to get the TSC value, the hardware first scales the TSC value with Guest TSC Scale and then adds the Guest TSC Offset.

As different cores and core complexes have different delays during boot time, especially compared to the bootstrap processor, there is a slight variance on the TSC values they observe. In all our measurements, we found no case with a difference of more than 2^{28} cycles, which corresponds to about 0.13 seconds at a frequency of 2 GHz.

3. High-Level Idea

The high-level idea behind our attack is to exploit that servers in a data center have unique uptimes. If precise-enough measurements of the real uptime within the virtual machines are possible it is easy to detect co-located virtual machines by a central coordinator. Servers, even within one rack, are typically not started at precisely the same time but cascaded to avoid load spikes on the data center's power supply. Until now timing sources in virtual machines could easy be randomized by the hypervisor as it can freely adjust the offset and frequency of `rdtsc`, thwart an attack like this. Additionally, the hypervisor already continuously mangles with the `TSC` value to, for example, compensate for virtual machine exists. It was already shown that timing sources in virtual machines are too inaccurate for microarchitectural attacks [44].

Our attack exploits the novel SecureTSC feature, available for AMD SEV-SNP confidential virtual machines that prevents the hypervisor from mangling with the `TSC` value inside a virtual machine while it is running. Figure 11.1 illustrates our attack with the example of two physical machines in the same data center. The root cause enabling our attack is the availability of a non-malleable timing source, *i.e.*, a non-malleable `rdtsc` instruction and that servers have unique uptimes. Confidential virtual machines with SecureTSC enabled have access to the scale, offset, and value of the timestamp counter at any time. With these values, the confidential virtual machine, or an external system provided with this data, can compute the precise CPU uptime in timestamp counter cycles. Based on the precise CPU uptime, we then determine which virtual machines are co-located on the same physical machine. In contrast to prior work, using one-to-one channels to detect co-location, e.g., cache contention [25, 52, 73], memory bus contention [63, 68], CPU frequency [31] our side channel on SecureTSC is a one-to-many channel: Instead of requiring time-consuming attempts between each two pairs of virtual machines to establish a channel for co-location detection, a remote coordination server can compare all non-malleable timestamp counter data at once and instantly determine the co-location status for all virtual machines involved. If only the co-location of any two virtual machines is required, the birthday problem shows that only a small number of virtual machines must be rented.

4. Design and Implementation of our Attack

In this section, we present the design and implementation of our attack. We show how an attacker can use values of SecureTSC to determine whether two or more SecureTSC-enabled AMD SEV-SNP virtual machines are co-located. We define the threat model for our attack and present an attack protocol, *TsCupid*, that we use to determine the co-location status of all connected virtual machines at once, using a single remote coordination network server.

4.1. Threat Model

The attacker has the goal to determine which of its confidential virtual machines are co-located on the same physical machine. With this knowledge, the attacker aims to mount subsequent attacks that benefit from the co-location of two or more confidential virtual machines, e.g., attacks on the cache directory which require multiple cores to exhaust the over-provisioned directory [70]. We assume the attacker has the resources to launch hundreds of confidential virtual machines, which are typically billed in short time frames (we evaluate the attack cost in Section 5.5). We also assume that starting a large number of confidential virtual machines eventually leads to co-location of some of them. Furthermore, we assume that an attacker has a coordination server. The coordination server can be one of the virtual machines within the same data center or an remote machine somewhere else.

Owners of AMD SEV-SNP confidential virtual machines can decide whether their machines are allowed to be live-migrated through the `MIGRATE_MA` field in the guest policy, a structure supplied by the guest owner that limits the actions that the hypervisor can take [6]. For this threat model, we assume that the AMD SEV-SNP confidential virtual machines disallow live migration. It must be noted that live migration of these confidential virtual machines is not yet supported [2, 22, 36]. Hence, live migration is no concern for our attack currently, in contrast to prior work, where the resource provisioning system (RPS) may have relocated virtual machines for load balancing [43]. However, we believe that our attack extends to the live-migration scenario as AMD SEV-SNP confidential virtual machines that allow live migration will know that they have been migrated by design [6].

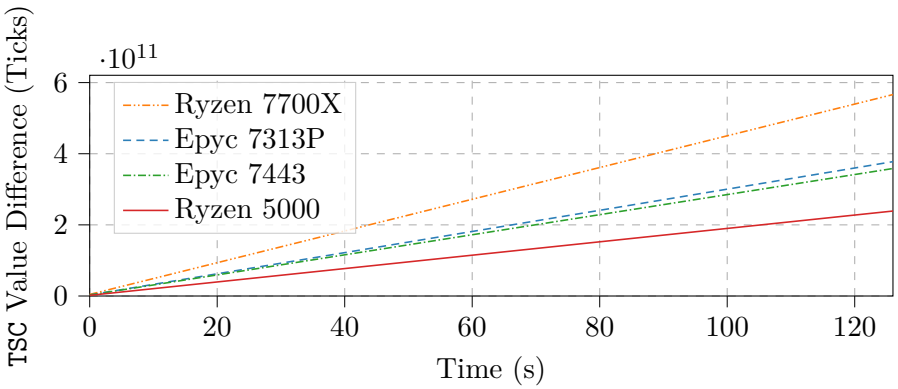


Figure 11.2.: The difference in `TSC` values reveals that different AMD CPUs increment their `TSC` at unrelated rates. For each machine, we plot the difference of a `TSC` read against the first `TSC` read. This was done on four machines for 128 s where a `TSC` read was performed every second.

