Fabian Rauscher

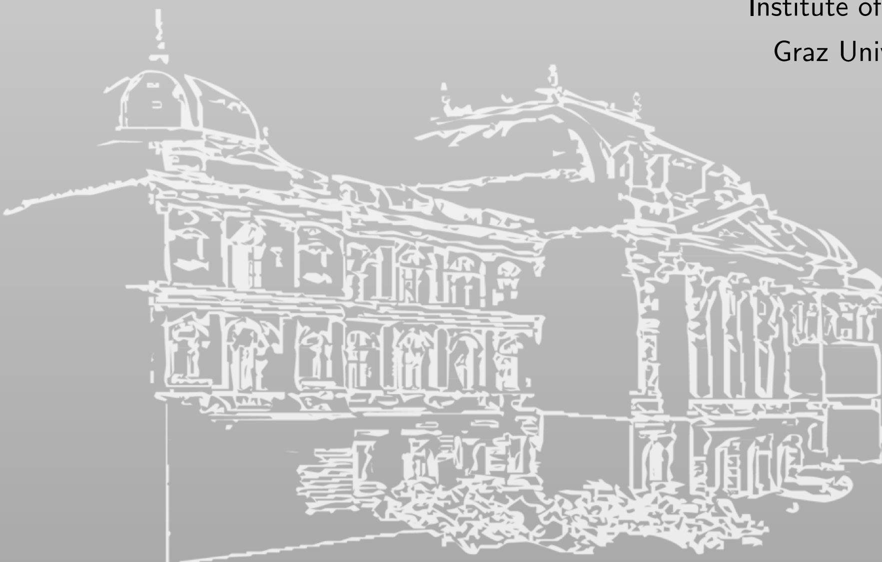
Advancing CPU Security through Attack Discovery and Systematization

PhD Thesis

Assessors: Daniel Gruss, Jo Van Bulck

May 2026

Institute of Information Security
Graz University of Technology



SCIENCE ▪ PASSION ▪ TECHNOLOGY

Abstract

In the modern age, we are extremely dependent on phones, desktop computers, servers in the cloud, and embedded devices. We trust these highly complex devices with our sensitive information. The CPUs in these systems are becoming more complex each year through microarchitectural changes and new instruction set extensions, continuously improving performance. While these changes are great for performance, they can also introduce new attack surfaces for malicious actors.

In this thesis, we advance the security of modern x86 CPUs by analyzing the attack surface introduced by new CPU features in the contexts of native code, regular virtual machines (VMs), and confidential virtual machines (CVMs). In the native and VM scenarios, we analyze 4 recently introduced instruction set extensions, WaitPKG, CLDemote, User Interrupts, and IPI Virtualization. We found that all of these extensions provide attackers with new attack techniques that can be used to recover a wide range of private and confidential information, e.g., websites visited, cryptographic keys, and sexual preferences. For the recently introduced Intel TDX, Intel's CVM implementation, we analyzed the mitigations against both single-stepping and performance-counter-based attacks and found flaws that can be used to bypass them. In the case of Intel's single-stepping mitigation, our discovered vulnerability allows attackers to turn the mitigation against itself and perform highly accurate single-stepping, which would not be possible without this mitigation. Finally, we performed a large-scale systematic analysis of the runtime and energy overheads of all hardware vulnerability mitigations and CVE patches introduced into the Linux kernel over an 8-year timeframe. Through our analysis, we found multiple instances where energy and runtime overheads diverge significantly, underscoring the need for energy-overhead measurements.

This thesis is split into two parts. Part one summarizes the main contributions, provides background, and discusses the state of the art. Part two consists of the main contributions in the form of my first-author publications. All included papers were accepted and published at double-blind peer-reviewed conferences (either A* or A ranked) and are identical[1] to their published version.

[1] They are identical in terms of text and content. Only formatting changes were made to fit the format of this thesis.

Acknowledgements

As most graduates can attest, a PhD is rarely completed without support or guidance from others, whether from the advisor, colleagues, friends, or family. Similarly, I received a lot of support and help from the people around me, whether through collaborative work on a paper, administrative assistance, or emotional support.

First and foremost, I would like to thank my advisor, Daniel Gruss. I am extremely grateful for all the time, effort, and support you provided throughout my PhD. You allowed me to pursue the research topics I am interested in while providing guidance and help whenever needed. Thank you for making my PhD such an enjoyable experience. I could not imagine a better advisor.

I want to thank all my past and present colleagues (in no particular order): Jonas Juffinger, Stefan Gast, Simone Franza, Hannes Weissteiner, Roland Czerny, Sudheendra Raghav Neela, Carina Fiedler, Martin Schwarzl, Lukas Giner, Andreas Kogler, Lukas Maar, Theresa Dachauer, and Martin Heckel. Thank you all for this lighthearted and overall just amazing work environment in which it is not only possible to do bleeding-edge high-quality research, but also joke around on a regular basis[2].

I want to thank the administrative staff of our institute for their essential role in keeping our institute going. In particular, I want to thank Ursula Urwanisch for being the backbone of our institute and for doing an amazing job handling the bureaucracy that exists at public universities in Austria, allowing us to focus on our research. I also want to thank our Sys Admins, Martin Glasner, Fabian Zelzer, and Philip Könighofer, for managing all of our IT infrastructure and handling all the support tickets I have created over the past years.

I would also like to thank Jo Van Bulck for taking the time and effort to assess this thesis and traveling to Graz to take part in the defense.

I want to thank my friends and family, in particular, my mother, Andrea. Raising two children as a single parent with a relatively low income is not easy, and while there were ups and downs, you did an amazing job. Thank you so much for prioritizing our education, making it possible for

[2]At the time of writing, our institute's messaging server contains 108 custom emojis, mainly added by the people listed here and myself.

me to not only complete the Matura at an HTL but also to continue on to do a Bachelor's, Master's, and, finally, a PhD. I also want to express my deepest thanks to my brother, Andreas.

I want to thank Martin for the regular evening gaming sessions. They are a much-appreciated break from my regular workweek, and even with those few hours we spend each week, I am certain that we will finish the factory-building game we have been playing for over a year now.

Finally, I want to thank my partner, Christina. Thank you for all the love and support you have given me over the past 5 years, and for helping me relax on the weekends. Thank you for tolerating my questionable work hours and understanding that sometimes last-minute revision requests from a shepherd forced me to postpone our plans[3]. I love you ♡.

[3]This happened multiple times...

Contents

I Introduction to the Security of Modern CPUs **1**

1. Introduction **3**
- 1.1 Main Contributions 5
- 1.2 Other Contributions 9
- 1.3 Outline 12

2. Background **15**
- 2.1 Memory Management 15
- 2.2 Virtualization 20
- 2.3 Cache Side-Channel Attacks 23

3. State of the Art **29**
- 3.1 Traditional Software-Based Side Channels 29
- 3.2 Transient-Execution Attacks 32
- 3.3 Novel Attacks on CVMs 34

4. Conclusion **41**

References **45**

II Publications **59**

List of Publications **61**

5. WaitPKG **65**

6. CLDemote **115**

7. User Interrupts & IPI Virtualization **173**

8. TDX Single-Stepping **223**

9. TELESCOPE **265**

10. Kernel Security Performance and Energy Costs **309**

Part I.

Introduction to the Security of Modern CPUs

1 Introduction

In the past 20 years, our entire lives have moved to digital devices, with most of our private information being stored on local devices or in the cloud. Particularly, online services, such as social networks, banks, and e-mail providers, hold vast amounts of sensitive information on us. Consequently, the security of these devices is becoming increasingly important.

Modern CPUs have a wide range of hardware features to protect privileged applications, e.g., the operating system and the hypervisor, from unprivileged users and unprivileged users from each other. These hardware features are well defined and, in most cases, provide protections easily verifiable in a vacuum. In a race to improve performance, new features are regularly added to improve specific use cases, such as machine learning or encryption, and microarchitectural optimizations are introduced. These changes in the architecture and microarchitecture open up new, unexpected attack surfaces between the wide range of existing parts of the architecture and the new additions. Recent x86 CPUs can include more than 1 000 different instructions [50]. Testing all of these instructions in every possible architectural and microarchitectural state with every existing optimization is impractical, making it unavoidable that vulnerabilities make their way into released CPUs.

Side-channel attacks extract information from metadata. On CPUs, there is a wide range of microarchitectural elements, such as caches, buffers, and other shared hardware resources, that can be used to collect metadata on other applications running on the system and collect information from them. Caches are one of these targets, as the cache state can change with every memory access of a victim's application. First described by Kocher [59] in 1996, cache side-channel attacks have since become a fundamental part of the microarchitectural attack landscape, used both as basic building blocks and as standalone attacks. Prime+Probe [76] and Flush+Reload [130] are the most prominent attacks, which detect

victim accesses through occupancy and timing accesses to shared memory, respectively. Not every cache attack can be deployed in every scenario due to the lack of accurate timing primitives, the absence of required instructions, or fundamental changes in CPU architectures over the past decades, e.g., a move to non-inclusive last-level caches. A wide variety of cache attacks are still being proposed to cover new and more restrictive environments [22, 42], increase the detection speed [95, 41, 81], and re-enable old attacks on modern hardware [137].

Trusted execution environments (TEEs) protect applications from a potentially malicious or compromised host through a hardware extension. They can be used to run confidential code on untrusted client systems and to protect code hosted in the cloud, e.g., proprietary machine learning models, confidential information, and digital rights management. The unique threat model, with the host being untrusted, enables new attack vectors, as attackers have full access to privileged instructions and monitoring functionality. Confidential virtual machines (CVMs) are a type of TEE that can run full virtual machines inside of them. The first commercially available CVM implementation, AMD SEV, was released in 2016 and was followed by a wide range of architectural and microarchitectural attacks [132, 126, 31, 101, 64, 98, 33, 99, 100] that uncovered flaws that were fixed in subsequent iterations. Intel's CVM implementation, Intel TDX, has only recently been released and offers a clear improvement over existing CVM implementations. Despite this, multiple works already demonstrate vulnerabilities in TDX [125, 3, 73, 100].

Mitigation proposals are a fundamental part of vulnerability-discovery research [125, 31, 41, 22, 130, 126, 128]. A wide range of microarchitectural attacks are regularly published, and most of them are not mitigated because vendors may deem them out of scope or not powerful enough to warrant a proper mitigation that results in significant overhead. More powerful microarchitectural attacks are often patched on already released CPUs through microcode updates, software mitigations, or a combination of both. These applied mitigations are accompanied by performance evaluations [5, 70, 79, 91, 39] to assess their impact on system performance, and can be deactivated if a user deems them not worth the cost. Energy overhead is usually ignored when deploying mitigations, as it is considered to correlate with runtime overhead. Prior work has shown that this assumption does not always hold, and in some cases, they can even differ drastically [46]. Herzog et al. [46] investigated the energy and runtime overhead of the Meltdown and Spectre mitigations in the Linux kernel. Since then, a wide

range of new mitigations against hardware vulnerabilities have been added to the Linux kernel with currently unknown energy overheads.

1.1. Main Contributions

In this thesis, we focus on discovering and mitigating vulnerabilities introduced by advances in CPU architectures, demonstrate how they can be exploited to leak sensitive information from other users, virtual machines, and even trusted execution environments, propose mitigation strategies, and finally analyze the performance and often overlooked energy overheads that already universally implemented mitigations have. In this section, we summarize the first-authored papers that are part of this thesis. This thesis includes 6 first-authored papers, of which 4 were published at A* conferences and 2 were published at A conferences.

A Systematic Evaluation of Novel and Existing Cache Side Channels [84]. On recent Intel Xeon processors, the shared last-level cache (LLC) is non-inclusive, meaning that data currently cached in the core private caches (L1 and L2) does not have to be cached in the LLC shared across cores. A non-inclusive LLC reduces duplicate data in the caches and prevents evicting data used by one core through memory accesses on a different core. Despite these advantages, a non-inclusive LLC can lead to increased access times when a core accesses data that is cached in the private caches of another core but not in the LLC. Intel introduced the `cldemote` instruction, which moves a cache line from caches closer to the core (L1 and L2) to the L3 (LLC) to combat this increased access latency. We abuse the `cldemote` to introduce three new cache-based attacks called Demote+Reload, Demote+Demote, and DemoteContention. We evaluate these new attacks and compare them with Flush+Reload, Flush+Flush, Evict+Probe, and Prime+Probe in a comprehensive analysis based on hit-miss margins, temporal precision, spatial precision, topological scope, attack time, blind-spot length, channel capacity, noise resilience, and detectability. The analysis provides interesting insights, even on the already well-known existing attacks. Furthermore, we demonstrate a KASLR break, an AES T-table attack, Collide+Power style leakage, an inter-keystroke timing attack, as well as one of the fastest existing cache covert channels using `cldemote` at 17.03 Mbit/s. This work was published at the Network and Distributed System Security (NDSS) Symposium in

2025 [84] in collaboration with Carina Fiedler, Andreas Kogler, and Daniel Gruss.

IdleLeak: Exploiting Idle State Side Effects for Information Leakage [87]. Modern instruction sets include a wide range of instructions that focus on optimizing the energy consumption of the CPU. One of the main focus points of such instructions is to optimize doing nothing, e.g., while the core is idling or while waiting for a lock. While the `hlt` instruction was used in the past to put a core to sleep, in recent years the `mwait` instruction has been used to allow the core to wait on specific events and enter deeper sleep states. These instructions were previously only accessible in kernel space, requiring users who want to sleep to perform an expensive switch to kernel mode. Recently, Intel introduced the new `tpause` and `umwait` instructions, which allow users to enter a light sleep state without requiring kernel assistance. To determine the security implications of allowing a user to control the sleep state we analyzed these new instructions and developed two new attacks, PassiveIdleLeak and ActiveIdleLeak. With ActiveIdleLeak, we monitor the performance change induced by a sibling logical core entering the new `C0.2` light sleep state to transfer information over a covert channel in native (7.1 Mbit/s) and cross-VM (46.3 kbit/s) scenarios. With PassiveIdleLeak, we monitor interrupts that occur not only on the logical core that is sleeping, but also on the sibling logical core with hyper-threading disabled. This extended way of interrupt monitoring across logical cores is the result of the new `C0.1` sleep state, a light sleep state introduced with these new instructions. To evaluate PassiveIdleLeak, we build a high-speed covert channel (656.37 kbit/s) and perform cross-VM website fingerprinting. Finally, we use PassiveIdleLeak to demonstrate the first interrupt-based video fingerprinting attack, achieving an open-world F_1 score of 85.2 %. This work was published at the Network and Distributed System Security (NDSS) Symposium in 2024 [87] in collaboration with Andreas Kogler, Jonas Juffinger, and Daniel Gruss.

Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs [85]. Interrupts are fundamental to the typical operation of modern operating systems. They are used to inform the operating system of external events, e.g., a network package arriving, for cross-core communication, and for inter-process communication. Previously, interrupts were only designed to be directly configurable and managed by the operating system.

Despite this, some interrupts are regularly forwarded to the user, e.g., for user space drivers or signal handling. To remove the overhead of regular context switches between the kernel and user, Intel recently introduced user interrupts, which allow the user to send and receive inter-processor interrupts, and on more recent CPUs, even timer interrupts. Based on these new low latency interfaces for unprivileged users we develop the IPI (inter-processor interrupt) side channel. While who can receive user interrupts is very restricted, user interrupts provide the user with direct access to the interrupt interface, allowing for precise timing measurements regarding how long interrupt delivery takes. We discover that the delivery delay is significantly affected by other interrupts on the system, allowing the user to monitor all interrupts delivered to any CPU core. Furthermore, this attack is even possible from inside a virtual machine, due to the new IPI virtualization feature, which is the virtual machine equivalent of user interrupts. We use the IPI side channel to build covert channels with capacities of 434.7 kbit/s (native) and 3.56 kbit/s (cross-VM), perform an inter-keystroke timing attack with an F_1 score of 97.9 % and website fingerprinting with F_1 scores of 89.0 % (native) and 71.0 % (cross-VM). This work was published at the ACM Conference on Computer and Communications Security (CCS) in 2024 [85] in collaboration with Daniel Gruss.

TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX [89]. Single-stepping is a powerful debugging feature for regular applications. Trusted execution environments (TEEs) do not support single-stepping under regular operation, as the code executed inside them should be protected from the host. Despite this, prior work has demonstrated that single-stepping is possible on a multitude of TEEs by setting up an external interrupt to trigger exactly after one instruction is executed in the trusted code. The external interrupt forces the control back to the host, which has to handle the interrupt. The resulting single-stepping primitive can be extremely useful on its own or as a base for building more powerful attacks on top of it. Intel TDX mitigates this technique by monitoring the number of instructions executed between entering and exiting a trust domain (TD) and introducing noise when this count is too low. With TDXploit, we turn the mitigation against itself by exploiting the fact that Intel chose a linear-feedback shift register (LFSR) to generate random numbers for the mitigation. We recover the secret state of the LFSR, allowing us to control the mitigation's behaviour fully. Using this, we can force the mitigation to single-step and even multi-step

the TD for us with an accuracy of >99.99 %. Furthermore, we discovered that Flush+Flush can be performed on TDX private memory, allowing an attacker to monitor memory accesses at cache-line granularity across the entire guest physical memory, without requiring any shared memory. Finally, we evaluate the applicability of multiple existing side-channel attacks on Intel TDX. This work was published at USENIX Security in 2025 [89] in collaboration with Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss.

TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters [88]. Performance counters are a powerful tool for debugging, benchmarking, and optimizing code by providing insight into architectural and microarchitectural events during execution. Because performance counters can expose sensitive information, they are accessible only to the kernel and are available to user space only for privileged users, e.g., the root user. The confidential virtual machine (CVM) includes a compromised host system, as the confidential code is protected at the hardware level from other code running on the system. As a result, performance counters become a viable attack vector in the threat model of CVM. Gast et al. [31] abused that AMD's confidential virtual machine (CVM) implementation ignores performance counters, as a potential threat vector to leak sensitive information from trusted applications. Intel's CVM implementation (Intel TDX) disables performance counters when entering trust domains (TDs), actively mitigating this type of attack. With TELESCOPE, we demonstrate that this mitigation is insufficient to eliminate the threat posed by performance-counter-based attacks fully. Performance counters that monitor the whole physical core can still be used to monitor code executed inside a TD from the sibling logical core. TELESCOPE exploits this oversight to bypass the mitigation and recover the full RSA-2048 private key from a TD running MbedTLS with an average bit error of only 0.92 bits and break the recently introduced inter-keystroke timing defense of OpenSSH (F_1 score of 99.6 %). Finally, we demonstrate that it is possible to leak information from Spectre-type attacks on TDs using the uOPs-executed performance counter, enabling the use of a large number of previously unusable Spectre gadgets. This work was published at the ACM ASIA Conference on Computer and Communications Security (AsiaCCS) in 2026 [88] in collaboration with Hannes Weissteiner and Daniel Gruss.

Systematic Analysis of Kernel Security Performance and Energy Costs [86]. Whenever a new mitigation or CVE patch is introduced, performance is one of the most important metrics. With global power consumption rising, the energy overhead of software has become a more important metric. Despite this, security research almost exclusively focuses on runtime overhead, completely ignoring energy consumption as a metric. To fill this gap, we systematically analyze the performance and energy overhead of Linux kernel CVE patches and software mitigations against hardware vulnerabilities using Intel RAPL. We automatically attribute patch commits to CVEs over an 8-year timeframe, use automated debugging to determine which benchmarks execute code affected by the patches, and finally benchmark the pre-patch and post-patch commits with each other to determine the energy and runtime overheads. Additionally, we determine the runtime and energy overhead of all software mitigations against hardware vulnerabilities that were available at the time of writing this work. To gain further insights into our measurement results, we also tracked a wide range of performance counters. We perform a high-level analysis and discuss particularly interesting cases in detail, using the measured overheads and changes in performance counter values. Our analysis demonstrates that runtime overhead is a very unreliable estimator of energy overhead, with the two metrics differing by a factor of 2 in some cases. One of these cases is the `retbleed` IBPB mitigation, which has a minimal runtime overhead of 0.4 % but reduces energy consumption by 7.1 % in the Apache Phoronix benchmark. This work was published at the ACM Asia Conference on Computer and Communications Security (AsiaCCS) in 2025 [86] in collaboration with Benedict Herzog, Timo Hönig, and Daniel Gruss.

1.2. Other Contributions

In this section, we briefly introduce peer-reviewed papers that I co-authored during my PhD. This includes 7 papers, 3 of which have been published at top-tier conferences. The other 4 papers have all been published at second-tier conferences. The papers focus mainly on side-channel attacks, microarchitectural attacks, and defenses.

DRAM addressing functions are vital for effectively performing Rowhammer and side-channel attacks on DRAM modules. We propose a new method to reliably verify the correctness of DRAM addressing functions.

To verify the effectiveness of our new approach, we compare it to 5 existing tools on 2 DDR3, 4 DDR4, and 4 DDR5 configurations. Furthermore, we found a new selection side channel, allowing us to perform row-conflict side-channel attacks on DDR5 with a leakage rate of 1.39 Mbit/s. Lastly, we perform a website fingerprinting attack with F_1 scores of 84 % (DDR4) and 74 % (DDR5). This work was published at the European Symposium on Research in Computer Security (ESORICS) in 2025 [44] in collaboration with Martin Heckel, Florian Adamsky, Jonas Juffinger, and Daniel Gruss.

Flush+Reload and Flush+Flush take advantage of the `clflush` instruction on x86 to detect memory accesses to a shared memory location, e.g., a shared library. These flush-based attacks have been used in the past to perform powerful attacks on native applications [130, 84, 41, 43] and even trusted execution environments [89]. We introduce WaitGuard, a novel detection technique for flush-based cache side-channel attacks. It uses the userspace monitor and wait instructions, which are designed to track writes to memory but can trigger spurious wake-ups when a cache line is evicted. WaitGuard is highly reliable, detecting Flush+Reload and Flush+Flush attacks with a detection rate of >99.99 %. Furthermore, we created WaitWatcher, a framework to find spurious wake-up triggers for the userspace wait instruction. This work was published at European Symposium on Research in Computer Security (ESORICS) in 2025 [62] in collaboration with Lukas Lamster, Martin Unterguggenberger, and Stefan Mangard.

In Secret Spilling Drive [55], we exploit a contention-based side channel present on modern NVMe SSDs. When exceeding the maximum I/O throughput an SSD can handle, we observe large latency spikes during accesses. By fine-tuning the number of memory accesses, an attacker can monitor accesses to the SSD by other users or applications. We analyzed 12 different SSDs and built a covert channel with a capacity of 1 503 bit/s across virtual machines. Finally, we performed an open-world website-fingerprinting attack using our novel side channel, achieving an F_1 score of 97 %. This work was published at the Network and Distributed System Security (NDSS) Symposium in 2025 [55] in collaboration with Jonas Juffinger, Giuseppe La Manna, and Daniel Gruss.

Prior work [31] used performance counters to attack AMD SEV-SNP, AMD's implementation of confidential virtual machines (CVMs). Simply disabling performance counters whenever a CVM is running is not a viable mitigation, as performance counters are used for load-balancing and

anomaly detection. We developed TEEcorrelate, a lightweight mitigation against performance counter-based attacks on CVMs while keeping them usable for legitimate use cases. TEEcorrelate statistically decorrelates the performance counters' values and temporal resolution. We demonstrate the effectiveness of TEEcorrelate through attacks on MbedTLS RSA 4096, a TOTP implementation, and a post-quantum HQC key encapsulation. With TEEcorrelate enabled, attack runtimes increase from 0.58 s to 429 s to 1-775.6 days even under optimal conditions. This work was published at USENIX Security in 2024 [121] in collaboration with Stefan Gast, Roland Czerny, Jonas Juffinger, Simone Franza, and Daniel Gruss.

SnailLoad [28] is a network-based side channel that exploits the varying response times systems exhibit when handling network packets. The precise response time can depend on the system's current activity, e.g., when the victim watches different videos or visits websites. As the SnailLoad attack relies solely on network latency, it does not require any code execution on the victim's system, but instead only a constant network connection to the victim, e.g., through the download of a picture from a malicious web server. To demonstrate the impact of SnailLoad, we performed a video-fingerprinting attack with an F_1 score of 98 % allowing us to determine which YouTube videos the victim is watching within only 90 seconds. Additionally, we performed an open-world website fingerprinting attack using SnailLoad with an F_1 score of 62.8 % on the top 100 visited websites. This work was published at USENIX Security in 2025 [28] in collaboration with Hannes Weissteiner, Robin Leander Schröder, Jonas Juffinger, Stefan Gast, Jan Wichelmann, Thomas Eisenbarth, and Daniel Gruss.

Modern GPUs are not only used for graphics processing, but also for general-purpose computing, e.g., encryption [72], matrix operations [8], and AI [111]. As a result, it is possible to run arbitrary code on GPUs to take advantage of their high parallelism and computational throughput. WebGPU enables websites to leverage GPUs, e.g., for rendering complex scenes and running browser games. We use WebGPU to perform a drive-by Prime+Probe attack on the GPU from the browser. Using this, we perform an inter-keystroke timing attack (F_1 score of 98 %) and leak a full AES key in 6 minutes. This work was published at the ACM Asia Conference on Computer and Communications Security (AsiaCCS) in 2024 [32] in collaboration with Lukas Giner, Roland Czerny, Christoph Gruber, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss.

DOmain Protection Enforcement (DOPE) [71] is a novel protection technique for the Linux kernel based on Intel's PKS hardware feature. PKS

allows us to assign mapped pages to domains via a model-specific register (MSR) that specifies which access rights are allowed for each domain at any given time. By prohibiting code access to memory not needed during normal operation, we can prevent data-oriented attacks on the kernel. Due to the hardware support, changing the access rights to different domains when switching between code segments only requires an MSR write, which is significantly faster than the page table changes and cache invalidations that would be required without PKS. This results in an average runtime overhead of only 2.3 % for real-world user applications. This work was published at the Annual Computer Security Applications Conference (ACSAC) in 2023 [71] in collaboration with Lukas Maar, Martin Schwarzl, Daniel Gruss, and Stefan Mangard.

1.3. Outline

We provide background on cache side channels, memory management, and virtualization in Chapter 2. In Chapter 3, we discuss the current state of the art for microarchitectural attacks, defenses, and the performance costs of defenses when applied in practice. Finally, we conclude Part 1 in Chapter 4. Part 2 includes a list of all first-authored and co-authored papers that I contributed to during my PhD, as well as the camera-ready versions of the main contributions that are part of this thesis.

2 Background

In this section, we provide background on virtual memory and caching, hardware-assisted virtualization, and cache side-channel attacks.

2.1. Memory Management

A memory access in a C program is as simple as dereferencing a pointer or simply using a variable. On modern CPUs, this simple operation travels through a multiple indirections, including address translations and lookups across multiple caches. These indirections exist to improve performance, security, and generalizability of applications. In this section, we discuss two fundamental parts of memory accesses relevant to this thesis. First, we discuss virtual memory, an indirection layer that provides every process with its own separate address space by introducing virtual addresses, which applications use to map to physical addresses on every memory access. Second, we take a look at caches in modern CPUs, a vital performance optimization that drastically improves memory access times. We will focus on long mode x86, also known as 64-bit x86, as it is the main architecture used throughout this thesis.

2.1.1. Virtual Memory

Virtual memory is a fundamental part of modern desktop and server CPUs. It introduces a mapping from virtual addresses used by processes to physical addresses. Virtual memory provides multiple advantages. First, it adds an additional layer of security, as each process has access only to its own virtual address space and the memory mapped into it. Second, it simplifies the operating system's cleanup process, as all the physical memory used by processes is mapped into their virtual address spaces.

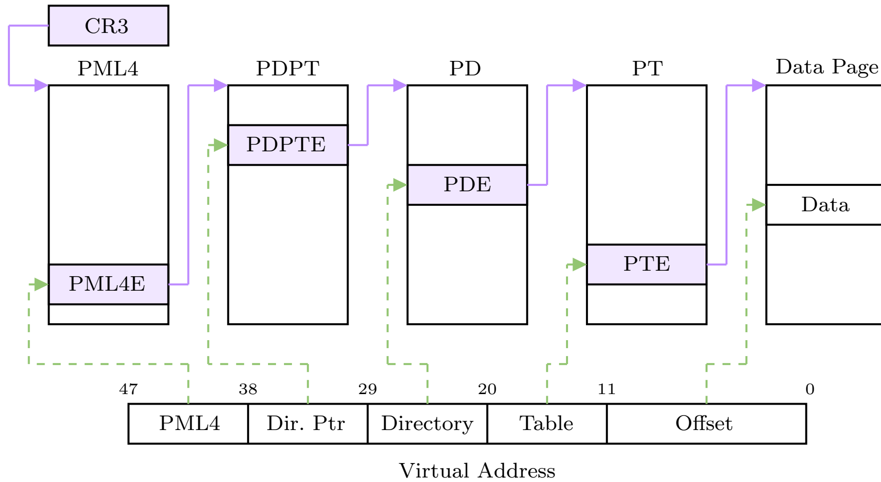


Figure 2.1.: Overview of 4-level paging on x86. The CR3 register holds the address to the root of the page table, the PML4. Each level consists of 512 entries, which, if present, hold the physical addresses of the next level. The virtual address is split into an index for each 9 bit level and a page offset. The last level (PT) holds physical addresses of the 4 kB data pages.

Third, it can reduce memory usage by mapping required memory on demand and deduplicating it if necessary. Lastly, it simplifies application development, as multiple applications can run simultaneously and use the same (virtual) addresses without unintentionally interfering with each other.

With virtual memory, memory is managed not byte by byte but in chunks called memory pages. In x86 long mode, there are currently three different page sizes, 4 kB, which is the default page size, and two larger sizes, 2 MB and 1 GB, referred to as huge pages. To translate from a virtual to a physical address, page tables[1] are used. The address translation for each memory access is done automatically, in hardware, by the memory management unit (MMU), using the page table. A page table is a tree data structure, similar to a radix tree. These page tables have either 4 or 5 layers on x86. A visualization of the page table structure with 4 layers is provided in Figure 2.1. The page table levels are called PT, PD, PDPT, PML4, and, if 5-level paging is used, PML5. Each level consists

[1]Commonly, the whole paging structure is referred to as a page table, although the lowest level of the x86 paging structure is also called page table. When we use the term page table we always refer to the whole structure, not just the lowest level.

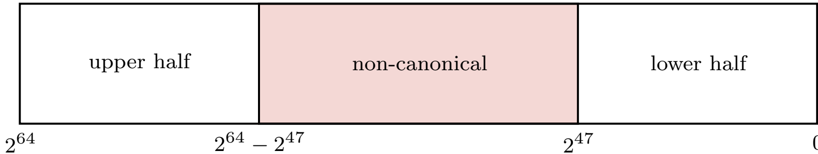


Figure 2.2.: Visualization of the address space layout on x86 with 4-level paging. Due to the sign-extension of the most-significant used bit, the usable virtual memory is split into an upper half and a lower half with an unusable non-canonical space in between.

of one 4 kB page filled with 512 entries, each 8 byte in size. Each 8 byte entry contains a present bit that determines whether the entry is valid, the physical address of the next level, and further permission bits, such as user-accessible, writable, and executable. Because there is a present bit at each level, the operating system can sparsely populate the page table, setting up only the levels required to map present data pages. Whenever a memory access violates the permissions set in the page table, an exception, called a page fault, occurs, e.g., when accessing a non-present page or writing to a read-only page. The operating system can handle this exception accordingly, typically by adjusting the page table and letting the process try again or killing the process in case of an invalid access.

The lowest page table level, the PT, directly maps to 4 kB data pages. 2 MB and 1 GB data pages, commonly referred to as huge pages, can be created by directly mapping these larger data pages into the PD or PDPT, respectively. Huge page entries are identified by a size bit in the corresponding entry, which indicates that this entry does not map to the next page table level but instead directly to a data page. The physical address of the top level (PML4 with 4 levels and PML5 with 5 levels) is stored in the CR3 register. The CR3 register defines which page table the CPU should currently use for address translation.

Virtual addresses can be split into multiple parts that are used to translate them through the page table. For a 4 kB page, the 12 least-significant bits of an address are the offset into the mapped page. Each subsequent 9-bit block is used to index into one of the page table levels, with the least-significant one indexing into the lowest level (PT) as shown in Figure 2.1. With 4-level paging this results in a 48-bit virtual address ($4 \times 9 + 12$), while with 5-level paging this results in a 57-bit virtual address ($5 \times 9 + 12$). The most-significant used bit for address translation is sign-extended to 64 bits. This sign-extension splits the virtual address space into two

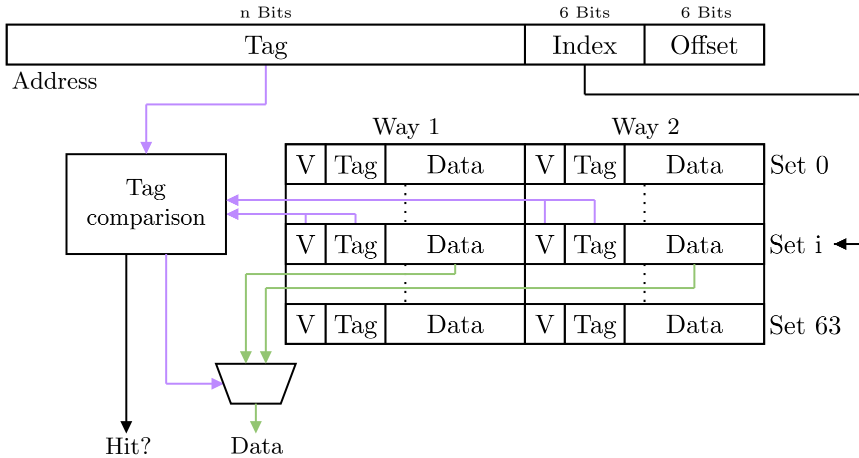


Figure 2.3.: Visualization of a 2-way set associative cache with a 6 bits index and 64 byte cache-line size. The address is split into a tag, index, and offset. The index is used to select the cache set, and the tag is compared to the stored tags of all the ways. If a tag matches and the cache line is valid, the corresponding cache line is forwarded, and a cache hit is reported.

parts, one upper half, typically used for the kernel, and one lower half, typically used for the user space, with unusable space in-between, called the non-canonical space, to which no memory can be mapped (Figure 2.2).

Compared to direct access to physical memory, which is only available on x86 in protected mode (32 bit) with paging disabled and real mode (16 bit), virtual memory adds significant overhead to memory accesses. In a worst-case scenario, virtual memory with 4-level paging requires 4 additional physical memory accesses, one per page table level, for each regular memory access. To minimize the overhead, address translations are cached in the translation lookaside buffer (TLB).

2.1.2. Caches

Caches are a vital part of modern processors, as they significantly speed up otherwise slow operations, e.g., memory accesses, by caching previous results in the CPU. Modern Intel and AMD CPUs have multiple layers of caches. An L1 cache, per-core and closest to the core; an L2 cache, a larger, slower cache also per core; and an even larger L3 cache, which

is often the last-level cache and shared between multiple cores. Some CPUs split the L1 cache into an instruction L1 cache (L1i) and a data L1 cache (L1d). Fetching data from the L1 cache takes <10 cycles, while an L3 cache access can take >50 cycles, and an access to DRAM can take >200 cycles [49]. This makes it vital that memory is cached when it is needed. Caches can be inclusive, non-inclusive, or exclusive to other caches. In older desktop CPUs, the LLC is inclusive with respect to all higher-level (closer to the cores) caches, meaning that everything cached in the higher-level caches must also be cached in the LLC. While this can simplify and improve caching of data used by multiple cores, with an increasing number of cores, this can become increasingly inefficient, as cache lines evicted from the LLC, e.g., due to space constraints, have to be evicted from all other higher-level caches. Recent CPUs contain a non-inclusive LLC, meaning that data cached in higher-level caches does not have to be present in the LLC [49].

Modern CPUs use set-associative caches for their data and address translations. A visualization for set-associative caches is provided in Figure 2.3. A set-associative cache is split into cache sets. Each cache set consists of multiple ways, usually 4 to 24, each containing one cache line [49]. Each cache line has a size of 64 bytes and holds 64-byte aligned memory. The CPU determines the cache line to use based on the memory address. The least-significant 6 bits are the offset within the cache line; the next n bits are used as an index to determine the cache set; and the remaining address bits are used as the tag. Any memory location with a matching index can be stored in any of the ways in a cache set. To fetch data from the cache, the set index is first extracted to determine the cache set, and then a parallel lookup is performed based on the tag. If one of the ways contains a cache line with the matching tag, this cache line is returned. A cache miss occurs when none of the ways contain a matching tag. Whenever a new cache line needs to be stored in a full cache set, one of the already stored lines in the set must be evicted to make space. The cache line to be evicted is chosen based on an eviction policy. While the specific eviction policy can vary depending on the CPU and cache, it is commonly a variation of the least-recently used (LRU) policy or an approximation called pseudo-LRU.

2.2. Virtualization

On server and desktop operating systems, applications run in their own separate execution environments managed by the operating system. In these environments, they have access to the whole user space part of the address space, can request memory to be mapped by the operating system, and have their memory and execution protected from other applications. Virtualization serves a similar role at the operating system level. Virtualization allows full operating systems to run in an environment managed by a hypervisor. These virtual machines (VMs) or guests get their own virtualized physical memory space, and certain operations, such as accesses to hardware components or certain CPU features, are emulated by the hypervisor. A hypervisor can be standalone software with the sole purpose of managing VMs or part of a regular operating system.

Hardware-Assisted Virtualization. Modern CPUs provide hardware extensions to allow for efficient virtualization of software, e.g., AMD SVM and Intel VMX. Both AMD SVM and Intel VMX include the same core features, with only slight variations in their exact implementations and naming. For readability, we will use Intel VMX terminology [51]. These hardware extensions include a second level of virtual memory to provide guests with access to their own hypervisor-managed physical address space, new traps for certain instructions that must be emulated, e.g., instructions that can access hardware devices, and most importantly, they manage context switches between the host and the guests. Since the guest has access to ring 0, the highest privilege level, they can modify registers that are not context-switched by existing hardware mechanisms, such as CR0, which holds bits for managing the current CPU mode and whether paging is enabled, requiring hardware support for this extended context switch. The guest state for these context switches is stored in a separate control structure called the virtual machine control structure (VMCS). Each VMCS represents the state of one virtual CPU core.

In virtualization, there are two types of context switches: VM entries, which switch from the host to the guest, and VM exits, which switch from the guest to the host. VM entries are performed by the host using a dedicated instruction, while VM exits can occur synchronously or asynchronously. Synchronous VM exits are mainly caused by the execution of specific instructions. Some instructions always cause a VM exit, e.g., `vmcall` and `cpuid`, while others can be configured to cause a VM exit, e.g.,

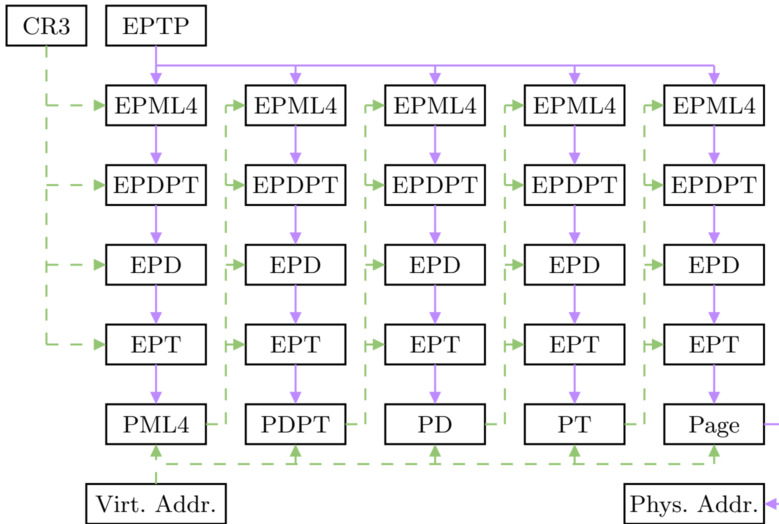


Figure 2.4.: A full address translation from a virtual address in the guest to a host physical address with a 4-level EPT and 4-level paging. The inputs are the EPTP (host physical address of the EPT PML4), the guests' CR3 (guest physical address of the guests' PML4), and the virtual address to be translated. Regular arrows represent physical addresses, and dashed lines represent indices extracted from guest physical addresses and the virtual address. The full translation requires 5 page table walks in the EPT, one per page table level of the guest's page table, and one additional one for the final data page. This results in up to 25 memory accesses in the worst-case scenario.

`rdtsc`. In addition to instructions, some CPU-state changes can also trigger synchronous VM exits, such as when the core is ready to receive interrupts, which is important for interrupt injection. Asynchronous VM exits can occur at any time and do not require any specific instructions to be executed by the guest or a specific guest state. Typical reasons for asynchronous VM exits include external interrupts and non-maskable interrupts [51].

To allow a guest to manage their own memory, virtualization extensions include a second level of virtual memory called extended page tables (EPT). Guests do not directly access physical addresses; instead, they use guest physical addresses, which map to actual (host) physical addresses just as virtual addresses map to physical addresses in a regular system without virtualization. Consequently, virtual addresses in the guest map to guest physical addresses, which map to host physical addresses. Unlike

regular x86 virtual memory, the guest physical address space does not have a non-canonical space, since the address is not sign-extended. While the use of an EPT simplifies management of guest memory and allows the host to exert a high degree of control over it, it can significantly slow down address translation. With EPT enabled, an access to a virtual address in the guest can result in up to 25 memory accesses (with 4-level paging and a 4-level EPT) compared to 5 with only virtual memory, as every access to each of the levels in the guest's page tables, which hold guest physical addresses, results in an address translation through the EPT. To minimize this additional overhead, EPT translations are also stored in the TLB [51].

Intel TDX. Intel Trust Domain eXtension (TDX) is Intel's CVM implementation based on Intel VMX and Intel Total Memory Encryption Multi-Key (TME-MK) [52]. TDX introduces a new execution mode, Secure Arbitration Mode (SEAM), and a trusted hypervisor, the TDX module, written and signed by Intel, which runs in SEAM mode. The guests running in SEAM mode are managed by the TDX module and are referred to as trust domains (TDs). To enter SEAM mode, the host can use a `seamcall`. Going back to the host from SEAM mode happens through a `seamret`, using a VMCS to save and restore the system state. For memory protection, TDX uses the already existing memory encryption provided through TME-MK. With TME-MK, it is possible to encrypt memory on a cache line granularity using keys stored in a table within the CPU. To select the correct key for a memory access, a key ID is added to the upper bits of the physical address. Parts of the key IDs are reserved for Intel TDX (private keys) while the rest are usable by the host (shared keys). These key IDs can only be used when in SEAM mode, *i.e.*, only by the TDX module and for the memory of the TDs. This ensures that the memory used by the TDX module and the TDs is inaccessible to the host. To protect against the host overwriting TDX memory, each encrypted cache line is stored together with a MAC. When a modified cache line is accessed, the MAC comparison will fail on the next victim access, causing the CPU to poison the cache line and throw an exception.

Each TD's physical address space is split into two parts: private memory and shared memory. Private memory is encrypted using a shared key and translated through an EPT managed by the TDX module. Shared memory is either unencrypted or encrypted through a shared key managed by the host. Shared memory can be used to efficiently transfer data between

the host and the TD and is not executable by the guest, minimizing the attack surface. The guest can choose between shared and private memory through the shared bit in the guest physical address, which is the most significant non-reserved bit. This strict separation decreases the chance of accidental confusion between public and private memory.

Synchronous VM exits result in a virtualization exception that is immediately injected into the guest. The guest can then decide whether to consult the host via a hypercall, in which the guest specifies which guest registers the host is allowed to read and modify. This functionality informs guests of VM exits and allows them to restrict the changes the host can make to the guest's state. Some hypercalls, e.g., those requesting certain `cpuid` leafs, are handled directly by the TDX module, while others are forwarded to the host. The host can read and modify the guest state accordingly, and then return control to the guest. To further protect the guest, the host cannot arbitrarily enable synchronous VM exit reasons, as most of the configuration is fixed by the TDX module. Asynchronous VM exits from external interrupts are forwarded to the host, with additional checks to protect against single-stepping and zero-stepping.

2.3. Cache Side-Channel Attacks

When applications execute, metadata is inadvertently generated, such as the amount of energy consumed, runtime, electromagnetic interference, or memory cached. Side-channel attacks extract information from metadata with a probability $<100\,\%$. By contrast, an attack is not considered a side-channel if the information is extracted with a probability of $=100\,\%$, as this merely indicates a different encoding of the information [40]. Side-channel attacks focus on extracting secret information through side channels, e.g., cryptographic keys, which websites a user accesses, or machine learning models. Cache side-channel attacks target the caches inside CPUs. They typically target data and instruction caches [76, 130, 41, 42, 95, 3], as they provide high-resolution information on the victim's execution, but can also target other caches, such as the TLB [37]. In the past, they have been used for powerful attacks to leak cryptographic keys [76, 130], detect which websites a user accesses [106, 27], to extract machine learning models [47, 13, 129], for inter-keystroke timing attacks [42], and to leak confidential information from trusted execution environments [125, 35, 33]. In this

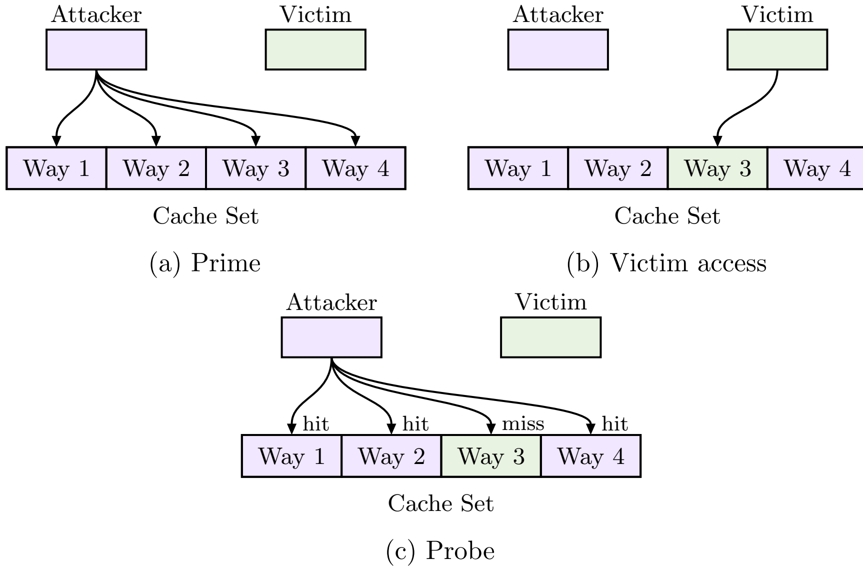


Figure 2.5.: Prime+Probe attack. First, the attacker fills the caches set with the eviction set (Figure 2.5a). Second, the victim access occurs, evicting a cache line (Figure 2.5b). Finally, the attacker measures the eviction set access time (Figure 2.5c). Slow access times indicate that parts of the eviction set were evicted by the victim.

section, we discuss 4 state-of-the-art attack techniques, Prime+Probe [76], Flush+Reload [130], Flush+Flush [41], and Evict+Reload [42].

Prime+Probe [76]: Cache lines are grouped into different cache sets based on their memory address. This mapping is fixed, meaning a specific memory address will always map to the same cache set. Because each cache set is limited in size, whenever memory has to be cached, and the set is already full, a cache line must be evicted to make space. Prime+Probe exploits the limited size of cache sets by occupying an entire cache set using a so-called eviction set. An eviction set is a set of memory addresses with at least one address per way of a cache set that all map to the same cache set. A visualization of Prime+Probe is provided in Figure 2.5. First, the attacker loads the eviction set, priming the cache set. Next, a victim access occurs to a memory location mapping to the target cache set. Finally, the attacker times a reload of the eviction set. As the cache set is filled with the attacker's eviction set, when a victim access occurs, a cache line from the eviction set is evicted. Consequently, this results in a

slower reload time for the attacker's eviction set, as at least one memory access will be a cache miss. In contrast, if the victim does not access the target memory, the reload time is faster due to fewer cache misses.

Prime+Probe has several advantages over other cache attacks, such as Flush+Reload and Evict+Reload. First, even if a victim memory access occurs while the attacker probes the memory, either a cache line is evicted that the attacker has not loaded yet, which the attacker would immediately notice, or a cache line is evicted that the attacker already probed, and this will be detected the next time the attacker probes the cache. Theoretically, this results in Prime+Probe having no blind spot. Second, it does not require shared memory with the victim. The main disadvantages for Prime+Probe are its resolution, as it only tracks whether anything on the system used the target cache set and not just the target access of the victim and the additional setup effort, as the attacker has to first build an eviction set, which can be complicated depending on system specific factors, e.g., the eviction policy and the prefetcher.

Flush+Reload [130]. With Flush+Reload, the attacker first evicts a cache line using the `clflush` instruction. The `clflush` instruction allows an application to evict a cache line from all caches. After some time has passed, the attacker can now probe the cache state by timing an access to the cache line. If the memory was accessed by the victim between the flush and the timed access, the access is fast. If the memory has not been accessed, access is slow because it is not in cache. Due to the large timing difference between a cache hit and a cache miss, Flush+Reload is very noise-resilient. Compared to Prime+Probe, Flush+Reload can monitor precise memory locations at cache-line granularity, while Prime+Probe can detect whether an access to any memory location that maps to the monitored cache set occurred. The main disadvantage of Flush+Reload is that it requires shared memory between the attacker and the victim, e.g., through page deduplication or a shared library. Additionally, Flush+Reload has a small blind spot between the probing memory access and the flush, which resets the cache state. Memory accesses between these two operations are missed by an attacker.

Flush+Flush [41]. The Flush+Flush attack only consists of a single timed execution of the `clflush` instruction. The execution time of `clflush` is highly dependent on the current state of the target memory.

If the memory is cached, it must be evicted from all caches, resulting in slow execution times. If the memory is not cached, nothing needs to be evicted, resulting in fast execution times. This allows an attacker to combine the probing step and the cache-state reset of the Flush+Reload attack into a single step, leading to significantly faster attack times and eliminating the blind spot. Flush+Flush can be used as a drop-in replacement to Flush+Reload, with the only disadvantage being a smaller timing difference.

Evict+Reload [42]. Flush+Reload and Flush+Flush require the availability of the `clflush` instruction for an attacker. This instruction is x86-specific and is not available in every context, e.g., in JavaScript. Evict+Reload replaces the `clflush` instruction execution of Flush+Reload with a cache eviction through occupancy, similar to Prime+Probe. Therefore, Evict+Reload requires building an eviction set to evict the target cache line from the cache and simulate a flush. This leads to a significantly longer attack time than Flush+Reload, as it replaces a single flush with multiple memory accesses, which may also be cache misses. Otherwise, Evict+Reload is very similar to Flush+Reload, including the blind spot between the probing and the cache reset via the eviction set, as well as the requirement for shared memory between the victim and the attacker.

3 State of the Art

In this chapter, we discuss the state of the art of software-based microarchitectural attacks and defenses. First, we discuss traditional software-based side-channel attacks, such as cache-based attacks and contention-based attacks, and their defenses in Section 3.1. Second, we look at the area of transient-execution attacks and defenses in Section 3.2. Finally, we provide an overview of new attacks on confidential virtual machines (CVMs) in Section 3.3.

3.1. Traditional Software-Based Side Channels

In this section, we discuss state-of-the-art software-based side-channel attacks and their defenses, with a focus on cache side channels and contention-based attacks. For cache-based attacks, techniques such as Prime+Probe [76], Flush+Reload [130], and Flush+Flush [41] have been introduced more than a decade ago and remain relevant to this day. Despite this, significant research continues to be published in this area [54, 80, 25, 81, 131, 27, 135, 24, 75, 2, 26]. Depending on the target, some primitives, e.g., the `clflush` instruction, are not available in every scenario; there might be no high-resolution timer available to an attacker, requiring new methods to work around these limitations. Purnal et al. [80] propose multiple amplification primitives for cache attacks, allowing an attacker to determine whether data is cached even with a very coarse-grained timer in both native and browser environments. Prime+Scope [81] is a Prime+Probe-based attack that replaces the probe step with a single memory access by priming the cache set so that this cache line is evicted from the LLC first, significantly improving probing speed. Horowitz et al. [48] further improved Prime+Scope using weird gates, which exploit transient execution to perform computations on the microarchitectural state [26]. In our work "A Systematic Evaluation of Novel and Existing Cache Side

Channels" [84] (Chapter 6), we introduced a new cache attack based on the recently introduced `cldemote` instruction, allowing for Flush+Flush- and Flush+Reload-style with significantly higher leakage rates, due to cache lines not being fully evicted from the cache. In addition to our new attacks, we provide a comprehensive systematic analysis of widely used attacks, such as Flush+Reload [130], Flush+Flush [41], Prime+Probe [76], and Evict+Reload [42], based on 9 metrics, providing novel insight on the advantages and disadvantages of not only our own new attacks based on `cldemote`, but also the existing attacks. One of our insights is that Prime+Probe, as it was initially described by Osvik et al. [76], can no longer be exploited on the shared last-level cache (LLC) of modern x86 CPUs, due to their now more complex replacement policies, and, most importantly, as the LLC is now often implemented as a non-inclusive cache. This makes eviction from the shared LLC as well as from the victim core's own private caches significantly more challenging. Zhao et al. [137] further analyzed the effectiveness of Prime+Probe on modern CPUs in the cloud. They also determined that traditional Prime+Probe on the LLC is not possible, but they can mount practical attacks on the Google cloud after overcoming multiple challenges posed by the LLC's non-inclusiveness.