4.2. Protocol for Co-location Detection

Basically, our co-location detection based on non-malleable timestamp counter values works by all virtual machines sending their current timestamp counter value to a central server. This central coordination server collects all timestamp counter values and compares them with each other. If two timestamp counter values from different virtual machines are identical, it means that the CPU they are running on has the same uptime and is, therefore, very likely, the same CPU. Hence, the virtual machines are deemed co-located.

4.3. Overview

However, this approach ignores multiple factors that negatively affect the co-location detection:

1. network latency and jitter that is possibly different on the virtual machines;
2. the high frequency of the timestamp counter;
3. the frequency of the timestamp counter can vary on different CPUs (see Figure 11.2);

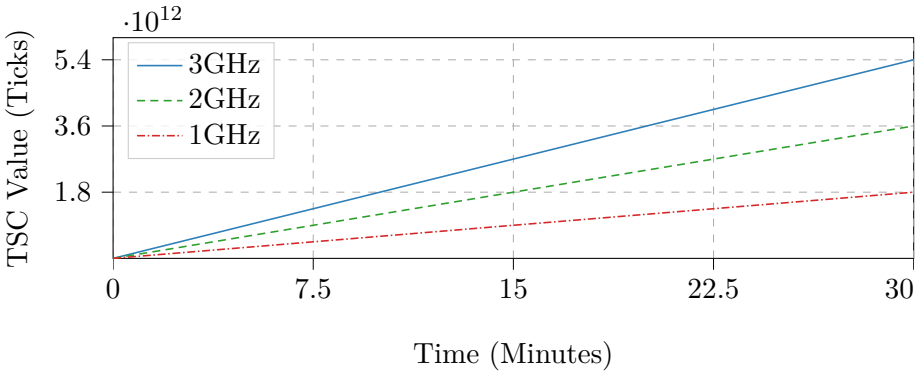


Figure 11.3.: SEV-SNP virtual machines running with different SecureTSC frequencies defined by the hypervisor. After virtual machine creation, the hypervisor can no longer change the frequency of the SecureTSC. The legend shows the configured frequency in GHz visible to the virtual machine.

4. the hypervisor can also specify a timestamp counter frequency *per* virtual machine (see Figure 11.3);

5. the timestamp counter offset field can be different *per* virtual machine.

The first point could be addressed by using an NTP-like transmission of the time value. NTP measures the round-trip time between two machines and under the assumption that both directions take approximately the same time, adds half of the round-trip time to the received time value [46]. But this solution does not address the other factors. Hence, we design a protocol, *TsCupid*, to take all five factors into account.

At its core, TsCupid collects timestamp counter information, including the value, offset and frequency on a remote coordination server. Based on this information, we can infer a *normalized timestamp* that is offset- and frequency-corrected. Storing the normalized timestamp and the frequency, we can also extrapolate the current normalized timestamp for any of the confidential virtual machines without exchanging further messages. This allows the coordination server to compare the normalized timestamp values across all confidential virtual machines regardless of when they sent in the timestamp counter information.

The normalized timestamps are still affected by the high frequency of the timestamp counter (*i.e.*, multiple increments per nanosecond). Conse-

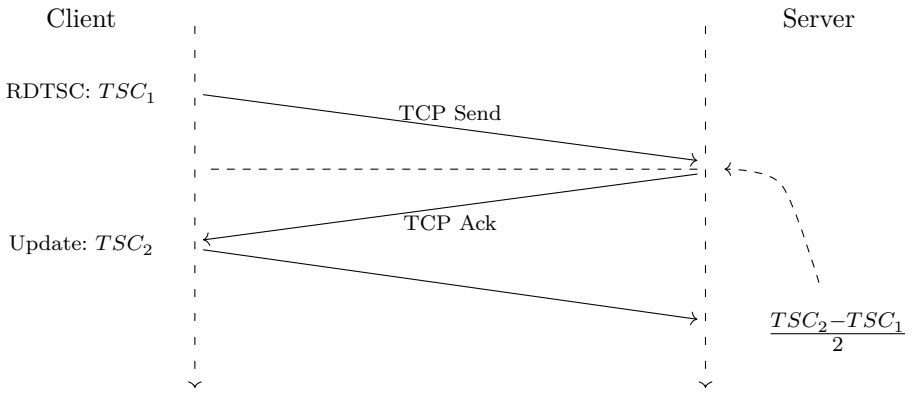


Figure 11.4.: TsCupid uses a NTP-style timestamp exchange. The round-trip time is taken into account by taking two measurements on the confidential virtual machine and one on the server. This way, the server can compute the precise timestamp counter value on the confidential virtual machine at precisely the point where it received the first message.

quently, the coordination server will practically never receive two identical timestamps. Instead, TsCupid uses a threshold to determine which timestamps are to be considered identical and the virtual machines co-located.

4.4. Implementation

The process of co-location detection with TsCupid uses specific messages:

First, the protocol starts with an initialization message (`TSCINIT`) sent by the confidential virtual machine, which contains the timestamp counter frequency and timestamp counter offset. This information is sufficient for the server to establish a record for this confidential virtual machine and to extrapolate and match further messages received.

Second, the confidential virtual machine sends a message (`TSCVALUE`) containing the current timestamp counter value. When receiving the message, the server stores its own local timestamp, as illustrated in Figure 11.4. The purpose for this is to know precisely what local timestamp corresponds to the timestamp in the confidential virtual machine. However, the timestamp counter value transmitted by the confidential virtual machine cannot be used yet as the network jitter and latency have not been accounted for.

As the **last** step, the confidential virtual machine keeps track of when the TCP acknowledgement for the `TSCVALUE` message arrives. Immediately after this acknowledgement, the confidential virtual machine sends a message (`TSCROUNDTRIP`) with the current timestamp counter value to the coordination server. This allows the coordination server to compute the time between the two timestamps from the confidential virtual machine and, thus, infer the round-trip time for the messages. Based on the round-trip time, the attacker can infer the precise point in time between the two timestamps, which is exactly when it stored its own local timestamp, as illustrated in Figure 11.4.

TsCupid has two further messages to coordinate the confidential virtual machines. First, a message (`TSCMATCH`) to inform co-located machines of their co-location. The server can send this message at any point in time but typically will do so after receiving a `TSCROUNDTRIP` message which resulted in a matching normalized timestamp in its local database. The `TSCMATCH` message contains the number of co-located confidential virtual machines and further information that may be relevant to the co-located confidential virtual machines. Second, a message `TSCWAIT`, which instructs confidential virtual machines to idle for a specified amount of time before providing a new timestamp counter value, e.g., to allow for reduction of noise with multiple measurements.

The coordination server can now compare the normalized timestamps of two confidential virtual machines as follows: Confidential virtual machine A is running at timestamp counter frequency f_A, with the timestamp counter offset o_A, and values $t_{1,A}$ and $t_{2,A}$ transmitted by `TSCVALUE` and `TSCROUNDTRIP`. The coordination server C took the timestamp $t_{A,C}$ when the `TSCVALUE` message arrived, and has its own timestamp counter frequency the current timestamp counter value available as f_C and t_C. Another confidential virtual machine B is running at timestamp counter frequency f_B, with o_B the timestamp counter offset, and values $t_{1,B}$ and $t_{2,B}$ transmitted by `TSCVALUE` and `TSCROUNDTRIP`. The normalized timestamp $n(A)$ for a confidential virtual machine A is computed as

$$n(A) = \left(\frac{t_{1,A} + t_{2,A}}{2} - o_A\right) + \frac{f_A}{f_C} \cdot (t_C - t_{A,C}) \,.$$

The coordination server can now compute whether A and B are co-located as

$$|n(A) - n(B)| < \varepsilon,$$

where ε is the detection threshold.

TsCupid stores all data to compute $n(x)$ for all confidential virtual machines in a database. Consequently, the coordination server can simply sort the data by $n(x)$ and compare the neighboring entries for being below the detection threshold ε. If the detection threshold ε is too small, the false negative rate increases, TsCupid does not correctly detect two co-located virtual machines. If ε is too large, the false positive rate increases, TsCupid erroneously labels virtual machines co-located.

5. Evaluation

In this section, we evaluate the effectiveness of our implementation of the *TsCupid* protocol on physical, local AMD SEV-SNP hardware. We show that we can effectively identify co-located SEV-SNP guest virtual machines (clients) by using our protocol. In Section 5.1, we describe our experimental hardware and software setup to study TsCupid. In Section 5.2 we evaluate the impact of network stress on our network's round trip time to aid our further noise evaluation. We run the experiments and discuss the results in Section 5.3. In Section 5.4, we look at sources of noise and challenges of TsCupid. Finally in Section 5.5, we calculate financial costs of deploying TsCupid in order to search for co-located machines.