While a significant amount of research focuses on x86 CPUs, an increasing number of recent publications look into both ARM and RISC-V CPUs. GhostCache [54] is a timer-free cache attack on ARM and RISC-V targeting the instruction L1 cache with a leakage rate of up to 1.68 Mbit/s. Using this new technique, GhostCache can be used to perform website fingerprinting with an accuracy of 92.02 %. Yu et al. [131] introduce a timerless cross-core cache attack on the Apple M1 that exploits hardware synchronization instructions. With attacks becoming more frequent across different CPU architectures, cache attacks have also expanded from CPUs to GPUs [27, 135, 24, 75, 2], due to their increased use in broader applications, particularly machine learning. Analogously, multiple works leak the parameters and architectures of machine learning models via cache attacks [129, 118, 68, 13].

When defending against cache-based attacks, completely eliminating shared cache is usually avoided, as this can result in a performance loss for multi-threaded applications and can complicate coherency. Secure caches aim to minimize or even eliminate the attack surface introduced by shared caches, without requiring a hard split of the available cache between applications [34, 122, 82, 109, 96, 120, 23, 38, 7, 104].

Contention-based attacks on CPUs target competitively shared resources between an attacker and a victim. The attacker continuously occupies or uses the shared resource and determines whether the victim also used it at a given time by observing throughput or timing changes. The information gained from such attacks does not directly leak data such as byte values; instead, it reveals what the victim is currently doing, which, in turn, can reveal the data they are processing. CPUs have a wide range of shared resources that can and have been targeted by these attacks, e.g., the memory bus [127] and the PCI bus [108, 107]. Hyperthreading, in particular, creates a significant attack surface for contention-based attacks, as it leads to two logical cores sharing large parts of a physical core and interleaving their instruction streams. These shared resources, which are located in the CPU core, can provide an attacker with much more fine-grained information than off-core components. A wide variety of CPU components have been attacked with the help of hyperthreading, such as execution ports [4, 93, 94], the CPU scheduler queue [30, 29], and the uOP cache [92].

Execution port usage contains a significant amount of information, as it allows an attacker to determine which types of instructions the victim executes when. This channel, named PortSmash, was initially exploited by Aldaya et al. [4] using hyperthreading in a native scenario. PortSmash has since been shown to be viable from JavaScript [93] and even through self-contention [94] (without hyperthreading). The viability of self-contention is particularly interesting, as it demonstrates that avoiding the sharing of resources is not enough to eliminate contention-based attacks fully.

In our works, we expand contention-based attacks by analyzing previously overlooked parts of the CPU. In “Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs” [85] (Chapter 7), we exploit contention on the cross-core system bus to enable cross-core interrupt detection attacks in both native and virtualized scenarios. We do this by exploiting user interrupts and IPI virtualization, which provide unprivileged access for sending and receiving interrupts between cores. While these interrupts are very restricted when it comes to which threads can send interrupts where, they provide a low-overhead interface to the system bus, allowing for precise measurements to detect and exploit contention. In “A Systematic Evaluation of Novel and Existing Cache Side Channels” [84] (Chapter 6), we also exploit contention effects for our novel DemoteContention attack, which triggers contention in the LLC, allowing an attacker to determine memory accesses by other cores to the monitored LLC cache slice. Finally,

in “IdleLeak: Exploiting Idle State Side Effects for Information Leakage” [87] (Chapter 5), we turn the concept of contention-based attacks around and focus on Intel’s new user space sleep states C0.1 and C0.2. Instead of focusing on detecting whether a resource is shared, we detect whether more resources become available on a logical core, due to the other logical core entering a sleep state. The easiest solution for eliminating contention-based attacks is to eliminate sharing, for example by disabling hyperthreading [31, 66, 30, 90, 4]. While eliminating sharing is extremely effective and can be almost trivial to implement, it can lead to significant performance and efficiency penalties, making it only a viable option for very powerful attacks or in scenarios where perfect isolation has to be achieved at all cost. Consequently, there is a significant amount of research that suggests a middle ground by using custom partitioning schemes and isolation only between security domains. Quarantine [78] implements this by isolating different security domains to different CPU cores, mitigating a large range of attacks. Taram et al. [110] suggest partitioning schemes to minimize cross-hyperthread leakage. Castes et al. [12] propose core-gapped CVMs to reduce hardware sharing between the host and the CVM.

Instead of preventing attacks from being performed, another option is to detect attack attempts and shut them down as a reactive measure. Performance counters have been used in conjunction with an anomaly-detection approach to detect potential side-channel attacks [21, 15, 45, 134, 77]. Kosasih et al. [61] analyzed the practicality of current state-of-the-art detection mechanisms that use performance counters in realistic scenarios. They determined that this approach for attack detection has significant flaws, including poor detection accuracy and slow detection speed. Finally, they demonstrate that attacks can remain undetected by modifying them so that current tools do not detect them.

3.2. Transient-Execution Attacks

Since the initial publication of Spectre [60] and Meltdown [67] in 2018, transient execution has become a common attack vector. There have been a wide range of transient execution attacks on x86 CPUs from both Intel and AMD [102, 112, 124, 74, 1, 103, 114, 97, 10, 83, 113, 90, 9]. Inception [112] manipulates the return stack buffer using phantom calls on AMD CPUs. Retbleed [124] poisons branch table buffer entries in an unprivileged process, allowing for arbitrary speculative code execution in

the kernel on Intel and AMD. Downfall [74] exploits the `gather` instruction on Intel CPUs to leak and inject data of SIMD registers, due to the instruction being able to operate on stale data during speculative execution. Agarwal et al. [1] demonstrate that Spectre attacks remain viable from JavaScript and are able to leak arbitrary memory from the address space of the process rendering the attacker's website. While mitigations are implemented against transient execution attacks, these mitigations might not always be correctly implemented and in some cases can be bypassed [123, 6]. With Apple's introduction of their M-series CPUs in 2020, they have also become a target of a wide range of transient execution attacks [58, 57, 53, 90].

To exploit vulnerabilities in transient execution, attack techniques are required that probe the microarchitectural state and make changes in it visible to an attacker. A common way to achieve this is by setting up the attack in a way that the information to be leaked is encoded into whether something is cached [67, 102, 74, 97], which can then be measured through traditional cache attacks, such as Flush+Reload and Prime+Probe. Depending on the attack, different probing techniques can be required to exfiltrate information. Zhang et al. [136] directly probes the branch target buffer (BTB) and encodes the result into the cache through two attack techniques called BunnyHop-Reload and BunnyHop-Probe. BunnyHop-Reload directly encodes whether the target branch was taken into the cache, while BunnyHop-Probe fills up the BTB with branches and then probes an eviction candidate to determine whether a new entry has been added to the BTB. Zhang et al. [133] abuse the `mwait` instruction to perform a timer-less cache side-channel attack on an AES T-Table implementation. The `mwait` instruction sleeps until a write is performed to a monitored address or a fixed timeout has passed. Instead of directly measuring memory access times, they monitor a memory location that they write to after accessing the monitored memory. In our work "TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters" [88] (Chapter 9), we propose to use the performance counter uOPs executed as a way to leak information from transient execution with a CVM target. As the performance counter also counts speculatively executed uOPs, it can be used to detect changes in instruction throughput, e.g., due to contention or different branches taken, to leak information from transient execution.

Modern operating systems include a wide range of mitigations against transient-execution attacks. Linux, in particular, allows users to specifically

deactivate and activate them at boot time [56]. These mitigations receive a significant amount of attention, due to their high overheads [46, 5, 70, 79, 91, 39]. The primary focus of research studying these overheads is the performance overhead. The energy overhead is rarely analyzed, although it roughly correlates with performance, and in some cases it may even significantly deviate from it [46].

Herzog et al. [46] evaluated the performance and energy overheads of the Meltdown [67] and Spectre [60] Linux kernel mitigations. They used a range of microbenchmarks to determine which kernel functionalities were required and created a model to predict energy and runtime overheads based on performance counters. Through their measurements, they found cases where mitigations result in negligible performance overhead but significantly increase energy consumption, underscoring the need to measure both metrics when evaluating a mitigation. We expand on this evaluation effort in our work "Systematic Analysis of Kernel Security Performance and Energy Costs" [86] (Chapter 10). We benchmarked all mitigations against hardware vulnerabilities present in Linux 6.2 as well as all CVE patches introduced over an 8-year timeframe, focusing on their performance and energy overhead. By combining automated filtering based on a debugger with the code changes for each CVE patch, we can reduce unnecessary benchmark runs, enabling a higher number of benchmark runs per CVE. Finally, we discuss interesting cases, e.g., where energy and runtime overhead significantly diverge, and determine possible reasons for their introduced overheads using performance counters. Our findings demonstrate that energy overhead should be treated as a separate metric for mitigations, as is already the case in other parts of the Linux kernel community [17, 105, 18].

3.3. Novel Attacks on CVMs

Confidential virtual machines (CVMs) have significantly increased in popularity in the security community over the recent years [33, 126, 125, 64, 69, 31, 100, 99, 132, 20, 19, 63, 101, 119, 65, 98, 14, 3, 73], due to their interesting threat model similar to regular TEEs, and their general applicability, being able to run full virtual machines inside of a trusted environment. Currently, there are only two publicly available implementations of CVMs, AMD SEV-SNP and Intel TDX. While they are similar in functionality, they differ in implementation: AMD SEV-SNP

relies mainly on hardware and microcode, while Intel TDX additionally introduces a trusted hypervisor, the TDX module, to implement most of its functionality. As the main purpose of CVMs is to provide a protected environment with similar guarantees as traditional TEEs, vulnerability research is necessary to determine potential attack vectors and mitigate them. In this section, we discuss 3 types of vulnerability research on CVMs: attacks on memory encryption, execution control techniques, and information leakage.

One of the main pillars of CVMs is memory encryption. It protects the binaries, sensitive data, encryption keys, thread states of not-scheduled threads, and execution-related states, all of which are stored in memory. If a malicious actor can read or modify a CVM's memory, the entire VM can be compromised, making it an attractive target. CIPHERLEAKS [64] monitors changes in the encrypted memory blocks on AMD SEV-SNP to determine which memory locations are modified by the guest and even infer information about the stored memory, as the encryption used lacks freshness, allowing an attacker to detect whether memory is changed back to a prior value. Li et al. [63] further studied this attack technique, finding a wide range of possible attack targets for the ciphertext side channel. In addition to attacks that can infer memory content, there are also multiple attacks that can modify CVM memory. CacheWarp [132] exploits the write-back, no-invalidate instruction to revert CVM memory to a previous state if the current state has not yet been written back to RAM on AMD. BadRAM [20] abuses a problem with some DDR5 memory modules, allowing them to remap pages, corrupt them, and even replay them on AMD. Schlüter et al. [98] exploit an issue in AMD SEV-SNP's memory management structure initialization process, allowing them to fully break the CVMs memory integrity.

Intel TDX has only been made publicly available at the end of 2023, resulting in significantly fewer publications than AMD SEV, which was introduced in 2016. One especially relevant publication is Google's Intel TDX security review [3], highlighting multiple issues and possible attack vectors, such as the coherency side channel, which allows for a Flush+Flush-style attack on CVM memory and is supposed to be mitigated in future CPUs, and a controlled-channel attack. Control over the CVMs execution, e.g., controlling how many instructions are executed at a time or forcing a VM exit on accesses to certain pages, can be extremely useful for mounting more complex attacks. Controlled-channel attacks [128] make pages inaccessible to the TEE until the TEE attempts to access them,

resulting in a page fault that returns control to the host. The host can then map the page again and continue execution. Controlled-channel attacks can be used by an attacker similarly to breakpoints during debugging, *i.e.*, by stopping guest execution during accesses to specific pages, making them a valuable tool for building further attacks [125, 126, 31]. Similarly, controlled-channel attacks are also possible on AMD SEV-SNP [126, 31].

The ability to control the victim's execution through single-stepping can be very problematic, as it allows a malicious actor to perform more precise attacks [66] or even directly leak the instructions that were executed [115]. SGX-Step [116] exploits the time-sensitivity of external interrupts to enable single-stepping on Intel SGX, as they force the control back to the host immediately so it can handle the event. By setting up a timer interrupt that triggers after one instruction is executed, the host can single-step SGX enclaves, even without official single-stepping support. AEX-Notify [16] eliminates single-stepping on Intel SGX, an Intel TEE, by prefetching the TLB entries used for the next instructions when entering an SGX enclave after an asynchronous exit. Single-stepping techniques that use external interrupts evict TLB entries used by the victim TEE, thereby increasing the time frame that must be hit by an external interrupt to achieve single-stepping. Therefore, having the TLB entries present before the next instructions are executed makes single-stepping through external interrupts impractical. Taking this one step further, TLBlur [117] prefetches the most recently used TLB entries by the SGX enclave to eliminate controlled channels. With most used pages in TLB when the SGX enclaves continue their execution, pages unmapped by the host will not lead to page faults, as translations are served by the TLB.

SEV-Step [126] extends this idea to AMD SEV-SNP, AMD's implementation of CVMs. Intel TDX attempts to mitigate precise single-stepping by detecting potential single-stepping attacks and introducing noise by letting the guest execute a random number of instructions if an attack is detected before returning control to the host [52]. Wilke et al. [125] bypass the single-stepping detection mechanism, which only checks the time elapsed using the timestamp counter, by decreasing the CPU clock frequency, causing the VM entry and exit to take long enough that the mitigation would never be triggered. This allowed single-stepping via the timer interrupt on Intel TDX, despite the presence of a mitigation. Additionally, they exploit the single-stepping mitigation; if triggered, single-step the guest. Therefore, by determining when the CPU accesses the guest's control structure, they can count the number of instructions executed whenever the

mitigation is triggered. Intel immediately patched the mitigation bypass with Instruction-Count Single-Step Defense (ICSSD) [52]. They replaced the timestamp counter detection technique with a performance counter, making it invariant to the CPU's clock frequency.

In our work "TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX" [89] (Chapter 8), we abuse Intel's mitigation to achieve extremely reliable single-stepping on Intel TDX. While Wilke et al. [125] targeted the detection technique, TDXploit exploits an improper implementation of the mitigation that is triggered once an attack is detected. The TDX module, TDX's trusted hypervisor, used an LFSR for random number generation, making it extremely predictable. By recovering the LFSR state, we were able to fully control the mitigation behaviour, allowing extremely precise single-stepping (>99.99 % accuracy) that would not be possible without the mitigation. Intel mitigated our attack by using the `rdrand` instruction for proper random number generation. In addition to the single-stepping defense, Intel TDX includes a controlled-channel defense that shuts down the CVM if multiple page faults at the same address occur in quick succession. This can be bypassed by simply alternating between two different pages [125, 88].

While external interrupts can be used by the host to force a VM exit, the host can also inject arbitrary interrupts into guests, as this is required for emulating external devices, e.g., network cards. HECKLER [100] and WeSee [99] abuse the interrupt injection mechanisms in AMD SEV-SNP and Intel TDX to change register values, by injecting the interrupt vector for the syscall interrupt and the virtualization exception, respectively. Both of these interrupts trigger interrupt handlers in the guest that modify registers, allowing a malicious host to corrupt data, bypass checks, and potentially even fully compromise the protected guest.

Performance counters allow for tracking of CPU events, e.g., cache hits, instructions executed, cycles stalled, enabling more detailed profiling of applications. As they provide detailed insight into the execution of an application, they can be a significant threat to CVMs. On AMD SEV-SNP, performance counters were initially untouched during context switches, allowing attackers to monitor applications running in CVMs through them. Lou et al. [69] use performance counters to perform website fingerprinting and keystroke detection attacks on AMD SEV-SNP. Gast et al. [31] take this one step further, by leaking an RSA private key and a TOTP secret from an AMD SEV-SNP CVM, showing that highly sensitive information

can be leaked through performance counters. Weissteiner et al. [121] propose TEEcorrelate, a lightweight mitigation against performance-counter-based attacks on CVMs that statistically decorrelates the measured values and temporal resolution on context switches. This mitigation allows for coarse grained performance monitoring of CVMs for cloud providers, while eliminating attacks that require fine-grained information such as the ones demonstrated by Gast et al. [31].

Intel TDX attempts to mitigate performance counter leakage by context switching performance counter registers. With our work "TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters" [88] (Chapter 9), we bypassed Intel's mitigation, abusing performance counters that track events across logical cores, making performance-counter-based attacks possible on Intel TDX. We achieved this by having the victim run on a single logical core within the CVM guest, while the attacker monitored the performance counter on the sibling logical core. Using this, we were able to not only extract an RSA private key from a CVM after only 400 encryptions, but also defeat the new inter-keystroke timing-attack mitigation in OpenSSH. There is currently no plan on mitigating our attacks, as Intel deems the developers of the applications responsible for hardening their software to run in CVMs, while the developers of OpenSSH do not care about the CVM threat model. This is particularly disappointing, as OpenSSH is recommended for managing CVMs in a recent guide on using Intel TDX that was written in collaboration with Intel [11]. The mitigation described by Weissteiner et al. [121] could also be used to mitigate TELESCOPE [88] when applying it on a hardware-level, globally and on all performance counter increments, not just the ones occurring inside of a CVM.

4 Conclusion

In this thesis, we presented novel attacks on x86 CPUs that exploit recently introduced features to leak information from other processes, virtual machines, and confidential virtual machines, and analyzed the impact of software mitigations against such attacks on energy consumption and performance. Based on our research presented in this thesis, we reach the following 3 conclusions:

New CPU features introduce new microarchitectural attack primitives. In the past decade, significant advances have been made in how an adversary can exploit the microarchitecture to leak information. Despite this, there is little emphasis on avoiding potential leakage when introducing new ISA extensions, beyond avoiding direct leakage of highly sensitive data similar to Meltdown. We demonstrated novel attack primitives based on 4 recently released ISA extensions, all released within a 2-year timeframe. We abused the `cldemote` instruction to perform significantly faster cache-based attacks compared to prior work (Chapter 6). Based on the new user-space wait instructions, we were able to perform cross-hyperthread interrupt detection, both native and from a VM, allowing an attacker to extract sensitive information such as which website a user accesses and which videos they watch on adult websites (Chapter 5). Lastly, we used user interrupts and IPI virtualization, extensions that allow unprivileged contexts to send IPIs with low overhead, to exploit contention on the system bus (Chapter 7).

Current CVM designs and their application lead to a false sense of security and weaker security guarantees than traditional TEEs. More traditional TEEs, such as Intel SGX and ARM TrustZone, have been around for a long time, resulting in a wide range of security improvements. For AMD SEV, in particular, multiple vulnerabilities were discovered that have already been found and fixed in Intel SGX years prior [132, 126, 31]. Even Intel does not fully leverage the lessons learned from attacks on Intel SGX in

its recently released CVM implementation, Intel TDX. Intel TDX includes a single-stepping mitigation that has been bypassed multiple times by prior work [125] and by research presented in this thesis (Chapter 8). In its current form, with both of these mitigation bypasses fixed, it is still less effective than the single-stepping mitigation in Intel SGX, due to the enormous overhead, low randomness, and the ability for an attacker to still determine precisely how many instructions were executed. While the host cannot directly modify the private memory mappings of a CVM on Intel TDX, they can block and unblock access to arbitrary pages through regular API interfaces. This feature enables controlled-channel attacks, powerful attack primitives, that would otherwise not be possible. Finally, the most significant issue CVMs face is an image problem. CVMs can run regular applications and do not require adaptation, unlike traditional TEEs such as Intel SGX. While running applications in CVMs is significantly more secure than running them on regular VMs, it can lead to a false sense of security. In this thesis, we discovered that an OpenSSH mitigation against inter-keystroke timing attacks can be bypassed when the victim runs in a CVM, because the attacker can perform more precise network latency measurements (Chapter 9). We responsibly disclosed our findings to both the OpenSSH team and Intel. The OpenSSH team does not consider CVMs part of their threat model. At the same time, Intel expects developers to mitigate this side-channel-based leakage, even though OpenSSH is recommended in a TDX guide published by Canonical in partnership with Intel [11]. Intel expects developers to ensure side-channel resilience, while most developers might not even know what a CVM is or care about this relatively niche use case.

Energy overhead needs to be considered when deploying mitigations. Energy overhead is widely considered to correlate with performance overhead. While this assumption can help provide a rough estimate, it does not always hold. A wide range of software-based mitigations have been introduced as band-aid-style solutions against hardware vulnerabilities in the past years [67, 60, 124, 113, 74, 36]. These mitigations result in a significant change in the microarchitectural behavior during a program's execution. This change in behavior can lead to a significant deviation between runtime and energy consumption as demonstrated by prior work for the Meltdown and Spectre mitigations deployed on Linux [46]. We further analyzed the relationship between energy and runtime for all mitigations in Linux 6.8 and for all CVE patches introduced into the kernel over the past 8 years (Chapter 10). We found multiple cases where runtime and energy overheads deviated significantly. The most surprising

finding was the slightly worse performance with a `retbleed` mitigation enabled, while the energy consumption decreased by more than 7 % in an Apache benchmark. Particularly for these high-overhead mitigations against hardware vulnerabilities, the deviation between performance and energy overhead can be enormous. Therefore, energy overhead should be separately considered when developing mitigations, as they can lead to significant energy savings due to their typical, immediate, large-scale deployment on server and client systems.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution. In: S&P. 2022 (pp. 32, 33).

[2] 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. 29, 30).

[3] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (TDX) security review. Tech. rep. Google, 2023 (pp. 4, 23, 34, 35).

[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. 31, 32).

[5] Nadav Amit, Michael Wei, and Dan Tsafrir. Dealing with (some of) the fallout from meltdown. In: SYSTOR. 2021 (pp. 4, 34).

[6] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In: USENIX Security. 2022 (p. 33).

[7] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In: PACT. 2013 (p. 30).

[8] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008 (p. 11).

[9] 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. 32).

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (p. 32).

[11] Canonical. Intel® Trust Domain Extensions (TDX) on Ubuntu. 2025. URL: https://github.com/canonical/tdx (pp. 38, 42).

[12] Charly Castes and Andrew Baumann. Sharing is leaking: blocking transient-execution attacks with core-gapped confidential VMs. In: ASPLOS. 2024 (p. 32).

[13] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. Side channel attacks for architecture extraction of neural networks. In: CAAI Transactions on Intelligence Technology (2021) (pp. 23, 30).

[14] Li-Chung Chiang and Shih-Wei Li. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In: ASPLOS. 2025 (p. 34).

[15] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. Real-time detection for cache side channel attack using performance counter monitor. In: Applied Sciences (2020) (p. 32).

[16] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In: USENIX Security. 2023 (p. 36).

[17] Luis Corral, Anton B Georgiev, Andrea Janes, and Stefan Kofler. Energy-aware performance evaluation of android custom kernels. In: GREENS. 2015 (p. 34).

[18] Pawel Czarnul, Jerzy Proficz, and Adam Krzywaniak. Energy-aware high-performance computing: survey of state-of-the-art tools, techniques, and environments. In: Scientific Programming (2019) (p. 34).

[19] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In: S&P. 2026 (p. 34).

[20] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In: S&P. 2025 (pp. 34, 35).

[21] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In: ACM SIGARCH Computer Architecture News (2013) (p. 32).

[22] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In: USENIX Security. 2017 (p. 4).

[23] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. In: ACM Transactions on Architecture and Code Optimization (TACO) (2011) (p. 30).

[24] 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 (pp. 29, 30).

[25] Philipp Ertmer, Robert Dumitru, and Yuval Yarom. Reverse-Engineering the Address Translation Caches. In: DIMVA. 2025 (p. 29).

[26] Dmitry Evtyushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A Eitel, Angelo Sapello, and Abhrajit Ghosh. Computing with time: Microarchitectural weird machines. In: ASPLOS. 2021 (p. 29).

[27] Ethan Ferguson, Adam Wilson, and Hoda Naghibijouybari. WebGPU-SPY: Finding Fingerprints in the Sandbox through GPU Cache Attacks. In: AsiaCCS. 2024 (pp. 23, 29, 30).

[28] 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 (p. 11).

[29] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote Scheduler Contention Attacks (Extended Version). In: arXiv:2404.07042 (2024) (p. 31).

[30] 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. 31, 32).

[31] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In: NDSS. 2025 (pp. 4, 8, 10, 32, 34, 36–38, 41).

[32] 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 (p. 11).

[33] Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In: DIMVA. 2025 (pp. 4, 23, 34).

[34] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In: USENIX Security. 2023 (p. 30).

[35] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In: EuroSec. 2017 (p. 23).

[36] Jean-Claude Graf, Sandro Rüegge, Ali Hajiabadi, and Kaveh Razavi. Vmscape: Exposing and exploiting incomplete branch predictor isolation in cloud environments. In: S&P. 2026 (p. 42).

[37] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security. 2018 (p. 23).

[38] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In: USENIX Security. 2017 (p. 30).

[39] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. 2018 (pp. 4, 34).

[40] Daniel Gruss. Transient-Execution Attacks and Defenses. 2020 (p. 23).

[41] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 4, 10, 23–25, 29, 30).

[42] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 4, 23, 24, 26, 30).

[43] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In: COSADE. 2015 (p. 10).

[44] Martin Heckel, Florian Adamsky, Jonas Juffinger, Fabian Rauscher, and Daniel Gruss. Verifying DRAM Addressing in Software. In: ESORICS. 2025 (p. 10).

[45] 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 (p. 32).

[46] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level. In: EuroSys. 2021 (pp. 4, 34, 42).

[47] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks. In: arXiv:1810.03487 (2018) (p. 23).

[48] Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-Scope: Cache Probing at Cache Speed. In: CCS. 2024 (p. 29).

[49] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2023 (p. 19).

[50] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, and 2D): Instruction Set Reference, A-Z. 2026 (p. 3).

[51] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (pp. 20–22).

[52] 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. 22, 36, 37).

[53] Hyerean Jang, Taehun Kim, and Youngjoo Shin. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In: CCS. 2024 (p. 33).

[54] Yu Jin, Minghong Sun, Dongsheng Wang, Pengfei Qiu, Yinqian Zhang, and Shuwen Deng. GhostCache: Timer-and Counter-Free Cache Attacks Exploiting Weak Coherence on RISC-V and ARM Chips. In: CCS. 2025 (pp. 29, 30).

[55] Jonas Juffinger, Fabian Rauscher, Giuseppe La Manna, and Daniel Gruss. Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS. 2025 (p. 10).

[56] The Linux Kernel. The kernel's command-line parameters. 2023. URL: `https://www.kernel.org/doc/html/v6.2/admin-guide/kernel-parameters.html` (p. 34).

[57] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In: USENIX Security. 2025 (p. 33).

[58] Jason Kim, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. iLeakage: browser-based timerless speculative execution attacks on apple devices. In: CCS. 2023 (p. 33).

[59] Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 3).

[60] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 32, 34, 42).

[61] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. Sok: Can we really detect cache side-channel attacks by monitoring performance counters? In: ASIA CCS. 2024 (p. 32).

[62] Lukas Lamster, Fabian Rauscher, Martin Unterguggenberger, and Stefan Mangard. WaitWatcher & WaitGuard: Detecting Flush-Based Cache Side-Channels through Spurious Wakeups. In: ESORICS. 2025 (p. 10).

[63] 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. 34, 35).

[64] 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. 4, 34, 35).

[65] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In: ACSAC. 2021 (p. 34).

[66] 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. 32, 36).

[67] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security. 2018 (pp. 32–34, 42).

[68] Zhibo Liu, Yuanyuan Yuan, Yanzuo Chen, Sihang Hu, Tianxiang Li, and Shuai Wang. Deepcache: Revisiting cache side-channel attacks in deep neural networks executables. In: CCS. 2024 (p. 30).

[69] Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Guo Shangwei, and Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In: DSN. 2024 (pp. 34, 37).

[70] Michail Loukeris. Efficient computing in a safe environment. In: ESEC/FSE. 2019 (pp. 4, 34).

[71] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023 (pp. 11, 12).

[72] Svetlin A Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: ICSPC. 2007 (p. 11).

[73] Nimish Mishra, Kislay Arya, Sarani Bhattacharya, Paritosh Saxena, and Debdeep Mukhopadhyay. "OOPS!": Out-Of-Band Remote Power Side-Channel Attacks on Intel SGX and TDX. In: Design Automation Conference (DAC). 2025 (pp. 4, 34).

[74] Daniel Moghimi. Downfall: Exploiting Speculative Data Gathering. In: USENIX Security. 2023 (pp. 32, 33, 42).

[75] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In: CCS. 2018 (pp. 29, 30).

[76] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 3, 23, 24, 29, 30).

[77] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In: ESSoS. 2016 (p. 32).

[78] Alexander Popov. Linux kernel heap quarantine versus use-after-free exploits. 2020. URL: `https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html` (p. 32).

[79] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, et al. Measuring the impact of spectre and meltdown. In: HPEC. 2018 (pp. 4, 34).

[80] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In: AsiaCCS. 2023 (p. 29).

[81] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (pp. 4, 29).

[82] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In: ISCA. 2019 (p. 30).

[83] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In: S&P. 2021 (p. 32).

[84] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025 (pp. 5, 6, 10, 30, 31).

[85] Fabian Rauscher and Daniel Gruss. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In: CCS. 2024 (pp. 6, 7, 31).

[86] Fabian Rauscher, Benedict Herzog, Timo Hönig, and Daniel Gruss. Systematic Analysis of Kernel Security Performance and Energy Costs. In: AsiaCCS. 2025 (pp. 9, 34).

[87] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (pp. 6, 32).

[88] Fabian Rauscher, Hannes Weissteiner, and Daniel Gruss. TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters. In: AsiaCCS. 2026 (pp. 8, 33, 37, 38).

[89] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX. In: USENIX Security. 2025 (pp. 7, 8, 10, 37).

[90] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: attacking ARM pointer authentication with speculative execution. In: ISCA. 2022 (pp. 32, 33).

[91] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In: SOSP. 2019 (pp. 4, 34).

[92] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In: ISCA. 2021 (p. 31).

[93] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In: AsiaCCS. 2022 (p. 31).

[94] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. CPU Port Contention Without SMT. In: ESORICS. 2022 (p. 31).

[95] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS. 2021 (pp. 4, 23).

[96] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security. 2021 (p. 30).

[97] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (pp. 32, 33).

[98] Benedict Schlüter and Shweta Shinde. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In: CCS. 2025 (pp. 4, 34, 35).

[99] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In: S&P. 2024 (pp. 4, 34, 37).

[100] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking Confidential VMs with Malicious Interrupts. In: USENIX Security. 2024 (pp. 4, 34, 37).

[101] Benedict Schlüter, Christoph Wech, and Shweta Shinde. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In: CCS. 2025 (pp. 4, 34).

[102] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 32, 33).

[103] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 32).

[104] Brian C. Schwedock and Nathan Beckmann. Jumanji: The Case for Dynamic NUCA in the Datacenter. In: MICRO. 2020 (p. 30).

[105] Claudio Scordino, Luca Abeni, and Juri Lelli. Energy-aware real-time scheduling in the linux kernel. In: SAC. 2018 (p. 34).

[106] 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. 23).

[107] Mert Side, Fan Yao, and Zhenkai Zhang. LockedDown: Exploiting Contention on Host-GPU PCIe Bus for Fun and Profit. In: Euro S&P. 2022 (p. 31).

[108] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible Probe: Timing Attacks with PCIe Congestion Side-Channel. In: S&P. 2021 (p. 31).

[109] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In: NDSS. 2020 (p. 30).

[110] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In: USENIX Security. 2022 (p. 32).

[111] TensorFlow. TensorFlow 2.16.1 API Documentation. 2026 (p. 11).

[112] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In: USENIX Security. 2023 (p. 32).

[113] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security. 2018 (pp. 32, 42).

[114] 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. 32).

[115] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 36).

[116] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (p. 36).

[117] Daan Vanoverloop, Andres Sanchez, Flavio Toffalini, Frank Piessens, Mathias Payer, and Jo Van Bulck. TLBlur: Compiler-assisted automated hardening against controlled channels on off-the-shelf Intel SGX platforms. In: USENIX Security. 2025 (p. 36).

[118] Han Wang, Syed Mahbub Hafiz, Kartik Patwari, Chen-Nee Chuah, Zubair Shafiq, and Houman Homayoun. Stealthy inference attack on DNN via cache-based side-channel attacks. In: DATE. 2022 (p. 30).

[119] 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. 34).

[120] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In: SIGARCH Computer Architecture News (2007) (p. 30).

[121] 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. 11, 38).

[122] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security. 2019 (p. 30).

[123] Johannes Wikner and Kaveh Razavi. Breaking the barrier: Post-barrier Spectre attacks. In: S&P. 2025 (p. 33).

[124] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In: USENIX Security. 2022 (pp. 32, 42).

[125] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In: CCS. 2024 (pp. 4, 23, 34, 36, 37, 42).

[126] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In: CHES. 2024 (pp. 4, 34, 36, 41).

[127] 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. 31).

[128] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (pp. 4, 35).

[129] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In: USENIX Security. 2020 (pp. 23, 30).

[130] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 3, 4, 10, 23–25, 29, 30).

[131] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W Fletcher. Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In: USENIX Security. 2023 (pp. 29, 30).

[132] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In: USENIX Security. 2024 (pp. 4, 34, 35, 41).

[133] 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 (p. 33).

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

[135] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive. In: USENIX Security. 2024 (pp. 29, 30).

[136] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the Instruction Prefetcher. In: USENIX Security. 2023 (p. 33).

[137] 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. 4, 30).

Part II.

Publications

List of Publications

During my PhD studies, I contributed to 13 publications in conference proceedings, 6 of which are included in this thesis as shown below.

Publications in this Thesis

1. **Fabian Rauscher**, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025.
2. **Fabian Rauscher** and Daniel Gruss. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In: CCS. 2024.
3. **Fabian Rauscher**, Benedict Herzog, Timo Hönig, and Daniel Gruss. Systematic Analysis of Kernel Security Performance and Energy Costs. In: AsiaCCS. 2025.
4. **Fabian Rauscher**, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024.
5. **Fabian Rauscher**, Hannes Weissteiner, and Daniel Gruss. TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters. In: AsiaCCS. 2026.
6. **Fabian Rauscher**, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX. In: USENIX Security. 2025.

Other Contributions

1. 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.

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

3. Martin Heckel, Florian Adamsky, Jonas Juffinger, **Fabian Rauscher**, and Daniel Gruss. Verifying DRAM Addressing in Software. In: ESORICS. 2025.

4. Jonas Juffinger, **Fabian Rauscher**, Giuseppe La Manna, and Daniel Gruss. Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS. 2025.

5. Lukas Lamster, **Fabian Rauscher**, Martin Unterguggenberger, and Stefan Mangard. WaitWatcher & WaitGuard: Detecting Flush-Based Cache Side-Channels through Spurious Wakeups. In: ESORICS. 2025.

6. Lukas Maar, Martin Schwarzl, **Fabian Rauscher**, Daniel Gruss, and Stefan Mangard. DOPE: DOmain Protection Enforcement with PKS. In: ACSAC. 2023.

7. 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.

5

IdleLeak: Exploiting Idle State Side Effects for Information Leakage

Publication Data

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

Contributions

Main Author.

IdleLeak: Exploiting Idle State Side Effects for Information Leakage

Fabian Rauscher, Andreas Kogler, Jonas Juffinger, Daniel Gruss

Graz University of Technology

Abstract

Modern processors are equipped with numerous features to regulate energy consumption according to the workload. For this purpose, software brings processor cores into idle states via dedicated instructions such as `hlt`. Recently, Intel introduced the C0.1 and C0.2 idle states. While idle states previously could only be reached via privileged operations, these new idle states can also be reached by an unprivileged attacker. However, the attack surface these idle states open is still unclear.

In this paper, we present IdleLeak, a novel side-channel attack exploiting the new C0.1 and C0.2 idle states in two distinct ways. Specifically, we exploit the processor idle state C0.2 to monitor system activity and for novel means of data exfiltration, and the idle state C0.1 to monitor system activity on logical sibling cores. IdleLeak still works regardless of where the victim workload is scheduled, *i.e.*, cross-core, due to the low-level x86 design. We demonstrate that IdleLeak leaks significant information in a native keystroke-timing attack, achieving an F1 score of 90.5 % and a standard error on the timing prediction of only 12 µs. We also demonstrate website- and video-fingerprinting attacks using IdleLeak traces, pre-processed with short-time Fourier transforms, and classified with convolutional neural networks. These attacks are highly practical with F1 scores of 85.2 % (open-world website fingerprinting) and 81.5 % (open-world video fingerprinting). We evaluate the throughput of IdleLeak side channels in both directions in covert channel scenarios, *i.e.*, using interrupts and performance-increasing effects. With the performance-increasing effect, IdleLeak achieves a true capacity of 7.1 Mbit/s in a native and 46.3 kbit/s in a cross-VM scenario. With interrupts, IdleLeak achieves 656.37 kbit/s in a native scenario. We conclude that mitigations against IdleLeak are necessary in both personal and cloud environments when running untrusted code.

1. Introduction

Modern CPUs have various features to meet the energy consumption, and performance demands, of users and workloads. One of the most significant performance increases came with out-of-order execution. While out-of-order execution enables parallel use of multiple execution units for different instruction, most workloads do not fully utilize this parallelism. Hence, vendors introduced simultaneous multithreading (SMT), giving the software level the illusion of having more CPU cores, so-called logical cores. The CPU interleaves the instructions from multiple logical cores on one physical core. Consequently, CPUs achieve substantially higher utilization of execution units, improving performance by roughly 30 % [58]. The reason for this performance gain is that many on-core and off-core resources are competitively shared (e.g., caches and TLB), and others are inexpensive to split statically (e.g., load and store buffer) [54]. Thus, SMT can still negatively impact the performance of workloads if resources are statically split or partially occupied by other workloads.

Over the past three decades, researchers investigated the security implications of these optimizations and found numerous side channels in various microarchitectural elements. A particularly long line of research focused on caches and buffers [66, 3, 34], both in scenarios where attacker and victim share a physical core [49, 10], as well as cross-core attacks [66, 31]. Other works investigated contention on execution ports [1, 43] and scheduler queues [9]. All these works focused on contention and interference on specific microarchitectural elements. However, the increased necessity for energy and performance improvements led to more advanced efficiency features in recent CPUs. These go beyond single microarchitectural elements and operate on the level of entire logical cores, physical cores, or entire packages. In particular idle states can introduce substantial efficiency gains. However, until recently, idle state management was only possible from kernel space. Still, prior work showed that these optimizations may still have security implications [68], in particular when attacking Trusted Execution Environments (TEEs).

With the efficiency and performance goals for new microarchitecture designs, Intel recently introduced the user space sleep primitives `tpause`, a timed pause, and `umwait`, a timed pause with a memory trigger. These instructions allow efficient waiting for timeouts and memory accesses while the processor is idle to save energy. Prior work showed that `umwait` can be exploited for Spectre-type attacks without a timing-based side channel

and interrupt monitoring in native code [68], similar to prior works showed for other side channels [42, 27, 48]. While the work [68] did show that the interrupt monitoring can be used for website fingerprinting in native code, they did not further investigate other undocumented interrupt wake-up triggers, the possible security implications of unprivileged modification of idle states, the behaviour of the idle states in VMs, or the related `tpause` instruction.

In this paper, we present IdleLeak, a novel attack exploiting control over the C0.1 and C0.2 idle states from user space and their undocumented behaviors. Our first attack technique, ActiveIdleLeak, shows that the performance-increasing effects of the C0.2 idle state yield unforeseen security implications on workloads co-located on a sibling logical core. Our second attack technique, PassiveIdleLeak, shows that both the C0.1 and C0.2 idle state are woken up by some processor operations, including events from workloads running on different physical cores, and, in general, different activity on the same core or a sibling logical core. That is, instead of attempting co-location with the victim, PassiveIdleLeak can run on an arbitrary physical core where either of the logical cores is influenced by the victim workload due to the way x86 implements interrupts.

In our first attack, ActiveIdleLeak, the attacker uses the unprivileged `tpause` instruction to put a logical core into the C0.2 idle state. The C0.2 idle state relinquishes some on-core resources (e.g., load buffer, store buffer, and reorder-buffer entries) to the other logical core. Consequently, the single-threaded performance of the other logical sibling core increases. The attacker uses this performance side effect to construct a covert communication channel between separate security domains. We evaluate the capacity of the ActiveIdleLeak channel in native and cross-VM covert channels, yielding 7.1 Mbit/s ($\sigma_{\bar{x}} = 0.004$ Mbit/s, $n = 512$) and 46.3 kbit/s ($\sigma_{\bar{x}} = 0.15$ kbit/s, $n = 370$) respectively.

For our second attack, PassiveIdleLeak, we exploit both idle states to spy on activities on the same core and the sibling logical core. Both idle states are left, e.g., on interrupts, special instructions, user input, and scheduling behavior, on the same core. C0.1 is also influenced by such activity on sibling logical cores. Surprisingly, both idle states are also influenced by such activity in the host and co-located virtual machines (VMs). This unexpected behavior opens up attack vectors for leaking user activity information from host to guest.

We present an unprivileged keystroke-timing attack using PassiveIdleLeak, with an F1 score of 90.5 % and a standard error on the timing prediction of only 12 µs. Our keystroke-timing attack works across cores, regardless of where the victim is scheduled. This is possible since the attacker can occupy multiple cores, including logical cores that belong to the physical core receiving the key-down and key-up interrupts.

We demonstrate that PassiveIdleLeak running inside a VM can also be used to fingerprint website and video accesses by the host. We show that we can target either the interrupt handling of a victim workload running on another physical core, or the victim workload itself if it is co-located on a logical sibling core. In an open- and closed-world evaluation with 100 websites, we achieve F1 scores of 85.2 % and 93.1 % respectively on an independent test set. We then demonstrate that we can distinguish videos accessed by a user in a video-fingerprinting attack on popular video streaming platforms, with F1 scores of 90.2 % (closed-world) and 81.5 % (open-world) for YouTube, and 75 % (closed-world) and 70.5 % (open-world) for Pornhub using only the first 10 s of a video. Our attack is the *first* to demonstrate video fingerprinting based on interrupt detection via a side channel. The information gained by this attack could be used for extortion campaigns, illustrating the severe and previously unknown privacy implications of the novel idle states. We evaluate the PassiveIdleLeak channel in a native covert-channel scenario, yielding a true capacity of 656.37 kbit/s ($\sigma_{\bar{x}} = 0.63$ kbit/s, $n = 1\,024$).

Since interrupt scheduling does not adhere to side-channel aware scheduling policies like gang scheduling [21], systems are unprotected against our attack. Contrary to prior works, we focus on the undocumented behavior of the novel idle states and the new instructions unexpected behavior inside VMs compared to existing instructions. We discuss possible mitigations against IdleLeak and find that efficient mitigations are not trivial due to the nature of external interrupts and the ambiguity of the interrupt receiver.

To summarize, we make the following contributions:

- We analyze the security properties of the C0.1 and C0.2 idle states that can be manipulated by an unprivileged attacker, resulting in our novel attack IdleLeak.
- We show that IdleLeak is fast and robust, leaking up to 7.1 Mbit/s ($\sigma_{\bar{x}} = 0.004$ Mbit/s, $n = 512$) in a native covert channel and 46.3 kbit/s ($\sigma_{\bar{x}} = 0.15$ kbit/s, $n = 370$) in a cross-VM covert channel respectively.

- We demonstrate that IdleLeak can be used to monitor inter-keystroke timings with an F1 score of 90.5 % and a standard error on the timing prediction of only 12 µs.
- We demonstrate website fingerprinting IdleLeak attacks, with F1 scores of 93.1 % (closed world) and 85.2 % (open world) over the top 100 websites.
- We demonstrate video fingerprinting IdleLeak attacks on two video streaming platforms, with F1 scores of 90.2 % (YouTube) and 75 % (PornHub) over the top 20 videos using only the first 10 s of a video.

Outline Section 2 provides background on SMT, side channels and power states. Section 3 presents the idea behind IdleLeak. Section 4 evaluates native and cross-VM IdleLeak covert channels. Section 5 presents our keystroke timing attacks and Section 6 our website- and video-fingerprinting attacks. Section 7 discusses implications and mitigations. Section 8 discusses related work. Section 9 concludes.

2. Background

In this section, we provide background on SMT and side-channel attacks. Finally, we discuss processor idle states, how the running software can influence them, and how they influence the performance and energy consumption.

2.1. Simultaneous Multithreading (SMT)

Modern CPUs have multiple execution units that execute different instructions simultaneously and out of order. While this speeds up the instruction throughput and can improve the wall-clock performance of workloads, for many workloads, the execution units are idling. Thus, to maximize performance and efficiency, modern CPUs have multiple execution threads (logical cores) that run separate instruction streams from different execution contexts on the same physical core. Intel calls this technique hyperthreading, more generally known as simultaneous multithreading (SMT). With SMT, many microarchitectural elements within the same physical CPU core can be shared (e.g., reorder buffer, load and store

buffer, caches, the TLB) [54]. Similarly, processors achieve substantially higher utilization of execution units, improving performance by roughly 30 % [58], since, even on personal computers, many execution threads are constantly running in parallel. However, for single-threaded workloads, SMT can have a negative performance impact, as on-core resources are statically split or competitively shared, reducing the throughput for the single-threaded workload.

2.2. Side-Channel Attacks

Side-channel attacks on computer systems were first reported in the 1990s [22] and have since significantly influenced the system security area. These attacks obtain meta-data about a secret processed through a (possibly unintentional) channel and derive the secret fully or partially from this meta-data. Historically side-channel research focused on cryptographic primitives [35, 3] as they have a very well-defined threat model with a valuable secret, the secret key, and meta-data that obtainable by an attacker, such as execution time [22, 37], power consumption [24, 23], and EM radiation [39].

One line of side-channel research focused on user input, mainly obtaining inter-keystroke timings but in some cases even precise key press timings for a small set of keys or even single keys [42, 11, 27, 57, 32, 50]. Inter-keystroke timings already contain a significant amount of information as the finger movements across the physical layout of a keyboard influence the inter-keystroke timings in unique ways. While any such timing differences depend on the specific user, previous work showed that written text can still be recovered from them, possibly with an initial learning phase [51, 52, 67, 48], using e.g., machine learning [27, 51, 52].

Another important direction of side-channel research investigates using side channels to establish covert communication channels. In this scenario, the victim and attacker collaborate and form a sender-receiver pair of a communication channel [64, 31, 45]. Covert channels have since been demonstrated on various microarchitectural elements [60, 62, 7], across VMs in the cloud [31], and in browsers [33, 44].

Side-channel research typically evaluates these basic attack targets in various attack scenarios, such as native code attacks [66], browser- or VM-based attacks [33, 38], or even attacks on and from TEEs such as Intel SGX [47, 63, 56, 59]. Each of these scenarios comes with different security

properties and, hence, influences the applicability and impact of a potential side-channel attack. Another important aspect of these scenarios is the relative location of the attacker and victim. In many attacks, the attacker and victim run on the same core or two sibling logical cores [10, 1, 49, 46]. This is not surprising, as many of the targeted microarchitectural elements are not shared across cores, and thus, influences are only visible on the same core or sibling logical cores. Cross-core attacks are possible only for microarchitectural elements shared across the core [66, 30]. Consequently, research in the 2010s focused more on cross-core shared microarchitectural elements [38, 59] and less on private per-core microarchitectural elements. More recently, studies have focused more on these elements again [1, 28, 9].

2.3. Processor Power States (C-States)

C-States are processor power states defined in the Advanced Configuration, and Power Interface (ACPI) [55]. The ACPI defines power states C0 to Cn, where C0 is the running state in which the CPU executes instructions and C1 through Cn are idle states where the processor consumes less power. The time to enter and exit a C-State depends on the depth of the power state (with C1 being light sleep), where deeper idle save more power but have a higher cost to enter and exit.

Older Intel x86 processors only had the privileged `hlt` instruction to put the processor into the C1 idle state. To avoid the expensive system calls to briefly move the processor into an idle state, Intel subsequently introduced the `pause` instruction. The `pause` instruction for several processor generations was the most efficient way for user space programs to implement a busy wait while staying in C0. The *MONITOR* x86 ISA extension introduced the `mwait` instruction which allows waiting for a write to a memory location and provides a simple way to switch to deeper C-states than C1. Recently Intel introduced the *WAITPKG* x86 ISA extension with the Tremont and Alder Lake microarchitectures adding the `umwait` and `tpause` instructions on all privilege levels. While the privileged `mwait` instructions allows entering all C-States, `tpause` and `umwait` are restricted to two sub-states of the C0 running state (C0.1 and C0.2). The `umwait` instruction can be used to monitor a write accesses to a specific memory range. The `tpause` instruction allows to generically wait for a deadline specified in the `EDX:EAX` registers to optimize busy waits. Thus, `tpause` provides an efficient and fast way for user programs to switch to C0.1 and

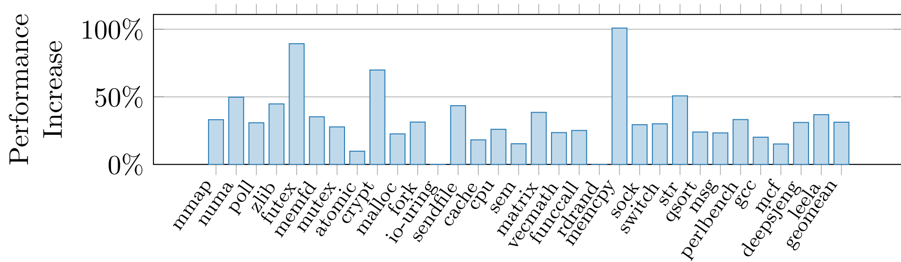


Figure 5.1.: Performance increase in the Phoronix Test Suite and SPEC CPU 2017 on a logical core while the sibling logical core is in the C0.2 idle state. We observe an average increase of 31 %.

C0.2 and is now the most efficient option for short waits [16]. Thereby it reduces the energy consumption and, in the case of C0.2, increases the performance of sibling logical cores. Both of these idle states offer small power savings (5-13% compared to a busy wait) and a fast wake-up time (about 22% increase compared to a busy wait) [4]. Bityutskiy [4] found no significant latency advantage of C0.1 over C0.2, justifying the use of only the C0.2 in recent Linux kernel patches. For longer waiting times, the user can still rely on operating system support. Intel recommends the use of the `tpause` instruction for user-level busy polling, synchronization, and asynchronous I/O, to reduce energy consumption while maintaining a much lower wake-up latency than previously available methods [15].

The operating system can prevent user programs from setting excessively high deadlines by setting the maximum sleep time through the `IA32_UMWAIT_CONTROL` model-specific register. However, besides the time limit, `tpause` is also woken up by non-maskable interrupts, system management interrupts, machine check exceptions, and external interrupts, regardless if interrupts are enabled (`RFLAGS.IF`) [15, 18].

3. Idle State Side Effect Information Leakage

In this section, we present the attack primitives of IdleLeak. We first analyze the side effects of the C0.1 and C0.2 idle states and how they can be exploited. As these new idle states C0.1 and C0.2 are reachable from userspace, their behavior and effects can be observed by an unprivileged

attacker. We then build two attack primitives, ActiveIdleLeak and PassiveIdleLeak. With ActiveIdleLeak, we use side effects of the C0.2 idle state to leak information from a restricted environment to an attacker-controlled environment. With PassiveIdleLeak, we use side effects of the C0.1 and C0.2 idle state to spy on co-located workloads, network activity, user input, and system activity. We also demonstrate both attack primitives in virtual machines and show how they can even be used for information leakage across a VM-host boundary. Our attacks demonstrate the negative security implications of adding user-controlled idle states, as they allow observing various system-level events with a high accuracy, even in practical open-world attacks.

3.1. Performance Side Effects of the C0.2 Idle State

For ActiveIdleLeak, we investigate the side effects of the C0.2 idle state on the system performance, in particular the performance of co-located workloads. While performance gains are documented [4], their security implications are not. Therefore, we analyze the performance effects in more detail using micro-benchmarks and macro-benchmarks to infer how they can be used in side-channel and covert-channel scenarios.

As macro-benchmarks, we use SPEC CPU 2017[1] and Stress-NG from the Phoronix Test Suite. We run the benchmarks on an Intel i7-13900K on one core while the sibling logical core enters idle state C0.2 using `tpause`. We compare the benchmark results to a run with a busy wait on the sibling logical core. All benchmarks except for two show a significant performance increase of 31 % on average when the sibling logical core is in the C0.2 idle state (cf. Figure 5.1). Only two benchmarks from the Phoronix Test Suite show no significant change: `rdrand` and `io-uring`. The reason for these two outliers is that the performance limitations lie outside of the core: For `rdrand`, the corresponding random-number generator module is located outside of the processor core and shared across all cores [7]. For `io-uring`, testing the performance of the `io-uring` asynchronous I/O framework on Linux, the default setting is an interrupt-driven mode [25], *i.e.*, the performance of the test is largely not limited by core-internal resources. The benchmarks with the highest performance gains are primarily single-core compute-bound, e.g., `memcpy` at +100 %, `futex` at +89 %, and `crypt` at +70 %.