5.1. Experimental Setup

Although SEV-SNP machines can be created on Google Cloud and Amazon's AWS, they are still in preview and do not support SecureTSC from the hypervisor side. Therefore, we evaluate TsCupid in our local network with a single machine, an AMD Epyc 7313P. The network has a link speed of 1 Gbit/s as we assume that this as a lower bound for real cloud environments. We launch 16 co-located clients with a TSC frequency of 2 GHz on our AMD Epyc 7313P. The central coordination server runs on another machine in the same network.

It is enough to evaluate our co-location detection with only a single AMD machine with only co-located virtual machines. Without SecureTSC support on any large cloud provider we can only estimate the probable differences in uptimes between different machines in a server farm. Therefore, we use a single machine to measure the minimum threshold ϵ required to

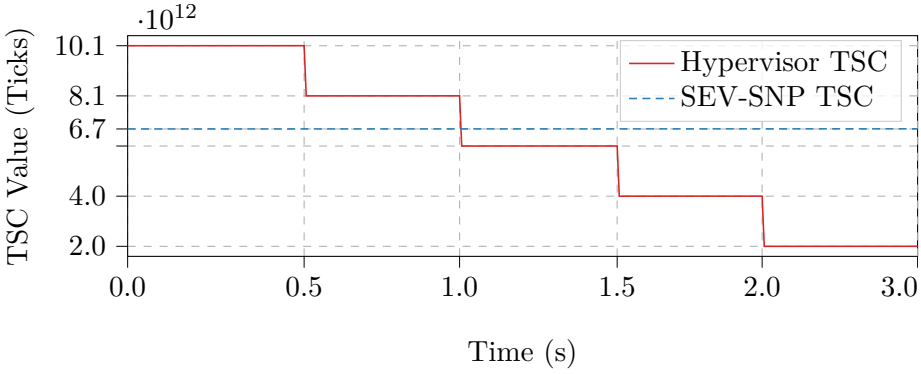


Figure 11.5.: Changing the `TSC` MSR on the hypervisor has no effect on the SecureTSC value returned by a read to the guest `TSC` MSR. Over a period of 2.5 s, the hypervisor's `TSC` MSR was decreased in intervals of fifths from 10.1×10^{12} ticks to 2.0×10^{12} ticks. The guest's `TSC` MSR stayed unaffected at a value of 6.7×10^{12} ticks.

correctly detect co-located machines. This corresponds to a co-location rate of 100 % as all our machines are co-located. With this information we can show that our found threshold ϵ of 0.13 ms is lower then the probable difference in machine uptimes. To test the correct functionality of our implementation of the client and server code and rule out any software bugs, we additionally tested our protocol on non-co-located clients and TSCupid correctly identified them as non-co-located.

Messages from the hypervisor take an average of 0.15 ms to reach the server, whereas messages from the clients take an average of 0.45 ms. We test 17 thresholds ε ranging from 8 µs to 0.5 s . For each threshold ε, we run 100 experiments each with and without network stress, totaling 3 400 experiments.

We install AMD's kernel patches for the guest [16] and the hypervisor [17], to add SecureTSC functionality. Although the guest patches redirect attempts to read the TSC MSR directly with `rdmsr` by executing `rdtsc`, we found that directly reading the TSC MSR within a guest resulted in the same value returned by executing `rdtsc`. We do not know why AMD chose to implement their guest patches like this. Moreover, the guest TSC MSR is not affected by writes to the hypervisor TSC MSR as is shown in Figure 11.5 resolving the concerns by Alder et al. [1].

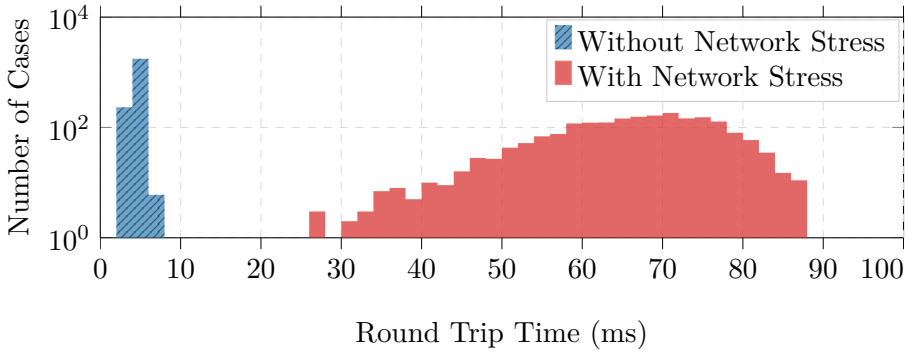


Figure 11.6.: Network stress has a discernible effect on packet round trip times. For 2 000 packets, it took an average of 4.14 ms without network stress to make a round trip to Cloudflare's `1.1.1.1` public DNS resolving server from the system we attack. With network stress, it took an average of 66.46 ms to perform the same round trip.