[1]We excluded 648.exchange2, 620.omnetpp, 657.xz, 625.x264, and 648.xalancbmk as they did not compile on our test systems.

The performance is also influenced by the frequency of accesses to potentially uncached data, which induces pipeline bubbles and stalls. Hence, benchmarks like `memcpy`, `crypt`, and `str` show a much higher performance gain than benchmarks with a higher cache miss frequency, e.g., `cache` and `qsort`. Similarly, besides `io-uring`, also other benchmarks, e.g., `atomic` and `sem`, may require cross-core operations or the invocation of coherency protocols, leading to a lower performance gain than more compute-bound workloads.

These results already indicate that there are workload-dependent influences that an attacker could exploit. To understand the performance effects of C0.2 on an instruction level, we perform micro-benchmarks with specific operations we will subsequently use in our attacks. To determine the optimal instruction to construct a channel, we select nine candidate x86 instructions that we expect to be mainly influenced by core performance rather than external device or memory latency. Since we focus on the core performance influence, we test all instructions with register operands with randomized inputs.

We measure their change in execution speed when the sibling core is in C0.2 compared to a busy wait. Each instruction is measured 10 000 times with a busy wait on the sibling core and 10 000 times while the sibling core is in idle state C0.2. We repeat the target instruction 8 192 times to ensure that the CPU can take full advantage of the increased space of internal buffers and pipeline elements released by the idle sibling core. For each measurement, the target instruction executions are surrounded by `serialize` instructions, to avoid reordering of the measurement code.

The results of our measurements are shown in Figure 5.2. For fast instructions that take a single cycle to execute, such as `add`, `xor`, and `mov`, we observe a median speedup of roughly 80 %. We observe a speedup of less than 10 % for `pause`, `div`, and `mul`, even though they have no memory operands. This indicates that the throughput limitation for these instructions is not in any pipeline element or buffer shared between the sibling logical cores. The `lea` instruction receives a speedup of ≈40 % and `rdtsc` receives a speedup of ≈20 %. The `nop` instruction receives the most significant performance boost of 90 %. This is not surprising, as `nop` should only be limited by the throughput of the core's frontend and reorder buffer but no execution unit. Consequently, having the full reorder buffer available doubles the throughput. We use `nop` in combination with C0.2 to build a covert channel in Section 4.

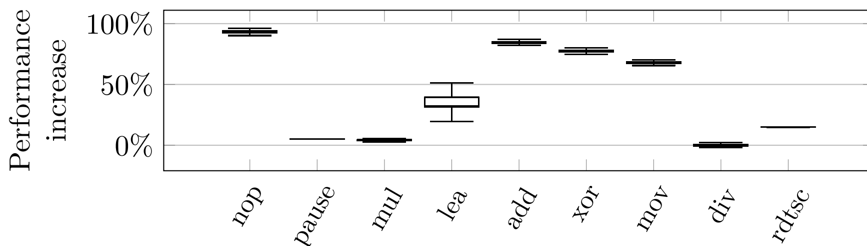


Figure 5.2.: Performance increase of a set of x86 instructions on an Intel i7-1260P, when the sibling logical core is in idle state C0.2 compared to a busy wait. We can see that for some instructions an increase of almost 100 % is possible, whereas other instructions see no performance gains.

3.2. Interrupt Detection

Zhang et al. [68] recently proposed the use of `umwait` for interrupt detection in native code as it is woken up on external interrupts [16]. Furthermore, `umwait` also wakes up when the specified memory is written to. Since we do not require the memory monitoring for any of our attacks, we use `tpause`, as a less noisy alternative to `umwait`. We further show undocumented behavior that allows us to use `tpause` and `umwait` inside a virtual machine to detect interrupts of co-located virtual machines and the host system.

There are three reasons for `tpause` to wake up: First, the instruction continues if the specified deadline is reached. Second, it continues if the the operating system's deadline is exceeded. Finally, it wakes up if an interrupt occurs.

For PassiveIdleLeak interrupt monitoring, we set the `tpause` deadline to the maximum value. This high deadline ensures that we never wake-up due to reaching the deadline.[2] The operating system's deadline, setting the real maximum deadline, cannot be influenced by the user. However, a wake-up caused by the operating system's deadline sets the carry flag. A wake-up due to an interrupt does not set the carry flag. Thus, we can reliably detect interrupts using `tpause`.

[2]A spurious wake-up is possible when the time-stamp counter (TSC) overflows, which will not happen within 10 years of the last reset according to Intel [17].

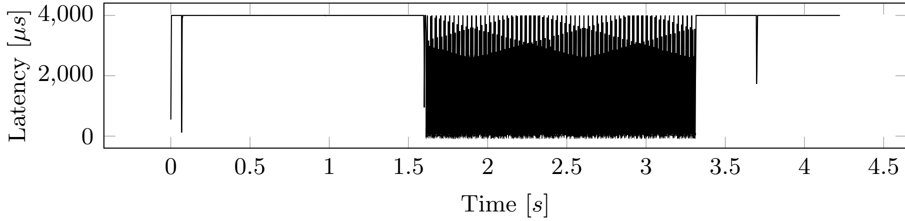
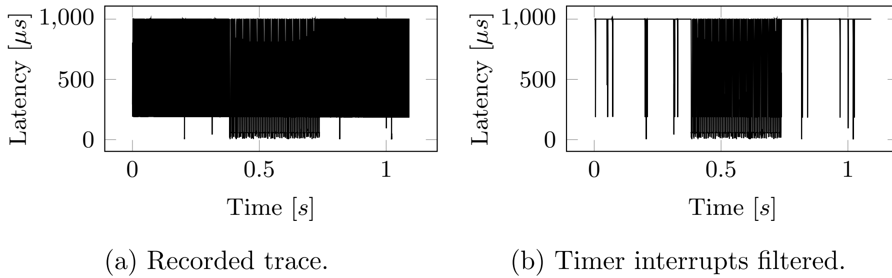


Figure 5.3.: Interrupts detected by a native attacker. The block with lower latencies in the middle, corresponds to a high number of interrupts caused by a touch pad.



(a) Recorded trace.

(b) Timer interrupts filtered.

Figure 5.4.: Touch-pad interrupts recorded by a VM-based attacker are clearly visible (indicated by the low latencies in the middle) when monitoring interruptions with `tpause` and filtering the timer interrupts.

To build our PassiveIdleLeak interrupt detection attack primitive, we repeatedly run `tpause` with the maximum sleep time until a wake-up occurs with the carry flag cleared. This is different to `umwait` which can also spuriously wake up due to memory activity, without revealing this wake-up reason to the user program, *i.e.*, effectively inducing noise into the channel.

Figure 5.3 shows an example of a PassiveIdleLeak interrupt trace recorded with the `tpause` instruction on an i7-1260P running a default-configured Ubuntu 22.04 (Linux 6.1.0). The y-axis represents the observed sleep time. Effectively, this is the time the logical core spent in the idle state. The x-axis shows the wall-clock time. One constant element the interrupt traces contains, is a continuous line, in this case at 4 000 μs, representing regular timer interrupts generated by the operating system for context switches. Any deviation from this line indicates the occurrence of a different interrupt. The interrupt trace in Figure 5.3 consists mainly of timer interrupts at 4 000 μs and touchpad interrupts between 1.6 s and 3.3 s. This trace already

shows the possibility to use PassiveIdleLeak to spy on user input activity. A similar trace recorded in a VM environment is shown in Figure 5.4a. The VM runs a default-configured Debian 11 (Linux 6.2.0), on a default-configured KVM Ubuntu 22.04 host, with guest timer interrupts every 1 000 µs. We observe wake-ups every 4 000 µs indicated by a drop in sleep time, despite the lack of received interrupts inside the guest. We find that these wake-ups are caused by the host's timer interrupts. The interrupts wake up the logical core from the idle state and let the execution return from `tpause` even though these interrupts are intended for the host. The interrupts cause an immediate VM exit and transfer control to the host. However, this still allows guests running in a VM to spy on interrupts intended for the host. We filtered the host timer interrupts by removing the regular interrupts every 4 000 µs shown in Figure 5.4b to highlight the other interrupts. This filtering results in a significantly clearer interrupt trace showing touchpad interrupts intended for the host between 0.4 s and 0.7 s.

For C0.1, we further observe that the CPU not only wakes up on interrupts intended for the waiting core but also on interrupts intended for sibling logical cores even while running inside a VM. On our Ubuntu 22.04 (Linux 6.2.0) we further observed that exceptions, e.g., divide-by-zero, page fault, general protection fault, on a sibling logical core result in a wake up for C0.1. This allows an attacker to use C0.1 to spy on interrupts and exceptions intended for the sibling logical core and, in the case of a cloud environment, on interrupts and exceptions intended for different VMs. Importantly, this behavior does not induce a VM exit on the attacker's logical core but still allows observing the interrupt. This has a significant advantage over the scenario where the interrupt arrives on the attacker core. Since the attacker core does not execute the interrupt service routine, it does not execute a VM exit. Instead, it can re-enter C0.1 immediately after detecting an interrupt and, thus, is immediately ready to detect the next interrupt. Since interrupts do not adhere to security-aware scheduling policies [21] and the x86 architecture implies the assignment of interrupts to a specific cores, it is not trivial to mitigate this issue without significant performance cost, as we discuss in Section 7.

We did not observe the idle-state-interrupting behavior upon sibling-logical-core interrupts in idle state C0.2. Since the C0.1 and C0.2 idle states were both introduced for the user-mode instructions `tpause` and `umwait`, disabling access to C0.1 is not possible without fully disabling access to `tpause` and `umwait`. In Section 5, we evaluate the PassiveIdleLeak

attack primitive in attacks on user input. In Section 6, we show that PassiveIdleLeak can also be used for website and video fingerprinting, with high accuracies despite the victim running on a separate physical core.

4. Covert Channel

We present two high-speed covert channels based on ActiveIdleLeak and on PassiveIdleLeak. The ActiveIdleLeak covert channel is based on the performance effects of the idle state C0.2 on different instructions. The PassiveIdleLeak covert channel is based on the wake-up of sibling logical cores from the idle state C0.1 when an interrupt or exception occurs.

4.1. Covert Channel Design

In this section, we explain the design of our two covert channels. For both the ActiveIdleLeak and the PassiveIdleLeak we use time slicing in combination with a primitive based on the corresponding attack to transmit data.

ActiveIdleLeak Transmission Primitive

In this scenario, we transmit a data stream bit-wise through the ActiveIdleLeak channel. The receiver repeatedly measures the execution time of a series of `nop` instructions as it showed the highest performance change in our micro-benchmark (cf. Section 3). When the sender enters idle state C0.2, the execution time of the `nop` series will go down substantially. When the sender does not enter an idle state, the execution time of the `nop` series will not be affected. Thus, we build the covert channel on top of this timing difference, transmitting ‘0’ and ‘1’ bits.

Figure 5.5 presents the high-level overview of our covert channel. We use one receiver thread and one sender thread, each running on one logical core of the same physical core. In this example, the sender transmits a sequence of ‘0110’. For a ‘1’-bit, the sender performs a busy wait. Consequently, the receiver sees a low performance, *i.e.*, a higher execution time. For the next bit, a ‘0’-bit, the sender enters the C0.2 idle state via `tpause`, increasing the performance of the victim, *i.e.*, lowering the execution time of the `nop` series. The same operation follows for the next bit, another ‘0’-bit.

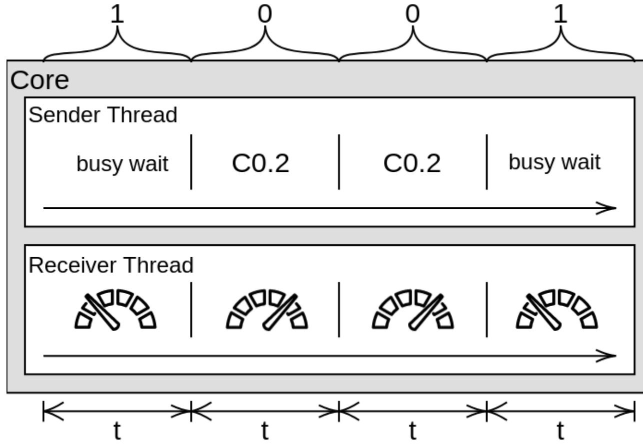


Figure 5.5.: Overview of a ActiveIdleLeak covert-channel transmission where t is the length of a time slice. For each time slice, the sender enters C0.2, which speeds up the receiver to send a '0'-bit or busy waits to send a '1'-bit.

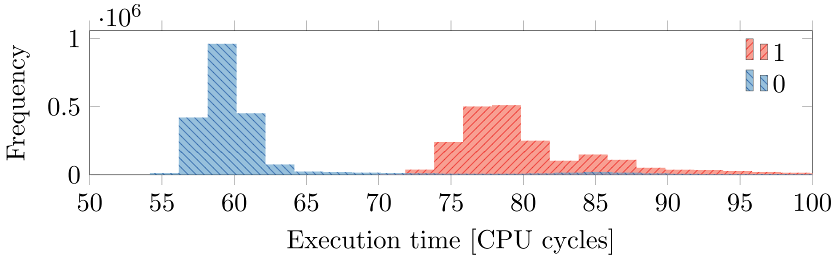


Figure 5.6.: The sender idling in C0.2 via `tpause` is used to encode a '0'-bit, busy-waiting is used to encode a '1'-bit.

For the fourth bit to transmit, a '1'-bit, the sender performs a busy wait, reducing the performance of the victim's `nop` series again. The receiver can now infer the full sequence '1001'.

Figure 5.6 shows the histograms for the two corner cases we use for transmission. Executing 512 `nop` instructions in the receiver takes 82 cycles ($\sigma_{\bar{x}} = 0.01$ cycles, $n = 2\,151\,841$) when the sender performs a busy wait. In contrast, executing 512 `nop` instructions the receiver only takes 60.3 cycles ($\sigma_{\bar{x}} = 0.01$ cycles, $n = 2\,151\,841$) cycles when the sender is in state C0.2. While the timings of the two cases are very clearly separated, a small number of outliers can be observed in the '0'-bit case in Figure 5.6, at 80

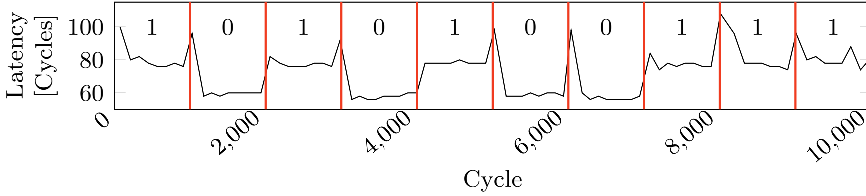


Figure 5.7.: The native ActiveIdleLeak covert channel transmission has a clearly visible difference in the latency between transmitting ‘0’ and ‘1’ bits.

to 90 cycles. These outliers result from the regular wake-ups from C0.2, caused by the operating system's deadline. Thus, for a short time, the sibling core is active before executing the next `tpause` to re-enter idle state C0.2. Since the observed execution time is then in the range of a busy wait, this effect introduces a small amount of noise into our side channel and our covert channel transmission.

PassiveIdleLeak Transmission Primitive

In this scenario, we transmit data bit-wise through the covert channel. We use one receiver and one sender thread, running on the logical cores of one physical core. The receiver measures the interrupt frequency of the sibling logical core with PassiveIdleLeak. The sender causes a wake-up of the receiver by generating exceptions. We use the divide-by-zero exception, as it can be triggered without a memory access. When the sender does not trigger exceptions, the receiver thread will detect fewer interrupts. Thus, we build the covert channel on top of this behavioral difference, transmitting ‘0’ and ‘1’ bits.

Synchronization

We synchronize our covert channels via the TSC. The transmission starts on a previously agreed-on TSC value and sends the bits in time slices of predefined length. For each time slice with our ActiveIdleLeak covert channel, the sender repeatedly executes `tpause` to stay in C0.2 until the end of the time slice to transmit a ‘0’ or runs a busy wait to send a ‘1’. To compensate for noise, the receiver averages the execution times of all `nop` instruction sequences within a time slice at the end and determines the

bit through a threshold. An example transmission with a time slice length of 1 000 cycles is shown in Figure 5.7. The time slices are indicated with vertical red lines. The ground truth is printed at the top of each time slice. The receiver's latency when receiving a '1' is around 80 cycles and for a '0' at 60 cycles. We can observe latency spikes of 100 cycles at the beginning of each time slice. These spikes are due to the sender determining which bit value should be sent in the upcoming time slice.

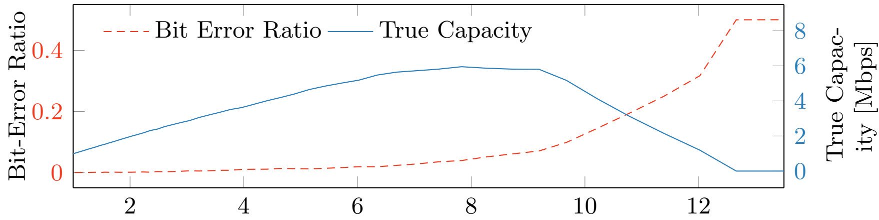
For each time slice with our PassiveIdleLeak covert channel, the sender generates exceptions until the end of the time slice to transmit a '1' or runs a busy wait to send a '0'. The transmission starts with 16 '1'-bits for initialization. The receiver counts the number of interrupts detected with PassiveIdleLeak for the first 16 time slices, averages them, and divides them by 2 to compute a threshold value. For the following time slices, the sender transmits the data. For each time slice after initialization, the receiver counts the number of interrupts and compares them with the threshold value. If the number of interrupts is above or equal to the threshold value, a '1'-bit was transmitted. If the number of interrupts is lower than the threshold, a '0'-bit was transmitted.

Relying on the TSC for synchronization has additional benefits for our IdleLeak covert channels. The wake-up deadline of `tpause` is specified by a TSC value at which the processor should leave the idle state again. Consequently, we inherently re-synchronize our covert channel through the used instruction and eliminate the need for additional synchronization logic.

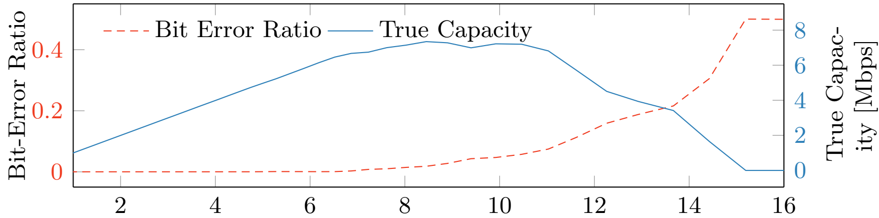
4.2. Evaluation

We evaluate both covert channels by sending data from `/dev/urandom` on an i9-13900K. We additionally evaluate our ActiveIdleLeak covert channel on an i7-1260P. We assume attacker and victim run in separate processes and that they can be scheduled on sibling logical cores of the same physical core. We assume they run native userspace code. Furthermore, there is no legitimate communication channel between the processes and no bugs could be exploited for communication.

For our initial tests, to obtain the optimal configuration parameters, we run the system idle without interfering workloads. Later on we evaluate the influence of different types of noise for the native ActiveIdleLeak channel. We evaluate different time slice lengths and record the raw capacity and



(a) Alder Lake Core i7-1260P



(b) Raptor Lake Core i9-13900K

Figure 5.8.: The raw capacity of our native ActiveIdleLeak channel, and the corresponding bit-error ratio and true capacity. We can see that the optimal true capacity is reached between 5 and 10 Mbit/s of raw capacity.

bit-error ratio of the channel. Since our channels are based on time slices, the raw capacity is inversely linear in the time slice length. Thus, shorter time slices result in a higher transmission rate but due to the shorter time slices, more bit errors may occur. Consequently, while the raw bitrate increases with a time slice reduction, the actual channel capacity might decrease due to a higher bit-error ratio. To find the optimal transmission rate, we compute the true capacity based on the raw bitrate and the bit-error ratio.[3]

Native ActiveIdleLeak Covert Channel

Figure 5.8 shows the true channel capacity and bit-error ratio as a function of the raw capacity on our test systems. On the i9-13900K, shown in

[3] 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.

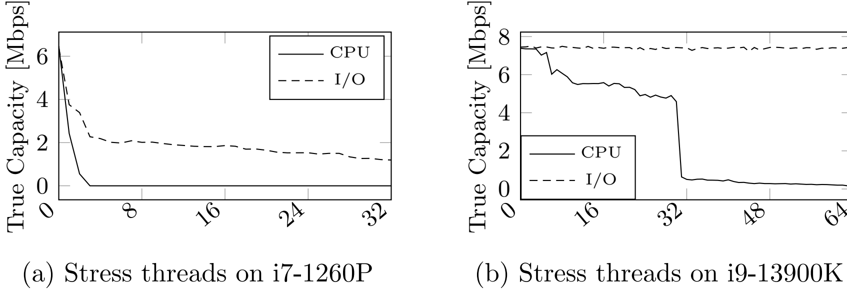


Figure 5.9.: Effect of I/O and CPU-related workloads simulated with the `stress` utility on the covert channel's true capacity. On the i9-13900K, the capacity reaches ~0 bit/s with more than 30 stress threads. I/O heavy workloads do not have a significant influence on the capacity. On the i7-1260P, capacity drops to ~0 bit/s with 4 CPU stress threads. I/O workloads decrease to 2 Mbit/s with 4 stress threads decreasing slowly with more threads.

Figure 5.8b, the bit-error ratio stays below 1 % up to a raw capacity of 7.2 Mbit/s. The true capacity then peaks at a raw capacity of 8.9 Mbit/s with a bit-error ratio of 3.2 % ($\sigma_{\bar{x}} = 0.01\,\%$, $n = 512$), corresponding to a true capacity of 7.1 Mbit/s ($\sigma_{\bar{x}} = 0.004\,\text{Mbit/s}$, $n = 512$). On the i7-1260P, shown in Figure 5.8a, the bit-error ratio is slightly higher. The true capacity here peaks at a raw capacity of 7.8 Mbit/s with a bit-error ratio of 4.4 % ($\sigma_{\bar{x}} = 0.18\,\%$, $n = 512$), corresponding to a true capacity of 5.9 Mbit/s ($\sigma_{\bar{x}} = 0.05\,\text{Mbit/s}$, $n = 512$).

To determine the effect of system noise on the covert channel transmission rate, we run our channel while other workloads run on the CPU. We use the optimal parameters for each CPU according to Figure 5.8. To simulate the system noise, we use the `stress` tool to run I/O and CPU workloads with different numbers of threads. For both of our tested CPUs (i7-1260P and i9-13900K), we run up to double the number of stress threads as there are logical cores for both I/O and CPU workloads. The results of our evaluation are shown in Figure 5.9. On the i9-13900K, I/O workloads do not significantly impact the covert channel, with the true capacity staying at 7.1 Mbit/s for all scenarios. With CPU workloads, the capacity decreases to 6 bit/s with 8 stress threads and to almost 0 Mbit/s with more than 30 stress threads. Together with our 2 threads taking part in the transmission, this results in 32 threads before the transmission breaks down, exactly the number of logical CPU cores. On the i7-1260P, I/O workloads have

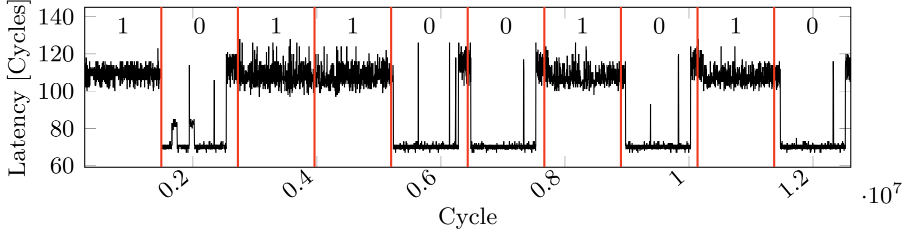


Figure 5.10.: The cross-VM covert channel transmission is more noisy but still has a clear difference in the latency between '0' and '1' bits.

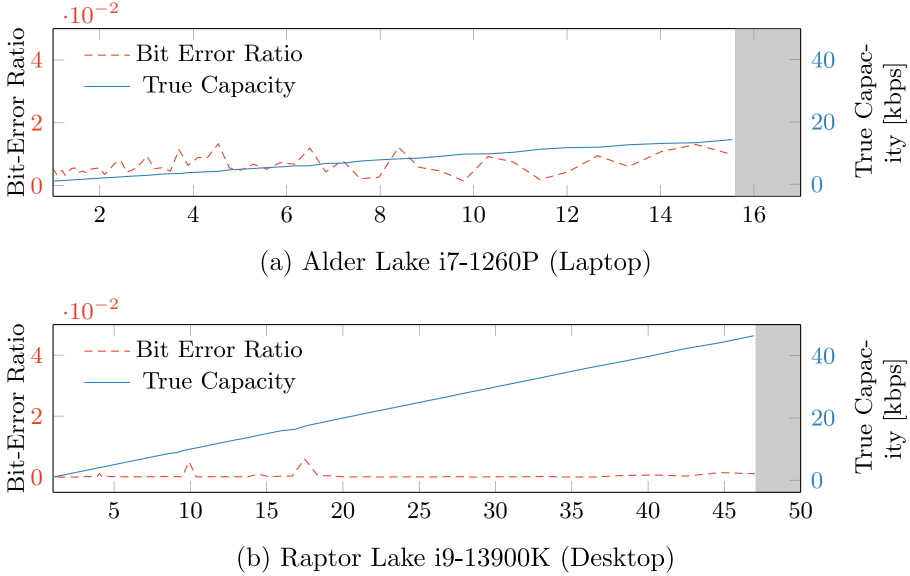
a significant effect on the transmission rate, dropping it from 6 Mbit/s to 2 Mbit/s with only 4 stress threads, decreasing slowly to ~1.8 Mbit/s with 32 threads after that. For CPU workloads, the covert channel drops to almost 0 bit/s with only 4 stress threads which are significantly less than the 16 logical cores the CPU has. During testing, we observed that the i7-1260P reached its thermal limits of 100 °C, causing it to thermal throttle. When fixing the fan speed to its maximum to avoid thermal throttling, we still observed high core frequency fluctuations throughout the experiment with an increasing number of stress threads. We presume these fluctuations to be the result of lower power limits and the stricter efficiency requirements of laptop CPUs.

Cross-VM ActiveIdleLeak Covert Channel

Our cross-VM covert channel evaluation is similar to our native code evaluation. Additionally to the separate processes for attacker and victim, we assume attacker and victim run in separate VMs on the same physical machine. Both VMs run a recent Debian 11 with Linux kernel 6.2.0 and can be scheduled on sibling logical cores. Attacker and victim have no means of communication besides the covert channel.

Figure 5.10 shows an example transmission of our cross-VM channel. As we are running in VMs, there is no shared time-stamp counter (TSC). We resolved this challenge with an initialization sequence of 5 alternations between '1' and '0', unlikely to be received through random noise. Random data follows after this sequence similar to Section 4.2.

Figure 5.11 shows the true channel capacity and bit-error ratio as a function of the raw capacity on our test systems. On the i9-13900K, shown



(a) Alder Lake i7-1260P (Laptop)

(b) Raptor Lake i9-13900K (Desktop)

Figure 5.11.: The raw capacity (x-axis in kbps) of our cross-VM channel, and the corresponding bit-error ratio and true capacity. The optimal true capacity is reached at 46.9 kbit/s of raw capacity for the i9-13900K and at 15.5 kbit/s of raw capacity for the i7-1260P. At a higher raw capacity the synchronization mechanism fails leading to no data transmitted, marked by the gray area.

in Figure 5.11b, the bit-error ratio stays below 0.3 % up to a raw capacity of 46.9 kbit/s. The true capacity then peaks at a raw capacity of 46.9 kbit/s with a bit-error ratio of 0.22 % ($\sigma_{\bar{x}} = 0.07\,\%$, $n = 370$), corresponding to a true capacity of 46.32 kbit/s ($\sigma_{\bar{x}} = 0.15$ kbit/s, $n = 370$). On the i7-1260P, shown in Figure 5.11a, the bit-error ratio is slightly higher. The true capacity here peaks at a raw capacity of 15.5 kbit/s with a bit-error ratio of 2.1 % ($\sigma_{\bar{x}} = 0.39\,\%$, $n = 60$), corresponding to a true capacity of 13.57 kbit/s ($\sigma_{\bar{x}} = 0.27$ kbit/s, $n = 60$). We have no data for higher raw capacities for either CPU as the synchronization mechanism fails.

Native PassiveIdleLeak Covert Channel

Figure 5.12 shows transmission rate and bit-error ratio compared to the raw capacity of the PassiveIdleLeak channel. The true capacity peaks at 656.37 kbit/s ($\sigma_{\bar{x}} = 0.63$ kbit/s, $n = 1\,024$) with a bit-error ratio of 9.22 % ($\sigma_{\bar{x}} = 0.02\,\%$, $n = 1\,024$) at 1 179 kbit/s of raw capacity. After the true

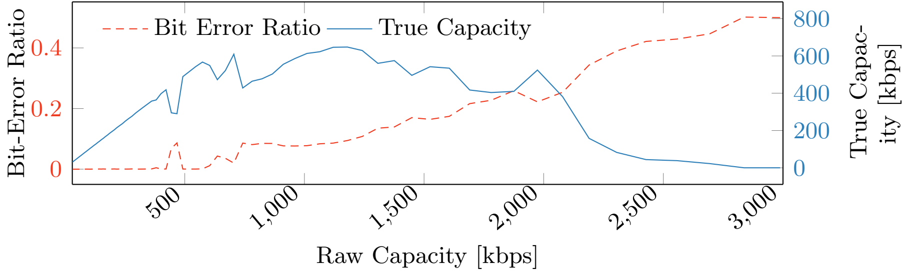


Figure 5.12.: The raw capacity of PassiveIdleLeak, and the corresponding bit-error ratio and true capacity on an i9-13900K. The optimal true capacity of 656.37 kbit/s ($\sigma_{\bar{x}} = 0.63$ kbit/s, $n = 1\,024$) is reached with a bit-error ratio of 9.22 % ($\sigma_{\bar{x}} = 0.02$ %, $n = 1\,024$) at 1 179 kbit/s of raw capacity.

capacity peak, the time slice length is smaller than the time to trigger an exception and recover from it. We conclude that PassiveIdleLeak can be used to leak data at a high transfer rate from sibling logical cores.

Previous Work

Both our ActiveIdleLeak (7.1 Mbit/s) and PassiveIdleLeak (656 kbit/s) covert channels achieve comparable or faster transmission rates than previous work. Zhang et al. [68] built a covert channel based on detecting speculative writes with `umwait` and achieved a transmission rate of 200 kbit/s. Gast et al. [9] exploit scheduler contention to leak and transmit 2.7 Mbit/s. Saileshwar et al. [45] use cache contention and achieve a transmission rate of 14.4 Mbit/s.

5. Keystroke Detection

In this section, we present our inter-keystroke timing attack. We exploit that keystrokes of USB keyboards generate interrupts and detect them with PassiveIdleLeak.

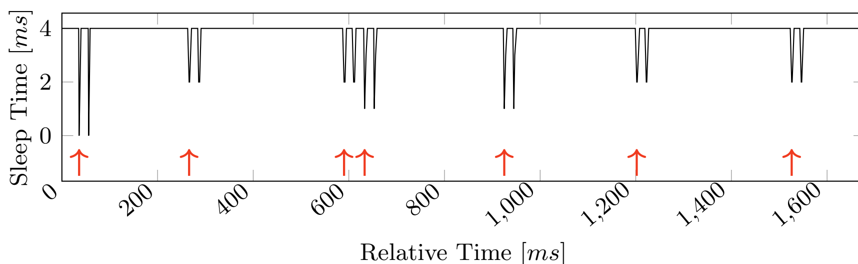


Figure 5.13.: Keystroke detection using PassiveIdleLeak on an i7-1360P. The downward spikes show the interrupts caused by key-down and key-up events, *i.e.*, where keystrokes are detected. The red arrows show the ground-truth.

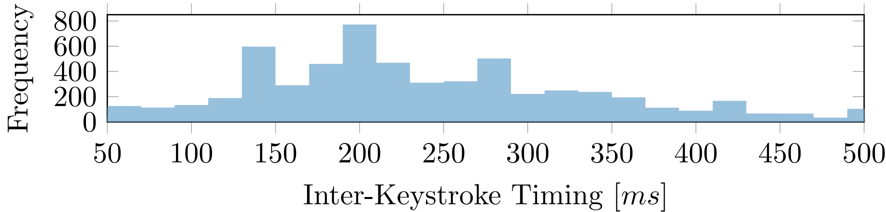
5.1. Threat Model

We assume the attacker has access to millisecond-accurate timers, e.g., `clock_gettime` or C++ standard clocks. We make no assumptions on the availability of high-resolution timers such as the TSC, as they can be manipulated. Linux assigns each external interrupt to one of the available cores. Interrupt-core assignments rarely switch between cores at runtime, as can be observed via `/proc/interrupts`. We assume the attacker can start multiple threads on different cores. Thus, we can assume that the attacker is eventually scheduled on the core that receives the keyboard device interrupts. We make no assumption on the core the victim code receiving the keystrokes is running on, as this is independent of the core that receives the keyboard device interrupts.

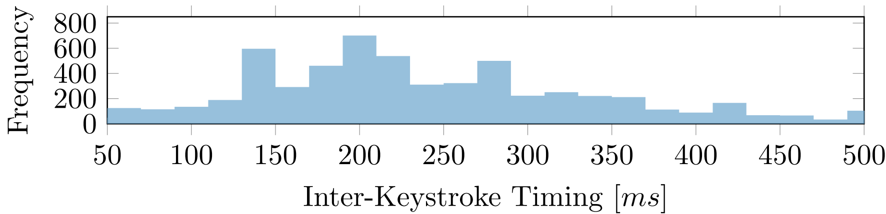
5.2. Attack Implementation

For our inter-keystroke timing attack, the attacker first records an interrupt trace using PassiveIdleLeak on the core that receives the USB interrupts. USB devices generate interrupts when sending data to the host system. A USB keyboard sends two types of interrupts for a single keypress: one for the key press (key down) and one for the key release (key up). Once recorded, the attacker analyzes the data and infers the precise inter-keystroke timings.

Figure 5.13 shows an interrupt trace recorded with PassiveIdleLeak. The key-down interrupts are marked with red arrows and result in a significant



(a) Distribution of inter-keystroke timings in the ground-truth.



(b) Distribution of inter-keystroke timings recovered by PassiveIdleLeak on an i7-1260P.

Figure 5.14.: The distributions of inter-keystroke timings and of the ground-truth. The recovered and ground-truth inter-keystroke timings are very similar.

dip in sleep time. After each key-down interrupt, a key-up interrupt occurs. The time between two interrupts depends on the writing speed but is usually in the tens of milliseconds [20]. We use the key-up interrupt to distinguish keystroke interrupts from other interrupts that occur on the same core. We exclude unlikely interrupt pairs that are too low (<10 ms) or too high (>100 ms). Thus, for every interrupt in our trace, we search for a possible key-up interrupt that is at least 10 ms and at most 100 ms after the possible keystroke. We make no assumptions on overlapping key presses. If such an interrupt exists, we log the first interrupt as the key down event, remove the interrupt pair from the trace and continue with the next interrupt.

5.3. Evaluation

To evaluate PassiveIdleLeak, we use an ARM Mbed LPC-1768 μ-controller board. This controller board acts as a USB keyboard device that we plug

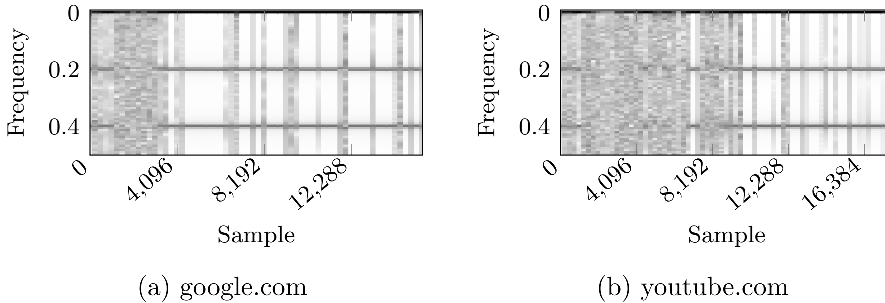
into our test machines to inject hardware keyboard interrupts. We use pre-recorded inter-keystroke timing data [29] covering a hundred participants typing an eight-letter word ten times. We use these 7 000 inter-keystroke timings and replay them using the μ-controller with a high-precision clock. Thus, we have highly accurate ground-truth data for the actual keystroke interrupts.

We evaluate our native-code inter-keystroke timing attack on an i7-1260P on Ubuntu 22.04 (Linux 6.1.0). We schedule the attacker on one of the two logical cores of the physical core that receives the keyboard interrupts. Our keystroke detection has a precision of 87.1 %, a recall of 94.1 %, and an F1 score of 90.5 %. For the timing, we measured and statically subtracted the average deviation, which was 15.2 µs, effectively minimizing the average deviation to 0. The detected and reference inter-keystroke timings of the correctly detected keystrokes are shown in Figure 5.14. The reference distribution (Figure 5.14a) and the detected distribution (Figure 5.14b) are almost identical. We achieve a standard deviation of 950 µs and a standard error of 12 µs for our correctly detected keystrokes. Therefore, we conclude that PassiveIdleLeak can accurately monitor interrupt-based singular events like keystrokes.

6. Website and Video Fingerprinting

In this section, we present our website and video fingerprinting attacks using PassiveIdleLeak. We show that it is possible to determine the website a user accesses in a closed-world setting over the top 100 websites from the Alexa top 1 million list [2] and an open-world setting with the top 100 websites and an `other`-class for websites not in the top 100. Furthermore, we demonstrate an open-world and a closed-world video-fingerprinting attack on two popular video-streaming websites, YouTube and PornHub, to distinguish between the top 20 trending videos in the US at the time of writing for YouTube and between the most viewed videos of the 20 most popular categories on PornHub. In this scenario, we assume an attacker runs code in a VM on the same machine as the victim.

Network devices generate interrupts when sending or receiving data. When accessing a website or video streams, the computer sends and receives numerous network packages. The number of packages depends on the content, while the time between packages can, among other things, depend on the content, server location, and server software, resulting in unique



(a) google.com

(b) youtube.com

Figure 5.15.: The STFTs of the interrupt traces of different websites show distinct patterns, here with the examples (a) google.com and (b) youtube.com. Both traces were acquired in a VM-based attack, attacking a browser running on the host machine. (ploted values are amplified for readability)

interrupt patterns for most content. We use PassiveIdleLeak to infer the exact website accessed or video stream watched by the user, based on a convolutional neural network (CNN) we train to classify the interrupt patterns.

6.1. Threat Model and Attack Setup

We run our measurements on an i7-1260P CPU with Mozilla Firefox 113.0.2 running Ubuntu 22.04. We assume that a user wants to run untrusted code in a secure way and, hence, runs it in a VM. While running the code in a VM, the user browses the web. We assume the attacker has access to millisecond-accurate timers in the VM, e.g., `clock_gettime` or C++ standard clocks. We do not use and do not make assumptions about the availability of high-resolution timers, such as the timestamp counter, as they can be manipulated by the host. Linux assigns each external interrupt to one of the available cores. Typically, these assignments do not switch at runtime, as can be observed via `/proc/interrupts`. On Linux with KVM, VM threads are scheduled like other threads, irrespective of interrupt routing. Thus, it is reasonable to assume that the attacker VM is eventually scheduled on the core that receives the host's network device interrupts or on a sibling logical core. We do not make any assumption on the core the web browser is running on, as this is independent of the core that receives the network device interrupts.

6.2. Attack

Website Fingerprinting

Our attack consists of an online data-collection phase and an offline phase for processing and evaluation of the traces. The online phase consists of a user space program running inside a VM continuously running PassiveIdle-Leak on the CPU core that receives the network device interrupts of the host as described in Section 3.2. To measure the execution time, we use an accurate millisecond timer, as we only need to distinguish guest timer interrupts from other interrupts. In the offline phase of our attack, we analyze the collected traces. Since we previously determined that a timer interrupt in our VM setup takes $\geq$1 ms, as shown in Figure 5.4b, a sleep time of $\geq$1 ms means only timer interrupts occurred and an execution time of $<$1 ms means a different interrupt must have occurred.

In the next step, we search for the time frame in which the website access occurred. When there is no network traffic, the core rarely receives interrupts except for the regular guest timer interrupts, and the regular host timer interrupts. This results in a change in interrupt frequency whenever a website access occurs. We compute the short-time Fourier transform (STFT) of the interrupt trace with a window size of 256 to analyze the frequency change. The STFT of an access to `google.com` and an access to `youtube.com` are shown in Figure 5.15. The x-axis of the plots shows the sample number, and the y-axis shows the change in frequency. Since the time between samples can vary depending on the number of interrupts that occur in a given time frame and we do not require an exact sampling frequency for further processing, we do not have a unit of measurement for our frequency scale. Due to this lack of a consistent sampling frequency, we directly refer to the frequency value returned by the STFT without any unit of measurement. In case of no network traffic and a lack of other interrupts, the 0.2, 0.4, and 0 frequency components are high, while all other components are almost 0. An example of almost no network traffic is shown in Figure 5.15b in the last fourth of the trace since most of the website is already loaded. A website access starts if the 0.2, 0.4, and 0 frequency components decrease over multiple STFT windows. Following the detected website access, we use the following 512 STFT windows and forward them to our classifier. Examples can be seen between samples 0 and 4 096 in Figure 5.15a and between samples 1 024 and 10 240 in Figure 5.15b. While some of the tested websites take longer to load than the 512 STFT windows we use, we determined through

manual testing that the used time frame is enough to uniquely identify a website.

After this pre-filtering step, we forward the 512 STFT windows to our convolutional neural network (CNN) for classification. We use the STFT instead of the interrupt trace since server response times can change slightly with each website access, shifting interrupt timings. The resulting shift in features in the time domain results in slightly different traces on every website access. Applying an STFT to the interrupt trace allows for efficient convolutions on the input rather than relying on less efficient, fully connected layers. Additionally, compared to a Fast Fourier Transform (FFT) over the whole trace, the STFT preserves part of the time domain by applying a Fourier transform on separate slices of the trace instead of the whole trace. This technique is well established in the field of signal classification [65, 6, 14]. Our CNN consists of 4 convolutional layers followed by 3 fully connected layers and outputs a match probability for each of the 100 websites.

Video Fingerprinting

For our video-stream fingerprinting attack, we measure the interrupt trace of the first 10 seconds of a video. Contrary to our website fingerprinting, we do not apply the STFT directly on the interrupt trace. As the interrupt trace of a video stream typically consists of a low number of interrupts, mainly from the timer interrupt, with short bursts of a high number of interrupts, an STFT directly on the interrupt trace becomes inefficient. Instead, for each millisecond, we count the number of interrupts that occurred, resulting in the number of interrupts per millisecond. We then perform an STFT on this transformed interrupt trace and feed the result into a CNN similar to our website fingerprinting attack. Our CNN for video-stream fingerprinting consists of 4 convolutional layers followed by 3 fully connected layers and outputs a match probability for each of the 20 videos.

6.3. Evaluation

We evaluate our open-world website fingerprinting, closed-world website fingerprinting, and video fingerprinting attacks with the attacker VM on the sibling logical core of the core that receives the network interrupts. We

additionally evaluate our closed-world website fingerprinting attack with the attacker VM on the core that receives the network interrupts. For each closed-world website fingerprinting scenario, we collected 5 000 traces (50 per website). For our open-world scenario, we collected 20 000 traces (200 per website) for each of the top 100 websites and 7 000 traces from 7 000 further websites from the Alexa 1 million list [2] (1 trace per website). For our closed-world video fingerprinting scenario on YouTube, we collected 2 900 10 s traces (145 per video) of the top 20 trending videos in the US at the time of writing; for PornHub, we collected 3 300 10 s traces (165 per video) of the most viewed videos in the top 10 default and gay categories. For our open-world video fingerprinting scenario on YouTube, we collected 1 500 10 s traces (75 per video) of the top 20 trending videos in the US at the time of writing; for PornHub, we collected 1 500 10 s traces (75 per video) of the most viewed videos in the top 10 default and gay categories. To represent the `other`-class we collected 1 000 traces of 1 000 random videos (1 trace per video) for both YouTube and PornHub.

For all attacks, we split the collected traces randomly into a test set (20 %) and a training set (80 %). Our CNN was trained with a validation split of 10 % of the training set.

Closed-World Same-Core Interrupt Website Fingerprinting

In this scenario, the attacker runs on the core that receives the network device interrupts. We tested our classifier on the test set which is not used for training and achieved an F1 score of 88.2 %. The full confusion matrix is shown in Figure 5.16. Each cell indicates the probability that our classifier labels the access to a website indicated by the row as a particular website indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no website being labeled correctly for less than 30 % of the time, which is significantly higher than random guessing at 1 %. The websites with the worst accuracies are `twimg.com` (10 %), `google.com` (40 %), `google.co.in` (40 %), and `google.com.hk` (40 %). The `twimg.com` domain is used by twitter for images and videos and only serves a page on subdomains but not under the direct URL `twimg.com`, at the time of testing, resulting in the low score. All other websites have accuracies of at least 50 %. Websites that our classifier frequently confuses with each other include `google.com`, `google.com.hk`, and `google.co.in` since all three domains forward the

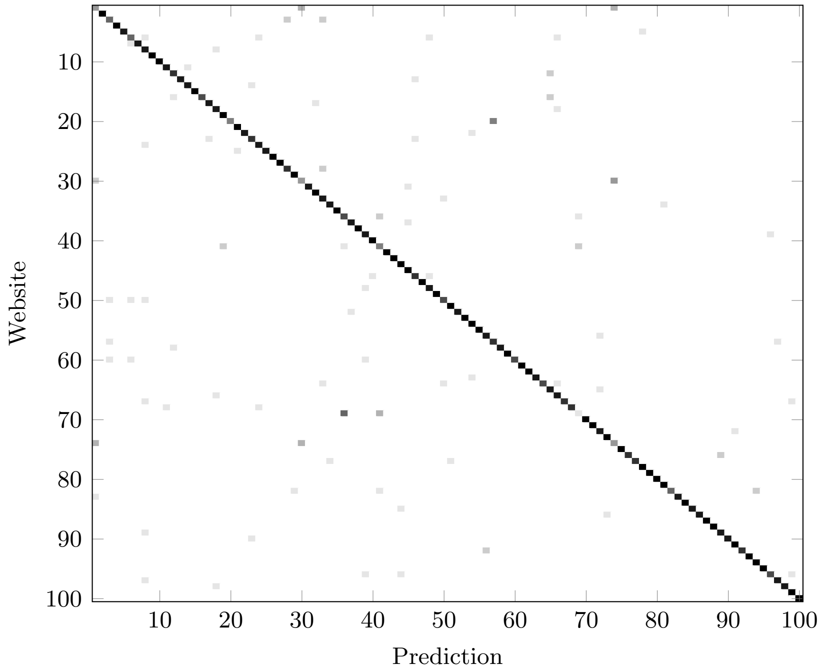


Figure 5.16.: The confusion matrix for our website-fingerprinting attack, with network interrupts arriving on the attacker's core. For the classification, 10 out of 50 samples serve as the test set, with the remainder as the training set, for each of the websites.

browser to the same server. Grouping all Google domains into one class results in an overall F1 score increase from 88.2 % to 90 % for our model.

Closed-World Sibling-Logical-Core Interrupt Website Fingerprinting

In this scenario, the attacker runs on a sibling logical core of the core that receives the network device interrupts. We tested our classifier on a test set which is not used for training and achieved an F1 score of 92.4 %. The full confusion matrix is shown in Figure 5.17. Each cell indicates the probability that our classifier labels the access to a website indicated by the row as a particular website indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no website labeled correctly for less than 20 % of the time, which is significantly higher than random guessing at 1 %. The

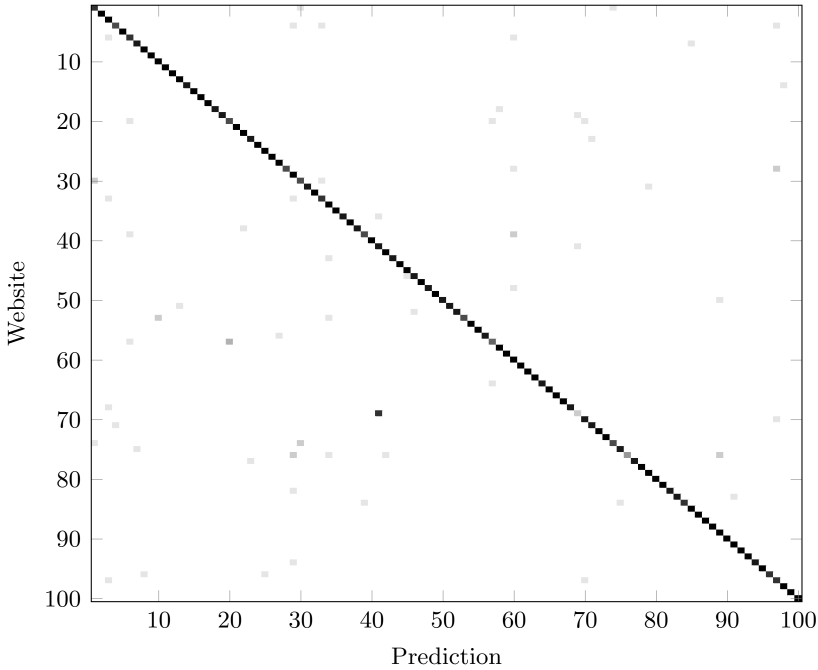


Figure 5.17.: The confusion matrix for our website-fingerprinting attack, with network interrupts arriving on a sibling logical core. For the classification, 10 out of 50 samples serving as the test set, with the remainder as the training set, for each of the websites.

websites with the worst accuracies are `twimg.com` (20 %, we described this issue in Section 6.3), `dzen.ru` (40 %), and `tencent.com` (60 %). All other websites have accuracies ≥70 %. Grouping all Google domains into one class results in an F1 score increase from 92.4 % to 93.1 % for our model.

Since PassiveIdleLeak has to use C0.1 to detect interrupts of the sibling logical core, the attacker detects all interrupts from both logical cores that are part of the physical core the attacker is running on. Despite the added noise from the higher interrupt frequency compared to Section 6.3, this scenario performs significantly better, with an F1 score of 93.1 %. The increase in F1 score is the result of the higher accuracy PassiveIdleLeak has for detecting interrupts of sibling logical cores as they do not require the attacker logical core to execute interrupt service routines and do not result in VM exits for the attacker. Thus, the attacker is immediately ready to detect the next interrupt, decreasing the number of missed interrupts significantly. Despite the significant advantage of this approach, the added

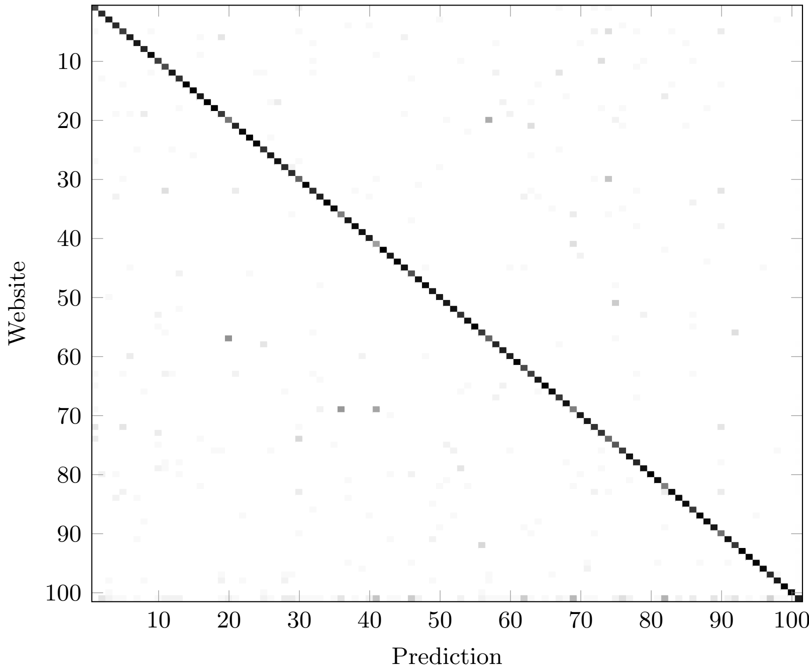
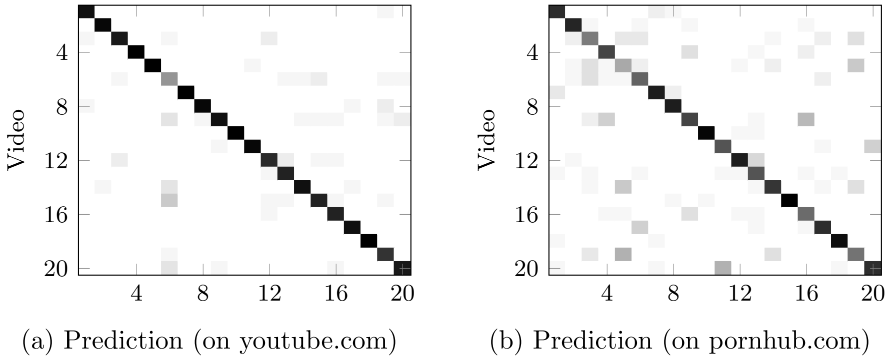


Figure 5.18.: The confusion matrix for our open-world website-fingerprinting attack, with network interrupts arriving on a sibling logical core.

noise from the interrupts on the attacker core can negate this advantage if a device generates a high frequency and number of interrupts, resulting in a lot of noise. In such a high noise scenario, PassiveIdleLeak on the same core as the target interrupts using C0.2 (as evaluated in Section 6.3) performs significantly better.

Open-World Website Fingerprinting

In this scenario, we add an `other`-class for websites not part of the top 100 from the Alexa top 1 million list [2] with the attacker running on a sibling logical core of the core that receives the network interrupts. For the training of the `other`-class, we use accesses to 5 600 websites from the top 1 million list. For testing of the `other`-class, we use accesses to 1 400 websites from the top 1 million list that are **not** in the training set. As the `other`-class test set websites have never been seen by our classifier during training, they result in a realistic accuracy measurement of our classifier on unknown websites.



(a) Prediction (on youtube.com)

(b) Prediction (on pornhub.com)

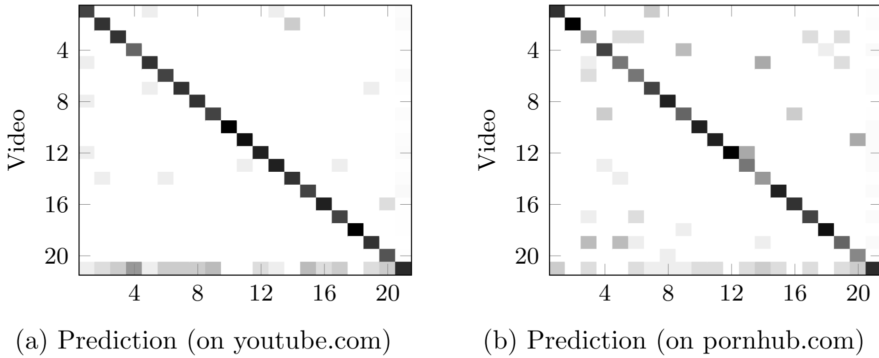
Figure 5.19.: The confusion matrices for our video-stream fingerprinting attack with 20 videos, with network interrupts arriving on a sibling logical core performed on youtube.com and pornhub.com. For the classification, we use a 20 % test split, with the remainder as the training set, for each video.

Our classifier achieved a macro-averaged F1 score of 85.2 % on the test set. The `other`-class, in particular, has an accuracy of 87.4 % on the 1 400 test traces. The full confusion matrix is shown in Figure 5.18. Each cell indicates the probability that our classifier labels the access to a website indicated by the row as a particular website indicated by the column. The last column and row correspond to the `other`-class. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no website being labeled correctly for less than 37.5 % of the time, which is significantly higher than random guessing at 1 %. The websites with the worst accuracies are `microsoftonline.com` (37.5 %), as direct access to this domain does not resolve to an IP address, `t.co` (50 %), `twimg.com` (50 %) and `jianshu.com` (50 %). All other websites have accuracies of over 50 %.

Video-Stream Fingerprinting

In this scenario, the attacker fingerprints video streams using PassiveIdleLeak. The attacker runs on a sibling logical core of the core that receives the network device interrupts.

For YouTube, our classifiers achieved macro-averaged F1 scores of 90.2 % (closed-world) and 81.5 % (open-world) on test sets which are not used for training. The full confusion matrix for the closed-world scenario is shown



(a) Prediction (on youtube.com)

(b) Prediction (on pornhub.com)

Figure 5.20.: The confusion matrices for our open-world video-stream fingerprinting attack with 20 videos and a seperate class for other videos, with network interrupts arriving on a sibling logical core performed on youtube.com and pornhub.com. For the classification, we use a 20 % test split, with the remainder as the training set, for each video.

in Figure 5.19a. Each cell indicates the probability that our classifier labels streaming a video indicated by the row as a particular video indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 41.4 % of the time, which is significantly higher than random guessing at 5 %. The videos with the worst accuracies are `Video 6` (41.4 %), and `Video 19` (79.3 %). All other videos have accuracies of over 80 %. The full confusion matrix for the open-world scenario is shown in Figure 5.20a. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 60 % of the time, which is significantly higher than random guessing at 4.8 %. The videos with the worst accuracies are `Video 4` (60 %), and `Video 20` (66.7 %). All other videos have accuracies of over 70 %. The `other`-class in particular has a test accuracy of 83 % on videos never seen during the training phase.

For PornHub, our classifiers achieved macro-averaged F1 score of 75 % (closed-world) and 70.5 % (open-world) on test sets which are not used for training. The full confusion matrix for the closed-world scenario is shown in Figure 5.19b. Each cell indicates the probability that our classifier labels streaming a video indicated by the row as a particular video indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 33.3 % of the time, which is significantly higher than random guessing

at 5 %. The videos with the worst accuracies are `Video 5` (33.3 %), `Video 3` (51.5 %), `Video 19` (54.5 %), and `Video 16` (57.8 %). All other videos have accuracies of over 60 %. The full confusion matrix for the open-world scenario is shown in Figure 5.20b. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 33.3 % of the time, which is significantly higher than random guessing at 4.8 %. The videos with the worst accuracies are `Video 3` (33.3 %), `Video 14` (40 %), and `Video 20` (46.7 %). All other videos have accuracies of over 53 %. The `other`-class in particular has a test accuracy of 82 % on videos never seen during the training phase.

The F1 scores for PornHub (75 % closed world, 70.5 % open world) are significantly lower than for YouTube (90.2 % closed world, 81.5 % open world), mainly for two reasons: First, Pornhub has a lower default video resolution of 720p (in some cases even 480p) on our system, whereas YouTube has 1080p resulting in fewer network packages within the same time frame. Second, platform- or company-specific intros at the beginning interfere with our fingerprinting. Both issues can be addressed by using more than the first 10 s for classification.

6.4. Previous Work

Our website fingerprinting attack achieves an F1 score of 93.1 % in a closed-world and 85.2 % in an open-world fingerprinting scenario, over the top 100 websites, which is on par and in most cases even better than previous work. Spreitzer et al. [53] achieved an accuracy of 89 % on 100 websites using the data-usage statistics on Android. Jana et al. [19] exploited the memory usage statistics of browsers and reported an accuracy between 30 % and 50 % for the top 100 000 websites. Gulmezoglu et al. [13] used hardware performance events and achieved accuracy of 86.3 % on 40 websites. Zhang et al. [68] performed website fingerprinting on the top 100 pages of the Alexa Top 1M list by monitoring interrupts using `mwait` and reported an F1 score of 70 % on an Intel CPU. Based on our F1 score of 93.1 %, we can conclude that our attack has a much higher accuracy.

Our video fingerprinting attack achieves an F1 score of 90.2 % in a closed-world fingerprinting scenario over the top 20 videos. Reed et al. [40] exploit the change in throughput of Dynamic Adaptive Streaming over HTTP (DASH) throughout a video to fingerprint videos streamed from Netflix

Table 5.1.: List of proposed mitigations, their performance impact measured with the Stress-NG benchmark suite, and if the mitigation partially (◐) or fully (●) mitigates our attacks.

Mitigation	Perf. Change	Security
Isolated IRQ Core	−4.8 %	◐
IRQ Randomization	∼0 %	◐
Disable `WAITPKG` (VMX)	−2 %	◐
Hardware Change	`unknown`	●

through a wireless network with an accuracy greater than 90 % in less than 5 min. Gu et al. [12] built a bitrate-independent video fingerprinting attack on DASH by monitoring network traffic throughput and matching video fingerprints and achieved an accuracy of 90 % after 3 min on custom videos. Reed et al. [41] use passive network traffic analysis to fingerprint 20 min long Netflix videos transmitted through HTTPS with an accuracy of 99.5 %. Based on our F1 score of 90.2 % on YouTube videos, our results are in line with existing attacks, with significantly shorter measurement times using only package frequency information through network interrupts.

7. Discussion and Mitigations

Our work shows previously unknown security implications of idle states, in particular on the cross-process and cross-VM confidentiality of highly sensitive privacy-related information. The `tpause` instruction offers a fast and energy-efficient alternative to unprivileged busy waits. While prior work already used `mwait` to detect interrupts [68], our work shows that the underlying root cause is the idle state, revealing a more generic problem, including the performance-related information leakage of C0.2, as shown in Section 3.1.

Our work shows that the potential security risks of idle states have previously not been fully understood. Consequently, apart from disabling the instruction set extension, the mitigations discussed by Zhang et al. [68] do not resolve the root cause of the security issue. Moreover, secure scheduling policies, e.g., core scheduling [21, 26, 8], do not protect against our attacks as the core assignment for interrupt handling is independent

of these policies for threads, processes, and VMs. In Table 5.1, we provide an overview of possible mitigations, their performance impact measured through the Stress-NG benchmark suite, and their effectiveness in mitigating the attacks proposed in this paper. We propose the following mitigations:

Disabling WAITPKG Currently, there is no option to disable only the `WAITPKG` instruction set extension completely. It is possible to disable `WAITPKG` for the user space, but only together with all other `TSC` related instructions, including `rdtsc` by setting the `TSD` bit in the `CR4` register [16]. This is not feasible as numerous user space applications rely on the availability of the `rdtsc` instruction. It is only possible to disable the C0.2 through the `IA32_UMWAIT_CONTROL` MSR forcing `umwait` and `tpause` to fall back to C0.1 if C0.2 is requested [16], which does not mitigate the security issues present in C0.1. Contrary to the native case, it is possible to disable `WAITPKG` for virtual machines. While most user space applications do not use `umwait` or `tpause`, as they are new instructions, the Linux kernel uses `tpause` with C0.2 for short delays and falls back to a busy wait if the instruction is not supported. Disabling the instruction set extension for security would mean sacrificing its potentially substantial performance gains, as shown in Section 2.3, which is not a practical solution for most systems. For disabling `WAITPKG` in virtual machines, we measured a performance degradation of 2 % on our i7-1260P with the Stress-NG benchmark.

Isolated IRQ Core To avoid a possible detection of external interrupts, the operating system can isolate at least one physical core for handling external interrupts. This interrupt isolation would make it no longer possible to detect external interrupts through `umwait` or `tpause` with C0.2 on the same core or with C0.1 on the sibling logical core, as the attacker can not be scheduled on the physical core that receives the interrupts. While this mitigates attacks targeting external interrupts, the undocumented wake-ups of C0.1 on exceptions and in/out-port instructions of sibling logical cores are still exploitable. Furthermore, isolating physical cores for interrupt handling comes with a significant performance impact, especially on multithreaded workloads. Specifically for larger server systems, multiple cores are needed to handle the interrupt load fully. On our test system with an i7-1260P, the performance in the Stress-NG benchmark decreased by 4.8 % when dedicating one physical core for interrupt handling.

IRQ Randomization By regularly randomizing the core assignments of external interrupts in short periods of time, the operating system can introduce a significant amount of noise to fingerprinting attacks. A higher reassignment frequency causes more noise for the attacker but also a higher performance and energy overhead. While this mitigation makes attacks monitoring external interrupts significantly more challenging, it does not entirely mitigate them, as an attacker can still monitor interrupts. Furthermore, exception monitoring by the attacker through the C0.1 idle state is still possible. On our i7-1260P, we did not observe a significant performance overhead in the Stress-NG benchmark with random interrupt affinity reassignment every 0.5 s.

Hardware Changes Completely removing wake-up on interrupts is not viable as software, such as the Linux kernel, relies on this functionality. We propose removing the undocumented wake-ups of C0.1 on exceptions and external interrupts of sibling logical cores. Furthermore, to completely mitigate interrupt monitoring from VMs, a solution would be to adopt the behavior of the `hlt` instruction, which does not wake up in case of a VM exit. These changes would completely resolve the interrupt-related security issue of `umwait` and `tpause` for the scenario of a virtual-machine-based attacker and limit the attack surface in a native scenario.

Other Potential Mitigations We conclude that this issue requires a hardware mitigation, as in software we can only fully disable `umwait` and `tpause` for virtual machines and other mitigation techniques are not sufficient and impose a possible negative performance impact. Removing wake-up reasons may be a viable approach for the cases where the sibling logical core is woken up. However, e.g., in our attack in Section 6.3, we exploited interrupt handling on the same core, which is a wake-up reason that cannot be eliminated. The older privileged `mwait` instruction, which is similar to the unprivileged `umwait`, allows the CPU to switch into deeper sleep states, and wakes up when a VM exit occurs [18]. It is, therefore, reasonable to assume that this is also the intended behavior for `umwait` and `tpause`. However, currently, this behavior is not documented in the Intel instruction manual and can be the source of security issues regarding virtual-machine-based software isolation.

Besides the interrupt-related issues of idle states, we also show how the performance gain results in security problems. As we can expect more

applications start to incorporate `umwait` and `tpause` in the near future, the attack surface will further increase where attackers can also infer, e.g., control-flow information on other applications.

Generic solutions include trapping idle-state instructions in virtual machines, and injecting fake keystrokes for noise against our inter-keystroke timing attacks [48]. However, these approaches come with performance and energy costs that might not be justified compared to just removing the idle state control. Moreover, for other attacks, e.g., our covert channel and the website and video fingerprinting, noise reduces the performance but does not fully mitigate the leakage.

Finally, prior work discussed the detection of side channels using performance counters [36]. However, as Zhang et al. [68] already noted, there are no performance counters tracking the use of idle-state controlling instructions so far. In contrast to many prior microarchitectural attacks, IdleLeak does not induce a negative performance impact in the victim, that could be detected by the victim. Instead, IdleLeak rather improves the performance of the victim workload. However, the victim cannot detect this irregularity as malicious behavior, as this legitimately happens, with identical idle state choices, when the corresponding processor core is legitimately idling.

In conclusion, our work highlights the necessity for future work on effective mitigation of idle-state side channels.

8. Related and Future Work

Interrupt Detection & Keystroke Attacks Interrupt detection has been used in several previous works. Ristenpart et al. [42] used interrupt detection to synchronize attacker and victim. Schwarz et al. [48] presented an interrupt-detection-based attack in native code using high-resolution timers such as the x86 instruction `rdtsc`. They observed that `rdtsc` values have larger jumps when an interrupt occurs, as the attacker is not scheduled in this time frame. They further proposed a countermeasure to keystroke timing attacks which injects uniform high-frequency stream of fake keyboard interrupts. Lipp et al. [27] demonstrated a keystroke timing attack from JavaScript using a counting thread instead of a high-resolution timer. Prior to these works, keystroke timing attacks have been performed

based on cache attacks [42, 11], smart phone sensors [5], and remote timing measurements [52].

Closely related to IdleLeak is the recent work by Zhang et al. [68], exploring the `umonitor` and `umwait` for side-channel attacks, primarily to translate microarchitectural states into architectural states in transient-execution attacks. They evaluate their side channel, continuously executing `umonitor` and `umwait`, in interrupt detection scenarios counting the number of wake-ups in fixed time periods and making deductions on system activity, similar to prior works. Clearly, interrupt detection attacks are not new but they are important for comparability with prior work. Thus, we follow their best practice and also evaluate our idle-state side channel in interrupt detection scenarios for comparability. Furthermore, we demonstrate the *first* video fingerprinting attack based on interrupt detection with the IdleLeak side channel. This attack illustrates the severe and previously unknown privacy implications of the novel idle states that IdleLeak uncovered: Information about sensitive online video consumption may be used for instance for extortion campaigns [61].

One further difference to Zhang et al. [68] is our attack technique: Zhang et al. [68] use a documented feature of an instruction to wake up on interrupts and can only be used in a native non-VM scenario. While they focused on the behavior of these instructions, they did not explore the behavior of the idle states nor the problem around the interrupt core affinity. In contrast, we discovered the undocumented effect that the `tpause` instruction wakes up upon several events, including interrupts but also other system-level events (hardware exceptions, e.g., page faults, divide by zero, VM exits, inport and outport operations), which is orthogonal to the findings of Zhang et al. [68] compared to prior works [42, 27, 48]. Our findings are surprising as the spurious wake-up behavior of `tpause` on VM exits due to, e.g., host interrupts is inconsistent with other sleep instructions such as `hlt`, which does not wake up on VM exits. While not the main focus of their work, they also briefly evaluated their approach in a website-fingerprinting scenario. In contrast to their work, we focused on the broader security implications of idle states in general and demonstrated different attack scenarios (same core, sibling core) and new attacks (e.g., the first interrupt-detection-based video fingerprinting). We investigated the performance-enhancing effects and discovered further wake-up causes, including interrupts of unrelated workloads even in other virtual machines and the host, and exceptions and interrupts of sibling logical cores. Therefore, with our discoveries, the mitigations proposed by

Zhang et al. [68] are not sufficient anymore and future work needs to find mitigations that are effective but maintain an acceptable efficiency.

9. Conclusion

The new idle states, C0.1 and C0.2, introduce novel leakage that can be used to monitor system activity, in particular interrupts, in the case of C0.1 even interrupts arriving on logical sibling cores. Since interrupts on x86 are scheduled regardless of the corresponding workload, the attacker can spy on victims running on separate physical cores, by focusing on interrupt activity instead. We evaluated both our techniques ActiveIdleLeak and PassiveIdleLeak with covert channels, achieving true capacities of 7.1 Mbit/s ($\sigma_{\bar{x}} = 0.004$ Mbit/s, $n = 512$) and 656.37 kbit/s ($\sigma_{\bar{x}} = 0.63$ kbit/s, $n = 1\,024$) in native code. In a cross-VM scenario, we still achieves 46.3 kbit/s with ActiveIdleLeak. We demonstrated native keystroke-timing attacks, website- and video-fingerprinting attacks, all with high F-Scores, and a low standard error on the timing. The highly sensitive information that an attacker can acquire through these attacks, potentially exposing even sexual preferences to an attacker, can be used in different ways such as extortion campaigns. While mitigations against IdleLeak may be expensive due to the way interrupts are implemented on x86, we conclude that further research on efficient and effective mitigations is necessary to thwart the exploitation of this side channel.

Acknowledgments

We thank the anonymous reviewers and our anonymous shepherd, for their guidance, comments, and suggestions. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I-6054-N and FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from AWS, Google, Intel and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] 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. 68, 73).

[2] Alexa Internet, Inc. The top 1 million sites on the web. May 2023. URL: `https://www.alexa.com/topsites` (pp. 91, 95, 98).

[3] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 68, 72).

[4] Artem Bityutskiy. Sapphire Rapids C0.x idle states support. 2023. URL: `https://lwn.net/Articles/925863/` (pp. 74, 75).

[5] Liang Cai and Hao Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In: USENIX HotSec. 2011 (p. 106).

[6] 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. 94).

[7] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS. 2016 (pp. 72, 75).

[8] Dario Faggioli. Re: [RFC PATCH v3 00/16] Core scheduling v3. 2019. URL: `https://lore.kernel.org/lkml/277737d6034b3da072d3b0b808d2fa6e110038b0.camel@suse.com/` (p. 102).

[9] 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. 68, 73, 88).

[10] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security. 2018 (pp. 68, 73).

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 72, 106).

[12] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. Walls have ears: Traffic-based side-channel attack in video streaming. In: INFOCOM. 2018 (p. 102).

[13] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to violate web privacy with hardware performance events. In: ESORICS. 2017 (p. 101).

[14] 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. 94).

[15] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2023 (p. 74).

[16] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 2023 (pp. 74, 77, 103).

[17] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (p. 77).

[18] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2023 (pp. 74, 104).

[19] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In: S&P. 2012 (p. 101).

[20] Kevin S Killourhy and Roy A Maxion. Comparing anomaly-detection algorithms for keystroke dynamics. In: IEEE/IFIP International Conference on Dependable Systems & Networks. IEEE. 2009 (p. 90).

[21] 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 (pp. 70, 79, 102).

[22] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 72).

[23] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to Differential Power Analysis. In: Journal of Cryptographic Engineering (2011) (p. 72).

[24] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In: CRYPTO. 1999 (p. 72).

[25] Alberto Lerner and Philippe Bonnet. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In: International Conference on Management of Data. 2021 (p. 75).

[26] Linux Kernel Documentation. Core Scheduling. 2022. URL: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html (p. 102).

[27] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (pp. 69, 72, 105, 106).

[28] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS. 2020 (p. 73).

[29] Chen Change Loy. Keystroke100 Dataset. 2021. URL: http://personal.ie.cuhk.edu.hk/~ccloy/downloads_keystroke100.html (p. 91).

[30] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 73).

[31] 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. 68, 72).

[32] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 72).

[33] 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. 72).

[34] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 68).

[35] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. In: Cryptology ePrint Archive, Report 2002/169 (2002) (p. 72).

[36] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In: ESSoS. 2016 (p. 105).

[37] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (p. 72).

[38] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security. 2016 (pp. 72, 73).

[39] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In: E-smart. 2001 (p. 72).

[40] Andrew Reed and Benjamin Klimkowski. Leaky streams: Identifying variable bitrate DASH videos streamed over encrypted 802.11 n connections. In: CCNC. 2016 (p. 101).

[41] Andrew Reed and Michael Kranch. Identifying https-protected netflix videos in real-time. In: CODASPY. 2017 (p. 102).

[42] 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. 69, 72, 105, 106).

[43] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In: AsiaCCS. 2022 (p. 68).

[44] Michael Rushanan, David Russel, and Aviel D Rubin. Mallory-Worker: Stealthy Computation and Covert Channels Using Web Workers. In: International Workshop on Security and Trust Management. 2016 (p. 72).

[45] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS. 2021 (pp. 72, 88).

[46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (p. 73).

[47] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 72).

[48] 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 (pp. 69, 72, 105, 106).

[49] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 68, 73).

[50] Martin Schwarzl, Erik Kraft, and Daniel Gruss. Layered Binary Templating. In: ACNS. 2023 (p. 72).

[51] Laurent Simon, Wenduan Xu, and Ross Anderson. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. In: PETS (2016) (p. 72).

[52] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security. 2001 (pp. 72, 106).

[53] 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 (p. 101).

[54] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In: USENIX Security. Aug. 2022 (pp. 68, 72).

[55] UEFI Forum, Inc. Advanced Configuration and Power Interface (ACPI) Specification Release 6.5. 2022 (p. 73).

[56] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security. 2017 (p. 72).

[57] Pepe Vila and Boris Köpf. Loophole: Timing Attacks on Shared Event Loops in Chrome. In: USENIX Security. 2017 (p. 72).

[58] Steve Walton. How Screwed is Intel without Hyper-Threading? 2019. URL: `https://www.techspot.com/article/1850-how-screwed-is-intel-no-hyper-threading/` (pp. 68, 72).

[59] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: CCS. 2017 (pp. 72, 73).

[60] Zhenghong Wang and Ruby B Lee. Covert and Side Channels due to Processor Architecture. In: ACSAC. 2006 (p. 72).

[61] Davey Winder. Has A 'Hacker' With Your Password Really Recorded You Watching Porn? 2022. URL: `https://www.forbes.com/sites/daveywinder/2022/11/28/has-a-hacker-with-your-password-really-recorded-you-watching-porn/` (p. 106).

[62] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyperspace: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: ACM Transactions on Networking (2014) (p. 72).

[63] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (p. 72).

[64] 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. 72).

[65] 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. 94).

[66] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 68, 72, 73).

[67] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security. 2009 (p. 72).

[68] 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. 68, 69, 77, 88, 101, 102, 105–107).

6

A Systematic Evaluation of Novel and Existing Cache Side Channels

Publication Data

Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025

Contributions

Main Author.

A Systematic Evaluation of Novel and Existing Cache Side Channels

Fabian Rauscher, Carina Fiedler, Andreas Kogler, Daniel Gruss

Graz University of Technology

Abstract

CPU caches are among the most widely studied side-channel targets, with Prime+Probe and Flush+Reload being the most prominent techniques. These generic cache attack techniques can leak cryptographic keys, user input, and are a building block of many microarchitectural attacks.

In this paper, we present the first systematic evaluation using 9 characteristics of the 4 most relevant cache attacks, Flush+Reload, Flush+Flush, Evict+Reload, and Prime+Probe, as well as three new attacks that we introduce: Demote+Reload, Demote+Demote, and DemoteContention. We evaluate hit-miss margins, temporal precision, spatial precision, topological scope, attack time, blind spot length, channel capacity, noise resilience, and detectability on recent Intel microarchitectures. Demote+Reload and Demote+Demote perform similar to previous attacks and slightly better in some cases, e.g., Demote+Reload has a 60.7 % smaller blind spot than Flush+Reload. With 15.48 Mbit/s, Demote+Reload has a 64.3 % higher channel capacity than Flush+Reload. We also compare all attacks in an AES T-table attack and compare Demote+Reload and Flush+Reload in an inter-keystroke timing attack. Beyond the scope of the prior attack techniques, we demonstrate a KASLR break with Demote+Demote and the amplification of power side-channel leakage with Demote+Reload. Finally, Sapphire Rapids and Emerald Rapids CPUs use a non-inclusive L3 cache, effectively limiting eviction-based cross-core attacks, e.g., Prime+Probe and Evict+Reload, to rare cases where the victim's activity reaches the L3 cache. Hence, we show that in a cross-core attack, DemoteContention can be used as a reliable alternative to Prime+Probe and Evict+Reload that does not require reverse-engineering of addressing functions and cache replacement policy.

1. Introduction

Modern CPUs have multiple cache levels with lower latencies and capacities in lower levels, closer to the execution core, and higher latencies and capacities at higher levels, further away from the core. Cache performance is so crucial that disabling caching on commodity systems effectively slows them down by multiple orders of magnitude. Caches cannot buffer all memory, meaning that some memory locations will be buffered and fast, whereas others have higher access times, introducing the problem of cache side-channel attacks: Which data is cached, is decided based on what was recently used, is frequently used, as well as what is predicted to be used in the near future. Hence, cache side-channel attacks measure the timing to then infer which memory locations were recently used [84, 25], when memory locations are used [25, 77], and which memory locations are predicted to be used [21, 41]. Side-channel attacks then use this information to infer the actual secrets that led to these memory accesses or predictions.

Several generic cache attack techniques have been discussed in the literature, with the most prominent examples being Prime+Probe [52] and Flush+Reload [84]. Flush+Flush [23] and Evict+Reload [25] are variations of Flush+Reload that can be beneficial in some use cases [42, 71]. A promising defense against Prime+Probe is the concept of randomized secure caches, minimizing [81, 56, 61] or even eliminating [18] the chance of priming and probing the cache successfully. However, flush-based attacks are typically excluded, *i.e.*, Flush+Reload and Flush+Flush, and instead, propose to disable `clflush` [81]. Other works try to mitigate flush-based attacks through detection [11, 26, 23]. Instead of introducing a secure cache, Intel decided to move to a non-inclusive L3 cache with their recent Sapphire Rapids microarchitecture. The non-inclusive L3 cache provides no guarantees on inclusiveness towards lower levels, effectively mitigating the possibility to evict cache lines from the private L1 and L2 caches of other cores. Hence, eviction-based cross-core attacks, e.g., Prime+Probe and Evict+Reload, are not possible anymore unless the victim's own activity repeatedly leads to data placement in the L3 cache. Furthermore, the addressing functions and replacement policy for Sapphire Rapids have not been reverse-engineered yet, and even with these steps, the input to these functions is physical addresses, *i.e.*, privileged information typically unavailable to an attacker.

In this paper, we present the first systematic evaluation using 9 characteristics of the 4 most relevant cache attacks, Flush+Reload, Flush+

Flush, Evict+Reload, and Prime+Probe, as well as three new attacks that we introduce: Demote+Reload, Demote+Demote, and DemoteContention. We evaluate hit-miss margins, temporal precision, spatial precision, topological scope, attack time, blind spot length, channel capacity, noise resilience, and detectability. In a comprehensive comparison, we show that our new attacks, Demote+Reload and Demote+Demote, perform better on some and worse on other characteristics but yield overall a similar attack performance as known attacks. Demote+Reload has a 60.7 % smaller blind spot than Flush+Reload, and Demote+Demote has the lowest attack runtime (185.8 cycles). Our blind-spot evaluation also reveals that Flush+Flush and Demote+Demote have no significant blind spot, whereas other attacks have considerable blind-spot lengths compared to their attack runtime, of up to 90.1 %. With a true capacity of 15.48 Mbit/s, Demote+Reload has a 64.3 % higher channel capacity than Flush+Reload, making it slightly faster than the previously fastest CPU covert channel [60]. Similar to Flush+Flush, Demote+Demote doesn't even trigger L1 or L2 misses, making it less visible to state-of-the-art detection mechanisms [11, 26, 23].

To evaluate all attacks further, we mount multiple attacks as a benchmark: We compare them in an AES T-table attack, showing that Demote+Reload outperforms the other attacks. Furthermore, we compare Demote+Reload and Flush+Reload in an inter-keystroke timing attack. Beyond the scope of the prior attack techniques, we demonstrate a KASLR (kernel address-space layout randomization) break with Demote+Demote, exploiting that `cldemote` also leaks information about the validity of kernel address mappings. Finally, we show how Collide+Power-style power side-channel leakage can be amplified with Demote+Reload, as the expensive and noisy cache eviction and cache reloading are avoided as compared to Flush+Reload.

Our analysis also revealed that the non-inclusive L3 cache of the recent Sapphire Rapids microarchitecture practically poses a significant restriction of what cross-core cache attacks can observe. Our results show that Flush+Reload and Flush+Flush are unaffected as they rely on the `clflush` instruction that evicts a cache line from all caches. However, attacks relying on eviction or `cldemote` do not lead to an eviction of the cache line from the other core's private cache. Hence, without the `clflush` instruction, an attacker can only spy on cache lines that the victim itself brings into the L3 cache. Based on this insight, we present a new cross-core attack, DemoteContention. DemoteContention does not rely on reverse-engineering

of addressing functions or cache replacement policy for eviction, neither of which has been reverse-engineered for Sapphire Rapids yet. As DemoteContention does not rely on shared memory between attacker and victim, it can be used as already as a reliable alternative to Prime+Probe and Evict+Reload on the non-inclusive L3 cache, even without privileged information about physical addresses.

In summary, we make the following contributions:

1. We provide the first **systematic evaluation** of 9 characteristics (hit-miss margins, temporal and spatial precision, topological scope, attack time, blind spot length, channel capacity, noise resilience, and detectability) of the 4 most relevant cache attacks, Flush+Reload, Flush+Flush, Evict+Reload, and Prime+Probe.
2. We present three novel cache attack techniques, Demote+Reload and Demote+Demote for same-core scenarios with shared memory, and DemoteContention, for cross-core scenarios without shared memory and physical addresses. We include all three new attacks in our systematic evaluation.
3. We compare all attacks in an AES T-tables attack as a benchmark as well as Demote+Reload and Flush+Reload in an inter-keystroke timing attack.
4. We show that `cldemote` leaks information beyond the prior generic techniques: We build a KASLR break with Demote+Demote, and show how Collide+Power leakage can be amplified with Demote+Reload.

Outline. Section 2 provides background. Section 3 presents our novel attacks. Section 4 presents a systematic evaluation of state-of-the-art cache side-channel attacks. Section 5 presents Demote+Reload case studies. We discuss mitigations in Section 6 and conclude in Section 7.

Responsible Disclosure. We responsibly disclosed our findings to Intel (November 28, 2023). Intel concluded the responsible disclosure process and did not consider our findings a vulnerability.

2. Background

In this section, we discuss caches, the state-of-the-art in cache side-channel attacks as well as mitigation techniques.

2.1. CPU Caches

CPU caches are small and fast memories the CPU uses to store copies of data from main memory to hide the latency of main memory accesses. Modern CPUs have different levels of cache, typically three, varying in size and latency: the L1 cache is the smallest and fastest, while the L3 cache, also called last-level cache, is larger and slower. Modern CPUs are set-associative, *i.e.*, a cache line is stored in a fixed set, as determined by either its virtual or physical address. The last-level cache is physically indexed and shared across cores of the same CPU. On many CPUs from the last decade, the L3 cache was inclusive with respect to L1 and L2, meaning that all data stored in L1 and L2 is also stored in the last-level cache. To maintain this property, every line evicted from the last-level cache is also evicted from L1 and L2 caches. Some Intel CPUs also have an L4 cache, which acts as a victim cache, shared across all cores. However, recent Intel Xeon CPUs, e.g., Xeon Silver 4410T, and some large Intel Core CPUs have a non-inclusive L3 but no L4 cache. In a non-inclusive L3, cache lines present in the L1 and L2 of CPU cores do not have to be present in the L3. However, the L2 in turn is now inclusive with respect to the L1, whereas the L3 now acts as a victim cache. The last-level cache, though shared across cores, is also divided into slices. The undocumented hash function that maps physical addresses to slices in Intel CPUs has been reverse-engineered for older CPUs [48, 85, 29], however, it has not yet been reverse-engineered for Intel Sapphire Rapids CPUs.

2.2. Cache Attacks

Cache attacks exploit timing differences between cached and non-cached memory. Access-driven attacks are most powerful, where an attacker monitors its own activity to infer activity of a victim, e.g., which cache lines or cache sets the victim accessed. Flush+Reload [84], Evict+Reload [25] and Flush+Flush [23] all use shared memory, which is then shared in the cache, to infer whether the victim accessed a specific cache line. The attacker evicts data either by using the `clflush` instruction (Flush+Reload and Flush+Flush), or accessing congruent addresses, *i.e.*, cache lines that belong to the same cache set (Evict+Reload). These attacks have a very fine granularity (*i.e.*, a 64-byte cache line) and are most relevant in native environments.

An access-driven attack that does not rely on shared memory and, hence, is widely applicable, is Prime+Probe [52, 45, 33]. As it does not share a cache line with the victim, it cannot use the `clflush` instruction but instead has to access congruent addresses to evict data from the victim. The granularity of the attack is noisier and coarser as an attacker only learns which cache set was accessed. Besides noise from other processes, replacement policies make it hard to guarantee eviction from a cache set [22]. Disselkoen et al. [14] and Gruss et al. [24] use TSX transactions to detect victim-induced evictions instead of the probe step as a variation of the attack (Prime+Abort). Purnal et al. [56] proposed a variant optimized for randomized secure caches called Prime+Prune+Probe and a variant that reduces the observer effect [57] (blind spot) by exploiting the specific replacement policy of the cache and allowing for a single cache access of the victim to be detected with a single cache access by the attacker. It is important to note that these attacks are specializations of Prime+Probe and, beyond the generic attack principle, exploit specific behaviors of the caches and CPUs they attack.

2.3. Mitigating Cache Attacks

Eliminating resource sharing in the cache can mitigate attacks [54] at substantially increased costs. Abandoning technologies like SMT (Simultaneous Multi-Threading, which shares L1 and L2 caches) would reduce performance by 25 % to 35 % [76]. Consequently, the trend is going in the other direction, towards more sharing on all hardware and software levels. Furthermore, attacks through remote interfaces [8, 3, 1] are playing an increasing role [72, 64, 73, 39, 78] where cross-domain sharing is not the root cause. Resource sharing is unavoidable on personal computers, which serve the purpose of executing third-party code both in the form of native binaries or JavaScript on a website. Hence, researchers try to find defense mechanisms that maintain sharing.

Eliminating Measurable Timing Differences. Bernstein [3] proposed constant-time (*i.e.*, no secret-dependent branches or memory accesses) to mitigate cache attacks, a technique that today is the standard means to protect cryptographic algorithms. However, writing truly constant-time code can still be challenging [86, 55]. Several independent works proposed to manipulate timers to remove the ability to measure timing differences through determinism [2, 46, 37] or fuzziness [74]. Furthermore, even without timing sources, counting threads [82, 42, 63, 62] and timeless

methods [73] are viable alternatives. Disabling `clflush` [84, 23] can be circumvented by using eviction [25].

Eliminating cache-line and cache-set sharing. Attack techniques like Flush+Reload require shared memory and could be stopped by removing shared memory [84]. Indeed, one source of shared memory, page deduplication, is more and more restricted to prevent its malicious use [70, 4, 66, 13]. However, other use cases of shared memory, particularly shared libraries, are unaffected and still available for attacks [65]. Mitigating Prime+Probe requires addressing the sharing issue on the level of cache sets, e.g., by coloring cache lines and assigning colors to security domains [68, 35, 19, 12]. Intel CAT [31] provides dedicated software control over cache ways and can be used to separate workloads into different parts of the cache [43]. Several works eliminate sharing via cache flushing during context switches between different domains [89, 19].

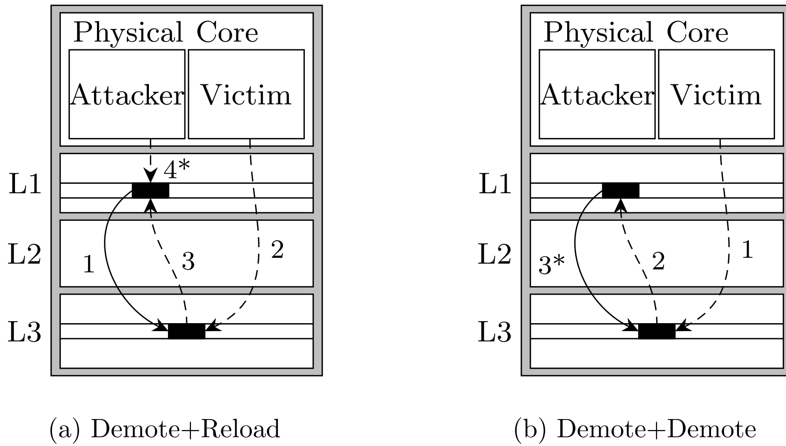
Detecting attacks. Many works researched the detection of cache attacks in binaries [25, 15, 32, 7], or at runtime using cache attacks [88, 25] or a range of performance counters [27, 10, 87, 53].

3. Novel `cldemote`-based Attacks

In this section, we present two attacks for same-core attack scenarios, Demote+Reload and Demote+Demote, and one cross-core attack, DemoteContention.

3.1. Same-Core Attacks

Demote+Reload and Demote+Demote offer fast and stealthy alternatives to existing cache attacks. Demote+Reload exploits the same hardware and software properties as Flush+Reload, whereas Demote+Demote exploits the same hardware and software properties as Flush+Flush. However, unlike Flush+Reload and Flush+Flush, Demote+Reload and Demote+ Demote do not induce a DRAM access. Cache misses are the largest contributor to the execution time of cache attacks, often taking hundreds of CPU cycles on average. Consequently, they are responsible for larger blind spots in attacks and lower attack frequencies. Furthermore, many detection mechanisms focus on detecting cache misses, e.g., with performance counters. Demote+Reload and Demote+Demote work across cores



(a) Demote+Reload

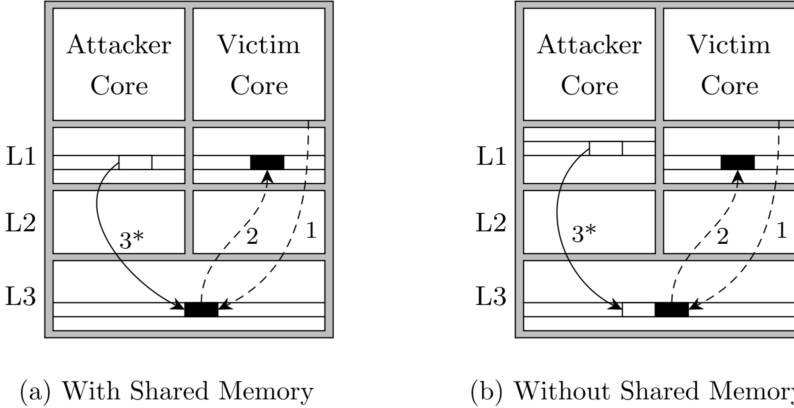
(b) Demote+Demote

Figure 6.1.: Working principle of Demote+Reload and Demote+Demote. The $*$ denotes the timed operation. Solid lines are `cldemote` executions and dashed lines memory accesses. Both attacks only move a cache line from L1 to L3. The victim activity is visible by timing either the reload (Demote+Reload) or the `cldemote` (Demote+Demote).

and in virtualized environments in a typical scenario where read-only shared memory with the victim process is available (e.g., shared libraries).

Demote+Reload and Demote+Demote build on the observation that the `cldemote` instruction can evict a cache line from L1 to L3. Subsequent victim accesses load the cache line into the L1 again, which the attacker can observe by timing the reload operation in Demote+Reload or `cldemote` in Demote+Demote. An attacker can also use Demote+Demote to observe when a victim running on another core loads a cache line into the cache and when it performs a write access on another core. Furthermore, like Flush+Flush, Demote+Demote can also be used to derive information on cache slices and CPU cores as the access latency to different cache slices varies.

The basic principle of our two attack techniques is illustrated in Figure 6.1. Demote+Reload follows the semantics of Flush+Reload: The attacker first demotes the cache line, then a victim operation possibly accesses this cache line, and afterward, the attacker times reloading the cache line, observing whether the victim accessed it (Figure 6.1a). Demote+Demote follows the semantics of Flush+Flush: The victim first possibly accesses this cache line, and afterward, the attacker times the `cldemote` instruction, observing whether the victim accessed the cache line in between (Figure 6.1b). Due



(a) With Shared Memory (b) Without Shared Memory

Figure 6.2.: Cross-Core DemoteContention. The ∗ denotes the timed operation. Solid lines are `cldemote` executions and dashed lines memory accesses. In both attacks, the victim repeatedly moves a cache line from L1 to L3. The attacker measures a memory access to the exact cache line the victim uses (Figure 6.2a) or a cache line in the same cache set (Figure 6.2b).

to the reliance on a load to the L1 by the victim, both attacks require that the attacker and victim run on the same physical core but potentially different logical cores with SMT.

3.2. Cross-Core Attacks

Cross-core attacks like Prime+Probe and Evict+Reload on previous systems exploited the inclusiveness of the L3 cache [47, 49]: Recent Xeon processors have a non-inclusive L3. Evicting a cache line from the L3 implies the removal of this cache line from all private L1 and L2 caches of all cores. This is not the case on these new CPUs anymore, where eviction from L3 has no implications on the private L1 and L2 caches of any core. Consequently, Prime+Probe and Evict+Reload can only sense when victim operations reach the L3 but not operations in the victim's private L1 and L2 caches. Beyond this limitation, the cache eviction in Prime+Probe and Evict+Reload brings another significant hurdle for an attacker: Eviction sets are generated based on physical addresses [47, 75], the undocumented cache addressing functions [17], and the undocumented cache replacement policy [22]. Mounting efficient Prime+Probe or Evict+ Reload would require to reverse-engineer these functions and additionally

to obtain privileged physical address information, e.g., possibly via a timing side channel [21, 59]. The non-inclusive L3 complicates Prime+Probe and Evict+Reload even further as the attacker's accesses to the eviction set need to reach the L3. This is possible by using `cldemote`, as we propose below, or by creating an eviction set that simultaneously evicts L1 and L2 caches, moves the corresponding cache lines to the L3 and thereby evicts the L3 cache set. Our experiments indicate that L1 and L2 eviction on Sapphire Rapids in general does not lead to placement of cache lines in the L3, *i.e.*, the L3 does not act primarily as a victim cache.

As an alternative to cross-core Prime+Probe and Evict+Reload, we present a new cross-core attack, DemoteContention (Figure 6.2). DemoteContention is a reliable alternative to Prime+Probe and Evict+Reload without reverse-engineering of addressing functions and cache replacement. DemoteContention relies on contention on the L3 cache set by continuously demoting a cache line. The attacker uses their own cache line, *i.e.*, no shared memory, located in the same L3 cache set as the victim cache line, e.g., found via profiling [25]. If the victim performed an operation that reaches the L3 caches, DemoteContention observes a spike in the `cldemote` execution time, even if the attacker-monitored cache line is not in the cache, allowing for continuous execution of `cldemote`.

The reason why DemoteContention works, is that `cldemote` has to interact with the L3 or cache directory, *i.e.*, leading to contention on the corresponding set in the L3 or the cache directory, regardless of the state of the cache line. The L3 and the cache directory have been investigated in previous works [83, 47] and identified as viable cross-core channels. Given that `cldemote`, under regular usage, updates either the L3 or cache directory, we suspect it performs a lookup in the cache directory or L3 even before it knows whether the cache line to be demoted is in the L1 or L2. Consequently, `cldemote` is influenced by concurrent contention on the corresponding set in the L3 or cache directory.

A significant limitation compared to same-core attacks, including Demote+Reload and Demote+Demote, is that, based on our experiments, we could only reliably trigger this situation by accessing a cache line and demoting. This limitation applies identically to cross-core Prime+Probe and Evict+Reload on Sapphire Rapids and Emerald Rapids.

Table 6.1.: Comparison of all tested attacks on an Intel Sapphire Rapids Xeon Silver 4410T (SR) and an Emerald Rapids Xeon Silver 4514Y (ER). The hit-miss margin and attack time are in cycles (C).

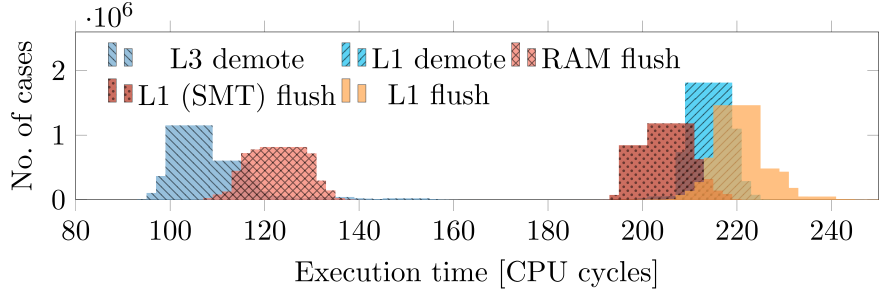
Attack	Topological Constr.	Shared Memory	Attack Margin		Temporal Prec. (SD)		Spatial Prec.	Blind Spot		Attack Time		Channel Cap. [Mbit/s]		Error Ratio	
CPU	SR/ER	SR/ER	SR	ER	SR	ER	SR/ER	SR	ER	SR	ER	SR	ER	SR	ER
Demote+Reload	Core	✓	48 C	34 C	±27 ns	±24 ns	cache line	42.8 %	42.6 %	424.3 C	320.6 C	6.38	6.09	0.7 %	0.7 %
Demote+Demote	Core	✓	86 C	60 C	±17 ns	±16 ns	cache line	0.0 %	0.0 %	185.8 C	137.6 C	8.34	9.36	4.6 %	2.3 %
DemoteContention	CPU	✗	–	–	±16 ns	±24 ns	L3 cache set	90.1 %	89.5 %	185.8 C	137.6 C	0.18	0.19	5.9 %	1.9 %
Flush+Reload	CPU	✓	170 C	124 C	±24 ns	±15 ns	cache line	89.9 %	86.3 %	614.1 C	462.8 C	1.94	2.15	4.9 %	3.6 %
Flush+Reload (SMT)	Core	✓	232 C	166 C	±20 ns	±20 ns	cache line	75.1 %	74.3 %	614.1 C	462.8 C	2.35	2.01	3.4 %	3.2 %
Flush+Flush	CPU	✓	86 C	38 C	±12 ns	±15 ns	cache line	0.0 %	0.0 %	192.0 C	155.2 C	8.80	10.26	1.2 %	1.9 %
Flush+Flush (SMT)	Core	✓	72 C	30 C	±19 ns	±18 ns	cache line	0.0 %	0.0 %	192.0 C	155.2 C	8.55	9.03	5.6 %	2.6 %
Prime+Probe (L1)	Core	✗	78 C	68 C	±33 ns	±33 ns	L1 cache set	23.9 %	24.8 %	281.2 C	245.0 C	3.62	3.78	10.9 %	1.2 %
Evict+Reload (L1)	Core	✓	10 C	4 C	±38 ns	±41 ns	cache line	21.4 %	15.5 %	390.0 C	339.8 C	2.51	3.32	6.8 %	1.4 %

4. Systematic Evaluation of State-of-the-Art Cache Side Channels

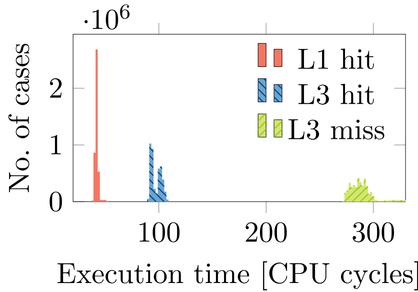
In this section, we provide a systematic evaluation of state-of-the-art cache attack techniques, including Demote+Reload, Demote+Demote, Flush+Reload [84], Flush+Flush [23], Evict+Reload [25] on the L1, and Prime+Probe [52] on the L1. All measurements are taken on the Sapphire Rapids Xeon Silver 4410T (CPU SR) and the Emerald Rapids Xeon Silver 4514Y (CPU ER) CPUs, as the very new `cldemote` instruction is only supported on Intel Xeon Sapphire Rapids and Emerald Rapids CPUs so far. Our setup runs Ubuntu 22.04 LTS, Linux 6.2.0, and gcc 11.4 on the CPU SR and Ubuntu 24.04 LTS, Linux 6.8.0, and gcc 11.4 on the CPU ER. We fixed the frequency for all experiments in this section, except for the covert channel and noise resilience tests, to reduce the noise of this comparison through frequency scaling. For the measurements, we use `rdtsc`, like prior work, providing a sub-nanosecond resolution timestamp. Similarly, we use `lfence` and `mfence` instructions to ensure that the instructions are ordered with respect to other instructions and memory operations. Our findings are summarized in Table 6.1. While we attempted to mount Prime+Probe or Evict+Reload on the L3, our experiments indicate that L1 and L2 eviction does not reliably lead to placement of the cache line in the L3. Furthermore, the addressing functions published by Gerlach et al. [17] did not yield reliable eviction on the more recent Sapphire Rapids microarchitecture either, possibly due to non-inclusiveness or a change in addressing functions. The only approach we found to place cache lines in the L3 reliably was `cldemote`. Additionally, physical addresses (for the addressing functions) are privileged information, whereas all other attacks in our comparison work with unprivileged information only. Hence, we also evaluate DemoteContention but not Prime+Probe or Evict+Reload on the L3.

4.1. Hit-Miss Margins

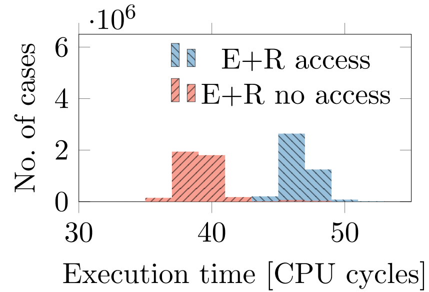
We define the hit-miss or attack margin for an attack as the difference between the 95th percentile of the faster and the 5th percentile of the slower case. The difference between percentiles is a more meaningful metric than a difference between averages, as the deviation around the mean would be ignored otherwise. The results of our measurements for all



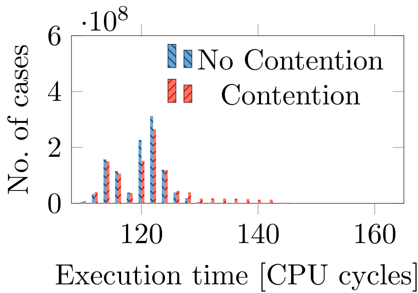
(a) `cldemote` & `clflush` Timings



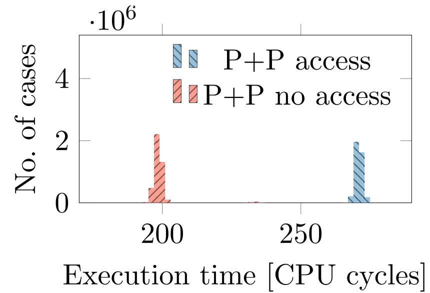
(b) Memory Access Timings



(c) Evict+Reload



(d) Cross-core DemoteContention



(e) Prime+Probe

Figure 6.3.: Timing histograms for all tested attacks on our Xeon Silver 4410T.

attacks are shown in Figure 6.3 (CPU SR) and Figure 6.13 (CPU ER). We conducted 10^6 measurements for each case.

The L1 hit, L3 hit, and DRAM access timings are shown in Figure 6.3b and Figure 6.13b. For Demote+Reload, we distinguish between L1 hits (victim access) and L3 hits (after `cldemote`). The resulting hit-miss margins are 48 cycles (CPU SR) and 34 cycles (CPU ER). SMT Flush+Reload uses L1 hits (victim access) and L3 misses (after `clflush`) with hit-miss margins of 232 cycles (CPU SR) and 166 cycles (CPU ER). Cross-core Flush+ Reload uses L3 hits and L3 misses with hit-miss margins of 170 cycles (CPU SR) and 124 cycles (CPU ER). The margin is significantly larger for Flush+Reload than Demote+Reload due to the long time it takes to fetch memory from DRAM.