5.2. Impact of Network Stress on Round Trip Time

We run our experiments in two network settings, with and without a network stress program that applies pressure on the network by checking internet download speed from `fast.com`. The network stress program runs natively on the system we attack. The bandwidth to the internet from our local network is high enough to cause significant stress also within the 1 Gbit/s local network. The packet round trip times to Cloudflare's `1.1.1.1` public DNS resolving server are shown in Figure 11.6. Without network stress, it took an average of 4.14 ms with a standard deviation of 0.31 ms. With network stress, packet round trip times took an average of 66.46 ms and a standard deviation of 10.0 ms.

5.3. Evaluation of TsCupid

We base TsCupid's success, *i.e.*, accuracy based on the percentage of clients where we correctly detected co-located for each threshold. We give TsCupid a maximum of 5 s before aborting the experiment. We provide a secondary metric for the time it takes to determine the maximum number of co-locations in each attempt. Figure 11.7 shows the percentage of co-locations detected and the time taken to detect them against the 17 thresholds. We determine that network stress has negligible consequences on the number of co-locations detected and some to little effect on the

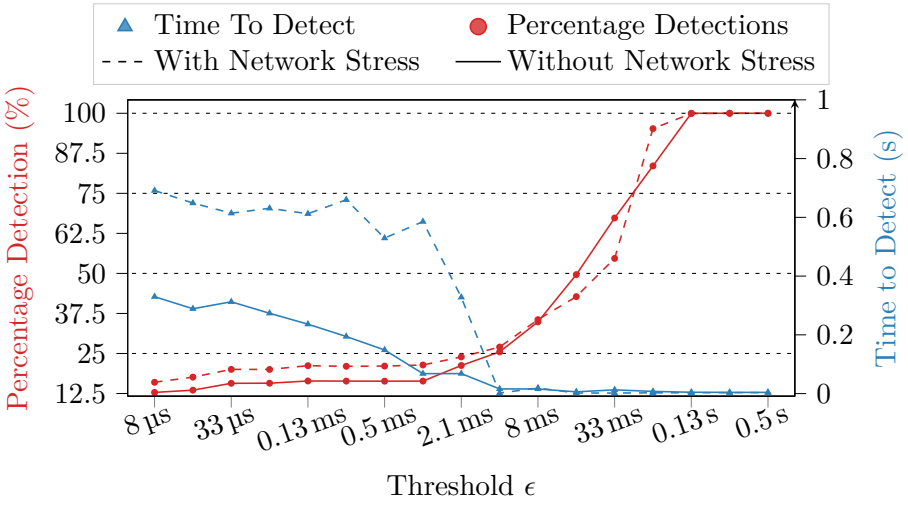


Figure 11.7.: The success rate of TsCupid with different TSC thresholds. A lower value allows for higher divergence of the TSC values. The detection percentage shows the percentage of all co-located virtual machines that were actually detected after a maximum of 5 s. The time to detect plots the time after which no new co-located virtual machines were detected. Network stress has only a small impact on the time to detect and negligible impact on the detection percentage.

time taken to detect them. We believe that this is due to TsCupid being a minimal protocol with every message (except for `TSCMATCH`) contained in one TCP segment. The `TSCMATCH` message is sent after co-location was detected so network stress has no negative impact on it.

The results of all thresholds are shown in Figure 11.7. With a threshold ϵ of 0.13 seconds, we achieve 100 % (n=100, $\sigma_{\bar{x}}$=0) detection rate without network stress and 100 % (n=100, $\sigma_{\bar{x}}$=0) detection rate with network stress. This is far lower than typical power-up delays for servers in a data center [14, 24], making our co-location detection very robust.

For low thresholds, the detection time is considerable higher with network stress than without. This can be explained by the high jitter of network packages under high network load as shown in Figure 11.6. While we tested the ping round-trip time to the internet in those experiments, the jitter seems to be lower inside the network, as the time to detect is already close to 0 with a threshold of 4.2 ms an higher.

Under the assumption that a single AWS data center has 100 000 individual servers [45] and 10 % of them use AMD CPUs that use AMD SEV-SNP with SecureTSC, there are 10 000 target servers. For TSCupid to work two machines must not have an uptime more similar than the chosen threshold ϵ. Further assuming that these server are randomly started over the span of a year, we can calculate the average number of pairs below the threshold the following way:

$$\lambda = \frac{\#\text{ Machines} - 1}{\text{Interval length}} = \frac{9999}{3\,600\,\text{s} \cdot 24 \cdot 356} = 3.25 \times 10^{-4}\,\text{s}^{-1}$$

$$P(Gap < \epsilon) = 1 - e^{-\lambda \cdot \epsilon} = 1 - e^{-6.5 \times 10^{-4} \cdot 0.13} = 4.23 \times 10^{-5}$$

$$\#\text{ False positives} = P(Gap < \epsilon) \cdot \#\text{ Machines} = 0.42$$

On average there is less than one false positive, another machine that has been booted within 130 ms of another machine.