The relevant timings for Demote+Demote and Flush+Flush are shown in Figure 6.3a and Figure 6.13a. For Demote+Demote, we distinguish between a `cldemote` on a cache line in the L1 (victim access) and a cache line in the L3 (after `cldemote`) with attack margins of 86 cycles (CPU SR) and 60 cycles (CPU ER). The similar SMT Flush+Flush attack distinguishes between a `clflush` on a cache line in the L1 (victim access) and a DRAM access (after `clflush`) with attack margins of 72 cycles (CPU SR) and 30 cycles (CPU ER). Cross-core Flush+Flush distinguishes between a `clflush` on a cache line in an L1 of a different core (victim access) and a not-present cache line with an attack margin of 86 cycles (CPU SR) and 38 cycles (CPU ER). We can observe that the margin for our Demote+Demote is significantly larger than for Flush+Flush. It takes slightly longer to demote a cache line from the L1 to the L3 than to flush a cache line accessed only by the same core. This difference is most likely the result of `cldemote` ensuring that the cache line is written to the L3 while `clflush` on an unmodified cache line only evicts the data from the cache.

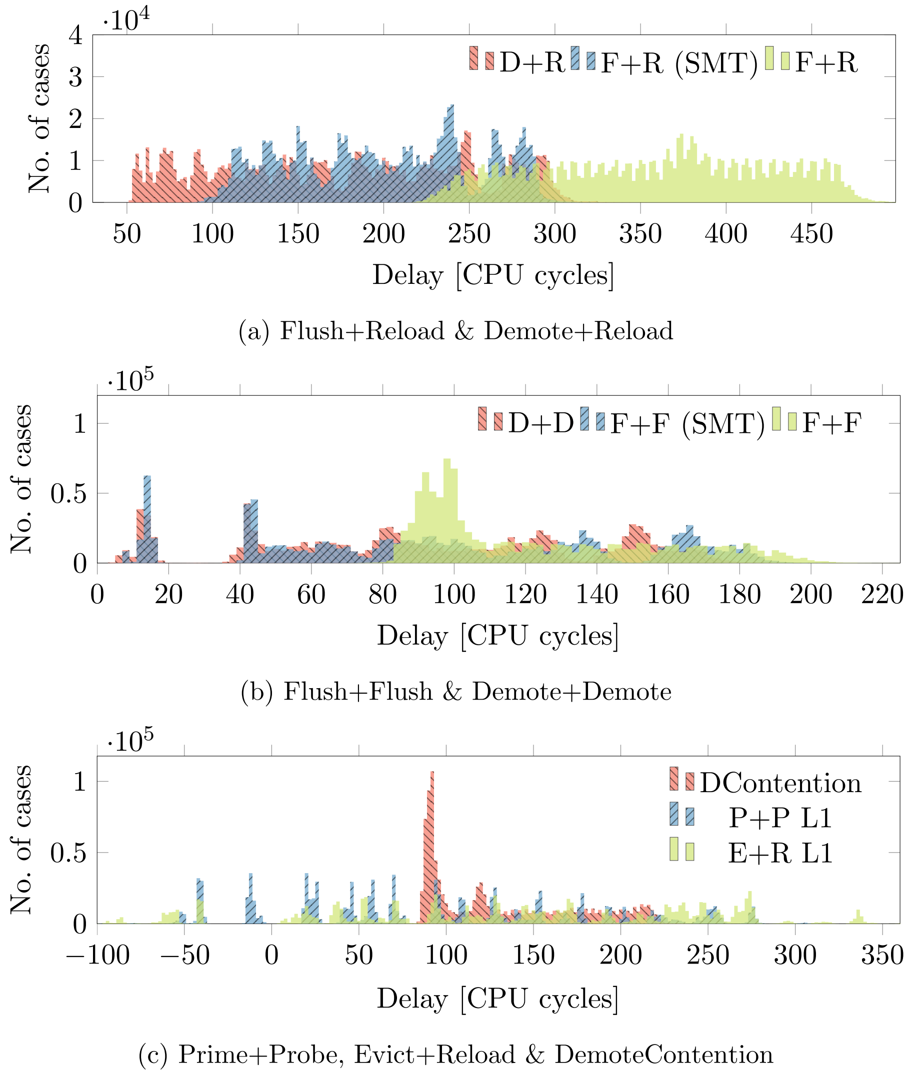
For Prime+Probe and Evict+Reload, we use an eviction-set size of 12 as the Xeon Silver 4410T has a 12-way set-associative L1 cache. L1 Prime+ Probe (Figure 6.3c and Figure 6.13c) has attack margins of 78 cycles (CPU SR) and 68 cycles (CPU ER). L1 Evict+Reload (Figure 6.3e and Figure 6.13e) has attack margins of 10 cycles (CPU SR) and 4 cycles (CPU ER). While the timings seem noise-free for Prime+Probe and Evict+ Reload, both attacks are highly susceptible to noise from unrelated memory accesses as they monitor accesses to whole cache sets in the relatively small L1.

Figure 6.3d shows the hit-miss histogram for DemoteContention. The contention case of DemoteContention (victim access) takes 121.4 cycles (CPU SR) and 101.4 cycles (CPU SR), and the no contention case (no victim access) takes 119.8 cycles (CPU SR) and 100.3 cycles (CPU SR). The two cases overlap almost entirely due to a large blind spot, which can not be easily counteracted, as further discussed in Section 4.4. Due to this, we do not have a realistic attack margin for DemoteContention. Despite this, the contention case has a significant number of measurements above 124 cycles (CPU SR) and 116 cycles (CPU ER), which are not present without contention, making the two cases distinguishable.

A high hit-miss margin is advantageous in scenarios with a significant amount of noise, such as frequently changing CPU frequency or other cache events on the CPU. This makes Flush+Reload a good choice in a high-noise scenario, given its large hit-miss margin. Despite this, a higher hit-miss margin can come at a cost, e.g., higher attack times (see Section 4.4), which is the case for Flush+Reload.

4.2. Temporal Precision

We measure the temporal precision by triggering accesses at a low frequency and measuring the time it takes for the attacker to detect them. The victim thread triggers an access at random intervals. All attacks are performed with a minimal attack loop, including the attack and a store to a memory location if an access is detected. We define the temporal difference as the timing difference between the start of the victim access, measured with `rdtsc` directly before the access, and the time the attacker detects the access. We specifically use the start of the attacker's measurement period that detects the access. We use the start of the measuring period instead of the end, as the end includes the attack time, which can result in more noise and, therefore, a higher standard deviation, an essential metric for attacks such as inter-keystroke timing attacks. Furthermore, we evaluate the attack time in a separate experiment (Section 4.4). Using the start of an attack's measurement period can result in a negative temporal difference, as accesses that occur directly after the measurement period starts can be detected by some attacks, as seen in Figure 6.4c. Only successfully detected accesses are included in this experiment, as blind spots are separately evaluated in Section 4.4. The results are shown in Figure 6.4 and Figure 6.16. We conducted 10^6 measurements for each case, and the standard error for all measurements is ≤ 0.1 cycles.



(a) Flush+Reload & Demote+Reload

(b) Flush+Flush & Demote+Demote

(c) Prime+Probe, Evict+Reload & DemoteContention

Figure 6.4.: The delay between memory access and the start of the detection period detected the access for all attacks on our Xeon Silver 4410T.

Our two CPUs perform very similarly for all attacks, except for DemoteContention and cross-core Flush+Reload. DemoteContention has a higher standard deviation (24 ns) on CPU ER than on (16 ns) on CPU SR. Flush+Reload has a significantly lower standard deviation at 15 ns on CPU ER than on CPU SR (24 ns). The histogram in Figure 6.16a shows that most measurements are at ≈ 180 cycles with some outliers (not false positives) at ≈ 350 cycles that increase the standard deviation. Without these outliers, the standard deviation is 8 ns. We presume this results from a change in inter-core communication in the microarchitecture related to memory accesses, as SMT Flush+Reload does not exhibit this difference but instead performs the same on both CPUs. Excluding this, cross-core Flush+Flush has the lowest standard deviations of 12 ns (CPU SR) and 15 ns (CPU ER). Demote+Demote has the lowest standard deviations of all same-core attacks, including same-core Flush+Flush of 17 ns (CPU SR) and 16 ns (CPU ER). Evict+Reload performs the worst with ± 38 ns (CPU SR) and ± 41 ns (CPU ER). Overall, all tested attacks have a standard deviation of tens of nanoseconds at most, which is ideal for variance-critical attacks such as inter-keystroke timing attacks.

4.3. Spatial Precision and Topological Scope

The spatial precision of the side channels we evaluate is defined by the microarchitectural element they attack. We provide the spatial precision for all attacks in Table 6.1. While the attacks have theoretical spatial precision, it is essential to note that practically, targeting multiple locations within a particular memory range can be challenging due to hardware mechanisms interfering, e.g., 4 kB due to the prefetcher, or parts of an 8 kB block scattered over 512 kB due to the DRAM row buffer. Demote+Reload, Demote+Demote, SMT Flush+Reload, and SMT Flush+Flush have a spatial precision of L1 cache lines (or L2 cache lines, depending on the threshold), allowing them to monitor if a 64 B memory region was recently accessed. L1 Evict+Reload has a spatial precision of L1 cache lines. Cross-core Flush+Reload and cross-core Flush+Flush have a spatial precision of L3 cache lines. L1 Prime+Probe has a spatial precision of L1 cache sets, and thus, can only detect whether any cache line is loaded into the monitored cache set. Cross-core DemoteContention has a spatial precision of L3 cache sets, *i.e.*, like Prime+Probe on the L3 cache.

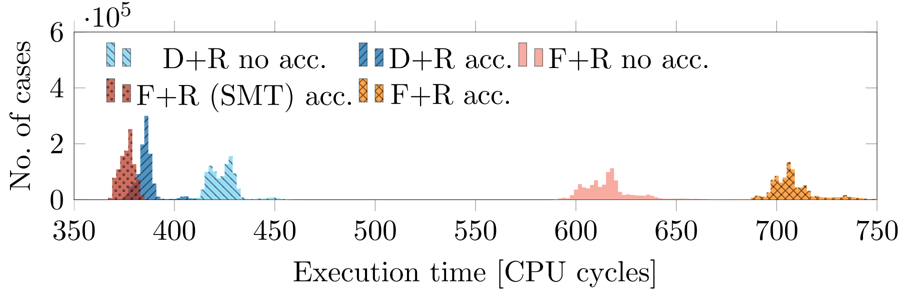
Another spatial aspect is the topological scope of an attack, which is influenced by the microarchitectural element under attack and aspects

such as the availability of shared memory between victim and attacker. Depending on these, an attack on a co-located victim using a specific technique may be viable or not. For instance, Flush+Reload requires co-location on the same physical CPU (on Intel) or core complex (on AMD) and, additionally, the use of a shared library or other read-only shared memory to mount an attack. Other attacks, e.g., L1 Prime+Probe, require co-location on the same physical core as the target is the L1 cache. Demote+Reload and Demote+Demote also require co-location on the same core, as the target is primarily the L1 cache. We provide an overview of all attacks in Table 6.1.

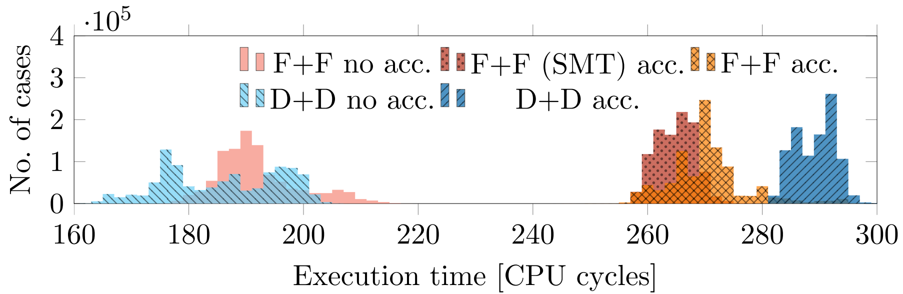
4.4. Attack Times and Blind Spot

The attack time is essential in determining the throughput an attack can achieve. To determine the attack time, we evaluated the attack round length and provide the results in Figure 6.5 and Figure 6.14. An attack round consists of the minimal code for the tested attacks, a check whether the resulting timing value is above or below a threshold and an access is detected, and a store of the result in an atomic variable for further processing in a separate thread. We conducted 10^6 measurements for each case. The standard error of the average for all results is ≤ 0.1 cycles. On both CPUs, the tested attacks perform similarly to each other, with the main change being a lower cycle count for all measurements on our CPU ER due to a different clock frequency compared to our CPU SR.

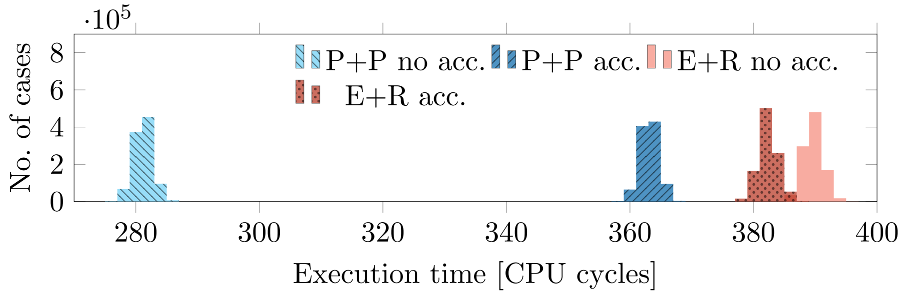
Demote+Demote has the lowest attack times, with 185.8 cycles (CPU SR) and 137.6 cycles (CPU ER) with no victim access, and 289.2 cycles (CPU SR) and 216.5 cycles (CPU ER) with a victim access (see Figure 6.5b and Figure 6.14b). The similar SMT Flush+Flush has attack times of 192.0 cycles (CPU SR) and 146.5 cycles (CPU ER) with no victim access and 264.9 cycles (CPU SR) and 197.7 cycles (CPU ER) with a victim access. The attack time without a victim access for Demote+Demote is slightly lower than for Flush+Flush. The attack time after a victim access for SMT Flush+Flush is slightly lower than for Demote+Demote; this is expected, as `cldemote` on an L1 cache line is slightly slower than a `clflush` on the cache line in the L1 of the attacking core (see Section 4.1). We consider the case without a victim access more relevant, as it is more common in most attack scenarios. Both Demote+Demote and Flush+ Flush consist of measuring a single instruction without further setup, unlike Flush+Reload and Demote+Reload, resulting in significantly lower



(a) Flush+Reload & Demote+Reload



(b) Flush+Flush & Demote+Demote



(c) Prime+Probe & Evict+Reload

Figure 6.5.: Execution time in cycles of a single attack iteration for all tested attacks on our Xeon Silver 4410T.

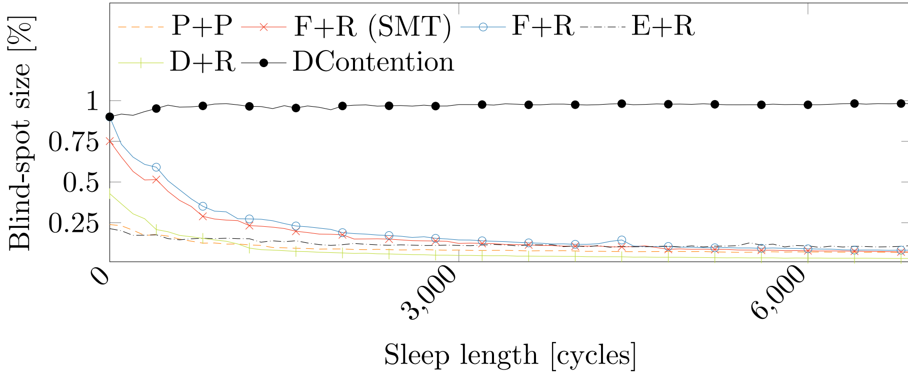


Figure 6.6.: Blind-spot size of all tested attacks that have a blind-spot for varying delay lengths after each attack iteration. The blind-spot size is given in the percent of a single attack loop iteration (including the delay) on our Xeon Silver 4410T.

attack times. Flush+Reload performs the worst with attack times of 614.1 cycles (CPU SR) and 462.8 cycles (CPU ER) with no victim access (see Figure 6.5a and Figure 6.14a).

The attack time for cross-core DemoteContention is the same as for Demote+Demote in the case of no victim access. The victim access case is challenging to evaluate as cross-core DemoteContention is based on contention with a huge blind spot and high noise. Despite this, the attack time with a victim access is only slightly longer than in the case of no victim access, as shown in Figure 6.3d and Figure 6.13d.

The times for the case with no victim access for all attacks are summarized in Table 6.1. With a lower attack time, an attacker can probe the cache more frequently and, therefore, can detect memory accesses that are closer to each other. This is particularly useful in attacks that benefit from additional information, such as fingerprinting attacks. While a low attack time is advantageous, the attack performance can still be worse as cache activity may be missed due to blind spots.

Cache side-channel measurements are typically destructive in the sense that they destroy the previous state of the microarchitectural element. Restoring the previous state takes time, during which a victim's operation may be missed. Therefore, this observer effect is called "blind spot" and has been studied as a limiting factor for attacks [23, 57]. To measure the size of the blind spot, we let one thread continuously execute the attack.

After a random number of cycles, a second thread accesses the memory location, which the first thread monitors. We log whether the attack detected the memory access. We repeat this measurement 1 000 times, resulting in a percentage of successfully detected memory accesses. We use this result as an approximation for the size of the blind spot a given attack has relative to its execution time. Furthermore, to show the effect the wait time after each attack iteration can have on the blind spot, we ran our evaluation for different sleep or delay periods after each attack execution. We additionally combine these results with the previously measured values for the attack time with no victim access to compute an estimate for the absolute blind-spot size in cycles. Similar to previously discussed metrics, the blind spot for all attacks is very similar for both of our tested CPUs. The sample size for all measurements is 200, and the standard error is $\leq 1\%$.

Our blind-spot evaluation results are shown in Figure 6.6 and Figure 6.15. We use a blind-spot percentage instead of a cycle estimate, as it directly represents the percentage of victim accesses that an attacker might miss and visualizes the effect a delay after each attack iteration has on the blind spot. Demote+Demote and Flush+Flush are excluded from this plot, as we did not observe a blind spot for both of them, resulting in a blind-spot size of $\approx 0\,\%$ regardless of the delay length. The blind spot for all attacks, except for cross-core DemoteContention, decreases with an increase in the delay length, approaching $\approx 0\,\%$. This is the expected behavior as a longer delay decreases the likelihood of the victim accessing the memory in the blind spot of the attack. Cross-core DemoteContention is the only exception, as it relies on contention with the victim. For cross-core DemoteContention, the victim and attacker have to access the same L3 cache set at roughly the same time. An increase in the delay length after each attack iteration decreases the likelihood for the attacker to trigger the contention, increasing the blind-spot size, as shown in Figure 6.6 and Figure 6.15. Ideally, the blind-spot size for cross-core DemoteContention would approach 1. However, the small attack margin, as discussed in Section 4.1 of cross-core DemoteContention, results in a high amount of noise, which, in turn, results in a high amount of false positives compared to the other tested attacks. These false positives lead to an underestimation of the blind spot. Also, the relative blind-spot size for Prime+Probe and Evict+Reload only slightly decreases over time as they are more susceptible to noise.

The full list of blind-spot sizes is provided in Table 6.1. We will discuss the results of Xeon Silver 4410T here, as the results from both CPUs are almost identical. The attacks with the lowest blind-spot size relative to the attack time are Demote+Demote and Flush+Flush, with ≈0 % on both tested CPUs. Next is Evict+Reload with 21.4 %, followed by Prime+Probe with 23.9 %. SMT Flush+Reload has a blind-spot length of 75.1 %. Cross-core Flush+Reload has a slightly higher blind spot than the SMT variant, with 89.9 %. The smaller blind spot for the SMT Flush+Reload variant could result from the CPU core merging loads. Demote+Reload has a blind-spot length of only 42.8 %, performing significantly better than the similar Flush+Reload. Finally, cross-core DemoteContention has a blind-spot size of 90.1 %.

The blind spots measured show that a delay between attack iterations can be vital for an effective attack. Especially for Flush+Reload, not using a delay makes the attack significantly less useful. While the blind spot for all listed attacks, except for DemoteContention, can be counteracted by a sufficiently large delay between attack iterations, adding this delay decreases the frequency in which the attacker can probe the cache, potentially losing information if the victim frequently accesses the monitored cache lines. When combining the attack times with the blind spot, Demote+Demote is optimal for SMT attacks, as it has no blind spot and the lowest attack time, and Flush+Flush is optimal for a cross-core attacker due to its also low attack time and non-existent blind spot.

4.5. Channel Capacity and Noise Resilience

The channel capacity is a standard evaluation metric for side channels. While prior work reported capacities for most of these side channels, comparing covert-channel capacities across different systems can be misleading as the channel capacity can be significantly influenced by the general performance of a CPU or its memory subsystem. Consequently, we take a different approach where we construct a simple covert channel that can be instantiated generically with any of the side channels we discuss. The basic construction uses time slices to transmit one or more bits through the channel. To provide a fair comparison, we assume perfect synchronization of the first time slice used by the channel. We use two threads for each channel, one receiver, and one sender thread. Each channel capacity listed in this section has a sample size of 100 and a standard error of the mean of ≤0.2 Mbit/s.

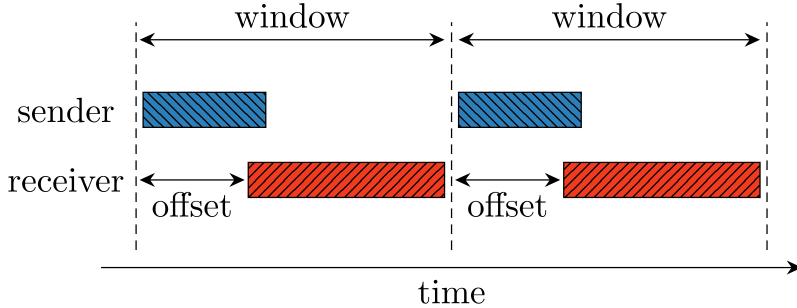


Figure 6.7.: Our basic covert channel construction and the dimensions to optimize. The `window` parameter defines the size of a single transmission window, and `offset` defines the head start the sender gets to prepare, e.g., access memory locations.

Table 6.2.: Single-bit (1-bit) and multi-bit (n-bit) covert-channel true capacity in **Mbit/s** and error ratio of all tested attacks on an Intel Xeon Silver 4410T.

Attack	Cap. (1-bit)	BER (1-bit)	Cap. (n-bit)	BER (n-bit)
Opt. Demote+Reload	11.47	0.2 %	15.48	2.0 %
Opt. Flush+Reload	5.42	0.7 %	9.42	0.3 %
Opt. Flush+Reload (SMT)	5.58	1.8 %	9.17	0.6 %
Demote+Reload	6.38	0.7 %	6.38	0.7 %
Demote+Demote	6.03	0.7 %	8.34	4.6 %
DemoteContention	0.18	5.9 %	0.18	5.9 %
Flush+Reload	1.94	4.9 %	1.94	4.9 %
Flush+Reload (SMT)	2.35	3.4 %	2.35	3.4 %
Flush+Flush	4.43	2.7 %	8.80	1.2 %
Flush+Flush (SMT)	4.56	1.1 %	8.55	5.6 %
Prime+Probe (L1)	3.62	10.9 %	3.62	10.9 %
Evict+Reload (L1)	2.51	6.8 %	2.51	6.8 %

This simple channel has two dimensions to optimize (Figure 6.7): First, the window length. Reducing the window length increases how many slices fit in a time frame, *i.e.*, increasing the transmission rate. Second, the offset the receiver has from the sender. The offset prevents receiving too early before the sender has sent the data, which could result in incorrect detections. While the error ratio can also be a parameter to optimize for, we avoid this additional dimension by working with the true capacity computed from the raw capacity and the error ratio. This way, we optimize for the optimal trade-off between error ratio and raw capacity. Furthermore, we can adjust the number of bits sent within a window to further optimize the true capacity.

Our approach to finding the optimal parametrization of the covert channel is to start with a single bit per window and reduce the receiver offset. As the next step, we decrease the window length. We optimize each attack for an increasing number of bits sent in parallel to maximize the true capacity. For some attacks, e.g., cross-core DemoteContention, the optimal capacity is achieved with a single bit per window.

Standard Demote+Reload and Flush+Reload variants are not optimized for fast covert channel transmission, as the receiver always accesses memory and afterward flushes or demotes it. As receiver and sender fully cooperate to transmit data, we tested optimized versions for the data transmissions of the two attacks in addition to the standard versions. With Demote+Reload, the sender demotes a cache line to send a '1' and does nothing to send a '0'. The receiver measures the access time to the memory location to detect if the cache line has been demoted. With Flush+Reload, the sender flushes a cache line to send a '1' and does nothing to send a '0'. The receiver measures the access time to the memory location to detect if the cache line was flushed. These approaches minimize the memory interactions, requiring only the receiver to access memory, resulting in significantly higher speeds.

The results for each side channel are provided in Table 6.2 (CPU SR) and Table 6.9 (CPU ER). With one bit per window, optimized Demote+Reload performs the best with 11.47 Mbit/s (CPU SR) and 11.10 Mbit/s (CPU ER). Demote+Reload and Demote+Demote perform similarly on our CPU SR with 6.38 Mbit/s and 6.04 Mbit/s, respectively. On our CPU ER, Demote+Demote is faster with 8.17 Mbit/s compared to Demote+Reload 6.09 Mbit/s, presumably due to microarchitectural changes. Optimized cross-core Flush+Reload has capacities of 5.42 Mbit/s (CPU SR) and 6.51 Mbit/s (CPU ER). SMT Flush+Flush performs significantly worse

than Demote+Demote using one bit per window at 4.56 Mbit/s (CPU SR) and 5.03 Mbit/s (CPU ER). DemoteContention performs the worst with 0.18 Mbit/s (CPU SR) and 0.19 Mbit/s (CPU ER). However, without reverse-engineering L3 addressing functions and replacement policies, this is the only cross-core attack available that does not require shared memory.

Sending more than one bit per window increases the capacities of optimized Demote+Reload to 15.48 Mbit/s (CPU SR) and 17.03 Mbit/s (CPU ER) using 12 bits, resulting in the overall fastest channel. Optimized cross-core Flush+Reload has capacities of 9.17 Mbit/s (CPU SR) and 10.01 Mbit/s (CPU ER) using 12 with SMT Flush+Reload performing similarly. Cross-core Flush+Flush increases to 8.80 Mbit/s (CPU SR) and 10.26 Mbit/s (CPU ER) using 800 bits with SMT Flush+Flush performing similarly. Flush+Flush performs similarly on CPU SR and slightly better on CPU ER using multiple bits. While in a single-bit transmission scenario, Flush+Flush is bottlenecked by the L3 misses of the sender, these delays can be hidden by sending multiple bits at once, leading to the similar performance of the two channels as the execution times of `cldemote` and `clflush` are similar (see Section 4.1). Demote+Demote increases to 8.34 Mbit/s (n=100, $\sigma_{\bar{x}}$=0.008) using 1 000 bits. The remaining attacks' channel capacity did not increase with more than one bit per transmission window.

We measure the noise resilience by starting on a completely idle system and measuring the true capacity of the optimized 1-bit covert channels, representing a typical attack scenario. We gradually increase the system load from 0 to one worker per logical CPU core, using cache thrashing worker threads from the stress-ng test suite. As shown in Figure 6.11 and Figure 6.12, all attacks suffer from noise.

Optimized Demote+Reload drops significantly until 10 workers and afterward decreases slowly to 5.24 Mbit/s (≈−56 %) on our CPU SR (see Figure 6.11a) and 2.23 Mbit/s (≈−80 %) on our CPU ER (see Figure 6.12a), performing the best. Cross-core DemoteContention performs the worst, dropping drastically, reaching almost 0 kbit/s at 10 (≈−100 %) on both tested CPUs. The remaining attacks perform similarly to each other, dropping slowly until they lose ≈80 % to 95 % of their channel capacity with the maximum number of workers on both CPUs, as shown in Figure 6.11 and Figure 6.12.

While the covert channel capacity is a metric to show how fast information can be transmitted covertly using a given attack, it also demonstrates how much information each channel can theoretically leak in a given time.

The 1-bit scenario is the typical attack scenario, where an attacker wants to monitor a single memory location. When monitoring a single memory location, Demote+Demote performs the best with a capacity similar to Demote+Reload on our CPU SR and outperforming it on CPU ER. This makes Demote+Demote the better choice in such a scenario over other attacks when attacking over other attacks such as Flush+Flush and Flush+Reload when targeting a victim over SMT. When monitoring many memory locations, Flush+Flush performs slightly better than Demote+ Demote while also working across cores.

4.6. Detectability

Cho et al. [11] recently developed a detection scheme using `mem_load_retired.l1_miss` (L1 miss), `mem_load_retired.l2_miss` (L2 miss), `mem_load_retired.l3_miss` (L3 miss), and `br_inst_retired.all_branches` (retired branches). This is similar to the approaches of Gulmezoglu et al. [26] and Gruss et al. [23]. To evaluate the detectability of the different attacks, we focused on the performance counters selected by Cho et al. [11].

We evaluated all attacks with no victim access for 10^6 iterations, as shown in Table 6.5 (CPU SR) and Table 6.7 (CPU ER). While the performance counter values in the base cases of the two CPUs are different, possibly due to the different CPU, kernel version, and compiler version, the changes in values for each attack are very similar. SMT Flush+Flush, cross-core Flush+Flush, Demote+Demote, Prime+Probe and DemoteContention are indistinguishable from the base cases (no attack being executed), with the variations mainly resulting from noise on both CPUs. As these attacks do not induce any accesses by themselves, they do not induce any L1, L2, or L3 misses without accesses from somewhere else. Evict+Reload significantly increases the L1 misses by $\approx 13 \cdot 10^6$, as it constantly *reloads* and evicts the victim cache line and the eviction set. SMT Flush+Reload increases the L1, L2, and L3 misses by $\approx 10^6$, as for every iteration, the victim cache line is loaded from the DRAM and flushed from all caches. Cross-core Flush+Reload increases the L1 and L2 misses by $\approx 2 \cdot 10^6$, and the L3 misses by $\approx 10^6$, as the cache line is loaded from DRAM once per iteration and then loaded into the victim and attacker cores. Demote+ Reload has $\approx 10^6$ extra L1 and L2 misses but does not increase the L3 misses, as the victim cache line is only moved between the L1 and the L3.

The number of retired branches is unchanged from the base case for each attack, as no attacks perform extra branches.

The results with a victim access are shown in Table 6.6 and Table 6.8, again with 10^6 iterations. The L1, L2, and L3 misses for DemoteContention do not change compared to no victim access, as the attack does not rely on a cache line being cached, and it does not directly interact with the victim cache line. This makes cross-core DemoteContention indistinguishable from running no attack. Evict+Reload performs similarly to the case of no victim access and increases the L1 misses by $\approx 13 \cdot 10^6$ compared to the base case and is otherwise indistinguishable from running no attack. Prime+Probe performs almost identically to Evict+Reload. SMT Flush+Reload performs almost identically with no victim access. Cross-core and SMT Flush+Flush perform the same as cross-core and SMT Flush+Reload, respectively, as in these attacks, the victim loads the cache line, and the attacker evicts it. Demote+Reload and Demote+Demote perform almost identically with $\approx 3.7 \cdot 10^6$ L1 and L2 misses, and no additional L3 misses. The number of retired branches is unchanged from the base case for each attack, as no attacks perform any extra branches.

Overall, DemoteContention is the only attack that can not be detected using any performance counters tested, as it does not trigger memory accesses or modify the state of victim cache lines. All other attacks result in more L1 misses (Prime+Probe and Evict+Reload), more L1 and L2 misses (Demote+Reload and Demote+Demote), or more L1, L2, and L3 misses (Flush+Flush, Flush+Reload). Still, Demote+Demote and Flush+Flush have the advantage of being indistinguishable from the base case if no victim access is performed, making them challenging to detect with low-frequency victim accesses.

5. Demote+Reload Attack Case Studies

In this section, we evaluate our attacks in multiple case studies: First, we compare Demote+Reload and Demote+Demote to existing attacks on an AES T-tables attack and an inter-keystroke timing attack. Second, we demonstrate that `cldemote` does not only result in leaks through access times but also through power consumption [36]. Finally, we show that `cldemote` can break KASLR, even on systems that do not officially support `cldemote`.

5.1. Attacking OpenSSL AES T-tables

A standard benchmark for cache side-channel attacks is OpenSSL AES T-tables. While the T-table implementation is not used anymore by OpenSSL, it is a common means to compare cache attacks in a realistic cryptographic attack scenario, *i.e.*, repeatable high-frequency attacks to accumulate leakage. We mounted a last-round attack following the approach by Irazoqui et al. [34]. All attacks were run in a same-core scenario, *i.e.*, the attacker triggers the encryption (with an inaccessible key) and then mounts the cache attack after the encryption. For statistical significance, we perform 1 000 key recoveries, corresponding to at least 10 million runs of each of the attacks. The success rate of the key recovery increases with the number of samples, *i.e.*, even with noisy channels the attack typically converges to the correct key with sufficient samples. An attacker can minimize the attack runtime by minimizing the number of encryptions until the success rate is not 100 % anymore. Hence, this is a good metric for comparison: We minimize the number of encryptions for each attack until we reach a 97 % to 99 % correct key guess range. As seen in Table 6.3, `sched_yield`, while costing > 100 cycles, can help improve the attack performance, as it indicates to the operating system that this is a cooperative thread: With default configuration, the kernel then less frequently preempts (interrupts) the thread. Furthermore, it increases the chance of running an idle thread, reducing power dissipation and thermal emissions, and hence, throttling effects.

We evaluated the use of `sched_yield` for all attacks. For Flush+Reload and Demote+Reload, we only observed a significant slow-down with `sched_yield`, but no significant increase in attack accuracy, resulting in a significantly higher runtime. We provide the numbers for all 6 attack techniques, including the `sched_yield`-optimized variants that resulted in competitive performance, in Table 6.3. For Prime+Probe, we did not reach a 97 % to 99 % correct key guess range, even with 2 orders of magnitude higher numbers of encryptions, and instead evaluated it for 1 million encryptions.

As shown in Table 6.3, Demote+Reload results in the overall lowest attack runtime, with 14.5 ms and 98.7 % of the key guesses correct. The number of encryptions required to reach this percentage is the same as with Flush+Reload, 11 000. This is on par with state-of-the-art key recovery attacks on AES that require, e.g., 6 000 to 10 000 encryptions [67, 34, 6]. However, Flush+Reload is 24 % slower than Demote+Reload. Only Demote+Demote

Table 6.3.: Comparison of attack techniques on Sapphire Rapids. Demote+Reload results in the lowest overall attack runtime.

Attack	Yield	Correct	Encrypt.	Runtime
Demote+Reload	✗	98.7 %	11 000	14.5 ms
Flush+Reload	✗	97.9 %	11 000	18.0 ms
Demote+Demote	✓	99.2 %	9 000	20.9 ms
Demote+Demote	✗	97.1 %	15 000	22.6 ms
Flush+Flush	✗	99.3 %	18 000	38.1 ms
Flush+Flush	✓	98.8 %	16 000	42.2 ms
Evict+Reload	✗	97.2 %	470 000	529.8 ms
Evict+Reload	✓	97.2 %	320 000	554.0 ms
Prime+Probe	✓	66.7 %	1 000 000	1 320.5 ms

Table 6.4.: Comparison of attack techniques on Emerald Rapids. Demote+Reload results in the lowest overall attack runtime. Yielding did not improve the attack performance for any of the attacks.

Attack	Yield	Correct	Encrypt.	Runtime
Demote+Reload	✗	98.6 %	13 000	16.7 ms
Flush+Reload	✗	99.2 %	13 000	20.4 ms
Demote+Demote	✗	99.2 %	16 000	23.6 ms
Flush+Flush	✗	99.2 %	20 000	37.8 ms
Evict+Reload	✗	97.2 %	360 000	1 263.9 ms
Evict+Reload	✓	97.2 %	320 000	1 369.8 ms
Prime+Probe	✓	53.4 %	1 000 000	3 246.6 ms

with sched_yield worked with a lower number of encryptions, namely 9 000, yet had a higher overall runtime due to the runtime overhead of sched_yield. Without sched_yield, there is more noise, requiring 15 000 encryptions to get to the same correct key rate. This is not unexpected as Demote+Demote has virtually no blind spot, and any interrupt will result in a timing deviation that is noise for the attack. For Flush+Flush, we observe a similar situation, where adding a sched_yield allows us to reduce the number of encryptions from 18 000 to 16 000, but this is not enough to compensate the runtime cost of sched_yield, resulting in the highest attack runtime in our test. Evict+Reload and Prime+Probe (on the L1) required the highest number of traces for key recovery. As we focus on the L1, the timing difference between hit and miss is very low (< 10 cycles), and thus, it is more susceptible to noise due to subtle timing variations, bringing up the attack runtime. Without sched_yield, the number of encryptions required is higher, but the overall runtime is lower. For a visual comparison of the three attacks, we provide a zero-key first-round attack matrix in Figure 6.8. The x-axis corresponds to the cache lines making up the T-Table, and the y-axis corresponds to the plaintext byte. A darker color means more detected accesses. A visible diagonal means that an attack identified the zero key correctly. The less visible the diagonal, the harder it is to identify the key with a given attack.

On the Emerald Rapids (Table 6.4), we find that the positive effects of sched_yield disappear largely. Only for Evict+Reload and Prime+Probe, we reaquire a lower number of encryptions with sched_yield. The number of encryptions required and attack runtimes overall increased slightly, as we see slightly more noise on this system. Still, Demote+Reload and Demote+Demote have high performance, comparable to the other state-of-the-art attacks.

5.2. Inter-Keystroke Timing Attack

Another standard benchmark for cache side channels is to mount an inter-keystroke timing attack, which can be used to infer the actual key press values and written text [69, 50]. Keystroke attacks are an excellent means to compare cache attacks in scenarios of low-frequency, non-repeatable attacks, where leakage cannot be easily accumulated, but the accuracy of a single attack is more relevant. For our attack, we first used a template attack to identify leaky offsets in a shared library using Flush+Reload [25]. Afterward, we use the same offset in the shared library for

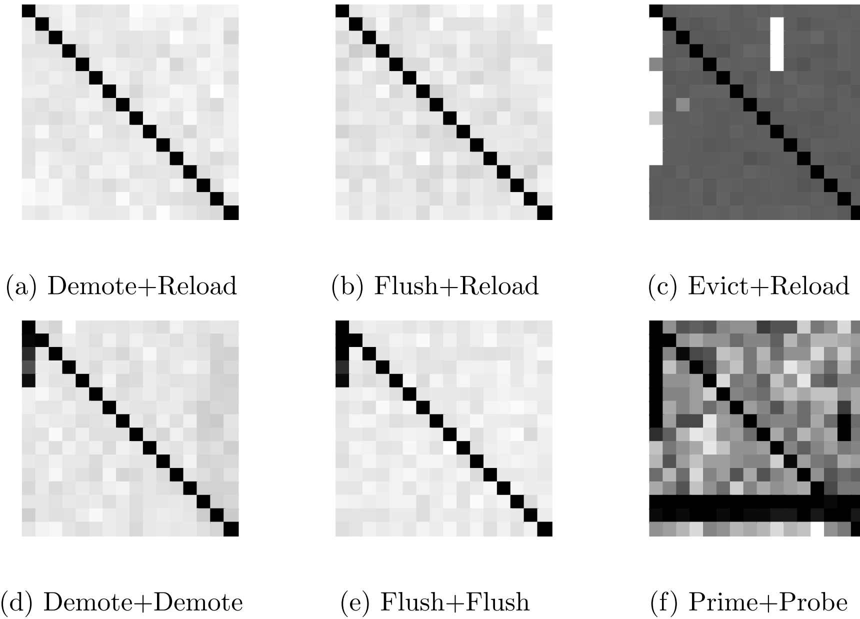


Figure 6.8.: Comparison of different attack techniques over 10 000 encryptions using a zero key. The y-axis is the plaintext byte value, and the x-axis is the T-table cache lines. A darker shade means more cache line accesses. A visible diagonal shows that the attack correctly identified the zero key.

Flush+Reload and Demote+Reload to compare their temporal precision and noise resilience.

For both our Flush+Reload attack and our Demote+Reload attack, we let a human perform 1000 keystrokes, typing into a program over multiple minutes to obtain the ground truth for the timings. In parallel, we run one of the two attacks, which have exactly the same implementation except for swapping the `clflush` and `cldemote` instructions and the corresponding hit-miss threshold. We observe that for both Flush+Reload and Demote+Reload, there are no false positive detections. This highlights the noise resilience of both attacks. With Flush+Reload, we ran ≈2.8 million, and with Demote+Reload ≈3.6 million attacks per second, with not a single measurement resulting in a false positive detection. For Demote+Reload, we observed no false negatives, and for Flush+Reload, only a single one, *i.e.*, Flush+Reload detected 999 out of 1000 keystrokes correctly. That is, in terms of the F-Score measure, which is often used to assess the accuracy

of keystroke detection [25, 23, 51], we achieve an F-Score of 1 in this experiment for Demote+Reload and 0.999 5 for Flush+Reload, showing that the attacks are on par. For the temporal precision, we observe a mean delta between the ground truth and the recovered timing of 288.5 ns (n=1000, $\sigma_{\bar{x}}$=3.95 ns) with Flush+Reload and 233.5 ns (n=1000, $\sigma_{\bar{x}}$=2.73 ns) with Demote+Reload. While an attacker can account for the mean delta, the standard error $\sigma_{\bar{x}}$ remains an inaccuracy. Thus, we can conclude that Demote+Reload has roughly 30 % more temporal precision in this attack scenario. It is essential to highlight that timing variations of humans are orders of magnitude higher (*i.e.*, in the millisecond range) [40], than either of the two attacks.

On Emerald Rapids, the noise resilience is unchanged, without a single false positive within 5.8 million Demote+Reload and 3.6 million Flush+Reload attacks per second. However, it appears that `sched_yield` has a more pronounced effect on the blind spot now. While for Demote+Reload, the number of false negatives remains at zero without `sched_yield`, yielding a strictly better attack performance. For Flush+Reload, omitting `sched_yield` increases the false negative rate from 1.5 % to 3.4 % and lowers the F-Scores correspondingly from 0.992 to 0.983. The temporal precision increases slightly to a mean delta of 245.1 ns (n=1000, $\sigma_{\bar{x}}$=3.54 ns) with Flush+Reload (with `sched_yield`), 182.8 ns (n=1000, $\sigma_{\bar{x}}$=3.23 ns) with Flush+Reload (without `sched_yield`), and 134.9 ns (n=1000, $\sigma_{\bar{x}}$=1.99 ns) with Demote+Reload. This is in line with the other observations, indicating a higher attack performance on Emerald Rapids, where some attacks are affected by noise more than on Sapphire Rapids.

5.3. Collide+Power

Collide+Power [36] exploits the power leakage of data collisions in the memory subsystem between security domains. These microarchitectural data collisions occur since the memory subsystem is shared between the attacker and victim domain, including caches and buses connecting the distinct cache levels. Therefore, data blocks traveling over the shared components immediately after another expose their Hamming distance, *i.e.*, the number of different bits, in the power domain, resulting in an exploitable signal to recover the actual data. We focus on a specific leakage effect of Collide+Power, the so-called *self-leakage* [36]: Here, the Hamming distance between the upper and lower 32 B of a cache line leak to one

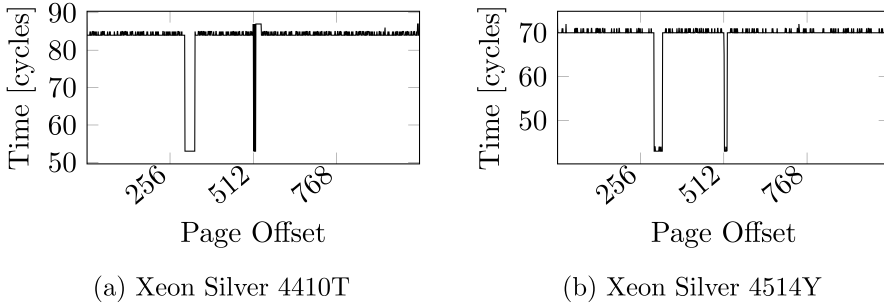
another when the cache line is moved into the L3 cache, reflecting the behavior of `cldemote`.

We show that `cldemote` amplifies the *self-leakage* effect for software-based power-analysis attacks when data is frequently demoted into the L3 cache, e.g., to make the data used in a multithreaded context visible to other physical cores faster as recommended by the Intel guidelines [30]. We measure the power consumption via the package domain of the Running Average Power Limit (RAPL) interface. We repeat a single store instruction once with and once without the added optimization in a loop for $\approx$ 12 ms to record a single measurement sample. The analysis framework of Collide+Power uses the recorded power samples to fit the power model $P(U, L) = x \cdot hd(U, L)$, where the coefficient x indicates the strength of the Hamming distance leakage between the upper (U) and lower (L) 32 B of the cache line.

We record 1.2 million samples per case and estimate a power leakage of $x = 310\,\mu\text{W}$ per bit difference between U and L when using `cldemote` on our Xeon Silver 4410T. For our Xeon Silver 4514Y, we record 3.5 million samples with an estimated power leakage of $x = 119\,\mu\text{W}$. In contrast, we do not observe a significant power leakage without the optimization. Finally, we compute the Pearson correlation coefficient of the power model and the measurements, resulting in correlations of 0.012 (Xeon Silver 4410T) and 0.004 (Xeon Silver 4514Y) for the case with `cldemote` and no significant correlation without the instruction. Therefore, using `cldemote` as an optimization to improve cross-core access latencies, as recommended by Intel [30], increases the attack surface for software-based power-analysis attacks.

5.4. Breaking KASLR

In this section, we demonstrate a KASLR break using the `cldemote` instruction on Ubuntu 22.04 LTS (Linux 6.2.0) on our Xeon Silver 4410T and Ubuntu 24.04 LTS (Linux 6.8.0) on our Xeon Silver 4514Y. We exploit the absence of faults by `cldemote` and its' TLB-dependent timing behavior, *i.e.*, `cldemote` needs to translate an address before determining if the demote operation is valid. KASLR is a low-cost security mechanism, randomizing kernel code and data addresses. Due to many KASLR breaks [28, 21, 20, 9, 38], the security value may be limited, but it is now a typical benchmark for new microarchitectural attacks. The TLB-induced timing



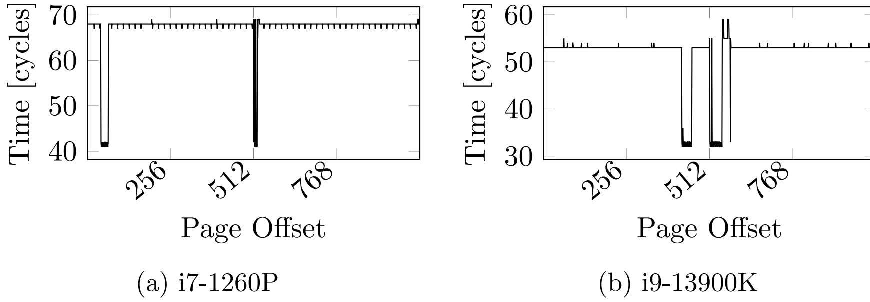
(a) Xeon Silver 4410T (b) Xeon Silver 4514Y

Figure 6.9.: Execution time of `cldemote` over the Kernel Text segment KASLR range for all 2 MB pages. The mapped regions show a significant drop in execution time, revealing the KASLR offset to an attacker at page offset 302 on our Xeon Silver 4410T and offset 298 on our Xeon Silver 4514Y.

difference we exploit can be used to detect if an address has a cached translation even if the address is not accessible, e.g., a kernel address. We assume the attacker can execute arbitrary unprivileged code on the victim system and has access to a high-precision timer such as the TSC but no access to privileged interfaces (e.g., `/proc/self/pagemap`).

As large parts of the kernel are regularly accessed through interrupts, syscalls, context switches, and other kernel activities, they have TLB entries. The `cldemote` timing for the first cache line of each 2 MB page starting at `0xffffffff80000000` covering the kernel binary mapping region is shown in Figure 6.9. For our Xeon Silver 4410T (Sapphire Rapids), shown in Figure 6.9a, the execution time for invalid addresses is ≈85 cycles. The dips to ≈53 cycles (starting at offset 302, virtual address `0xffffffffa5c00000`) indicate mapped pages. For our Xeon Silver 4514Y (Emerald Rapids), shown in Figure 6.9b, the execution time for invalid addresses is ≈70 cycles. The dips to ≈43 cycles (starting at offset 298, virtual address `0xffffffffa5400000`) indicate mapped pages. We verified these mappings for both CPUs through `/proc/kallsyms`. Scanning the kernel binary KASLR range multiple times to account for noise to find the base address takes <10 ms on both CPUs.

The `cldemote` is currently only supported on recent Xeon microarchitectures. On all other CPUs, `cldemote <register>` is interpreted as a `nop` that dereferences the register. Indeed, `cldemote` does not exhibit any unexpected timing behavior on any 10th-generation or older Intel



(a) i7-1260P

(b) i9-13900K

Figure 6.10.: Execution time of `cldemote` over the Kernel Text segment KASLR range for all 2 MB pages on two CPUs that do not support the instruction. The mapped regions show a significant drop in execution time on both CPUs, even though the instruction should be interpreted as a `nop`.

Core CPUs we tested. However, for our Alder Lake i7-1260P CPU and Raptor Lake i9-13900K CPU, the instruction has a timing behavior similar to the Xeon Silver 4410T when used on mapped but inaccessible kernel addresses. Even though this timing behavior for `cldemote` exists on these unsupported CPUs, `cldemote` does not trigger a cache line demotion on valid memory locations. The instruction triggers only a memory translation on these CPUs and does not further influence program behavior. This behavior seems specific to the `cldemote` op-code, as we could not find another `nop` instruction that exhibits the same timing behavior. The `cldemote` timings for both CPUs are provided in Figure 6.10. Similar to the Sapphire Rapids CPU, there are clear drops in execution time at the page offsets where the kernel is mapped, which is offset 43 (virtual address `0xffffffff85600000`) for our i7-1260P in Figure 6.10a, and offset 427 (virtual address `0xffffffffb5600000`) for our i9-13900K in Figure 6.10b.

The other attacks discussed in Section 4 can not be used to leak the full KASLR offset. Only Prime+Probe and DemoteContention can be used to leak the bits that are used for determining the cache set, but not the full address. This is a distinct advantage of Demote+Demote over other attacks.

6. Discussion and Mitigations

Cache attacks can be *prevented* at three levels: at the hardware level, at the system level, and finally, at the application level. At the hardware level, several solutions have been proposed to prevent cache attacks, either by removing cache interferences, or randomizing them. The solutions include new secure cache designs [80, 79, 44] or altering the prefetcher policy [16]. However, hardware changes are not applicable to commodity systems. At the system level, page coloring provides cache isolation in software [58, 35]. Zhang et al. [89] proposed a more relaxed isolation like repeated cache cleansing. These solutions cause performance issues, as they prevent optimal use of the cache. Application-level countermeasures seek to find the source of information leakage and patch it [5]. However, application-level countermeasures are bounded and cannot prevent cache attacks such as covert channels. In contrast to prevention solutions that incur a loss of performance, using performance counters does not prevent attacks but rather detects them without overhead.

7. Conclusion

Finding new generic cache attack techniques is crucial to understanding the attack surface of modern CPUs. We present three new attacks, Demote+Reload and Demote+Demote, that rely on the newly introduced `cldemote` instruction. We provide the first systematic evaluation of 9 characteristics of the most relevant cache attacks and our newly introduced attacks. We showed that Demote+Reload and Demote+Demote offer advantages on some characteristics, such as the blind spot and attack duration, and the high channel capacity of 15.48 Mbit/s. We performed further benchmarks, including AES T-table key recovery, an inter-keystroke timing attack, a fast KASLR break, and an amplified Collide+Power attack. This shows that our new attack techniques are an important extension and complement to the existing generic techniques.

Acknowledgments

We thank the anonymous reviewers and our anonymous shepherd for their guidance, comments, and suggestions. This research is supported in part

by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I6054). Additional funding was provided by a generous gifts from Red Hat, 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] Onur Acıiçmez, Werner Schindler, and Cetin K. Koc. Cache Based Remote Timing Attack on the AES. In: CT-RSA. 2006 (p. 122).

[2] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In: CCSW. 2010 (p. 122).

[3] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (p. 122).

[4] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P. 2016 (p. 123).

[5] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. In: Cryptology ePrint Archive, Report 2006/052 (2006) (p. 152).

[6] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M Moya. Cache misses and the recovery of the full AES 256 key. In: Applied Sciences 9.5 (2019), p. 944 (p. 144).

[7] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: S&P. 2019 (p. 123).

[8] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In: USENIX Security. 2003 (p. 122).

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (p. 149).

[10] 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) (p. 123).

[11] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. Real-time detection for cache side channel attack using performance counter monitor. In: Applied Sciences 10.3 (2020), p. 984 (pp. 118, 119, 142).