With a higher number of 50 000 target machines, started over the span of a year, there are on average 10.6 false positive machines. However, this is not a problem for an attacker. Following the use of TSCupid, potential co-located machines can use other methods [25, 63] to verify co-location. These one-to-one channels become viable for co-location detection after reducing the number of potentially co-located machines from, e.g., 50 000 to 10.

5.4. Sources of Noise & Challenges

One of the main assumptions of TsCupid is that the network latency between the clients and the server is symmetric, which is normally the case in local networks and within data centers. Based on this assumption we can compute the TSC of the client at the time the receiver received the message, as shown in Figure 11.4. Apart from an asymmetric network architecture, a potential challenge for an attacker is the hypervisor withholding network packets being sent out or received. A benign hypervisor may even inadvertently do this, as we observed that there were vastly different times between consecutive ping round trip times ranging from 0.1 ms to 4 ms. This is not surprising, as SEV-SNP machines are known to be subject to higher network latency [21]. We address this latency noise

problem by running the protocol multiple times, if necessary, until the result is clear, *i.e.*, detection of co-location or non-co-location.

5.5. Financial Costs of TsCupid

On Google Cloud, the cheapest way to rent an AMD SEV-SNP machine is to rent an N2D series machine with two vCPUs and 1GB of RAM running for four hours, with a cost of $0.75. On Amazon's AWS, renting a Linux, M6A.Large instance running for one hour is the cheapest way to rent a machine capable of AMD SEV-SNP, with a cost of $0.105. As discussed in Section 5.3, a TsCupid server takes significantly less than an hour to determine if clients are co-located. Therefore, an attacker would need to spawn an instance, run the client code, and delete the instance within an hour.

If an attacker wants to co-locate any two virtual machines in a data center, they need to know the number of servers in a data center. Under the, for the attacker pessimistic, assumption that a single AWS data center has 100 000 individual servers [45] and half of them use AMD CPUs and support AMD SEV-SNP with SecureTSC, there are 50 000 target servers. Applying the birthday problem, the attacker needs to rent only 264 virtual machines so that any two are co-located with a probability of 50 %. To co-locate two virtual machines with 90 % probability, 480 virtual machines are required. On AWS, this would cost $27.72 and $50.40 respectively to spin up, whereas it would cost $198.00 and $360.00 on Google Cloud.

The probability of co-location further increases when taking into account that new virtual machines are probably not distributed evenly over all servers as some servers may already be completely occupied. This is significantly lower than prior co-location attacks that worked through one-to-one checks, e.g., contention covert channels.

6. Potential Mitigations

The TSC frequency, TSC offset, and `rdtsc` value are the three pieces a guest VM needs to calculate the CPU's up-time — the determiner of co-location. The simplest way to prevent guests from computing the CPU up-time is to hide the Guest TSC Offset. Since letting the hypervisor control the Guest TSC offset could result in the hypervisor being able to

tamper with the guest's `rdtsc` value, the only remaining possibility would be for the AMD-SP to control it. Therefore, we propose that AMD hide the `GUEST_TSC_OFFSET` field entirely.

It is also possible to fingerprint CPUs based on their TSC frequency alone [75]. This is relevant in SEV-SNP guest VMs as the `GUEST_TSC_SCALE` is a ratio of the hypervisor-set `DESIRED_TSC_FREQUENCY` and the mean native frequency, *i.e.*, the native frequency of the TSC on the CPU. Since both `GUEST_TSC_SCALE` and `DESIRED_TSC_FREQUENCY` are available to the guest within its guest context, it is trivial for malicious guest to calculate the mean native frequency and use it to fingerprint based on methods suggested by Zhao et al. [75]. Therefore, we propose that AMD hide either one of `DESIRED_TSC_FREQUENCY` or `GUEST_TSC_SCALE`. Moreover, we suggest that hypervisors use a randomly generated `DESIRED_TSC_FREQUENCY` for each guest that is launched.

One challenge to hiding `GUEST_TSC_OFFSET` and `DESIRED_TSC_FREQUENCY` is its impact on live migration. Before and after migrating, SEV-SNP guest VMs that use the SecureTSC feature are required to update their TSC offset by normalizing the current TSC value (incremented with `GUEST_TSC_FREQUENCY`) to `DESIRED_TSC_FREQUENCY` and adding this value to the current TSC value. This way, a guest doesn't experience a discontinuity in its `rdtsc` value before and after migrating.

Since this is the responsibility of the guest, hiding the `GUEST_TSC_OFFSET` and `DESIRED_TSC_FREQUENCY` would prevent this operation. Our proposal is to have the AMD-SP update the TSC offset before migration and have the guest store the updated value either in its VMSA or guest context. After migrating, the guest can send its offset to the AMD-SP which updates it in a similar fashion to the formula above. We believe that this mitigation isn't technically hard to implement as SEV-SNP introduces the Reverse Map Table, a hardware-supported feature where pages can be owned by the hypervisor, guests, or the AMD-SP. We propose moving the `GUEST_TSC_OFFSET`, `GUEST_TSC_SCALE`, and `DESIRED_TSC_FREQUENCY` to a page owned by the AMD-SP. However, this mitigation is only possible if AMD decides to implement it, as guests of the targeted guest threat model cannot defend themselves otherwise.