[12] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In: CCS. 2014 (p. 123).

[13] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. On the effectiveness of same-domain memory deduplication. In: European Workshop on Systems Security. 2022 (p. 123).

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In: USENIX Security. 2017 (p. 122).

[15] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In: USENIX Security. 2013 (p. 123).

[16] Adi Fuchs and Ruby B. Lee. Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In: SYSTOR. 2015 (p. 152).

[17] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. Efficient and generic microarchitectural hash-function recovery. In: S&P (2024) (pp. 125, 128).

[18] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In: USENIX Security. 2023 (p. 118).

[19] Michael Misiu Godfrey and Mohammad Zulkernine. Preventing Cache-Based Side-Channel Attacks in a Cloud Environment. In: IEEE Transactions on Cloud Computing (2014) (p. 123).

[20] 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. 149).

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 118, 126, 149).

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 122, 125).

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 118, 119, 121, 123, 128, 136, 142, 148).

[24] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: USENIX Security. 2017 (p. 122).

[25] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 118, 121, 123, 126, 128, 146, 148).

[26] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. In: arXiv:1907.03651 (2019) (pp. 118, 119, 142).

[27] 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 (p. 123).

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 149).

[29] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In: CHES. 2016 (p. 121).

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 2023 (p. 149).

[31] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (p. 123).

[32] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. In: Cryptology ePrint Archive, Report 2016/1196 (2017) (p. 123).

[33] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P. 2015 (p. 122).

[34] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID. 2014 (p. 144).

[35] 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 (pp. 123, 152).

[36] 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. 143, 148).

[37] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In: USENIX Security. 2016 (p. 122).

[38] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In: EuroS&P. 2020 (p. 149).

[39] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In: S&P. May 2020 (p. 122).

[40] Po-Ming Lee, Wei-Hsuan Tsui, and Tzu-Chien Hsiao. The Influence of Emotion on Keyboard Typing: An Experimental Study Using Auditory Stimuli. In: PLOS ONE 10 (2015), pp. 1–16 (p. 148).

[41] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In: USENIX Security. 2022 (p. 118).

[42] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (pp. 118, 122).

[43] 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. 123).

[44] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In: MICRO. 2014 (p. 152).

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 122).

[46] Robert Martin, John Demme, and Simha Sethumadhavan. Time-Warp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: ACM SIGARCH Computer Architecture News (2012) (p. 122).

[47] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015 (pp. 125, 126).

[48] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 121).

[49] 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 (p. 125).

[50] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 146).

[51] John V Monaco. Feasibility of a Keystroke Timing Attack on Search Engines with Autocomplete. In: IEEE Security and Privacy Workshops (SPW). 2019 (p. 148).

[52] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 118, 122, 128).

[53] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In: ESSoS. 2016 (p. 123).

[54] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (p. 122).

[55] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make Sure DSA Signing Exponentiations Really Are Constant-Time. In: CCS. 2016 (p. 122).

[56] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In: S&P. 2021 (pp. 118, 122).

[57] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (pp. 122, 136).

[58] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource Management for Isolation Enhanced Cloud Services. In: CCSW. 2009, pp. 77–84 (p. 152).

[59] 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. 126).

[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. 119).

[61] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security. 2021 (p. 118).

[62] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 122).

[63] 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. 122).

[64] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 122).

[65] Martin Schwarzl, Erik Kraft, and Daniel Gruss. Layered Binary Templating. In: ACNS. 2023 (p. 123).

[66] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote Page Deduplication Attacks. In: NDSS. 2022 (p. 123).

[67] Milad Seddigh and Hadi Soleimany. Enhanced Flush+Reload Attack on AES. In: ISC International Journal of Information Security (2020) (p. 144).

[68] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In: Dependable Systems and Networks Workshops (DSN-W). 2011 (p. 123).

[69] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security. 2001 (p. 146).

[70] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In: EuroSys. 2011 (p. 123).

[71] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security. 2017 (p. 118).

[72] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In: CCS. 2015 (p. 122).

[73] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In: USENIX Security. 2020 (pp. 122, 123).

[74] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In: CCSW. 2011 (p. 122).

[75] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In: S&P. 2019 (p. 125).

[76] Steve Walton. How Screwed is Intel without Hyper-Threading? 2019. URL: `https://www.techspot.com/article/1850-how-screwed-is-intel-no-hyper-threading/` (p. 122).

[77] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In: NDSS. 2019 (p. 118).

[78] 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. 122).

[79] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In: MICRO. 2008 (p. 152).

[80] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH Computer Architecture News 35.2 (2007), p. 494 (p. 152).

[81] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security. 2019 (p. 118).

[82] John C Wray. An Analysis of Covert Timing Channels. In: Journal of Computer Security (1992) (p. 122).

[83] 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. 126).

[84] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 118, 121, 123, 128).

[85] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. In: Cryptology ePrint Archive, Report 2015/905 (2015) (p. 121).

[86] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In: JCEN (2017) (p. 122).

[87] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In: RAID. 2016 (p. 123).

[88] 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 (p. 123).

[89] Yinqian Zhang and MK Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: CCS. 2013 (pp. 123, 152).

Table 6.5.: Performance counter values for 10^6 runs of each tested attack without a victim access on our Xeon Silver 4410T.

Attack	L1 misses	L2 misses	L3 misses	Retired Branches
Base (SMT)	2 544 058	846 906	1 713	11 334 083
Base	4 004 011	11 758	1 834	12 101 850
Demote+Demote	2 303 139	717 459	1 566	11 156 613
DemoteContention	4 007 309	8 975	1 691	12 155 189
Demote+Reload	3 646 498	1 674 391	1 835	11 916 259
Flush+Flush (SMT)	2 130 205	709 578	1 640	10 781 589
Flush+Flush	3 839 776	12 712	1 923	12 127 306
Flush+Reload (SMT)	3 767 829	1 824 473	1 001 707	11 681 115
Flush+Reload	4 964 559	1 010 402	1 001 811	12 189 253
Evict+Reload	15 011 970	7 252	1 775	11 495 609
Prime+Probe	2 112 567	6 537	2 118	11 355 660

8. Appendix

Covert Channel and Noise Resilience

To determine the noise resilience, we run the 1-bit covert channel for all attacks with a varying number of background workers that perform cache-heavy operations. The results of our measurements are shown in Figure 6.11 and Figure 6.12. Optimized Demote+Reload (Figure 6.11a and Figure 6.12a) performed the best, while DemoteContention (Figure 6.11b) performs the worst, dropping to roughly 0 Mbit/s capacity. All other attacks lose $\approx 80\% - 95\%$ of their capacity with the maximum number of worker threads.

Performance Counter Values

The performance counter values of L1, L2, and L3 misses, as well as retired branches for all tested attacks after 10^6 iterations without a victim access, are shown in Table 6.5 and Table 6.7. Demote+Demote, cross-core DemoteContention, Flush+Flush, and cross-core Flush+Flush are indistinguishable from no attack. The performance counter values for all tested attacks after 10^6 iterations with victim access are shown in Table 6.6 and Table 6.8. Only cross-core DemoteContention can not be detected with the tested performance counters. All other attacks show at least a significant increase in L1 misses.

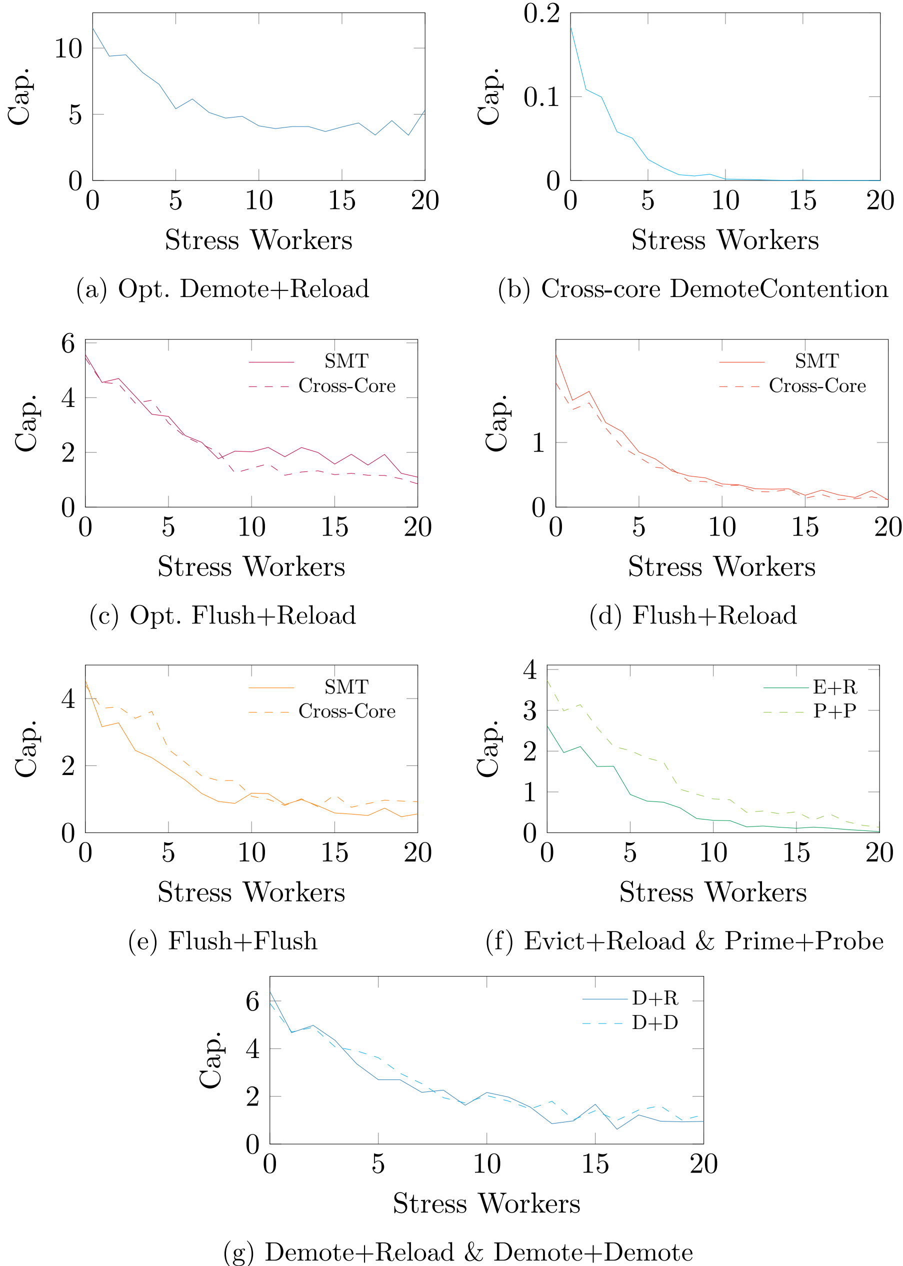
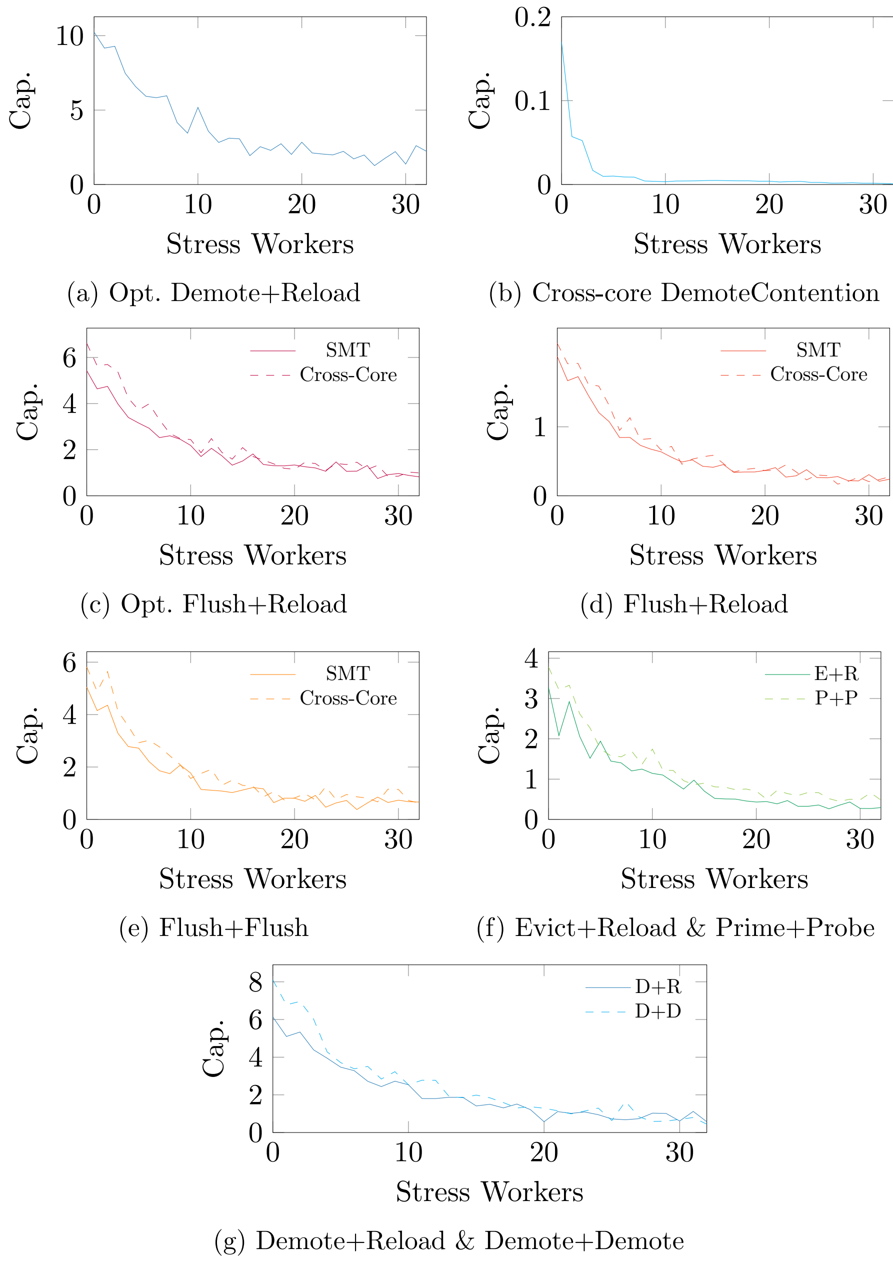


Figure 6.11.: Noise resilience of single-bit covert channels of all tested attacks. We increase the system noise with stress-ng worker threads until the number of cores of our Xeon Silver 4410T and provide the true capacity in **Mbit/s**. The capacity deteriorates for all attacks, indicating their respective noise resilience.



(a) Opt. Demote+Reload

(b) Cross-core DemoteContention

(c) Opt. Flush+Reload

(d) Flush+Reload

(e) Flush+Flush

(f) Evict+Reload & Prime+Probe

(g) Demote+Reload & Demote+Demote

Figure 6.12.: Noise resilience of single-bit covert channels of all tested attacks. We increase the system noise with stress-ng worker threads until the number of cores of our Xeon Silver 4514Y and provide the true capacity in **Mbit/s**. The capacity deteriorates for all attacks, indicating their respective noise resilience.

Table 6.6.: Performance counter values for 10^6 runs of each tested attack with a victim access on our Xeon Silver 4410T.

Attack	L1 misses	L2 misses	L3 misses	Retired Branches
Base (SMT)	2 463 241	852 141	1 457	11 210 752
Base	4 010 026	10 842	1 797	12 130 563
Demote+Demote	3 785 925	1 827 874	1 651	11 770 045
DemoteContention	4 012 453	9 195	1 888	12 139 868
Demote+Reload	3 756 115	1 811 665	1 723	11 892 382
Flush+Flush (SMT)	3 975 578	1 985 406	1 001 658	12 213 584
Flush+Flush	5 910 518	1 902 374	1 001 977	12 201 676
Flush+Reload (SMT)	3 985 752	1 991 979	1 001 816	12 240 320
Flush+Reload	6 885 299	2 902 632	1 001 760	12 207 778
Evict+Reload	15 010 561	6 685	2 180	11 830 557
Prime+Probe	15 010 450	6 686	2 062	11 647 487

Table 6.7.: Performance counter values for 10^6 runs of each tested attack without a victim access on our Xeon Silver 4514Y.

Attack	L1 misses	L2 misses	L3 misses	Retired Branches
Base (SMT)	47 121	11 299	1 496	10 630 295
cc Base	2 895 991	2 886 008	1 412	11 744 772
Demote+Demote	44 239	9 136	1 367	10 886 159
DemoteContention	2 988 775	2 975 436	1 338	11 781 288
Demote+Reload	1 052 979	1 011 615	1 337	10 884 189
Flush+Flush (SMT)	40 709	9 722	1 446	10 884 115
Flush+Flush	2 795 098	2 784 195	1 781	11 714 560
Flush+Reload (SMT)	1 075 286	1 016 118	1 001 531	10 882 563
Flush+Reload	4 042 492	4 026 895	1 001 513	11 713 327
Evict+Reload	13 023 103	10 946	1 468	10 881 358
Prime+Probe	41 488	9 483	1 753	10 874 922

Table 6.8.: Performance counter values for 10^6 runs of each tested attack with a victim access on our Xeon Silver 4514Y.

Attack	L1 misses	L2 misses	L3 misses	Retired Branches
Base (SMT)	55 672	13 004	1 582	10 629 320
Base	2 933 579	2 922 245	1 452	11 682 440
Demote+Demote	1 045 714	1 011 050	1 525	10 882 479
DemoteContention	2 979 165	2 964 159	1 468	11 745 555
Demote+Reload	1 053 159	1 011 066	1 409	10 885 386
Flush+Flush (SMT)	1 051 465	1 011 173	1 001 311	10 887 482
Flush+Flush	4 291 916	4 280 915	1 001 030	11 742 261
Flush+Reload (SMT)	1 068 255	1 012 946	1 001 391	10 885 194
Flush+Reload	5 444 751	5 429 372	1 001 418	11 748 249
Evict+Reload	13 021 365	11 188	1 569	10 885 999
Prime+Probe	13 020 591	12 295	1 470	10 881 335

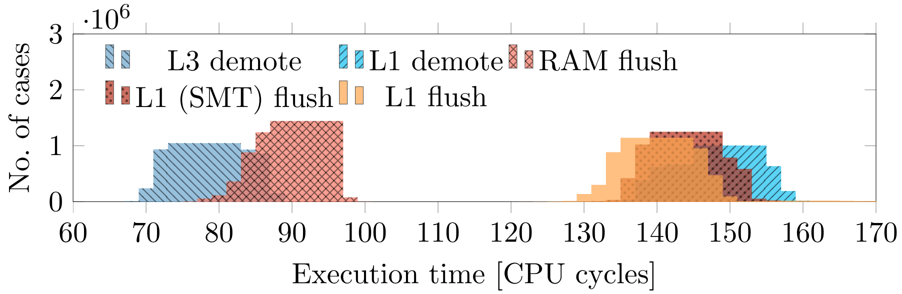
Further Emerald Rapids Data

In this section, we present visualizations and summarizations of further data collected on our Xeon Silver 4514Y. Histograms of all the attack's hit-miss timings are shown in Figure 6.13. Similar as on our Sapphire Rapids CPU, SMT Flush+Reload has the largest margin at 166 cycles as it distinguishes between a fast L1 hit and a slow L3 miss (Figure 6.13b). Evict+Reload has the lowest margin at only 4 cycles, which is the difference between an L1 hit and an L2 hit (Figure 6.13c).

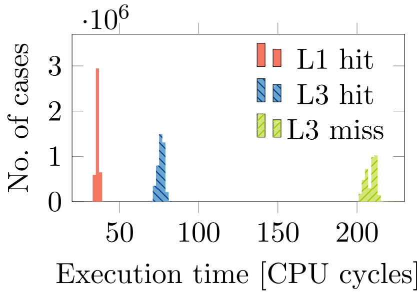
The attack times for all attacks are visualized in Figure 6.14. The results on our Xeon Silver 4514Y are very similar to the measurements on our Xeon Silver 4410T, with Flush+Reload having the highest attack time at 462.8 cycles, and Demote+Demote and DemoteContention having the lowest attack times at 137.6 cycles.

The blind-spot size and the effect of a delay between each attack iteration are shown in Figure 6.15. The relative blind spots are almost identical to the ones measured on our Sapphire Rapids CPU, with DemoteContention having the largest blind spot at 89.5 %, closely followed by cross-core Flush+Reload with 86.3 %. Flush+Flush and Demote+Demote have no measurable blind spot.

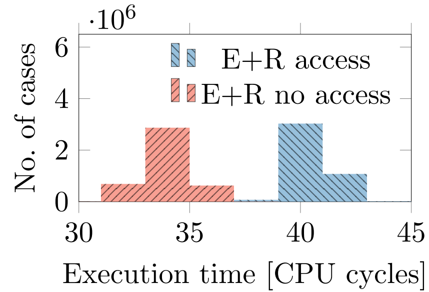
The temporal difference is visualized in Figure 6.16. Similar to the other metrics, the temporal differences are very similar to our measurements on our other Sapphire Rapids CPU, with the only exception being cross-core



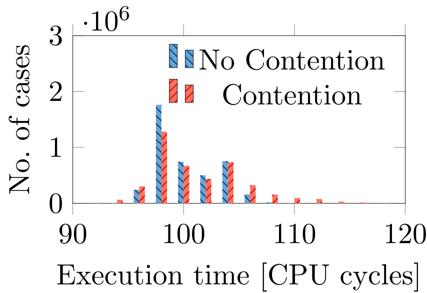
(a) `cldemote` & `clflush` Timings



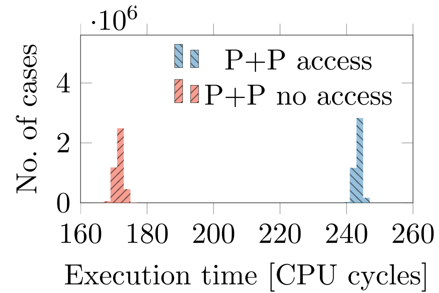
(b) Memory Access Timings



(c) Evict+Reload

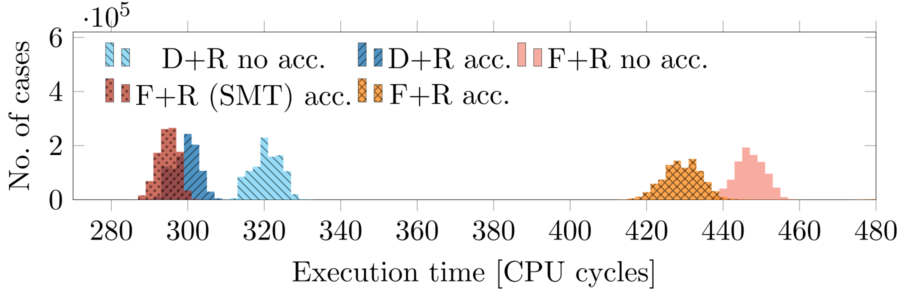


(d) Cross-core DemoteContention

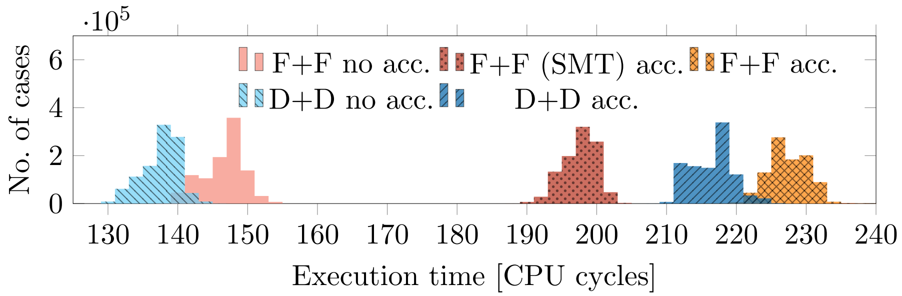


(e) Prime+Probe

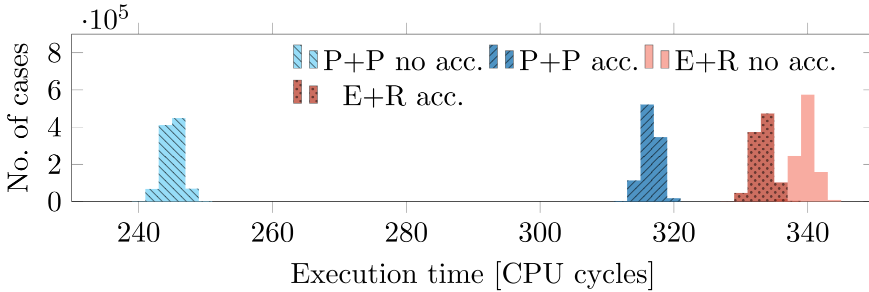
Figure 6.13.: Timing histograms for all tested attacks on our Xeon Silver 4514Y.



(a) Flush+Reload & Demote+Reload



(b) Flush+Flush & Demote+Demote



(c) Prime+Probe & Evict+Reload

Figure 6.14.: Execution time in cycles of a single attack iteration for all tested attacks on our Xeon Silver 4514Y.

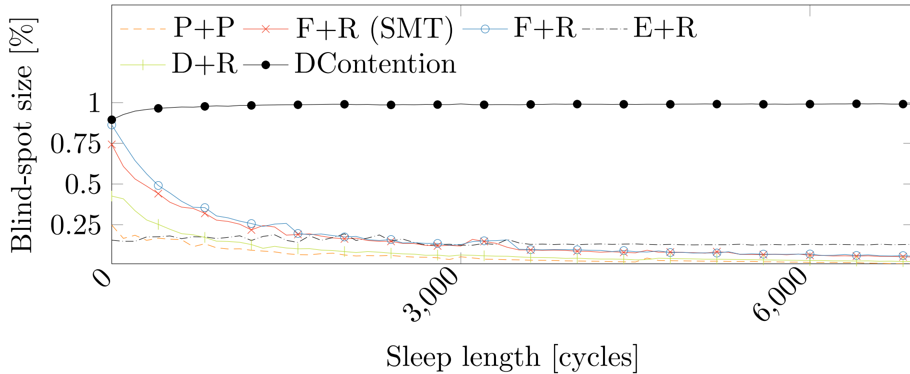
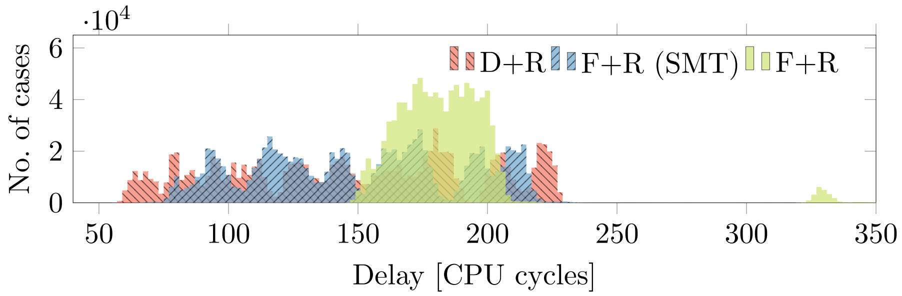


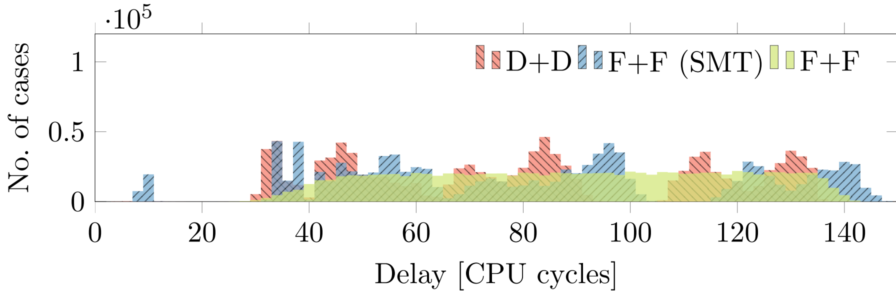
Figure 6.15.: Blind-spot size of all tested attacks that have a blind-spot for varying delay lengths after each attack iteration on our Xeon Silver 4514Y. The blind-spot size is given in the percent of a single attack loop iteration (including the delay).

Flush+Reload. The standard deviation for cross-core Flush+Reload is significantly lower at 15 ns compared to the 24 ns measured on our Sapphire Rapids CPU. This is most likely the result of some microarchitectural changes in the CPU.

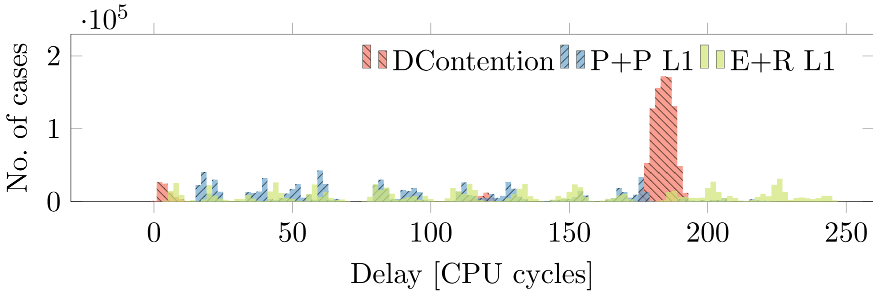
The channel capacities for single-bit and multi-bit covert channels are summarized in Table 6.9. In both single-bit and multi-bit optimized Demote+Reload performs the best with 11.10 Mbit/s and 17.03 Mbit/s respectively. Demote+Demote has the second-best in single-bit performance at 8.17 Mbit/s and cross-core Flush+Flush has the best multi-bit performance at 10.26 Mbit/s.



(a) Flush+Reload & Demote+Reload



(b) Flush+Flush & Demote+Demote



(c) Prime+Probe, Evict+Reload & DemoteContention

Figure 6.16.: The delay between memory access and the start of the detection period that detected the access for all attacks on our Xeon Silver 4514Y.

Table 6.9.: Single-bit (1-bit) and multi-bit (n-bit) covert-channel true capacity in **Mbit/s** and error ratio of all tested attacks on an Intel Xeon Silver 4514Y.

Attack	Cap. (1-bit)	BER (1-bit)	Cap. (n-bit)	BER (n-bit)
Opt. Demote+Reload	11.10	0.6 %	17.03	1.4 %
Opt. Flush+Reload	6.51	2.2 %	10.01	1.2 %
Opt. Flush+Reload (SMT)	5.31	1.7 %	9.24	1.0 %
Demote+Reload	6.09	0.7 %	6.09	0.7 %
Demote+Demote	8.17	1.7 %	9.36	2.3 %
DemoteContention	0.19	1.9 %	0.19	1.9 %
Flush+Reload	2.15	3.6 %	2.15	3.6 %
Flush+Reload (SMT)	2.01	3.2 %	2.01	3.2 %
Flush+Flush	5.74	2.0 %	10.26	1.9 %
Flush+Flush (SMT)	5.03	2.1 %	9.03	2.6 %
Prime+Probe (L1)	3.78	2.0 %	3.78	2.0 %
Evict+Reload (L1)	3.32	1.4 %	3.32	1.4 %

7

Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs

Publication Data

Fabian Rauscher and Daniel Gruss. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In: CCS. 2024

Contributions

Main Author.

Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs

Fabian Rauscher and Daniel Gruss

Graz University of Technology

Abstract

Interrupts are fundamental for inter-process and cross-core communication in modern systems. Controlling these communication mechanisms historically requires switches into the kernel or hypervisor, incurring high-performance costs. To alleviate these costs, Intel introduced new hardware mechanisms to send inter-processor interrupts (IPIs) from user space without switching into the kernel and from virtual machines without switching into the hypervisor. However, it is unclear whether this direct, unsupervised interaction between unprivileged (or virtualized) workloads and the underlying hardware introduces a significant change in the attack surface.

In this paper, we present the IPI side channel, a novel side-channel attack exploiting the recently introduced user interrupts and IPI virtualization features on Intel Sapphire Rapids and the upcoming Intel Arrow Lake processors. The IPI side channel is the first cross-core interrupt detection side channel, allowing an attacker to monitor interrupts delivered to any physical core of the same processor. Our attack is based on precise measurements of the hardware delivery time of interrupts from user space and virtual machines. More specifically, we exploit that interrupts are delivered through a cross-core bus, leading to timing variations on the attacker's local IPIs. We present multiple case studies to compare the IPI side channel with the state of the art: First, we present an unprivileged cross-core covert channel with a native true capacity of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) and a cross-VM capacity of 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.01). Second, we demonstrate a native inter-keystroke timing attack with an F_1 score of 97.9 %. Third, we present an open-world website fingerprinting attack on the top 100 websites, achieving an F_1 score of 89.0 % in a native scenario and an F_1 score of 71.0 % in a cross-VM (thin client) scenario. Furthermore, we discuss the broader context of the IPI side channels and categorize interrupt side channels and mitigations.

1. Introduction

Modern computer systems are highly parallelized, with hundreds of tasks with different privileges and access permissions that are isolated from each other through process isolation or virtualization. Process isolation and virtualization are enforced by the hardware and configured by privileged software, e.g., operating system or hypervisor [32]. However, tasks still frequently need to communicate with each other, e.g., to exchange data or synchronize operations, the system software provides means for cross-process and cross-core communication, including shared memory, signals, pipes, and various interrupts. All of these mechanisms require interaction with the system software, in some cases, e.g., for signal and interrupts, even for every single instance. As this incurs high context switch overheads, Intel introduced two new features called *user interrupts* [35] and *IPI virtualization* on Intel Sapphire Rapids processors. While these features are currently only available on Xeon CPUs, they will be available on the next generation of Intel consumer CPUs called Arrow Lake. User interrupts and IPI virtualization are intended to minimize cross-process and cross-core communication overheads by allowing user tasks and virtual machines to directly send and receive interrupts without invoking the kernel or hypervisor.

Side-channel attacks exploit information channels that carry information derived from a secret value. In particular, timing side-channel attacks [40] gained a significant amount of attention as they can easily be mounted by an adversary controlling a piece of software on a victim system [63], or even remotely [4]. Besides caches as a popular attack target [60, 25, 104], also other microarchitectural components have been attacked [16, 64, 17, 20, 1]. Several works studied information leakage from interrupt timings, as they carry information about user input, e.g., mouse movements or keystroke presses on the keyboard, which typically trigger interrupts [80, 13, 106, 67]. If an attacker can accurately detect interrupts, they can infer the inter-keystroke timings, and consequently the written text, in a side-channel attack [81, 105]. Recent works also demonstrated these attacks from JavaScript [46]. Another common attack scenario using the interrupt channel is website- or video-fingerprinting. Cook et al. [13] used interrupt detection with a hot loop to mount a website fingerprinting attack. Zhang et al. [106] and Rauscher et al. [67] exploited interrupt detection to fingerprint websites and videos. These previous works on interrupt side channels achieved high F_1 scores in fingerprinting scenarios.

However, they require physical placement of the attacker on the specific physical core that receives the interrupts. Hence, if the attacker cannot run on the core receiving the interrupts, these attacks are thwarted.

Therefore, we ask the following questions:
How can an attacker observe interrupts received by other cores? Do the unprivileged user interrupts and the IPI virtualization features facilitate new interrupt side-channel attacks?

In this paper, we present the IPI side channel, the first cross-core cross-VM interrupt side channel: Our attack observes all interrupts irrespective of the interrupt target cores, *i.e.*, the victim can run on any other core, in a different process or virtual machine. We exploit the recently introduced user interrupts and IPI virtualization features on Intel Sapphire Rapids and the upcoming Intel Arrow Lake processors. User interrupts allow an attacker to send inter-processor interrupts (IPIs) to the attacker's own threads without any privileges and measure their delivery time. IPI virtualization allows an attacker to send IPIs inside of virtual machines without hypervisor intervention allowing for precise IPI time measurements. Our IPI side channel exploits that even when IPIs do not go to the attacker's own core, they run through a cross-core system bus [31]. Consequently, any activity on the system bus leads to timing variations on the attacker's local IPIs.

We evaluate the IPI side channel in multiple case studies and compare it with other state-of-the-art side channels: First, we present a cross-core covert channel between two unprivileged user processes using user interrupts and a cross-core covert channel between two virtual machines using virtualized IPIs. We achieve a true capacity of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) in a native scenario and 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) in a cross-VM scenario. Second, we demonstrate an inter-keystroke timing attack using user interrupts with an F_1 score of 97.9 %. Third, we present a website fingerprinting side-channel attack on the top 100 websites. We evaluate our attack both in a closed-world native fingerprinting scenario, achieving an F_1 score of 91.7 %, and in an open-world native fingerprinting scenario with an additional class for other websites, achieving an F_1 score of 89.0 %. Furthermore, we demonstrate our website fingerprinting attack in a cross-VM attack achieving an F_1 scores of 80.4 % (closed-world) and 71.0 % (open-world).

Our IPI side channel is a significant improvement over prior interrupt side-channel attacks: Prior IPI side channels either relied on a software interface that is easy to constrain and already constrained on many systems [19],

or on same-physical-core interrupt detection techniques [46, 74, 13, 67]. Unrestricted cross-core and cross-VM interrupt detection substantially shifts the threat model, enabling attacks regardless of where attacker or victim are scheduled and which core receives the interrupts. We discuss this context of our work as well as mitigations against interrupt side channels.

To summarize, we make the following contributions:

- We present the first cross-VM cross-core interrupt detection side-channel attack, based on direct access to IPIs from user space (user IPIs) and virtual machines (virtualized IPIs).
- We show that the IPI side channel can be used to leak up to 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) in a native cross-core covert channel and 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) cross-VM.
- We present a native inter-keystroke timing attack exploiting the IPI side channel with an F_1 score of 97.9 %.
- We present website-fingerprinting attacks with F_1 scores of 91.7 % (closed-world) and 89.0 % (open-world) with native user IPIs, and 80.4 % (closed-world) and 71.0 % (open-world) in a cross-VM scenario without attacker access to the core receiving the victim interrupts.

Outline. Section 2 provides background on side channels and interrupt detection. Section 3 explains user interrupts and IPI virtualization and discusses the basic idea of the IPI side channel. Section 4 evaluates the IPI side channel using native and cross-VM cross-core covert channel. Section 5 presents our inter-keystroke timing side-channel attack. Section 6 presents our website fingerprinting attack in a closed- and an open-world scenario. Section 7 presents our systematic comparison of published interrupt side-channel attacks and mitigations and discusses related work. Section 8 concludes.

Responsible Disclosure. We responsibly disclosed our findings to Intel (February 21, 2024) and shared a paper draft with them. Intel recommends software authors follow Intel's software guidance on side-channel resistance.

2. Background

In this section, we provide background on side-channel attacks, interrupt detection, and website fingerprinting attacks as a typical side-channel evaluation scenario.

2.1. Hardware-based Virtualization

Virtualization allows running a full system (kernel and userspace programs) inside a virtual environment. A hypervisor monitors and manages these virtual machines (VMs) and mediates access to the hardware. Initial virtualization technology was based on full system emulation and later para-virtualization [95]. To reduce the performance overheads of these purely software-based solutions, hardware vendors introduced instruction set extensions to drastically improve the performance of virtualization. Intel introduced their Virtualization Technology extension (VT-x) [89], featuring a new CPU execution mode allowing a hypervisor to run at a higher privilege level than the operating system (*i.e.*, VMX root operation) [32]. The hypervisor controls a VM control structure to configure the VM. Accessing hardware resources or interacting with the hypervisor is done through interrupts or `vmcall`, which trigger VM exit operations, handing over control to the hypervisor. However, context switches from and to a VM are expensive [44] as they require configuring the VM control structures and often the flushing of buffers and even caches [98]. Consequently, to optimize the performance of systems running VMs, it is crucial to avoid expensive context switch mechanisms [44]. Following this intuition, Intel also introduced more hardware extensions to facilitate the reduction of context switches. A recent addition in this direction is IPI virtualization [33], introduced with Intel Xeon Sapphire Rapids and Arrow Lake CPUs. IPI virtualization allows VMs to directly send IPIs without the expensive switch to the hypervisor [33].

2.2. Side-Channel Attacks

Instead of exploiting software vulnerabilities, side channels exploit side effects of the implementation such as timing [40], power consumption [41], or radiation [66]. Many works focus on cryptographic primitives [40, 4, 6], leaking keys of vulnerable cryptographic implementations of e.g., AES [4,

60], RSA [104, 6], or ECDSA [103]. More recent works also focus on leakage on the system level, e.g., kernel information [29], user input [69, 23, 57], and other system activity [21]. Several works attempt to automate side-channel attacks, especially software-based side-channel attacks [23, 18, 68, 7, 78]. Among software-based side channels, especially cache side channels have taken a central role in the system security research community with generic techniques like Flush+Reload [104], Prime+Probe [60, 50, 54], and Flush+Flush [22]. Observing inter-keystroke timings is an interesting target for side-channel attacks, as it bypasses the security guarantees of any cryptographic algorithm applied by targeting the user's password instead [57]. Attacks on keystrokes are also difficult to mitigate as they run through an extensive code path: The software starts handling an interrupt in low-level kernel interrupt handler code and goes through kernel processing, library processing, and application processing until the keystroke shows a visual response to the user or is transmitted to the target buffer [74]. Inter-keystroke timings contain so much information that an attacker cannot only infer typed text [57] but also obtain privacy-related information, e.g., identify specific users [57]. Many works use machine learning to derive the secret input from the inter-keystroke timing trace [80, 81, 105].

In a more controlled setting, covert channels are a standard means to evaluate a side channel. In practice, covert channels can be relevant to exfiltrate secrets from co-located VMs [101, 53, 55, 84]. Covert channels are suitable to estimate the amount of noise and accuracy of a channel, and consequently, to provide a practical upper bounds for the leakage rate of the side channel [55]. Covert channels exploiting SMT, *i.e.*, two workloads share a physical core, and covert channels exploiting the cache, often reach the range of multiple megabytes per second [22, 70]. However, covert channels on other microarchitectural elements are often in the range of a few bytes per second [101, 16].

2.3. Interrupt Detection

Interrupts are commonly categorized into interrupts, faults, and traps [32]: Traps are intentionally configured to interrupt a process when a certain condition is reached. This can, for instance, be a certain memory access or reaching a specific location in the code. Faults occur when the processor cannot handle an issue in the instruction stream, handing over control to the operating system to decide what to do. Examples for faults are

page faults, general protection faults, or a division by zero. Interrupts occur upon events that are not part of the implemented instruction stream of the program. For instance, keystrokes can occur at any time and need to interrupt the running workload. The same also holds for other external interrupt sources, including network interrupts, disk interrupts, or interrupts caused by other input devices, e.g., the mouse. Consequently, prior work showed that observing interrupts, inherently allows to spy on these events [15, 80, 46] or even learn about the interrupted instruction stream [91]. Hence, observing interrupts and mitigating their observability has been identified as a direct path to inter-keystroke attacks and their mitigation [74].

Schwarz et al. [74] exploited that jumps in the timestamps returned by the `rdtsc` instruction indicate whether an interrupt occurred. Lipp et al. [46] demonstrated a similar attack using a JavaScript-based counting thread. Zhang et al. [106] and Rauscher et al. [67] exploited the `umwait` and `tpause` instructions to detect interrupts. They evaluated their attacks in website fingerprinting attacks, video-fingerprinting attacks, inter-keystroke detection, and covert channels to measure the side-channel capacity.

2.4. Website Fingerprinting

Website fingerprinting is a common side-channel evaluation scenario, *i.e.*, it serves as a benchmark to compare performance side channels. Consequently, there is a broad range of different side channels that perform website fingerprinting attacks, reporting various accuracies: Spreitzer et al. [82] achieved an accuracy of 89 % on 100 websites using the data-usage statistics on Android. Jana et al. [37] exploited the memory usage statistics of browsers and reported an accuracy between 30 % and 50 % for the top 100 000 websites. Gulmezoglu et al. [26] used hardware performance events and achieved accuracy of 86.3 % on 40 websites. Qin and Yue [65] used the power side channel on Android to fingerprint websites, achieving an accuracy of 55 %. Shusterman et al. [79] reported a website fingerprinting accuracy between 45.4 % (on the Tor browser) and 91.4 % (in the best case). Cook et al. [13] reported an accuracy of up to 97.2 % in an open-world website fingerprinting scenario and up to 96.6 % in a closed-world website fingerprinting scenario with the top 100 websites, based on an interrupt timing side channel similar to that of Lipp et al. [46]. Zhang et al. [106] monitored interrupts using idle states from native code and reported an F_1 score of 70 % over the Alexa top 100 websites. In a similar attack,

Rauscher et al. [67] achieved F_1 scores of 85.2 % and 93.1 % in an open- and closed-world evaluation over the Alexa top 100 websites.

3. IPI Side Channel

In this section, we present the attack primitive we use for cross-core and cross-VM interrupt detection. We provide an overview of user interrupts and how they can be triggered and received in userspace by an unprivileged attacker. We then provide an overview of IPI virtualization, allowing a VM to send inter-processor interrupts (IPIs) without hypervisor intervention, allowing for precise observation of their behavior and side effects. Lastly, we show the potential timing leakage of these new interrupt features to build an attack primitive. With these attack primitives, we can observe other interrupts on the same CPU to leak information from co-located workloads, such as network activity or keystrokes.

3.1. User Interrupts

User interrupts were introduced with the Intel Xeon Sapphire Rapids CPUs. The user interrupts feature **will** also be available on the upcoming Arrow Lake consumer CPUs. They allow the user to send and receive user inter-processor interrupts (user IPIs) using the `senduipi` instruction. Additionally, the system software can post user interrupts and send user-interrupt notifications. User IPIs, in particular, allow for fast inter-process communication.

At the time of writing, user interrupts are not officially supported by the Linux kernel. For our experiments, we use Linux with the official Intel implementation for user interrupt support [35]. While we use Intel's official patch, the exact software support implementation does not affect our results, as the attacks proposed in this paper rely mainly on the hardware implementation of user interrupts.

Both sending and receiving user IPIs is only possible in user space. If a user IPI is sent while the receiver thread is in kernel space or not running, the user IPI is buffered until the receiver thread is scheduled in user space. To allow for fast delivery when the receiver thread is running, the operating system reserves an interrupt vector in the advanced programmable interrupt controller (APIC) for user interrupts called the

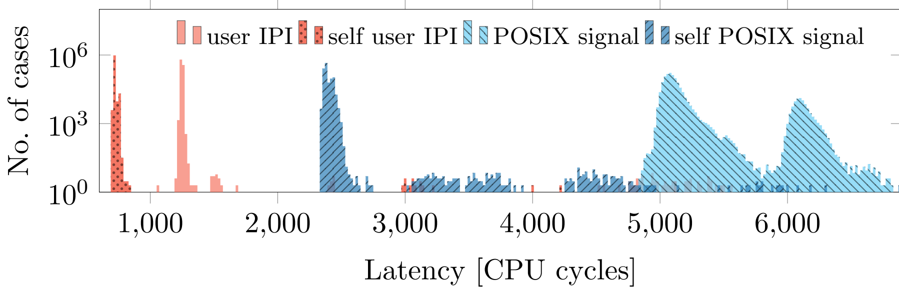


Figure 7.1.: User IPI and POSIX signal delivery time in cycles. User IPIs are 4 times faster (3 in case of a self-interrupt).

user-interrupt notification vector (UINV). The `senduipi` instruction sends an IPI with vector UINV to the core the receiver thread is running on. When the APIC receives an interrupt with the vector UINV, the CPU automatically performs checks for outstanding user interrupts instead of performing normal interrupt handling through the interrupt descriptor table (IDT). If the current thread has outstanding user interrupts, a user interrupt is triggered if the thread is in user space or set to pending if it is in the kernel. The CPU also checks for pending user interrupts when a thread state is restored, e.g., on a context switch.

While sending and receiving user IPIs can be done entirely in user space, user interrupts require support from the kernel. Among other things, the kernel manages the target cores for user interrupts, stores the handler address for the interrupts, determines which user interrupts are currently active, and which threads can receive which user interrupts. Specifically, in the setup process of user interrupts, support by the kernel is required. Intel proposes syscalls for registering and unregistering an interrupt handler as well as interrupt vectors for the receiver. Registering an interrupt vector as a receiver returns a file descriptor. Another thread can use this file descriptor to register as a sender for a given interrupt vector [35]. After the initial setup, the sender can trigger user IPIs with the `senduipi` instruction. Consequently, it is only possible for a thread to send user IPIs to threads that opt-in to receive them and only if the receiver shares the file descriptor with them.

The delivery delay of user IPIs and POSIX signals is shown in Figure 7.1. We measure the delay by executing `rdtsc` right before sending the POSIX signal or user IPI and as the first instruction in the handler function. The difference between the two TSC values is the minimum time between the

signal or IPI being sent and the handler being executed from the user's perspective. User IPIs perform better by a factor of 4 with a delay of only 1256.4 cycles (n=10^6, $\sigma_{\bar{x}}$=0.05) compared to POSIX signals at 5179.6 cycles (n=10^6, $\sigma_{\bar{x}}$=0.29). When the sender and receiver are the same thread, user IPIs perform better by a factor of 3 with a delay of only 726.8 cycles (n=10^6, $\sigma_{\bar{x}}$=0.06) compared to POSIX signals at 2398.0 cycles (n=10^6, $\sigma_{\bar{x}}$=0.07). POSIX signals have a significant kernel overhead, as sending a signal involves a syscall, which then delivers the signal, possibly by sending a regular IPI or setting it pending, waiting for the receiver to detect it. User IPIs have no significant kernel overhead, resulting in fast delivery times. These results show that user IPIs are a valuable addition for fast inter-process communication.

3.2. IPI Virtualization

IPI virtualization was introduced with the Intel Xeon Sapphire Rapids CPUs and Intel Arrow Lake CPUs. Furthermore, IPI virtualization **will** be available on Arrow Lake CPUs, the next generation of consumer Intel CPUs. This feature allows a VM to post IPIs without generating a VM exit. Contrary to user IPIs, IPI virtualization is already part of the Linux kernel and is activated per default (if available) with KVM since Linux 6.0 [38].

While it was already possible for the host to send so-called virtual interrupts to running VMs on a different core using the process posted interrupts feature and for the guest to receive them through virtual interrupt delivery without a VM exit, a VM couldn't send IPIs without causing a VM exit. Similar to user IPIs, posted-interrupt processing uses an interrupt vector that the operating system can designate for virtual interrupts. When a core running a VM receives such an interrupt, the core will check for any open posted interrupts and trigger them immediately, if possible. Previously, when sending IPIs, writes to the corresponding advanced programmable interrupt controller (APIC) field would cause a VM exit. With IPI virtualization, these writes are virtualized and post interrupts directly, using the posted-interrupt processing mechanism. Thus, the guest can send IPIs between cores without VM exits while sender and receiver are running, reducing the overhead of IPIs in VMs significantly. According to an Intel engineer, this reduces the delivery time of IPIs inside of VMs by up to 22.21 % [24]. As operating systems regularly use IPIs for functionalities such as TLB shootdowns, faster IPI processing can

```
1 volatile size_t end;
2
3 void __attribute__ ((interrupt))
4 handler(struct __uintr_frame *, unsigned long) {
5   end = rdtsc();
6 }
7 void attack() {
8   int ret = uintr_register_handler(handler, 0);
9   int uintr_fd = uintr_create_fd(1, 0);
10   stui();
11   int uipi_handle = uintr_register_sender(uintr_fd, 0);
12
13   for (;;) {
14     end = 0;
15     size_t start = rdtsc();
16     senduipi(uipi_handle);
17     while (!end);
18     auto x = end-start;
19     if (x < 900) continue;
20     //<interrupt detected>
21     //<further processing>
22   }
23 }
```

Listing 7.1: User-interrupt side-channel measurement code.

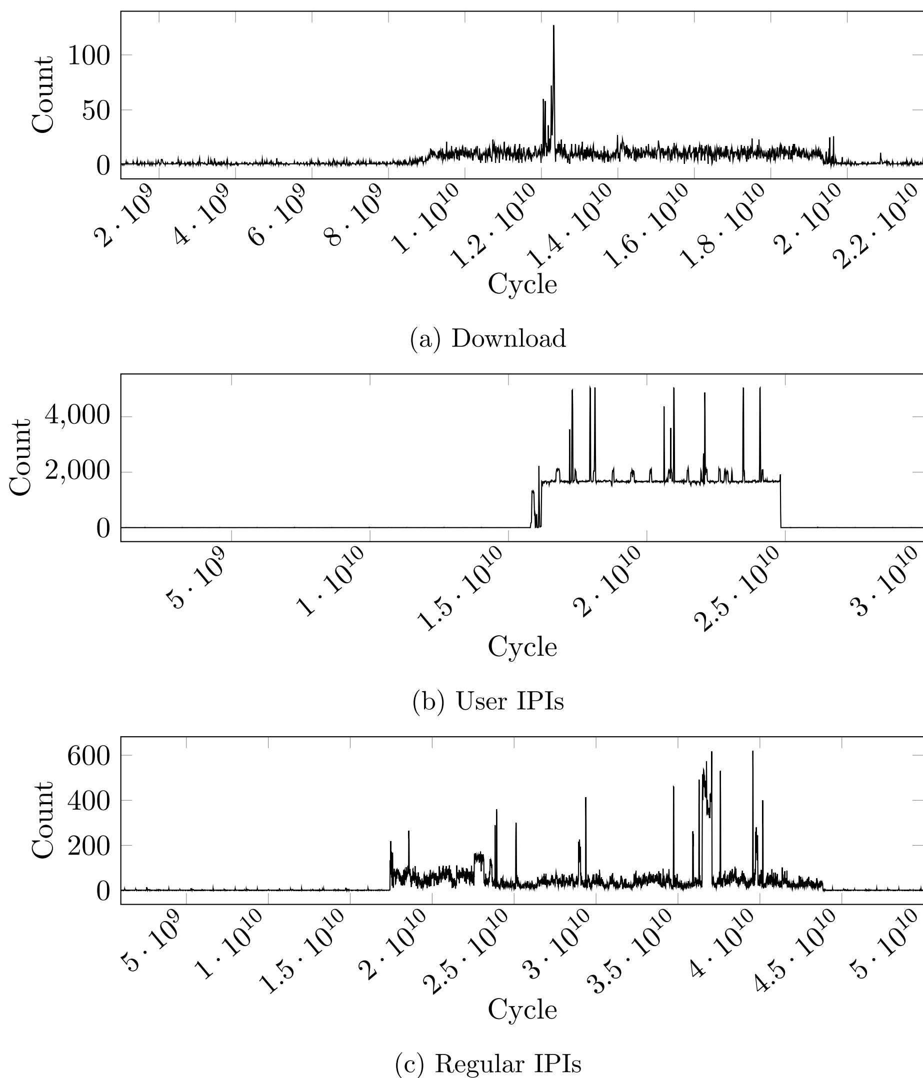
have a significant performance impact on inter-processor communication. Furthermore, this feature allows VMs to also take full advantage of user IPIs, as user IPIs can use IPI virtualization to not cause VM exits.

3.3. Timing Behaviour

In this section, we discuss the timing behavior of user IPIs and IPIs sent through IPI virtualization.

User IPIs

To determine the impact of other interrupts on the delivery time of user IPIs, we ran a measurement thread that continuously sends user IPIs to itself and measures the time between the `senduipi` instruction and the interrupt service routine using `rdtsc`. An example of the measurement



(a) Download

(b) User IPIs

(c) Regular IPIs

Figure 7.2.: User IPI timings with different tasks generating interrupts on other cores in the background.

code is shown in Listing 7.1. We trigger different interrupt types to arrive on other cores. Figure 7.2 shows the result of these measurements. To provide a better overview, we filtered all measurements in the typical user IPI delivery range from the core to itself (< 900 cycles). We grouped the remaining interrupts into 10 million cycle large bins. Figure 7.2 shows how many unusually slow user IPIs were measured at a given time. We

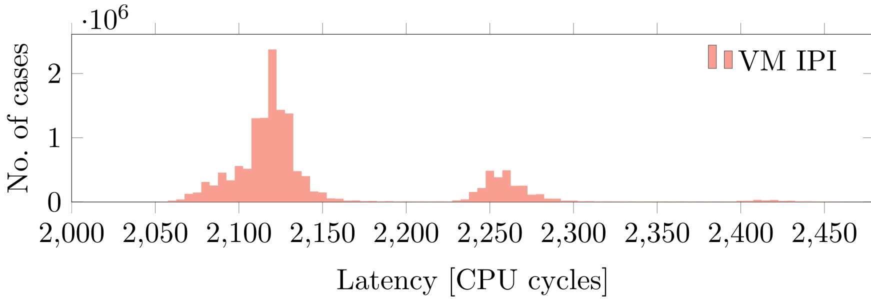


Figure 7.3.: IPI timings of a core sending an IPI to itself.

refer to this trace as an *interrupt trace*. We performed these measurements with a download in the background and the network interrupts arriving on a different core (Figure 7.2a), with a different thread sending user IPIs to another thread (Figure 7.2b), and a different thread sending regular IPIs (TLB shootdowns) to another thread (Figure 7.2c). In Figure 7.2a, the download starts at $\approx$ 10 billion cycles, resulting in a slight increase in unusually slow interrupts. The download ends at $\approx$ 20 billion cycles, resulting in a slight increase in unusually slow interrupts. In Figure 7.2b, the user IPIs start at $\approx$ 16 billion cycles, with a drastic increase in unusually slow interrupts. This increase is significantly higher than for network interrupts, as the user IPIs are sent at a significantly higher frequency than the typical arrival rate of network packets. The interrupt trace reaches almost 0 at $\approx$ 25 billion cycles when the user IPIs stop. In Figure 7.2c, the regular IPIs start at $\approx$ 20 billion cycles, resulting in a drastic increase in unusually slow interrupts. The interrupt trace reaches almost 0 at $\approx$ 45 billion cycles when the IPIs stop.

IPI virtualization

To determine the impact of other interrupts on the delivery time of IPIs inside VMs, we ran a measurement thread that continuously sends IPIs to itself inside of a VM (virtualized IPIs) and measures the time between the interrupt sent and the interrupt service routine using `rdtsc`. While the core sends IPIs to itself, we do not use self-IPIs. Self-IPIs are a special kind of IPI that allow a core to send an IPI to itself with low-performance overhead. As self-IPIs are not targeting other cores, they do not cause any contention on the shared system bus. Instead, we use regular IPIs, which allow for a target core to be specified, which we set to the core that

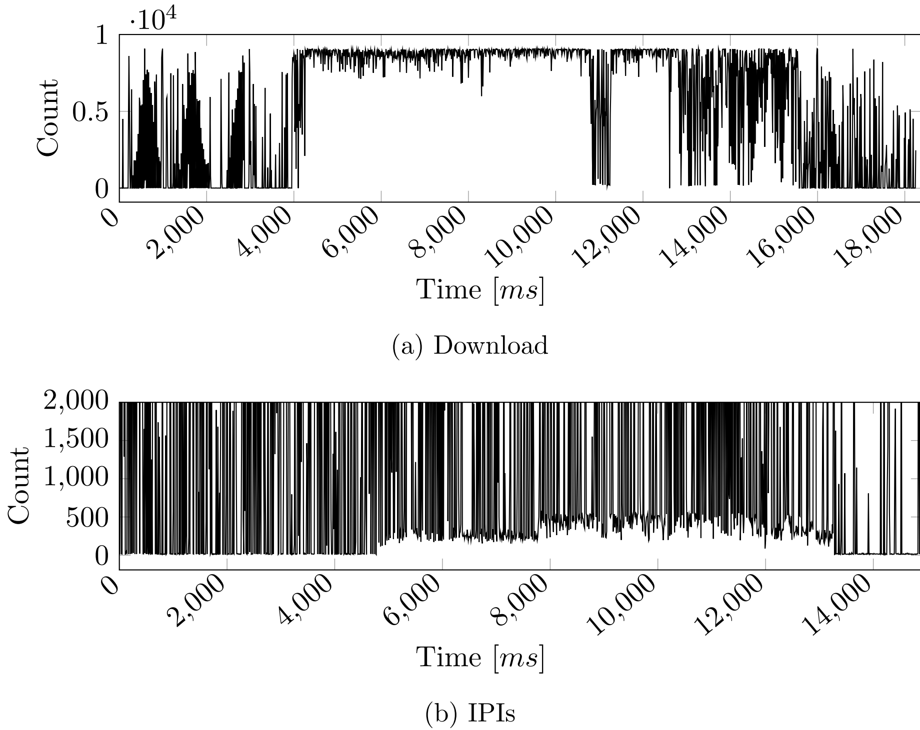


Figure 7.4.: IPI timings with different tasks generating interrupts on other cores inside of VMs in the background.

sends the IPI. This will trigger an IPI that is sent out to the system bus and received by the core, making it affected by possible contention. Such measurements on an idle system are shown in Figure 7.3. An IPI from a core to itself takes 2148.1 (n=$15 * 10^6$, $\sigma_{\bar{x}}$=68.3) cycles.

Our measurement code consists of a kernel module that registers a custom interrupt handler, repeatedly sends virtualized IPIs to itself, and measures the delivery time. We trigger different interrupt types to arrive on other cores. Figure 7.4 shows the result of these measurements. To provide a better overview, we filtered all measurements in the typical IPI delivery range from the core to itself ($<$ 2200 cycles). We grouped the remaining interrupts into 10 ms large bins. Figure 7.4 shows how many unusually slow IPIs were measured at a given time. We performed these measurements with a download in a separate VM, and the network interrupts arriving on a different core (Figure 7.4a) and a different thread in a VM sending regular IPIs (TLB shootdowns) to another thread (Figure 7.4b). In Figure 7.2a,

the download starts at 4 000 ms, drastically increasing the number of unusually slow interrupts. The download ends at 16 000 ms, resulting in a slight increase in unusually slow interrupts. In Figure 7.4b, the IPIs start at 5 000 ms, increasing the number of unusually slow interrupts. The interrupt trace goes back to almost 0 at 13 000 ms when the IPIs stop. The external interrupts from downloads, such as shown in Figure 7.4a, seem to have a significantly higher impact on the IPI latency than IPIs shown in Figure 7.4b.

Conclusion

These measurements show that it is possible to detect external interrupts and other IPIs, originating and targeting cores independent of the attacker. We assume this timing behavior results from contention on a shared bus used for delivering interrupts. According to the Intel manual, this shared bus is the system bus on Xeon CPUs [30]. Despite this, we cannot detect other events that should result in an access to the system bus, such as cache coherency-related events between cores. Furthermore, we cannot detect other system activity, such as a high amount of cache evictions on other cores or from the shared L3 using the IPI side channel. This is not surprising, as cache evictions from other cores or the L3 should not significantly affect the attacker core due to the non-inclusive L3 of the Xeon CPU used. Therefore, even if the cache lines of the attacker are evicted from the L3 by other cores, they stay in the private L1 and L2 of the attacker's core. The IPI side-channel signal also disappears when replacing the IPI measurement code with a cache attacker or constant time code, or turning off IPI virtualization in case of the cross-VM attack. This further validates that the signal stems from the IPI delivery delay.

4. Covert Channel

In this section, we present a cross-core and a cross-VM covert channel based on the IPI side channel. The covert channel is based on the performance impact other interrupts on the system have on user IPIs and virtualized IPIs. Covert channels are the most commonly used scenario to evaluate new side channels [85].

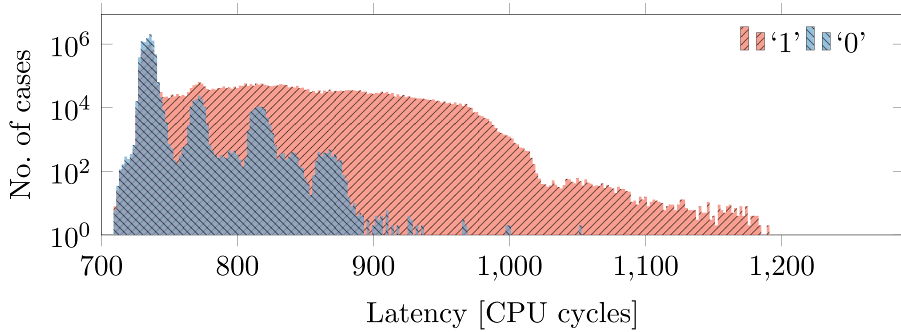


Figure 7.5.: Difference between a ‘1’ and a ‘0’ for our user IPI covert channel.

4.1. Covert Channel Design

In this section, we provide a high-level overview of our user-interrupt covert channel. We use time-slicing in combination with the IPI side channel shown in Section 3.3 to transmit data.

We transmit data by either performing user IPIs (native) or virtualized IPIs (cross-VM) in the sender or by busy waiting and measuring the time IPIs take in the receiver. When a ‘1’-bit is sent, the sender sends IPIs to itself, and when a ‘0’-bit is sent, the sender busy waits. The receiver sends IPIs to itself and measures the timings.

The timings for the ‘1’ and ‘0’ cases for user IPIs are provided in Figure 7.5. A ‘1’-bit results in measured user IPI timings of 785.4 cycles (n=$8 \cdot 10^9$, $\sigma_{\bar{x}}$=0.023) and a ‘0’-bit results in 735.7 cycles (n=$8 \cdot 10^9$, $\sigma_{\bar{x}}$=0.003) making them clearly distinguishable. Despite this clear difference in the average IPI time, both cases significantly overlap, as shown in Figure 7.5. This overlap results in a noisy but still functional covert channel.

The timings for the cross-VM covert channel in a sample transmission are shown in Figure 7.6. A ‘1’-bit results in frequent spikes in the transmission time, while there are no spikes when a ‘0’-bit is transmitted. The base latency of IPIs inside of VMs changes frequently, as shown in Figure 7.6, where the base latency is at ≈2 100 cycles for the first half of this example and at ≈2 250 cycles for the second half. This frequent change in the base latency is most likely due to other system events. A higher base latency also lessens the effect other IPIs have on the receiver’s IPI latency. While the frequently changing base latency does make it more challenging to extract the sent bits, they are still distinguishable.

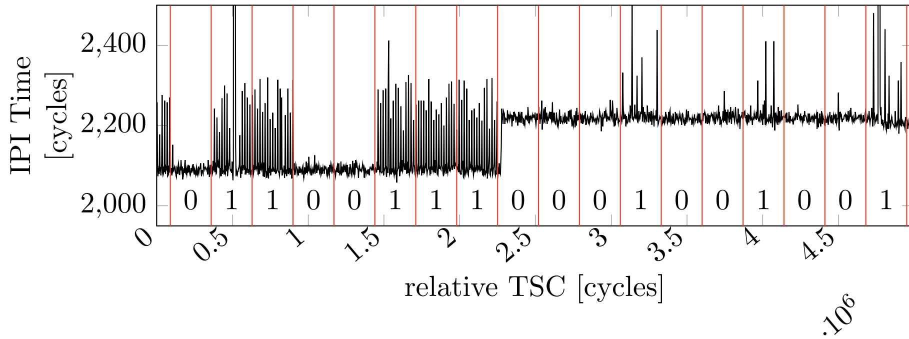


Figure 7.6.: Example transmission of our cross-VM covert channel using virtualized IPIs.

To synchronize the covert channel, we use the processor's time stamp counter (TSC) with the `rdtsc` instruction. The transmission for the native covert channel starts at a fixed TSC value or TSC modulo overflow. This reliably synchronizes sender and receiver. There is no direct communication between the sender and receiver. In the cross-VM covert channel, the initial synchronization occurs through the sender transmitting an initialization sequence, as the two VMs do not share the same TSC. Despite this, the TSC still increments at the same rate, allowing us to use it for further synchronization after establishing the beginning of the transmission.

The transmission itself is divided into fixed-sized transmission windows. Within each transmission window, one bit is sent. To send a bit, the sender either sends IPIs to itself throughout the transmission window or busy waits. The receiver continuously sends the IPIs to itself. After the transmission window is finished, the receiver determines the bit received using the timings of all its IPIs within this window.

Figure 7.7 provides an overview of a typical transmission. In the first transmission window, the sender continuously sends IPIs to itself, slowing the IPIs of the receiver down to transmit a '1'. In the second window, the sender busy waits, not affecting the receiver, to transmit a '0'. Finally, in the third window, the sender, again, sends IPIs to itself, slowing the IPIs of the receiver down to transmit a '1'. This results in the transmission of the bit sequence '101'.

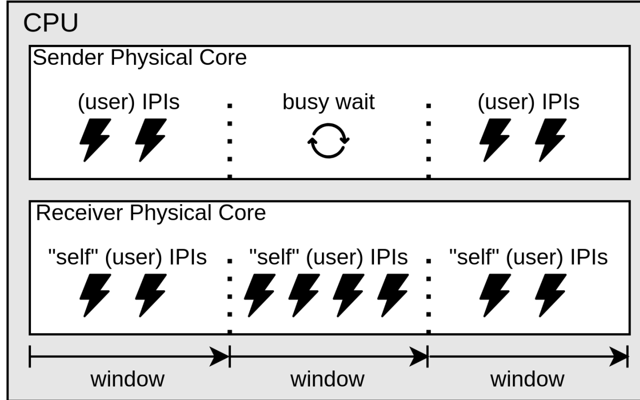


Figure 7.7.: Overview over a short sample transmission using user IPIs. When a '1' is sent, the sender runs self-user IPIs, slowing down the receiver's user IPIs. When a '0' is sent, the sender busy waits, which does not affect the receiver's user IPIs.

4.2. Evaluation

We evaluate our covert channel using random data on an Intel Xeon Silver 4410T. We tested two scenarios, native using user IPIs and cross-VM using virtualized IPIs. The sender and receiver run in separate processes, or VMs in the case of cross-VM, and are scheduled on two separate physical cores. Furthermore, we assume that there is no legitimate communication channel between them.

To determine the optimal transmission speed, we evaluate the covert channel for different transmission window lengths and record the raw capacity and bit error ratio. As our channel is based on time slices, decreasing the transmission window length increases the raw capacity. With a decrease in window length, the bit error ratio may increase, as there is less time for the receiver to determine the sent bit correctly. This decreases the true-channel capacity if the window length is too short due to the higher bit error ratio. To determine the optimal window length, we compute the true capacity of our channel using the binary symmetric channel model.[1]

[1] We 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.

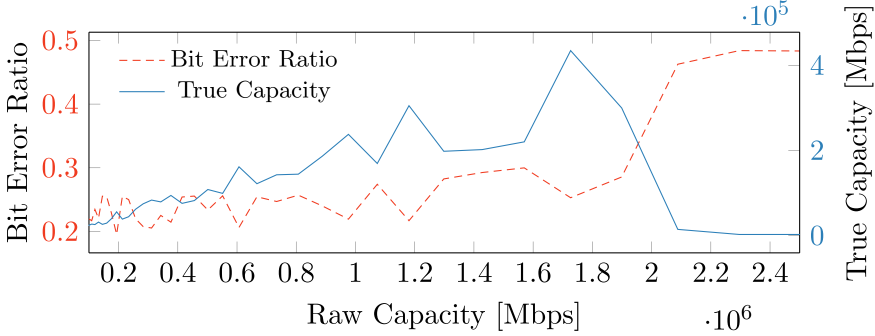


Figure 7.8.: The raw capacity and corresponding bit error ratio of our covert channel, as well as the resulting true capacity. The optimal true capacity is reached at a raw capacity of 1 726.6 kbit/s and a bit error ratio of 25.3 % (n=100, $\sigma_{\bar{x}}$=1.4).

Native We take the average for the true capacity and the bit error ratio over 100 runs for each transmission window length. The results of our optimization are shown in Figure 7.8. The user-interrupt covert channel is noisy due to the low attack margin (see Figure 7.5). Furthermore, our covert channel is affected by all external interrupts and IPIs on the same CPU, resulting in a high bit error ratio. The optimal transmission speed of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) is reached at a raw capacity of 1 726.6 kbit/s and a bit error ratio of 25.3 % (n=100, $\sigma_{\bar{x}}$=1.4). With a raw capacity higher than 1 726.6 kbit/s, the bit error ratio increases significantly, leading to a lower true capacity.

Cross-VM We optimized the cross-VM covert channel by hand due to the additional challenge of the initial synchronization of sender and receiver and the constantly changing base IPI latency in a VM scenario. This covert channel is affected by all external interrupts and IPIs on the same CPU, as well as further noise from the VMs, resulting in a high bit error ratio. Our cross-VM covert channel has a true capacity of 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.01) and a bit error ratio of 18.9 % (n=100, $\sigma_{\bar{x}}$=0,01). This lower capacity compared to the native scenario is the result of further noise introduced by the VM scenario, as well as the constantly changing base IPI latency, which makes bit extraction more challenging.

Noise Resilience To determine the noise resilience of the IPI side channel, we ran our native covert channel with a variety of stressors. We

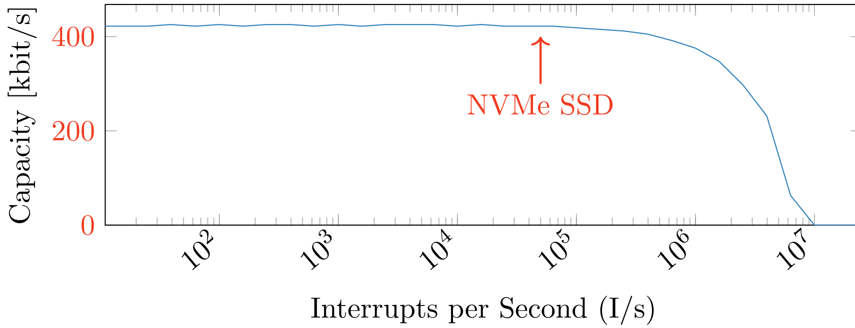


Figure 7.9.: The channel capacity of our native covert channel with an increasing amount of interrupt noise using IPIs. The capacity is unaffected until a interrupts noise of ≈600 kI/s interrupts per second and drops to ≈0 bit/s at ≈10 MI/s. As a reference point, the NVMe SSD that is part of our test system can generate up to ≈50 kI/s.

ran the stressors and the attack on separate cores to rule out a drop in capacity due to the attacker not being scheduled, as we want to focus on the noise directly generated by the stressors. We used the stress-ng CPU benchmark to generate compute-intense noise and the stress-ng I/O benchmark to generate noise through constant disk accesses and, therefore, NVMe SSD interrupts. As expected, the CPU benchmark does not negatively influence the channel capacity when all cores run the stress-ng CPU stressor except for the sender and receiver cores. The stress-ng I/O stressor also does not influence the channel capacity. While the stress-ng I/O benchmark does constantly generate disk accesses, even in this high I/O scenario, our NVMe SSD only generates ≈50k interrupts per second (I/s), which is significantly lower than the covert channel's ≈2 MI/s. The interrupt stress generated by the NVMe SSD is too low to impact the attack significantly.

To determine the interrupt frequency required to affect the channel capacity, we ran additional measurements using a custom interrupt stressor that generates IPIs on other cores at various frequencies. The results of these measurements are shown in Figure 7.9. The channel capacity starts to be affected by the other interrupts at ≈600 kI/s, dropping from 434.7 kbit/s to 391.9 kbit/s, slowly decreasing with more interrupt noise. The channel reaches a capacity of ≈0 bit/s at ≈10 MI/s. Our NVMe SSD was only able to generate ≈50 kI/s in our tests, which is significantly lower than the ≈600 kI/s required to measurably affect the channel.

Previous Work Saileshwar et al. [70] exploit contention on shared hardware resources on the CPU and achieve a cross-core capacity of 14.4 Mbit/s using shared addresses. Gruss et al. [22] time the `clflush` instruction to detect if a cache line is present in shared memory and achieve a cross-core capacity of 3.4 Mbit/s. Liu et al. [50] use Prime+Probe on the L3 with a capacity of 600 kbit/s, only slightly faster than our attack. While some of these attacks are faster, they require either information about physical memory, addressing functions, or shared memory with the main goal of observing cache accesses. Our attacks require access to IPIs with the main goal of observing the delivery of other IPIs and interrupts. The most closely related covert channel is by Rauscher et al. [67], using the new `tpause` instruction to detect interrupts and other events on the same physical core, achieving a true capacity of 656 kbit/s with an error rate of 9.2 %. While our attack is slightly slower with 434.7 kbit/s, we can detect interrupts from other physical cores.

5. Keystroke Detection

In this section, we present our inter-keystroke timing attack using user interrupts. Contrary to previous interrupt detection-based attacks, our attacker is not required to run on the physical core that receives the victim's keyboard interrupts. Instead, we measure the IPI delivery time of IPIs from the attacker core to itself to detect them. We do not directly infer text from these measurements but instead use them to determine the channel quality by comparing our measurements with the keystroke-timing ground truth. Few works recover text from timings, e.g., Song et al. [81], as this has become a standard but training-intense machine learning task.

5.1. Threat Model and Attack Setup

We run our measurements on an Intel Xeon Silver 4410T CPU with Ubuntu 22.04 and the Linux 6.0 kernel published by Intel [35] that supports user interrupts. We assume that the attacker can run unprivileged code on the victim system and has access to a high-precision timer, e.g., `rdtsc`. The attacker may be running on an isolated core that does not receive or handle any hardware interrupts. Finally, we assume that a user is providing input to the system via keystrokes while the attacker program is running.

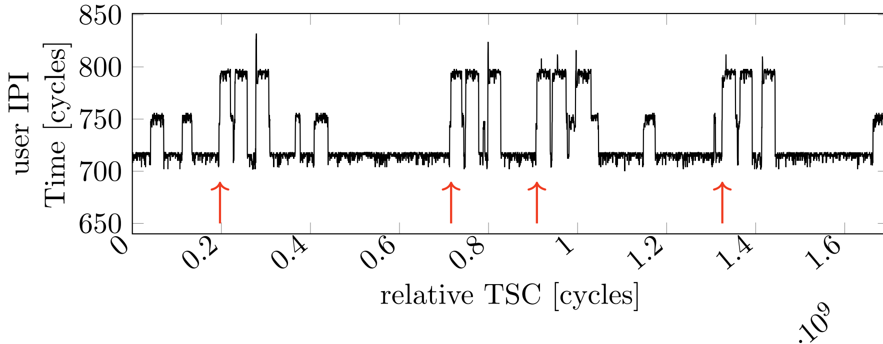


Figure 7.10.: Keystroke detection using user IPIs. For visualization purposes, we filtered the noise by only plotting the minimum delay of 400 measurements each. At every keystroke, indicated by red arrows, there are 3 distinct upward ticks in the user IPI delays, making them distinguishable from background noise.

Table 7.1.: Keystroke Detection Rates and F_1 Scores reported in other works.

Paper	Co-location	Det. Rate	F_1 Score	Temp. STD
our work	cross-core	98.2 %	97.9 %	6.15 ms
Schwarz et al. [74]	same core	100 %	94 %	≈1 ms
Rauscher et al. [67]	same core	94.1 %	90.5 %	0.95 ms
Lipp et al. [46]	same core	81.75 %	n/a	n/a

5.2. Attack Evaluation

For our inter-keystroke timing attack, the attacker continuously sends user IPIs to itself. By monitoring the time until the user IPI is handled, we can observe whether other system events, e.g., keyboard interrupts.

As the raw user IPI timings can be noisy, we apply a moving minimum filter, which returns the minimal timing observed in a 400 sample window. Figure 7.10 shows a trace of the filtered user IPI timings while a user is typing on the system. We can observe a distinctive pattern with 3 upward ticks in the user IPI delay. Based on this pattern, we can detect keystrokes using a similarity measure with a sliding window over a trace, e.g., with a window size of 200 million cycles.

We evaluated our attack with a human typing on the keyboard into a program measuring the ground truth. Based on this ground truth,

we computed the false negative and false positive rate, as well as the temporal deviation from the ground truth. Overall, we have a ground-truth trace with 571 keystrokes. The trace recorded with our IPI side channel contains 575 keystrokes. However, 14 of these keystrokes were false positives, meaning that we also had 10 false negative keystrokes, *i.e.*, only 561 true positive keystroke detections. Consequently, we have a precision of 97.6 % and a recall of 98.2 %. Based on this, we compute an F_1 score of 97.9 %. This is slightly higher than the F_1 scores and identification rates reported in other works, as shown in Table 7.1.

On the temporal scale, we observe a slightly higher standard deviation of 6.15 ms, which is expected as our attack is a cross-core attack in contrast to the other same-core attacks. Still, our temporal standard deviation is significantly lower than the average inter-keystroke interval of 120 ms and standard deviation of 11 ms for fast typists [14] and in the same order of magnitude as same-core inter-keystroke timing attacks.

6. Website Fingerprinting

In this section, we present a website fingerprinting attack using user interrupts and virtualized IPIs. We show a cross-core native website fingerprinting attack in both a closed-world and open-world scenario on the top 100 websites from the Alexa top 1 million list [2] using user IPIs to detect network interrupts. For open-world website fingerprinting, we use a separate `other` class for all websites not in the top 100. Furthermore, we present a cross-core cross-VM website fingerprinting attack in both a closed-world and open-world scenario on the top 100 websites from the Alexa top 1 million list [2] using virtualized IPIs to detect network interrupts. Contrary to previous interrupt detection-based fingerprinting attacks, our attacker is not required to run on the physical core that receives the victim browser's network interrupts.

6.1. Threat Model and Attack Setup

In this section, we discuss the thread model and overall setup of our attack.

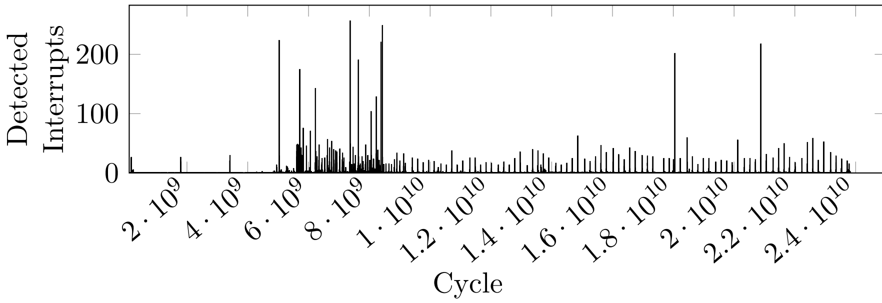
Native We run our measurements on an Intel Xeon Silver 4410T CPU with a default-configured Google Chrome 121.0. We assume that the attacker can run code on the victim system and has access to a high-precision timer, e.g., `rdtsc`. While the attacker code is running, the victim browses the web. We make no assumptions on whether the attacker is able to run on the core that receives the network interrupts, as the interrupts could be rerouted to a separate isolated core as proposed to mitigate interrupt detection attacks by previous works [67]. The attacker is not able to monitor interrupts through any system interfaces such as `/proc/interrupts`. Furthermore, we do not make any assumptions about the core the web browser is running on.

Virtual Machine Attack Scenario We assume a thin client scenario, where attacker and victim run co-located on the same VM host. Thin client devices have limited computing capability and only access a VM containing a full desktop environment. Thin clients offer companies lower initial hardware and maintenance costs compared to traditional desktops and are offered by major cloud providers [39]. While the attacker code is running, the victim browses the web.

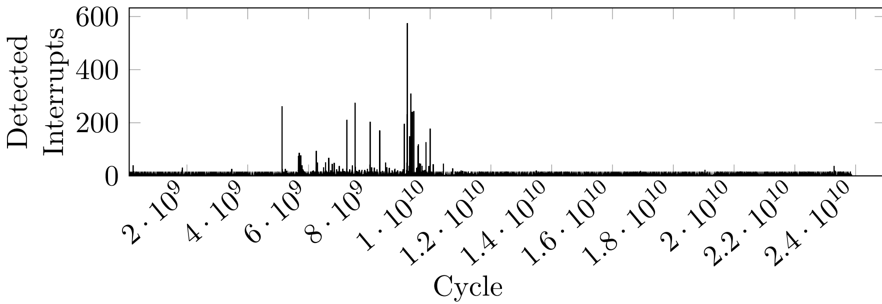
We run our measurements on an Intel Xeon Silver 4410T CPU with default-configured Google Chrome 121.0. For the hypervisor we use KVM with disk caching disabled, as recommended by Red Hat [27], on a stock Ubuntu 22.04. No custom kernel is required for this attack, as IPI virtualization is part of the Linux kernel and enabled by default. We assume that the attacker and victim run inside separate VMs in the cloud and are co-located on the same CPU, but both are running on separate physical cores. We assume that the attacker has root access inside its own VM and can load kernel modules, allowing for precise measurements of IPI timings. We make no assumptions on whether the attacker is able to run on the core that receives the network interrupts, as the interrupts could be rerouted to a separate isolated core, as proposed to mitigate interrupt detection attacks by previous works [67], or on a core the attacker does not run on.

6.2. Attack

Our attack consists of a data-collection phase and an offline phase for processing and classification of collected traces. The collection phase consists of the attacker running on the victim system collecting interrupt traces



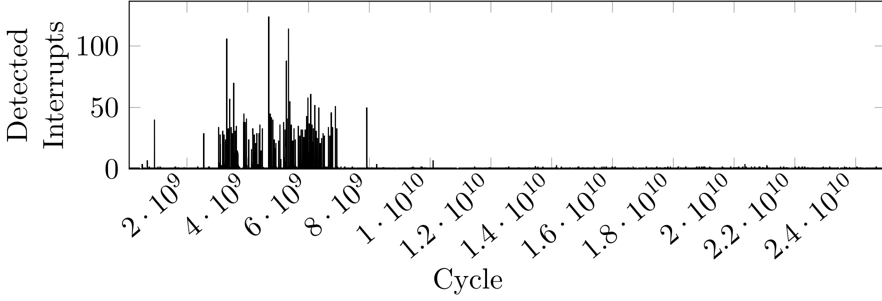
(a) `google.com`



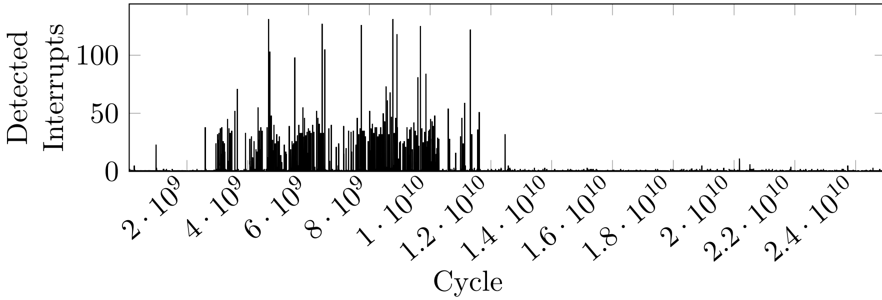
(b) `youtube.com`

Figure 7.11.: Interrupt traces of `google.com` (Figure 7.11a) and `youtube.com` (Figure 7.11b) measured by the attacker with user IPIs.

through either user IPIs, in the case of the native scenario, or virtualized IPIs, in the case of the cross-VM scenario. Such interrupt traces for `google.com` and `youtube.com` using user IPIs are shown in Figure 7.11 and for the cross-VM scenario using virtualized IPIs in Figure 7.12. The x-axis represents the CPU cycle relative to the beginning of the trace, and the y-axis represents the number of interrupts detected through our measurements. For the native user interrupts scenario (Figure 7.11) `google.com` and `youtube.com` have distinct traces, with `google.com` having a larger amount of interrupts at the beginning and multiple interrupts regularly throughout the trace and `youtube.com` having most interrupts occur at the start of the website access with one exceptionally high spike at 10 million cycles. For the cross-VM scenario (Figure 7.12) `google.com` and `youtube.com` also have distinct traces, with `google.com` having a large number of interrupts for a short period of time with distinct peaks, and `youtube.com` having a similar amount of interrupts over a longer period.



(a) `google.com`



(b) `youtube.com`

Figure 7.12.: Cross-VM interrupt traces of `google.com` (Figure 7.11a) and `youtube.com` (Figure 7.11b) measured by the attacker with virtualized IPIs.

As seen in both Figure 7.11 and Figure 7.12, there are more detections in our cross-VM scenario from the additional noise introduced by the VMs.

In the offline phase, these traces are first preprocessed with a short-time Fourier transform (STFT). The STFT performs multiple Fourier transforms on short windows of the trace, resulting in 2D data consisting of the frequency information for each window on one axis and the time on the other axis. The preprocessing through an STFT allows us to perform convolutions on our data, making it possible to use a convolutional neural network (CNN) for the classification. This is a well-established signal processing technique [102, 11, 28, 67]. Our CNN consists of 4 convolutional layers and 3 fully connected layers and outputs a probability for each website and one for the `other` class in the case of the open-world scenarios. The output is the probability that the input belongs to a given website.

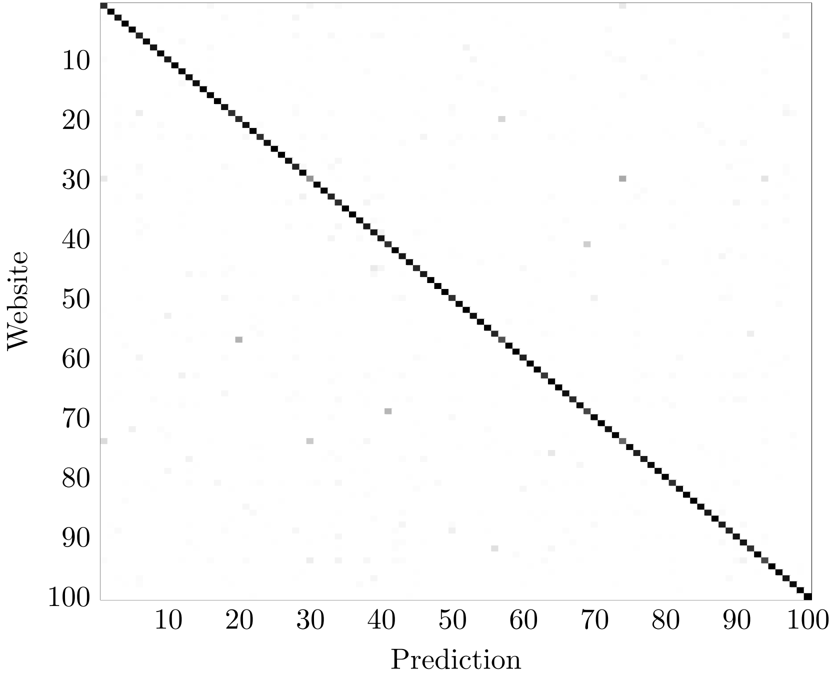


Figure 7.13.: Native closed-world website fingerprinting confusion matrix.

6.3. Evaluation

We evaluate our attack on closed-world and open-world scenarios with the attacker running on a physical core that does **not** receive network interrupts or run the browser. We collected 200 traces for each of the explicitly classified websites (20 000 in total). For the open-world scenario, we additionally collected the traces of `5000` additional websites (one per website) from the Alexa top 1 million list [2], which are not in the top 100. To evaluate our attack, we randomly split our data for each class into 5 equally large parts and performed 5-fold cross-validation. The test set for each run does `not` overlap with the training set. Due to this, in the open-world scenarios, website traces in the test set of the `other` class belong to websites that the model has never seen during training. We train our CNN with a validation split of 10 % of the training set.

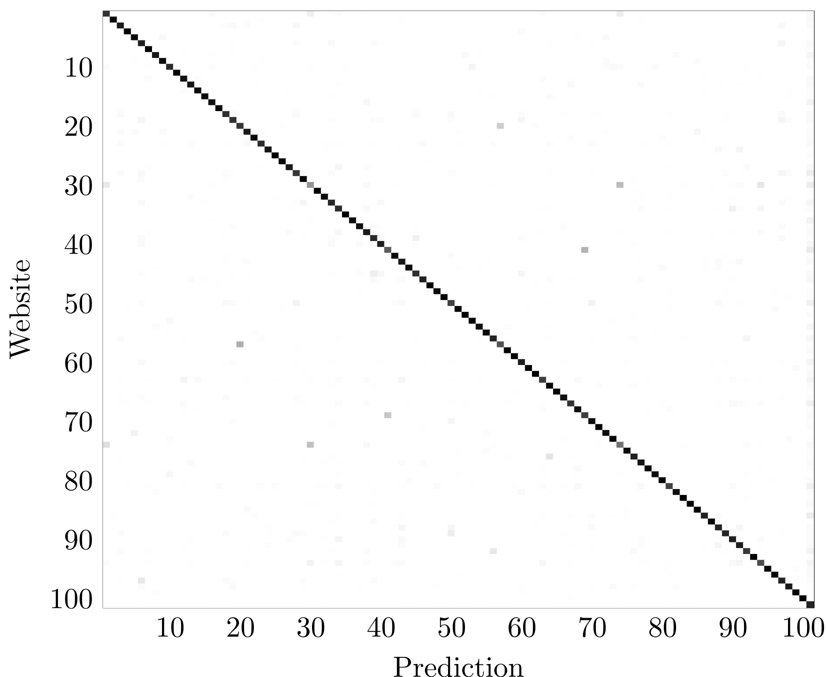


Figure 7.14.: Native open-world website fingerprinting confusion matrix.

Native Closed-World Website Fingerprinting

In this scenario, we only classify the top 100 websites in a native scenario using user IPIs. Our classifier achieved an F_1 score of 91.7 %. The confusion matrix is shown in Figure 7.13. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. The worst-performing websites are `google.com.hk` at 42 %, `google.co.in` at 59 %, and `dzen.ru` at 68 %. All other websites have accuracies of above 70 %. The `google.com.hk` and `google.co.in` sites, in particular, are often misclassified as other Google domains.

Native Open-World Website Fingerprinting

In this scenario, we classify the top 100 websites and use an `other` class for all other websites in a native scenario using user IPIs. Our classifier achieved a macro averaged F_1 score of 89.0 %, showing a high accuracy

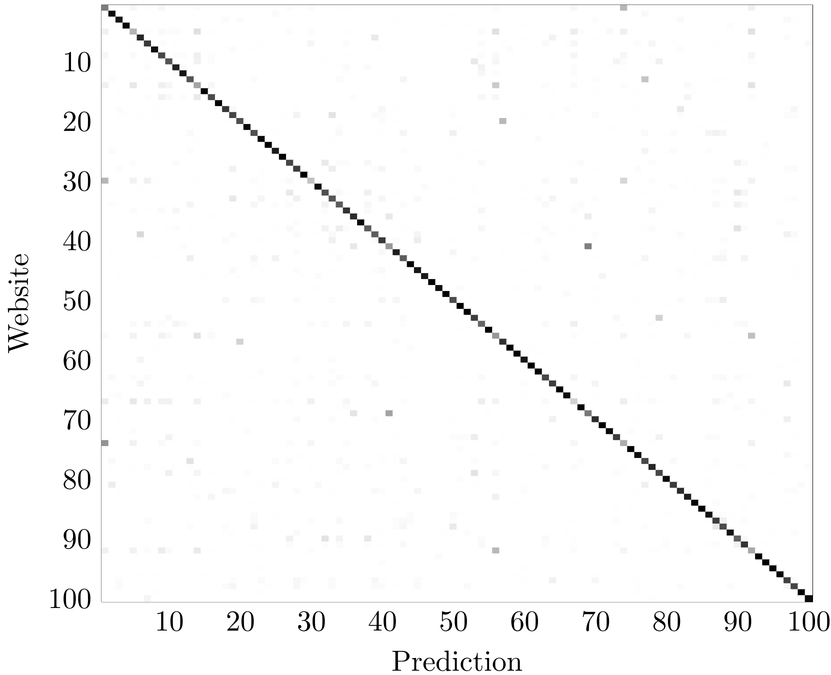


Figure 7.15.: Cross-VM closed-world website fingerprinting confusion matrix.

across all classes, and an accuracy of 85.2 % on the `other` class. The confusion matrix is shown in Figure 7.14. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. The worst-performing websites are `google.com.hk` at 41.5 %, `google.co.in` at 54.5 %, `microsoftonline.com` at 63 %, and `dzen.ru` at 67 %. All other websites have accuracies above 70 %. Similar to the closed-world scenario, the `google.com.hk` and `google.co.in` websites, in particular, are often misclassified as other Google domains. The `microsoftonline.com` domain performs poorly relative to other websites, as only sub-domains of it are accessible, e.g., `login.microsoftonline.com`, while we only tested the exact domains listed in the Alexa top 1 million list [2].

Cross-VM Closed-World Website Fingerprinting

In this scenario, we only classify the top 100 websites in a cross-VM scenario using virtualized IPIs. Our classifier achieved an F_1 score of 80.4 %. The confusion matrix is shown in Figure 7.15. Each cell represents

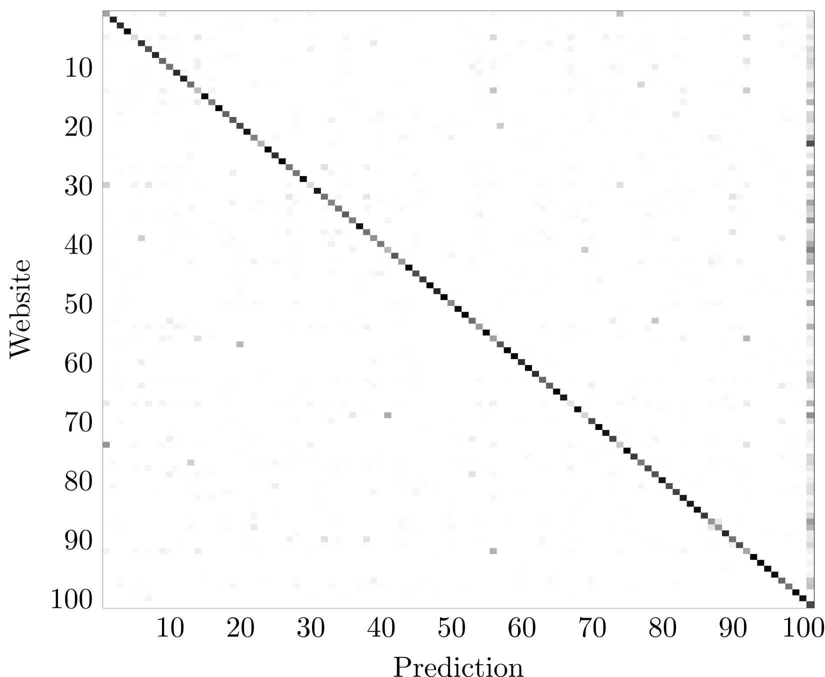


Figure 7.16.: Cross-VM open-world website fingerprinting confusion matrix.

the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. The worst-performing websites are `indeed.com` at 18 %, `google.com.hk` at 24 %, and `facebook.com` at 29.5 %. All `indeed.com` and `facebook.com`, in particular, are fast-loading websites when already cached, making them more challenging to distinguish with the added noise from the VMs. The Google domains are, similar to the other scenarios, often misclassified as other Google domains.

Cross-VM Open-World Website Fingerprinting

In this scenario, we only classify the top 100 websites in a cross-VM scenario using virtualized IPIs. Our classifier achieved a macro averaged F_1 score of 71.0 %, showing a high accuracy across all classes, and an accuracy of 71.7 % on the `other` class. The confusion matrix is shown in Figure 7.16. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. Similar to the

Table 7.2.: Overview of Different Interrupt Side-Channel Works

Attack	`Software Interface`	Hardware-Based Same-Core	Hardware-Based Cross-Core
Inter-Keystroke Timing	[105, 74]	[88, 87, 46, 74]	**our work**
Website Fingerprinting	[51]	[13, 106, 67, 107]	**our work**
Application Fingerprinting	[15, 86, 51]		
Cryptographic Key Leakage		[107]	
DNN Model Stealing	[51]	[107]	

closed-world scenario, the worst-performing websites are `indeed.com` at 11 %, `google.com.hk` at 13.5 %, and `facebook.com` at 13.5 %. The Google domains are, similar to the other scenarios, often misclassified as other Google domains.

6.4. Previous Work

Our closed-world attack achieves an F_1 score of 91.7 % (native) and 80.4 % (cross-VM), while our open-world attack achieves an F_1 score of 89.0 % (native) and 71.0 % (cross-VM), performing similarly to previous works while not requiring to run on the core where the network interrupts arrive. Zhang et al. [106] achieved an F_1 score of 78 % on the top 100 sites using the `mwaitx` instruction to detect interrupts on the core that receives the network interrupts. Gulmezoglu et al. [26] exploit performance counters, achieving an accuracy of 86.3 % on 40 websites. Spreitzer et al. [82] use data-usage statistics on Android to classify 100 websites with an F_1 score of 89 %. Rauscher et al. [67] exploit the `tpause` instruction in VMs to classify 100 websites with an F_1 score of 93.1 %.

Table 7.3.: Mitigations for Interrupt Side-Channel Attacks

Approach	Mitigations
Constrain interfaces	[105, 13]
Constrain co-location	[67]
Constrain timers	[94, 52, 47, 36, 61, 48, 45, 106]
Non-leaking timers	[3, 94, 52, 42]
Inject Noise	[74]

7. Related Work & Discussion

A series of works have investigated side-channel attacks exploiting interrupts or their effects. The information channels can coarsely be divided into three categories (cf. Table 7.2):

The **first** category of attacks uses software interfaces to obtain information about interrupts, e.g., `/proc/interrupts` [105, 15, 86, 74, 51]. While this system information is provided architecturally without noise and regardless of the attacker's core scheduling, they are trivial to mitigate by making the corresponding software interface privileged and these interfaces are generally not available from within VMs for the host system. Prior work demonstrated inter-keystroke timing [105, 74], website-fingerprinting [51], and application-fingerprinting [15, 86, 51] attacks. Some works have even demonstrated leakage of DNN models [51] and cryptographic keys [107]. Still, by constraining the software interface, all of these attacks can be mitigated in practice. The other two categories use side channels to monitor hardware behavior.

Concretely, the **second** category is hardware-based same-core attacks that typically exploit that the interrupt has to be executed by the core under attack. Several attacks use busy loops of timing measurements and measure when they are interrupted as timer jumps, *i.e.*, latency spikes [88, 87, 46, 74, 13]. More recent works exploit CPU features that do not depend on busy-looping the SMT thread under attack or a co-located SMT thread on the same physical core [106, 67, 107]. Hence, all of these attacks depend on being co-located on the same core as the attacker. In this setting, powerful attacks are possible, including inter-keystroke timing attacks [88, 87, 46, 74], website-fingerprinting attacks [13, 106, 67, 107], as well as leakage of DNN models and cryptographic keys [107].

The **third** category is cross-core hardware-based attacks that do not exploit system interfaces and also do not require a busy loop on the victim core. Our work is the first attack to demonstrate this possibility by exploiting both the user interrupts feature as well as the virtualized inter-processor interrupts feature. Our attack generalizes in the virtualized setting to any VM that can send and measure the latency of inter-processor interrupts.

Mitigations To mitigate the IPI side channel effectively and efficiently, hardware changes may be necessary. We believe that the current implementation sends IPI messages and lets cores check and decide which IPI messages they accept. However, it is unclear why the system bus transmitting interrupts between cores has to be used for local and I/O interrupts that only affect a single core. This approach inherently allows us to probe the corresponding system bus and, thereby, the interrupt activity of other cores. A different design of this system bus could affect our attack. Similarly, changing how cores check whether they should receive an interrupt would also have the potential of mitigating our attack.

While our conclusion is that closing the IPI side channel requires redesigning the corresponding interrupt handling hardware, we still want to discuss mitigations proposed by the academic community (cf. Table 7.3) against prior interrupt-driven attacks:

The **first** category is to remove the `/proc` interface or make it privileged [105, 13]. Virtual machines also cannot access the host's `/proc` interface for security reasons. This approach has been implemented on various systems [74, 107], leaving only other channels open to mitigate. However, with the `/proc` interface disabled or unavailable inside a VM, our attack still works. For user interrupts, we could also limit the attack surface by only allowing a selected number of trusted applications to use user interrupts. As user interrupts require kernel support to register a receiver and a sender thread, we suggest to restrict these kernel interactions to certain user groups or applications, e.g., specific drivers. This restriction would no longer allow an attacker to take advantage of the user-interrupt side channel while making user interrupts available for applications that need them. We do not consider entirely disabling user interrupts a viable option due to their low delay times compared to POSIX signals. This restriction is not possible for IPI virtualization, as modern operating systems require IPIs for fast inter-processor communication,

and disabling IPI virtualization for all untrusted VMs would remove the performance improvements gained by IPI virtualization for these VMs.

The **second** category of mitigations is to constraint either the co-location of the attacker workload with the victim or its placement on the interrupt-receiving core [67]. However, as we can detect interrupts on any core, this approach does not affect our attack, even when isolating interrupts to a separate physical core. Also, randomizing interrupt core assignments does not affect our attack.

The **third** category is to constrain timers, e.g., by making them privileged, as has been discussed in numerous works [94, 52, 47, 36, 61, 48, 45, 106]. Similarly, the **fourth** category is to modify timers in a way that they do not depend on the secret information anymore [3, 94, 52, 42]. Both approaches have limited effect in practice as the community has found many ways to bypass them, e.g., using counting threads [100, 47, 76, 73] and timeless methods [93].

A **fifth** category is to introduce noise [74]. Noise has been studied as a mitigation against power side channels as well [43]. However, from this context, it is also known that noise only reduces the side channel signal but cannot eliminate it [43]. Consequently, adding noise also can only reduce the signal, which, according to Schwarz et al. [74], is sufficient against inter-keystroke timing attacks. However, it is unclear whether similar approaches to inject noise could be sufficient to mitigate interrupt side-channel attacks in other attack scenarios such as website- and application fingerprinting, or leakage of cryptographic keys or DNN models.

Generic Side-Channel Mitigations A generic side-channel mitigation covering interrupt side channels as well, is side-channel detection, e.g., using performance counters [62]. Intel suggests that user interrupts can be tracked through architectural Last Branch Records (LBRs) and Intel Processor Trace, which record user interrupts the same way as normal interrupts [32]. Despite this, it is difficult to distinguish a benign workload using the IPI side channel for high-frequency message passing between processes and a malicious workload using user IPIs or virtualized IPIs for an attack.