7. Conclusion

In this paper, we presented the first co-location detection technique on confidential virtual machine hosts. We showed that the new AMD SecureTSC feature, providing a trusted timing source to confidential virtual machines, can be utilized by an attacker to detect whether two confidential virtual machines are co-located, based on their trusted and non-malleable timestamp counter values. Our minimal network protocol, *TsCupid*, runs fully parallelized and allows to detect co-located machines across hundreds of confidential machines at once. It is the first one-to-many channel for co-location detection that is able to find co-located virtual machines within 0.13 seconds while requiring only a comparably small number of virtual machines due to the birthday problem. In a data center with 50 000 AMD SEV-SNP machines an attacker only needs to spawn 480 virtual machines to get two co-located virtual machines with 90 % probability. We propose several concrete changes to the SecureTSC feature that would mitigate our attack.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback on this work. We furthermore thank Stefan Gast, Andreas Kogler for valuable feedback and Martin Glasner for the BIOS random boot up delay talks. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Red Hat and Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Fritz Alder, Gianluca Scopelliti, Jo Van Bulck, and Jan Tobias Mühlberg. About Time: On the Challenges of Temporal Guarantees in Untrusted Environments. In: SysTEX. 2023 (pp. 366, 380).

[2] Amazon AWS. AMD SEV-SNP Considerations. 2024. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html (p. 374).

[3] Amazon AWS. The EC2 approach to preventing side-channels. 2024. URL: https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/the-ec2-approach-to-preventing-side-channels.html (p. 370).

[4] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. 2020. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf (p. 369).

[5] AMD. AMD64 Architecture Programmer's Manual. 2023 (p. 371).

[6] AMD. SEV Secure Nested Paging Firmware ABI Specification. Sept. 2023 (pp. 372, 374).

[7] Fatima M. Anwar, Luis Garcia, Xi Han, and Mani Srivastava. Securing Time in Untrusted Operating Systems with TimeSeal. In: RTSS. 2019 (p. 366).

[8] ARM. Security technology building a secure system using trustzone technology. 2009. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture (pp. 366, 369).

[9] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. On detecting co-resident cloud instances using network flow watermarking techniques. In: International Journal of Information Security 13 (2014), pp. 171–189 (p. 370).

[10] Johann Betz, Dirk Westhoff, and Günter Müller. Survey on covert channels in virtual machines and cloud computing. In: Transactions on Emerging Telecommunications Technologies (2016) (p. 366).

[11] 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. 369).

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (p. 366).

[13] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJá Vu. In: AsiaCCS. 2017 (pp. 366, 371).

[14] Cisco Systems Inc. Cisco UCS C-Series Servers Integrated Management Controller GUI Configuration Guide for C22 M3, C24 M3, C220 M3 and C240 M3 Servers, Release 3.0. 2024. URL: https://www.cisco.com/c/en/us/td/docs/unified_computing/ucs/c/sw/gui/config/guide/3_0/b_Cisco_UCS_C-series_GUI_Configuration_Guide_301/b_Cisco_UCS_C-series_GUI_Configuration_Guide_201_chapter_011.htm%5C#d71727e3886a1635 (p. 382).

[15] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (pp. 366, 369).

[16] Nikunj A Dadhania. [PATCH v7 00/16] Add Secure TSC support for SNP guests. 2023. URL: https://lore.kernel.org/all/20231220151358.2147066-1-nikunj@amd.com/ (pp. 371, 380).

[17] Nikunj A Dadhania. SecureTSC Hypervisor Patches. 2023. URL: https://github.com/nikunjad/linux/tree/snp-host-latest-securetsc_v5 (p. 380).

[18] Weiqi Dai, Hai Jin, Deqing Zou, Shouhuai Xu, Weide Zheng, and Lei Shi. TEE: A Virtual DRTM Based Execution Environment for Secure Cloud-End Computing. In: CCS. 2010 (pp. 366, 369).

[19] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. In: arXiv:1712.05090 (2017) (p. 369).

[20] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. In: Workshop on Mobile Security Technologies (MoST). 2014 (p. 369).

[21] Google. Confidential Computing concepts. 2024. URL: https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview (p. 383).

[22] Google. Confidential Computing: Supported configurations. 2024. URL: https://cloud.google.com/confidential-computing/confidential-vm/docs/supported-configurations (p. 374).

[23] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In: ACM SIGPLAN Notices 52.7 (2017), pp. 129–142 (p. 369).

[24] HP. Setting the power-on delay. 2024. URL: https://support.hpe.com/hpesc/public/docDisplay?docId=sd00001068en_us&page=GUID-D7147C7F-2016-0901-0A72-000000000D92.html (p. 382).

[25] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-location detection on the cloud. In: COSADE. 2016 (pp. 367, 370, 373, 383).

[26] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. In: Cryptology ePrint Archive, Report 2015/898 (2015) (pp. 367, 370).

[27] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (p. 369).

[28] Intel. Intel Trust Domain Extensions. 2023. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html (pp. 366, 369).

[29] Intel. Intel Trust Domain Extensions Module Base Architecture Specification. 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html (pp. 366, 371).

[30] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Cross-VM Side Channels and Their Use to Extract Private Keys. In: Big Data and Cloud Computing. 2014 (p. 366).

[31] Manuel Kalmbach, Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. TurboCC: A Practical Frequency-Based Covert Channel With Intel Turbo Boost. In: arXiv:2007.07046 (2020) (p. 373).

[32] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. 2016 (pp. 366, 369).

[33] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security. 2012 (p. 366).

[34] 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 (p. 369).

[35] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In: USENIX Security. 2017 (p. 369).

[36] Tom Lendacky. QEMU not working with virt-install. 2022. URL: https://github.com/AMDESE/qemu/issues/%5C#issuecomment-1171302037 (p. 374).

[37] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In: S&P. 2022 (pp. 366, 369).

[38] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected {I/O} Operations in {AMD's} Secure Encrypted Virtualization. In: USENIX Security. 2019 (p. 366).

[39] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In: USENIX Security. 2021 (pp. 366, 369).

[40] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (p. 369).

[41] 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 (p. 369).

[42] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: HPCA. 2016 (p. 366).

[43] Hosein Mohammadi Makrani, Hossein Sayadi, Najmeh Nazari, Khaled N Khasawneh, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Cloak & Co-locate: Adversarial Railroading of Resource Sharing-based Attacks on the Cloud. In: Secure and Private Execution Environment Design (SEED). 2021 (pp. 367, 370, 374).

[44] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 366, 373).

[45] Rich Miller. Inside Amazon's Cloud Computing Infrastructure. 2015. URL: `https://www.datacenterfrontier.com/design/article/11431484/inside-amazon8217s-cloud-computing-infrastructure` (pp. 383, 384).

[46] David Mills et al. Network Time Protocol. Tech. rep. RFC-958, M/A-COM Linkabit, 1985 (p. 376).

[47] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In: EuroSec. 2018 (p. 369).

[48] Mathias Morbitzer, Sergej Proskurin, Martin Radev, and Marko Dorfhuber. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In: WOOT. 2021 (p. 369).

[49] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In: USENIX ATC. 2018 (pp. 366, 371).

[50] Florian Pfarr, Thomas Buckel, and Axel Winkelmann. Cloud Computing Data Protection – A Literature Review and Analysis. In: HICSS. 2014 (p. 366).

[51] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: CCS. 2019 (p. 369).

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS. 2009 (pp. 370, 373).

[53] Keegan Ryan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In: CCS. 2019 (p. 369).

[54] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 370).

[55] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (p. 369).

[56] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (p. 370).

[57] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 369).

[58] Sushrut Shringarputale, Patrick McDaniel, Kevin Butler, and Thomas La Porta. Co-residency attacks on containers are real. In: CCSW. 2020 (p. 370).

[59] Mary-Jane Sule, Maozhen Li, and Gareth Taylor. Trust Modeling in Cloud Computing. In: SOSE. 2016 (p. 366).

[60] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In: NDSS. 2018 (pp. 366, 370).

[61] 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 (p. 369).

[62] 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. 366).

[63] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In: USENIX Security. 2015 (pp. 370, 373, 383).

[64] 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. 369).

[65] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In: ESORICS. 2016 (p. 369).

[66] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In: AsiaCCS. 2019 (p. 369).

[67] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity–Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In: S&P. 2020 (p. 369).

[68] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: ACM Transactions on Networking (2014) (pp. 371, 373).

[69] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In: USENIX Security. 2012 (p. 366).

[70] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In: S&P. 2019 (p. 374).

[71] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. In: IACR Cryptology ePrint Archive, Report 2016/980 (2016) (p. 369).

[72] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: RAID. 2016 (p. 366).

[73] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P. 2011 (pp. 366, 370, 373).

[74] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (pp. 366, 370).

[75] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS. In: ASPLOS. 2024 (pp. 370, 371, 385).

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used anything 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.

Signed by:	Jonas Raffael Juffinger
Date:	24.06.2025 10:41:37

This document is digitally signed!

This document bearing a qualified electronic signature has the same legal standing as a handwritten document in accordance with Article 25 (2) of Regulation (EU) No. 910/2014 of 23 July 2014 ('eIDAS-VO')

Validation: Information for checking the electronic signature can be found at: www.a-trust.at/pdf