Beyond Side Channels Exploitation of Inter-Processor Interrupts Inter-processor interrupts provide multi-core and multi-processor

systems with a means to communicate and synchronize across cores. A common example is the invalidation of a virtual memory mapping, which is cached by the TLB, requiring a so-called TLB shootdown, a coordinated operation across multiple processors or processor cores to invalidate the corresponding TLB entries across all cores. Wang et al. [96] exploit this behavior to spy on the accessed bit in the page-table entry of an SGX enclave. Zhang et al. [108] exploit IPIs to preempt a victim, amplifying their Prime+Probe attack. Zhang and Reiter use IPIs to continuously flush the caches of other cores to mitigate cache attacks on these [109]. However, none of these works studies the timing of IPIs themselves.

Trusted Execution Environments Considering the emerging concept of trusted execution environments (TEEs) such as Intel TDX, Intel SGX, and AMD SEV-SNP, interrupt side channels may pose a relevant attack vector. We consider **two** attack scenarios.

The **first** category is the case of a **malicious host**. With SGX-Step [92] (Intel SGX) and SEV-Step [99] (AMD SEV-SNP), the host sets up the APIC timer to trigger an external interrupt shortly after entering the protected guest to single- and zero-step the guest. This can be used to determine instructions executed [99], amplify power side channels [49], or assist microarchitectural attacks [90, 75, 5, 56, 91]. Intel TDX includes a mitigation against single- and zero-stepping [34] to prevent these kinds of attacks. As the host has to be able to inject interrupts, e.g., for device emulation, TDX and AMD SEV-SNP allow the host to arbitrarily inject interrupts. Schlüter et al. [72] showed that a malicious host can inject interrupt vectors typically used for software interrupts, such as syscalls, to change register values on AMD SEV-SNP and Intel TDX. WeSee[71] injects interrupts with the interrupt number reserved for the virtualization exception leading the guest to assume that such an exception occurred on AMD SEV-SNP. Sridhara et al. [83] send signals to SGX enclaves to modify the enclave state. Constable et al. [12] propose a hardware ISA extension to make SGX enclaves interrupt aware, allowing an enclave to detect and mitigate interrupt-based attacks. While our IPI side channel does work in a malicious host scenario, the virtual machine host receives all external interrupts and only forwards them to the guest if necessary. Hence, the host already knows which interrupts occur at a given time without requiring a side channel.

The **second** category is the case of a **malicious guest**. As the code executed inside of an SGX enclave, a TDX guest, or an SEV-SNP guest,

is not directly accessible by the host, nefarious activity by the guest can be challenging for the host to detect. While there are no interrupt-related attacks from inside a TEE yet, there are multiple works discussing and showing the threat of malicious enclaves in Intel SGX [73, 77]. There are also multiple works that try to detect or defend against malicious SGX enclaves [97, 110, 58]. Similar to existing interrupt-detection-based attacks, an attacker could use `rdtsc` [74] or a counting thread [73] to detect interrupts on the same core in TDX and SEV-SNP or the `tpause` instruction to detect interrupts on a sibling logical core in TDX [34, 67]. Counting threads have also been explored in SGX enclaves as a defense against interruption-based attacks [10, 9, 8, 59]. The IPI side channel does not work inside AMD SEV-SNP, Intel TDX, or Intel SGX at the time of writing. Both AMD SEV-SNP and Intel TDX require the host to handle the routing and injection of IPIs sent from inside guests. The feature set inside Intel SGX is even more limited and excludes sending of user or regular IPIs. Hence, it is not possible to mount the IPI side channel from inside these TEEs.

8. Conclusion

While user interrupts and IPI virtualization drastically reduce the performance cost of cross-process and cross-core signaling, our work shows that these new features can be exploited to detect interrupts delivered to any core in native and cross-VM scenarios. We used the IPI side channel for cross-core covert communication between two processes that send IPIs to themselves, with a native true capacity of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) and cross-VM true capacity of 3.47 kbit/s (n=100, $\sigma_{\bar{x}}$=0.01). We presented an inter-keystroke timing attack with an F_1 score of 97.9 % and a standard deviation of 6.15 ms. Furthermore, we demonstrated a cross-core website fingerprinting attack that achieves an F_1 score 89.0 % in an open-world native scenario and 71.0 % in an open-world cross-VM scenario, highlighting the security and privacy implications. While there are mitigations against interrupt side-channel attacks, the change in the attack scenario (*i.e.*, cross-core cross-VM) also bypasses several mitigations. We conclude that bringing interrupts to userspace and providing VMs with low latency access to IPIs can have unforeseen side effects, resulting in an increased attack surface for security- and privacy-related applications.

Acknowledgments

We thank the anonymous reviewers and our anonymous shepherd for their guidance, comments, and suggestions. We would also like to thank our DIMVA reviewers for their feedback on an earlier version of this paper. Furthermore, we thank Andreas Kogler for his feedback and insightful discussions. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I6054). Additional funding was provided by generous gifts from Red Hat, 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] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In: S&P. 2018 (p. 176).

[2] Alexa Internet, Inc. The top 1 million sites on the web. May 2023. URL: `https://www.alexa.com/topsites` (pp. 197, 201, 203).

[3] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In: CCSW. 2010 (pp. 206, 208).

[4] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 176, 179).

[5] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In: USENIX Security. 2022 (p. 209).

[6] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. Photonic Side Channel Attacks Against RSA. In: HOST. 2017 (pp. 179, 180).

[7] Sebastien Carre, Victor Dyseryn, Adrien Facon, Sylvain Guilley, and Thomas Perianin. End-to-end automated cache-timing attack driven by Machine Learning. In: Journal of Cryptology (2019) (p. 180).

[8] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating Speculative-Execution Attacks on SGX with HyperRace. In: Dependable and Secure Computing (DSC). 2019 (p. 210).

[9] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: closing hyper-threading side channels on SGX with contrived data races. In: S&P. 2018 (p. 210).

[10] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJá Vu. In: AsiaCCS. 2017 (p. 210).

[11] 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. 200).

[12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. {AEX-Notify}: Thwarting Precise {Single-Stepping} Attacks through Interrupt Awareness for Intel {SGX} Enclaves. In: USENIX Security. 2023 (p. 209).

[13] 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. 176, 178, 181, 205–207).

[14] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. Observations on typing from 136 million keystrokes. In: CHI Conference on Human Factors in Computing Systems. 2018 (p. 197).

[15] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In: S&P. 2016 (pp. 181, 205, 206).

[16] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS. 2016 (pp. 176, 180).

[17] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (p. 176).

[18] Anders Fogh. Covert Shotgun: automatically finding SMT covert channels. 2016. URL: https://cyber.wtf/2016/09/27/covert-shotgun/ (p. 180).

[19] Google Issue Tracker. Android O prevents access to /proc/stat. 2017. URL: https://issuetracker.google.com/issues/37140047 (p. 177).

[20] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security. 2018 (p. 176).

[21] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (p. 180).

[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 180, 195).

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (p. 180).

[24] Zeng Guang. KVM Commit "[v9,0/9] IPI virtualization support for VM". 2022. URL: https://patchwork.kernel.org/project/kvm/cover/20220419153155.11504-1-guang.zeng@intel.com/ (p. 184).

[25] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P. 2011 (p. 176).

[26] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to violate web privacy with hardware performance events. In: ESORICS. 2017 (pp. 181, 205).

[27] Jiri Herrmann, Yehuda Zimmerman, Dayle Parker, and Scott Radvan. Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide. 2019 (p. 198).

[28] 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. 200).

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 180).

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 2023 (p. 189).

[31] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (p. 177).

[32] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2024 (pp. 176, 179, 180, 208).

[33] Intel. Intel Architecture Instruction Set Extensions and Future Features. 2022 (p. 179).

[34] 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. 209, 210).

[35] Intel. UINTR Linux Kernel. 2024. URL: `https://github.com/intel/uintr-linux-kernel` (pp. 176, 182, 183, 195).

[36] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Preventing microarchitectural attacks before distribution. In: CODASPY. 2018 (pp. 206, 208).

[37] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In: S&P. 2012 (p. 181).

[38] Linux Kernel. https://cdn.kernel.org/pub/linux/kernel/v6.x/ChangeLog-6.0. In: Linux Kernel Change Log 6.0. 2022 (p. 184).

[39] Simon Kissler and Owen Hoyt. Using Thin Client Technology to Reduce Complexity and Cost. In: ACM SIGUCCS conference on User services. 2005 (p. 198).

[40] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (pp. 176, 179).

[41] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (p. 179).

[42] David Kohlbrenner and Hovav Shacham. Trusted Browsers for Uncertain Times. In: USENIX Security. 2016 (pp. 206, 208).

[43] Itamar Levi, Davide Bellizia, David Bol, and François-Xavier Standaert. Ask Less, Get More: Side-Channel Signal Hiding, Revisited. In: IEEE Transactions on Circuits and Systems 67.12 (2020), pp. 4904–4917 (p. 208).

[44] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. In: ACM SIGARCH Computer Architecture News 43.3S (2015), pp. 375–387 (p. 179).

[45] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In: USENIX Security. 2022 (pp. 206, 208).

[46] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (pp. 176, 178, 181, 196, 205, 206).

[47] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (pp. 206, 208).

[48] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS. 2020 (pp. 206, 208).

[49] 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. 209).

[50] 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. 180, 195).

[51] Haoyu Ma, Jianwen Tian, Debin Gao, and Chunfu Jia. On the Effectiveness of Using Graphics Interrupt as a Side Channel for User Behavior Snooping. In: Transactions on Dependable and Secure Computing 19.5 (2021), pp. 3257–3270 (pp. 205, 206).

[52] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: ACM SIGARCH Computer Architecture News (2012) (pp. 206, 208).

[53] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 180).

[54] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In: RAID. 2015 (p. 180).

[55] 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 (p. 180).

[56] Daniel Moghimi. Downfall: Exploiting Speculative Data Gathering. In: USENIX Security. 2023 (p. 209).

[57] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 180).

[58] Soo Jung Moon, Hoorin Park, and Wonjun Lee. Preventing enclave malware with intermediate enclaves on semi-honest cloud platforms. In: BigComp. 2021 (p. 210).

[59] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In: USENIX ATC. 2018 (p. 210).

[60] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 176, 180).

[61] Yoshihiro Oyama. How does malware use RDTSC? A study on operations executed by malware with CPU cycle measurement. In: DIMVA. 2019 (pp. 206, 208).

[62] Matthias Payer. HexPADS: a platform to detect “stealth” attacks. In: ESSoS. 2016 (p. 208).

[63] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (p. 176).

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

[65] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In: TrustCom/BigDataSE. 2018 (p. 181).

[66] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In: E-smart. 2001 (p. 179).

[67] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (pp. 176, 178, 181, 182, 195, 196, 198, 200, 205, 206, 208, 210).

[68] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In: NDSS. 2017 (p. 180).

[69] 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. 180).

[70] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS. 2021 (pp. 180, 195).

[71] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious# VC Interrupts to Break AMD SEV-SNP. In: arXiv preprint arXiv:2404.03526 (2024) (p. 209).

[72] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking Confidential VMs with Malicious Interrupts. In: USENIX Security. 2024 (p. 209).

[73] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 208, 210).

[74] 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 (pp. 178, 180, 181, 196, 205–208, 210).

[75] 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. 209).

[76] 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. 208).

[77] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (p. 210).

[78] Martin Schwarzl, Erik Kraft, and Daniel Gruss. Layered Binary Templating. In: ACNS. 2023 (p. 180).

[79] 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. 181).

[80] Laurent Simon, Wenduan Xu, and Ross Anderson. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. In: PETS (2016) (pp. 176, 180, 181).

[81] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security. 2001 (pp. 176, 180, 195).

[82] 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. 181, 205).

[83] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. SIGY: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals. In: arXiv preprint arXiv:2404.13998 (2024) (p. 209).

[84] 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. 180).

[85] Jakub Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. In: Cryptology ePrint Archive, Report 2016/479 (2016) (p. 189).

[86] Xiaoxiao Tang, Yan Lin, Daoyuan Wu, and Debin Gao. Towards Dynamically Monitoring Android Applications on Non-rooted Devices in the Wild. In: WiSec. 2018 (pp. 205, 206).

[87] Albert Tannous, Jonathan T. Trostle, Mohamed Hassan, Stephen E. McLaughlin, and Trent Jaeger. New Side Channels Targeted at Passwords. In: ACSAC. 2008 (pp. 205, 206).

[88] Jonathan T Trostle. Timing Attacks Against Trusted Path. In: S&P. 1998 (pp. 205, 206).

[89] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. In: Computer 38.5 (2005), pp. 48–56 (p. 179).

[90] 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. 209).

[91] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (pp. 181, 209).

[92] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (p. 209).

[93] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In: USENIX Security. 2020 (p. 208).

[94] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In: CCSW. 2011 (pp. 206, 208).

[95] VMware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. 2007 (p. 179).

[96] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: CCS. 2017 (p. 209).

[97] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In: RAID. 2019 (p. 210).

[98] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018. URL: https://foreshadowattack.eu/foreshadow-NG.pdf (p. 179).

[99] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In: TCHES (2024), pp. 180–206 (p. 209).

[100] John C Wray. An Analysis of Covert Timing Channels. In: Journal of Computer Security (1992) (p. 208).

[101] 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. 180).

[102] 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. 200).

[103] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. In: Cryptology ePrint Archive, Report 2014/140 (2014) (p. 180).

[104] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 176, 180).

[105] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security. 2009 (pp. 176, 180, 205–207).

[106] 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. 176, 181, 205, 206, 208).

[107] Xin Zhang, Zhi Zhang, Qingni Shen, Wenhao Wang, Yansong Gao, Zhuoxi Yang, and Jiliang Zhang. SegScope: Probing Fine-grained Interrupts via Architectural Footprints. In: HPCA. 2024 (pp. 205–207).

[108] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In: CCS. 2012 (p. 209).

[109] Yinqian Zhang and MK Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: CCS. 2013 (p. 209).

[110] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer. In: AsiaCCS. 2021 (p. 210).

8

TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX

Publication Data

Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX. In: USENIX Security. 2025

Contributions

Main Author.

TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX

Fabian Rauscher[1], Luca Wilke[2], Hannes Weissteiner[1],
Thomas Eisenbarth[2], Daniel Gruss[1]

[1]Graz University of Technology, [2]University of Lübeck

Abstract

Intel TDX is a trusted execution environment (TEE) protecting arbitrary code, e.g., an entire OS, from the host system in trust domains (TDs). While TDX isolates the memory of TDs, side channels are still a threat due to shared hardware. Prior work showed that single-stepping is a powerful technique for attacking TEEs. After TDX was found vulnerable to these attacks, Intel improved their mitigations with TDX module version 1.5.06, stopping all known single-stepping attacks.

In this paper, we introduce TDXploit, a novel technique to revive single-stepping attacks on Intel TDX. TDXploit exploits a fundamental flaw in Intel's single-stepping mitigation, ironically, achieving a higher (>99.99 %) single-stepping accuracy than without mitigations. We recover the mitigation's internal state using an attacker-controlled TD. We not only predict the mitigation's behavior without any side channel but also manipulate it for reliable single- and multi-stepping. TDXploit can perform one single-step every 3.7 ms. We evaluate TDXploit with an attack on ECDSA in OpenSSL. Furthermore, we systematically evaluate 6 state-of-the-art side-channel attack techniques on TDX and their compatibility with TDXploit. A key finding is that `clflush` bypasses Intel's defenses, allowing Flush+Flush attacks on TDX guest physical memory. Compared to all previous Flush+Flush attacks, our Flush+Flush attack requires no shared memory and can target any memory location of a TD. We demonstrate the impact of this finding in a full key recovery on an AES T-Table implementation, requiring only 8 986 encryption traces. Finally, we combine our novel Flush+Flush with TDXploit to leak TOTP secrets with a single trace. We conclude that further mitigations against single-stepping and side channels on TDX are necessary.

1. Introduction

Outsourcing tasks involving confidential data, e.g., customer data and company secrets, to the cloud can be a threat to data confidentiality and privacy as the cloud provider has complete access to all the data when using traditional virtual machines (VMs). Therefore, a data breach in a cloud provider's infrastructure can have severe consequences for their customers. Trusted execution environments (TEEs) provide an environment where the host system does not have access to the memory and register state of the application. Traditional TEEs, e.g., Intel SGX, require applications to be written specifically for this environment [13], limiting its applicability to a narrow set of use cases. Using a library OS approach, prior work demonstrated that generic applications can be ported to Intel SGX [47], but there are still several limitations due to the process-scoped nature of SGX. The recently introduced AMD SEV-SNP and Intel TDX extensions are TEEs that allow entire VMs to run inside the secure environment, allowing the deployment of applications without any TEE-specific adaptations.

As the TEE threat model includes a malicious host, the attack surface of them is significantly larger than of non-TEE environments [13]. Intel SGX, in particular, received a significant amount of attention from the scientific community [53]. Early works focused on cache attacks [9, 16, 42, 35, 54], demonstrating that the isolation between the TEE and the rest of the system is limited due to shared hardware. A very powerful attack technique in the TEE threat model is single-stepping. The ability to single-step TEEs can be used to mount attacks such as instruction counting [36], determining executed instructions through timing [51], targeting specific code parts with side channels [15], or enabling new side-channel attacks that would otherwise suffer from too much unrelated code being executed [35, 32]. On SGX, Van Bulck et al. [52] use APIC timer interrupts to force enclave exits for single-stepping. Given these high-impact works, single-stepping is often considered the most powerful attack primitive on TEEs due to the instruction-granular control over the victim's execution.

Wilke et al. [56] demonstrated the first single-stepping attacks on AMD SEV-SNP VMs. However, Intel TDX includes a mitigation against single-stepping, which detects single-stepping attempts and steps a random number of guest instructions. To bypass this detection, Wilke et al. [55] decrease the core frequency, allowing them to, again, use the timer interrupt for the first single-stepping attack on TDX. Intel mitigated their attack

with TDX module version 1.5.06 by introducing Instruction-Count Single-Step Defense (ICSSD). ICSSD utilizes performance counters for the single-stepping detection instead of the time-stamp counter, safeguarding the detection mechanism against CPU frequency manipulations [24, §17.3.2]. Wilke et al. [55] also present a second attack primitive that is not mitigated by ICSSD. However, it only leaks fuzzy information about the number of executed instructions and does not enable single-stepping. Thus, currently, there is no known single-stepping primitive for Intel TDX thwarting the most powerful attacks on TEEs.

In this paper, we introduce TDXploit, a novel single-stepping attack on Intel TDX that completely bypasses Intel's recent ICSSD defense. TDXploit exploits a fundamental flaw in Intel's single-stepping mitigation, which, ironically, results in a higher single-stepping accuracy (>99.99 %) than without any mitigations without relying on side channels. Our attack exploits two flaws in the random number generator (RNG) of the mitigation: First, we exploit that the RNG state is *per-core*, not per-TD. Second, we exploit that the state of the RNG can be reverse-engineered with only 32 samples. TDXploit continuously triggers Intel's single-stepping mitigation using an attacker-controlled TD to recover the RNG state and predict all future outputs, *i.e.*, the number of instructions executed by the next invocation of the mitigation. When the next output is '1', we schedule the victim TD, which then, ironically, is single-stepped by the mitigation itself. Afterward, we pause the victim until the RNG state allows for the next single-step. In contrast to prior work, this also enables controlled multi-stepping of TEEs for the first time.

We evaluate TDXploit in a microbenchmark on 2 end-to-end attacks and 6 state-of-the-art side-channel attacks. For this purpose, we systematically analyze the viability of 6 state-of-the-art side-channel attack techniques on Intel TDX, comprising several cache attack variants and the PortSmash attack. We find that 4 of the analyzed attacks work in host-to-guest attacks, *i.e.*, in the traditional TEE threat model. We also find that 2 of the analyzed attacks work in guest-to-host attacks, and 2 work in guest-to-guest attacks, *i.e.*, in the malicious TEE threat model, which has not been studied for Intel TDX before. During our analysis, we discovered that the `clflush` instruction ignores all the architectural isolation used by TDX, allowing the host to perform Flush+Flush attacks on the TD memory. Compared to non-TEE Flush+Flush attacks, this does not require shared memory and can target any TD memory location. Our results indicate a change in the microarchitecture on 5th generation Xeon Scalable

CPUs, as prior work [1, 55] that primarily analyzed 4th generation CPUs found `clflush` ineffective for flushing TD memory. Unlike cache attacks demonstrated in these works, our `clflush`-based attack does not depend on parts of the coherence protocol that will most likely be abandoned with future CPU generations [24, §9.8]. We demonstrate the impact of our Flush+Flush attack as a standalone primitive by performing a full key recovery on the OpenSSL AES T-Table implementation that leaks the secret key after 8 986 encryptions.

Our microbenchmark of the single-stepping primitive shows that we can perform one single-step every 3.7 ms and successfully single-step in >99.99 % of cases. We perform an end-to-end attack on a time-based one-time password (TOTP) token generation library used in prior work on AMD SEV [15]. While their attack uses performance counter leakage that is already mitigated on Intel TDX, we use the Flush+Flush primitive in combination with TDXploit to leak TOTP secret keys with a single trace. In a second end-to-end attack, we re-create the attack on OpenSSL ECDSA by Wilke et al. [55] that Intel mitigated with ICSSD, showing again that TDXploit can reliably bypass Intel's mitigation. In line with previous results, we leak the full secret key by observing 33 biased signatures, which requires approximately 200 000 signature generations due to the low probability of the event that introduces the bias.

In summary, our work makes the following contributions:

- We introduce TDXploit, exploiting a fundamental flaw in Intel's single-stepping mitigation that not only can be bypassed but even yields a significantly more reliable single-stepping attack than without any mitigation.
- We systematically analyze 6 state-of-the-art side-channel attack techniques and their effectiveness with Intel TDX in host-to-guest, guest-to-host, and guest-to-host attack scenarios.
- We discover that `clflush` works on Intel TDX private memory and demonstrate a Flush+Flush attack on TDX guest physical memory, *i.e.*, we do not require shared memory and can attack all memory of the TD.
- We mount 2 end-to-end attacks with TDXploit, leaking TOTP secret keys with a *single trace* and re-enabling the previously mitigated attack on OpenSSL ECDSA.

Outline. We provide background in Section 2. In Section 3, we introduce and evaluate our novel single-stepping attack TDXploit. Next, we show Flush+Flush on TDX private memory and the potential of the attack in Section 4. Afterward, we evaluate the applicability of several other state-of-the-art side-channel attacks on TDX in Section 5. We discuss our results and their impact in Section 6. Finally, we conclude in Section 7.

Responsible Disclosure. We responsibly disclosed our findings to Intel on November 27, 2024. Intel confirmed the issue on January 22, 2025. They plan to publish the vulnerability in August 2025.

2. Background

In this section, we first discuss different trusted execution environments (TEEs). We then briefly discuss side channels, in particular in the context of TEEs. Finally, we discuss single-stepping as a powerful primitive for attacks on TEEs.

2.1. Trusted Execution Environments

TEEs are specialized environments that offer increased confidentiality and integrity guarantees, even against privileged or physical access [22, 5, 24, 7]. TEE technology is often seen as split into two generations: The first generation targets primarily personal computers, with the goal of protecting application components from a malicious user or compromised system. This is particularly interesting for storing and handling sensitive data, e.g., fingerprints or cryptographic material, or for digital rights management (DRM). Widely used examples are Intel Software Guard Extensions (SGX) [22] and ARM TrustZone [8], which have been practically deployed on millions and billions of devices. The second generation targets primarily cloud servers, with the goal of protecting entire VMs from a malicious or compromised host. These VMs are also called confidential virtual machines (CVMs). Practically deployed are AMD Secure Encrypted Virtualization (SEV) [4] and Intel Trust Domain Extensions (TDX) [23]. However, Arm also published a draft for their own TEE for the cloud scenario called Confidential Compute Architecture (CCA) [7]. Confidential virtual machines do not require writing code specifically for the TEE;

instead, they allow the entire VM to run without modification in a secure environment.

2.2. Intel TDX

Intel Trust Domain Extensions (TDX) is Intel's second generation TEE, which allows running entire VMs as TEEs on Intel CPUs [24]. These guests are also referred to as trust domains (TDs). Guest memory and stored guest state are encrypted and managed by the TDX module in the trust domain virtual processor state area (TDVPS). The TDX module, an open-source software module signed by Intel, runs in the new SEAM root execution mode protected from the host and handles interactions between the TEE and the host, e.g., to create VMs, map pages, and run VMs. For VM management, Intel TDX uses the existing Intel virtual machine extension (VMX). To protect the guest's memory while still allowing for fast communication with the host, the guest's physical memory is split into a private encrypted part, only accessible by the guest and the TDX module, and a shared part, accessible by the host and the guest. The page tables of the private memory are managed by the TDX module, while the host manages the page tables of the shared memory. The guest can differentiate between shared memory and private memory through a bit in the guest physical address, making it possible for the guest to only interact with unsafe shared memory when it intends to. Furthermore, shared memory is only readable and writable for the guest but not executable to avoid certain bug types. Intel TDX makes use of Intel's Total Memory Encryption - Multi Key (TME-MK) to encrypt the memory of TDs. To this end, the key ID (HKID) range of TME-MK is split into a public and a private range, with the private range being exclusive to the TDX module and TDs. To protect the private memory against corruption, e.g., through Rowhammer, each 64-bit memory region can be protected with a cryptographic MAC that is automatically validated on each memory access. If the integrity check fails, an exception is thrown. Still, the underlying hardware of the CPU itself and the memory subsystem are shared across mutually untrusted TDX guests as well as between the TEE and any host workloads.

2.3. Attacks on Trusted Execution

Both practically and scientifically, TEEs are an attractive attack target. The reason for this is the combination of holding the most valuable security assets on the one side and, on the other hand, operating in a threat model that permits high-privilege adversaries, e.g., attacks from the kernel level. Consequently, numerous attacks have been published attacking TEEs. The first side-channel attacks on SGX were cache attacks on SGX enclaves by Brasser et al. [9] and Götzfried et al. [16], as well as cache attacks from SGX enclaves by Schwarz et al. [42]. Weiser et al. [54] demonstrated an attack on RSA key generation in SGX. Moghimi et al. [35] showed that the SGX threat model even allows amplifying cache side channels, given the ability of the host to control enclave execution. Wang et al. [53] presented a systematic analysis of many side channels in the context of SGX enclaves. Evtyushkin et al. [14] and Huo et al. [20] exploited the pattern-history table (PHT). Similarly, Lee et al. [28] presented a side channel exploiting collisions in the branch-target buffer (BTB). Both the PHT and the BTB were later on also exploited in Spectre attacks [27], e.g., by Chen et al. [11] on SGX enclaves. Other works demonstrated software-based power side channels [32], controlled channels [57], and the interrupt side channel [52]. Lou et al. [33] and subsequently Gast et al. [15] investigated leakage from AMD-SEV CVMs through performance counters. Gast et al. [15] demonstrated several attacks, including the recovery of RSA keys as well as the leakage of TOTP tokens and TOTP keys. Similar to many other works, they relied on precise single-stepping of TEEs, as we detail in Section 2.4. Fewer works explored the possibility of attacks from the inside of TEEs [42, 44, 17, 26], mainly focusing on side channels and Rowhammer attacks.

2.4. Single-Stepping Trusted Execution

Crucial for the success of many attacks on TEEs is the possibility to reliably single-step [48, 11, 43, 38, 49, 32, 59, 56, 55, 15]. Hence, single-stepping frameworks have been published both for Intel SGX and AMD SEV [52, 56], building a cornerstone for many sophisticated attacks [32, 49, 59, 56, 15, 55]. Van Bulck et al. [52] used the APIC timer to interrupt SGX enclaves continuously after each executed instructions. Wilke et al. [56] implemented a similar mechanism for AMD SEV. In response, Intel's TDX module contains a mitigation against single-stepping attacks [24].

The original version of this mitigation uses the timestamp counter and instruction pointer differences to detect single-stepping attempts. If TDX detects a single-stepping attempt, it masks all external interrupts and single-steps the guest for a random number of instructions before returning control back to the host. Wilke et al. [55] bypassed this initial single-stepping mitigation in Intel TDX by reducing the CPU frequency and counting VM entries through a side channel. Consequently, with TDX module version 1.5.06, Intel improved their defense by using performance counters to more effectively detect single-stepping attempts. While this mitigates the single-stepping attack by Wilke et al. [55], it still leaks information about the number of instructions the mitigation executed in the TD.

3. TDXploit

In this section, we present TDXploit. First, we detail the inner workings of the Intel TDX single-stepping mitigation. Second, we describe our novel single-stepping attack TDXploit and how it exploits the mitigation. Third, we evaluate TDXploit by evaluating it against a synthetic target and by re-creating a previously demonstrated end-to-end attack against the ECDSA implementation of OpenSSL. Lastly, we discuss possible mitigations for TDXploit.

3.1. Single-Stepping Mitigation

To combat single-stepping, Intel TDX includes a mitigation as part of the TDX module [25]. The mitigation consists of two parts: the detection of a single-stepping attempts by the host and the introduction of noise in case of a detected attack.

For detection of single-stepping attacks, the mitigation focuses on interrupt-based VM exits (external interrupts, non-maskable interrupts, system management interrupts, and INIT interrupts), as past single-stepping attacks are based on triggering an external interrupt after one instruction in the guest is executed [52]. Unlike synchronous VM exits, such as VM calls, interrupts can occur at any time and are often required to be handled as soon as possible. Otherwise, it might result in the slowdown of other cores, e.g., for TLB shootdowns, or the slowdown of external devices. Hence, interrupts are used in existing single-stepping attacks as a simple way to

force the control back to the untrusted host at the attacker's will. On VMX, an interrupt-based VM exit would normally result in control being forwarded back to the host, which then handles the interrupt. However, with TDX all VM exits return control to the TDX module instead. The TDX module will eventually invoke the untrusted host to handle the exit. Intel TDX employs two methods to detect malicious interrupts. Which method is used depends on the guest configuration. The first method uses a heuristic to detect potential attacks. If the last VM entry occurred less than 2 µs to 3 µs ago and the instruction pointer changed by less than 32 B (two times the maximum length an instruction in x86 can have), the TDX module assumes an attack is in progress. The time required for a VM entry and a VM exit is highly dependent on the CPU and the frequency the CPU is running on, with the latter being fully controllable by the untrusted host. Therefore, this detection approach can be bypassed on CPUs that take more than 2 µs to 3 µs to execute a VM entry and exit on their minimum operating frequency [55].

The second detection method, Instruction-Count Single-Step Defense (ICSSD), is Intel's response to the aforementioned attack. ICSSD takes advantage of performance counters to determine the number of instructions executed by the guest since the last VM entry. If this number is too low, it decides that an attack is occurring and triggers the mitigation. While ICSSD is significantly more reliable than the heuristic-based approach, it is only activated if the guest is not allowed to use performance counters. Otherwise, the mitigation falls back to the previously described heuristic.

When a potential attack is detected by the TDX module, noise is introduced to thwart the attack. This noise consists of a random number of instructions, between 1 and 32, being executed in the guest. The random number is generated through a per core 32 bit LFSR (linear-feedback shift register), initialized only once when the TDX module is loaded. LFSRs output the least significant bit of their state and generate new bits through an XOR of multiple bits at fixed positions of their internal state. To ensure that the correct number of instructions are executed in the guest, TDX uses the VMX monitor flag to single-step the guest. During this process, all external interrupts are masked, and control is **not** returned to the untrusted host. Consequently, the host is unable to determine the exact number of instructions executed within the guest, effectively mitigating single-stepping-based attacks [25].

Unlike previous single-stepping attacks on TEEs, such as TDXdown [55] and SGX-Step [52], TDXploit does not rely on cache invalidation for

reliability. Therefore, prefetching address translations and other information every time a guest continues, as proposed by the AEX-Notify [12] mitigation for SGX, would not mitigate TDXploit. Informing the guest about frequent single-stepping attempts would allow the guest to react accordingly to a potential attack, but in itself does not prevent single-stepping.

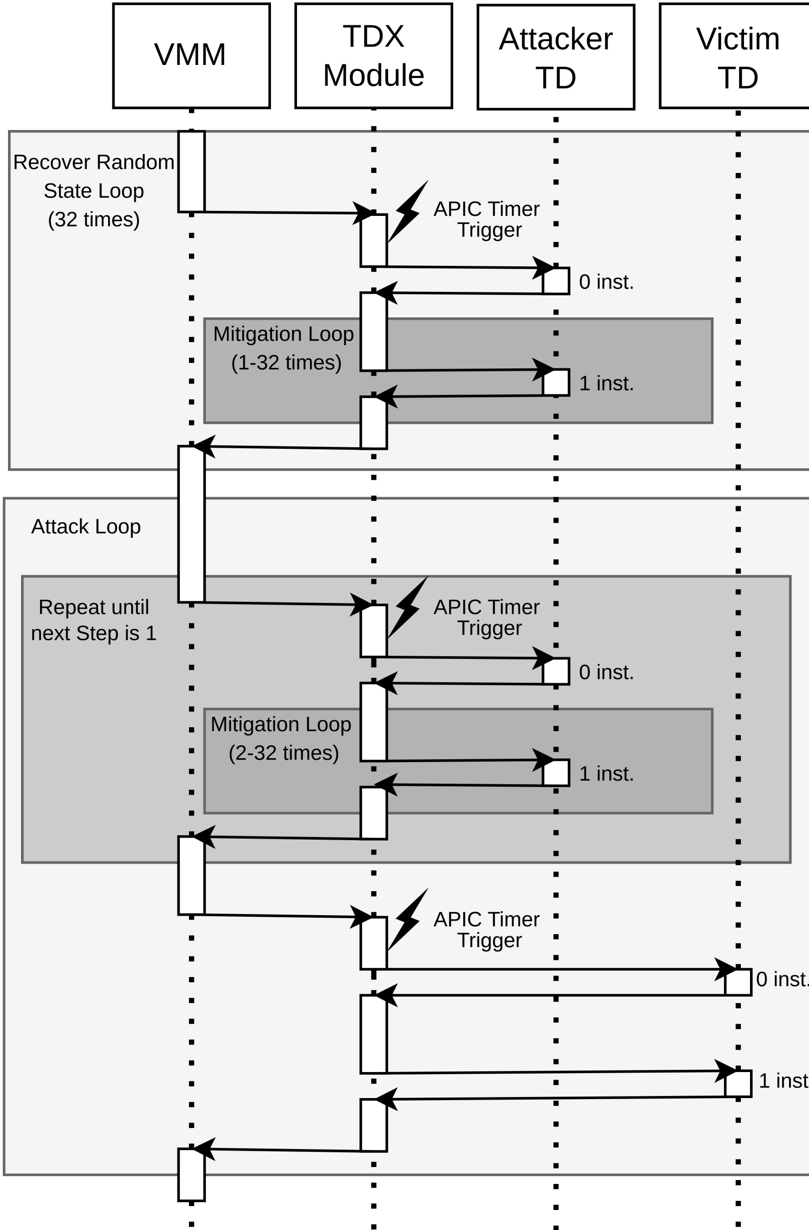
3.2. Threat Model

We assume the classical TEE threat model of a compromised host [43, 38, 48, 55, 56, 52, 59]. Specifically, we exploit that the host has full control over TD scheduling, can launch its own host-controlled TDs, and is able to arbitrarily trigger external interrupts, e.g., through the APIC timer. The CPU runs the most recent version of the TDX module at the time of writing (TDX module 1.5.06 [25]). The victim guest enforces that the ICSSD method for single-stepping detection is enabled and that hyper-threading is disabled. These assumptions follow the TDX threat model provided by Intel [23], which assumes a compromised cloud provider.

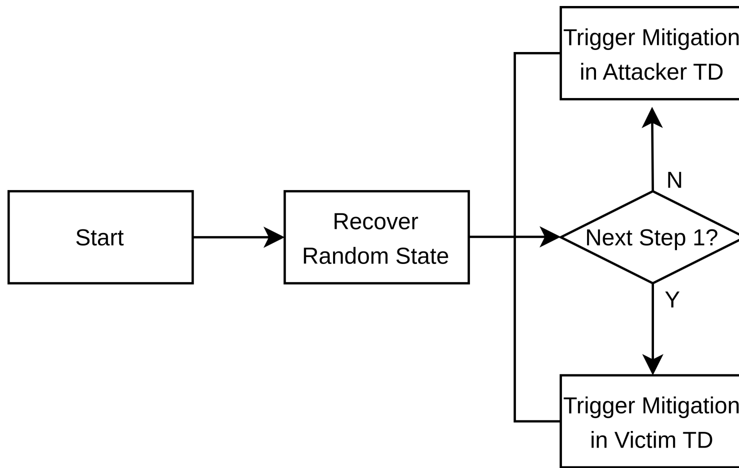
3.3. Attack

A high-level overview of our TDXploit attack is provided in Figure 8.1 as a flow chart in Figure 8.1b and a sequence diagram in Figure 8.1a visualizing the internal details of the attack. Contrary to previous single-stepping attacks on TEEs [55, 56, 52], TDXploit exploits the attacker's ability to spawn malicious TEE instances. The attack code consists of two parts. First, the compromised virtual machine monitor (VMM) is used to modify the scheduling of the TDs and to trigger the single-stepping mitigation through external interrupts. Second, an attacker-controlled TD is used to relay information about the current mitigation state to the VMM.

To mount the attack, first, the attacker has to recover the LFSR state on the logical CPU core used for the attack, as shown in the upper part of Figure 8.1a. As an LFSR outputs its internal state as a random number and the updates to the state consist of linear operations, the full state can be reversed by observing the output. The polynomial used for the LFSR is `0xB4BCD35C` and is publicly available [25]. The 5 least significant bits of the LFSR output plus 1 is the number of single-steps the mitigation does before returning to the host. The plus 1 is required to ensure that at



(a) Sequence Diagram



(b) Flow Chart

Figure 8.1.: A flow chart (Figure 8.1b) for a rough overview of TDXploit and a sequence diagram Figure 8.1a providing more detailed information. First, we recover the internal RNG state by triggering the mitigation on the attacker TD and reversing the state. Next, we schedule the attacker TD and trigger the mitigation until the RNG state is not 1. Only when the next random number is 1 the victim TD is scheduled.

```
1 1: xor rax, rax;     \\TDCALL leaf VMCALL
2 mov rcx, 0xff00;     \\TDCALL register bitmap
3 mov r11, 42;         \\VMCALL code (custom)
4 tdcall;
5 .rept 32;
6 add qword [rsi], 1; \\add to shared memory
7 .endr;
8 jmp 1b;              \\reset
```

Listing 8.1: Attacker TD assembly loop to leak the LFSR state.

least one instruction is executed. To recover the output of the LFSR, we exclusively schedule our attacker-controlled TD and continuously trigger the mitigation. The attacker TD runs the small code gadget depicted in Listing 8.1 to report the number of executed instructions to the VMM. In more detail, this step works as follows. The attacker TD signals the VMM through a VM call that it is ready for a measurement. To start the measurement, the VMM sets up the APIC timer to immediately trigger an external interrupt and resumes the attacker TD. As interrupts are masked until the attacker TD is entered, the pending interrupt causes a VM exit before the attacker TD can execute an instruction. As no instruction is executed in the guest, the single-stepping mitigation is triggered by ICSSD. The instructions in the attacker TD following the VM call consist of multiple `add` instructions that increase a counter residing in an unencrypted memory location that is shared with the VMM. When control is returned to the VMM, they can infer the number of instructions executed by the mitigation by computing the delta of the counter in the shared memory region.

We use the least significant bit of the number of instructions executed by the mitigation for the LFSR state recovery. While it is possible to use all 5 recovered bits for the state recovery, this would only marginally improve the attack, as each new mitigation trigger only reveals one new additional bit. As we recover one bit per mitigation trigger, we schedule the attacker TD and trigger the mitigation 32 times to recover the full LFSR state. With the state recovered, we are able to predict the mitigation behavior on this core. This information can already be directly used for instruction counting attacks on victim TDs by triggering the mitigation every time the victim TD is scheduled. However, by abusing control over scheduling, this can also be used to carry out single-stepping attacks, as depicted

in the lower part of Figure 8.1a. Again, the attacker TD and the victim TD are scheduled to run on the same logical core, exploiting the shared LFSR state. The VMM constantly triggers the mitigation and schedules the attacker TD until the next invocation of the mitigation would execute exactly one instruction. At this point, the VMM schedules the victim TD. Afterward, the VMM again schedules the attacker TD until the mitigation allows for the next single-stepping opportunity. The resulting attack, TDXploit, is highly reliable, as the number of instructions executed is architecturally ensured by the TDX module, removing the possibility for unintentional zero-steps or multi-steps entirely. Furthermore, we do not rely on any side channels, removing possible sources of noise and inaccuracies. To perform targeted multi-steps, e.g., to skip uninteresting victim code, the same method can be used by scheduling the victim when the next random number equals the desired number of steps. This provides a middle ground between the fast but coarse-grained page fault-controlled channel and fine-grained but slow single-stepping.

3.4. Evaluation

In this section, we evaluate the speed and accuracy of TDXploit and demonstrate its feasibility in an end-to-end attack against the ECDSA implementation of OpenSSL. As previously mentioned, the evaluation was performed on an Intel Xeon Silver 4514Y running Ubuntu 24.04 using the TDX-enabled software stack from Canonical [10]. In line with prior work [56, 55], we assume a VM with a single core to ease the implementation effort.

Accuracy In our first evaluation scenario, we use the established [52, 56, 55] synthetic example of a tight loop consisting only of `nop` instructions. We single-step the loop for a total of 85 000 000 instructions and observe only 2 misclassifications, achieving an accuracy of >99.99 %. A single-step takes 3.7 ms ($\sigma_{\bar{x}} = 0.06$ ms, $n = 10\,000$) on our system allowing for 270 single-steps per second.

Attack on OpenSSL In this section, we use our TDXploit primitive to perform the end-to-end attack against the modular reduction implementation of ECDSA curves in OpenSSL that was presented in [55]. ECDSA is a signature scheme that requires a nonce for each signature, which

must remain secret. In addition, the nonce must be smaller than the group order of the ECDSA curve. One approach for generating such a nonce is modular reduction, where an initial random value k' is reduced until it matches the imposed size limitations. The attack exploits that the modular reduction code of the ECDSA implementation in OpenSSL executes a different amount of instructions depending on the secret value k', which leaks information about the final nonce k used for the signature. For the brainpoolp224r1 curve, certain instruction counts leak the values of the 7 most significant bits of the nonce k, as detailed in Listing 8.2. For brainpoolp224r1, executions where the control flow passes Line 7 but not Line 11 correspond to signatures with a biased nonce. Such nonce biases can be used to recover the secret key [2]. The attack requires 33 biased signatures, which amounts to approximately 200 000 observed signature generations due to the low probability of the event. The vulnerability was at least present since OpenSSL version 3.2.0, and, to the best of our knowledge, there was no constant time alternative available. OpenSSL mitigated the attack by switching to rejection sampling starting with version 3.3.1. To orchestrate the attack, we use the attackers' ability to force and observe page faults to look for a unique page fault pattern that indicates that the vulnerable function is about to execute. This pattern has previously been determined in an offline step. Afterward, we single-step the victim TD until the end of the vulnerable code section is reached, which we again infer via page faults. For our PoC, we assume that the attacker has already located the OpenSSL library in the GPA space of the TD as, e.g., demonstrated in [37, 30, 29, 31]. With single-stepping we need 119.7 ms per signature without we need 0.062 58 ms per signature. Compared to the now mitigated single-stepping attack in [55], our attack is slower by a factor of 4. We did not further analyze the cause of the performance difference.

3.5. Mitigations

Unlike existing single-stepping attacks that are based on external interrupts forcing control back to the host, TDXploit exploits the single-stepping mitigation itself to achieve reliable single-stepping. There are multiple parts in the TDX single-stepping mitigation that make TDXploit possible and need to be changed to not only mitigate TDXploit but also make similar future attacks more challenging.

```
1 int bn_div_fixed_top(BIGNUM* dv, rm, num, divisor,
  -> BN_CTX *ctx) {
2 for (i = 0; i < loop; i++, wnumtop--) {
3   for (;;) {
4     if ((t2h < rem) ||
5         ((t2h == rem) && (t2l <= n2)))
6       break;
7     q--;
8     rem += d0;
9     if (rem < d0) //don't let rem overflow
10      break;
11    if (t2l < d1)
12      t2h--;
13    t2l -= d1;
14  }}};
15
```

Listing 8.2: Simplified version of bn_div_fixed_top from `openssl/crypto/bn/bn_div.c`. This function divides *num* by *divisor* and is called during the nonce generation. Based on figure from [55]

Improved RNG LFSRs are very useful for generating number sequences, but they are very bad random number generators (RNGs), especially for security-related purposes such as Intel's mitigation. The linearity of LFSRs makes their internal state trivial to reverse. Furthermore, they output part of their internal state directly as the random number. Replacing the LFSR with a proper pseudo RNG would prevent state recovery entirely. While LFSRs have the advantage of being extremely fast in generating numbers, performance is not an important factor for the RNG used in this mitigation. As the mitigation, if triggered, results in multiple extremely expensive VM entries and VM exits, the additional overhead of a proper pseudo RNG would not significantly affect the overall overhead of the mitigation. Furthermore, the mitigation, and therefore the random number generation, is rarely triggered under normal execution, which is the main reason a mitigation with such a large overhead is even viable to begin with.

Per vCPU RNG State TDXploit relies on an attacker-controlled TD to recover the RNG state and to skip undesirable step counts. This only works because of the per-core RNG state. Using a per TD or per TD vCPU RNG state would prevent our attack approach, even if a bad RNG like an LFSR is used. While the former requires slightly less memory, the latter

would work without locking. Furthermore, the TDVPS mechanism already allows storing per TD vCPU state, making the addition of the RNG state an easy change. However, the StumbleStepping attack from Wilke et al. [55] might still be used to recover the LFSR state, although its noisy signal might make reversing the LFSR more challenging. In summary, using a per TD (vCPU) RNG state significantly reduces the attack surface but is not sufficient on its own to protect a weak RNG like an LFSR.

Improved RNG range A larger value range for the amount of instructions executed by the mitigation would make it harder for any attacker to exploit side effects of the mitigation. In its current state, the mitigation already excludes the number 0 from its random number generation, as it would result in no progress. We would further recommend removing at least 1 as well, as this would result in a single-step, which is what this mitigation is trying to actively defend against. Even with this change, the range of noise the mitigation introduces is very limited, with a maximum of 32 steps. We suspect that the main reason for this low number is the enormous overhead each additional step introduces with the currently used stepping mechanism. Thus, we propose changing the mechanism the TDX module uses to execute additional instructions inside the TD. The VMX-preemption timer could be a promising solution. This is a counter that decrements with every cycle executed in the guest and triggers a VM exit when it reaches 0. Thus, the number of VM entries and exits is constant instead of scaling with the number of executed steps. As a result, significantly more instructions could be executed in the guest without any additional performance overhead. This mechanism would also significantly reduce the attack surface for side channels that try to infer the amount of executed instructions. As the VMX-preemption timer relies on the time stamp counter (TSC) and not on instructions executed, the performance counter already used for the ICSSD detection method can be used to ensure that a certain range of instructions were executed eliminating the possibility of a vulnerability similar to TDXdown [55]. The preemption timer would, therefore, only speed up the instruction execution, while the performance counter ensures that enough instructions are executed.

An alternative to the TSC-based preemption timer would be performance monitoring interrupts (PMIs). PMIs can be configured to trigger whenever a performance counter overflows. With one performance counter already counting instructions retired for ICSSD, the same counter can be used to trigger a PMI after a random number of instructions executed when the mitigation is active. As PMIs can only be delivered when they are not

masked, the CR8 register can not be used to mask all external interrupts. Alternatively, the "acknowledge interrupt on VM exit" feature can be enabled for the TD to avoid returning to the host, and encountered interrupts can be buffered by the TDX module. The buffered interrupts can then be triggered again through self-IPIs (inter-processor interrupts) directly before returning to the host when the mitigation is done. Similar to the VMX-preemption timer method, this would also not require any hardware changes while at the same time significantly increasing the possible number of instructions executed by the mitigation without the high overhead of constant VM entries and VM exits.

In summary, we recommend using a secure RNG that cannot easily be reversed to mitigate our attack and to switch to a dedicated RNG state per TD vCPU to hedge against potential attacks. Finally, using a larger range for the amount of instructions that can be executed by the mitigation would further increase its security.

4. TDX Flush+Flush

In this section, we show that Flush+Flush can be used on TDX private memory, contrary to prior findings. First, we demonstrate that the `clflush` instruction ignores the HKIDs used by the memory encryption system. In combination with the VMM's ability to create mappings to all memory locations, this enables Flush+Flush attacks on arbitrary TD memory without requiring shared memory. To demonstrate the effectiveness of this attack, we perform a last-round AES T-Table attack on OpenSSL from the host on a TDX guest, recovering the full encryption key.

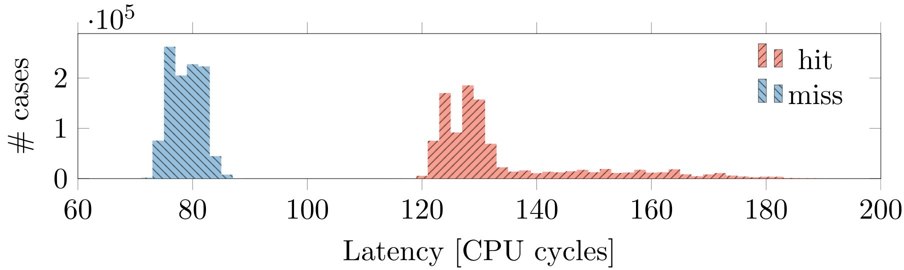
4.1. Attack

The Flush+Flush attack introduced by Gruss et al. [18] exploits the timing difference between a `clflush` on a cached memory location and an uncached memory location. If the memory location that is being flushed is in the cache, `clflush` has to evict the cache line, causing an increased latency. Using this time difference Flush+Flush can detect if a memory location has been recently accessed. Furthermore, Flush+Flush is fast and has no blindspot, making it a very powerful attack primitive [41]. The main disadvantage of traditional Flush+Flush is that it requires shared memory between the attacker and the victim.

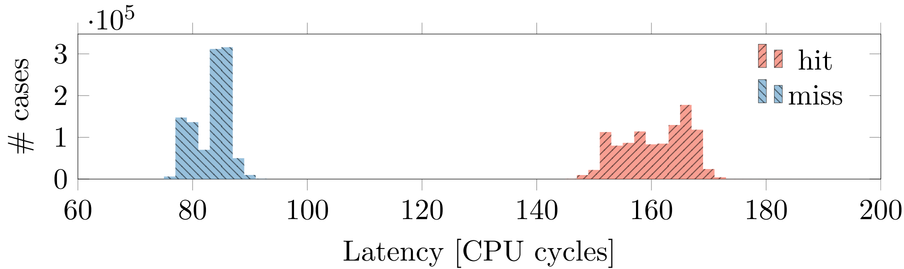
On Intel TDX, guest private memory is encrypted using TME-MK, which allows the use of multiple encryption keys specified in the upper parts of the physical address to encode the HKID. With TDX, any read access by the host to a memory location encrypted with a private HKID returns all-zero data. Returning all-zero data instead of the ciphertext or the decryption of the ciphertext with the public HKID is crucial for preventing ciphertext side-channel attacks [31]. Nonetheless, accessing a memory location with a different HKID still loads the data into the cache. Since the HKID is encoded in the physical address bits and thus influences the cache tag, it, in principle, enables multiple decryptions of the same memory location to reside in the cache at the same time. However, an additional coherence mechanism flushes any existing cache entries that only differ in the HKID. Prior work [1, 55] exploited this mechanism to build a cache attack where the attacker continuously loads the targeted address with a different HKID to observe latency spikes due to the coherence protocol. Intel implied that they plan on removing this coherence mechanism in future CPUs [24].

While the TDX documentation implies that `clflush` on TDX private memory should not be possible [24, §8.5.1], we discovered that the `clflush` instruction ignores HKIDs and flushes all cache lines related to a physical address. Prior work that primarily analyzed 4th generation Xeon Scalable CPUs reported the expected `clflush` behavior, leading us to assume that only 5th geneneration Xeon Scalable CPUs upwards are affected by our attack. As TDX-enabled 4th generation Xeon Scalable CPUs are not publicly available, we are unable to verify this assumption.

The results of our measurements are shown in Figure 8.2. Figure 8.2a contains the hit-miss histogram for Flush+Flush in a native scenario without TDX. Cache misses (memory not cached) require $\sim$80 cycles, and most cache hits require $\sim$130 cycles with outliers in the range of 135 to 180 cycles. Figure 8.2b contains the hit-miss histogram for Flush+Flush on TDX private memory with a host attacker. The host executes `clflush` on the target physical memory using the zero HKID, while the victim TD accesses the memory through its private mapping. Similar to the native scenario, the misses are at $\sim$80 cycles, which is to be expected as nothing has to be done. The hit case is at $\sim$160 cycles and thus significantly higher than the $\sim$130 cycles required for the majority of cases in the native scenario. As we also observed multiple instances in the native scenario where `clflush` required $\sim$160 cycles, this might be a less optimal case for `clflush` that is reliably triggered by flushing TD private memory.



(a) Native



(b) TDX

Figure 8.2.: Flush+Flush hit-miss histograms for a regular native attack (Figure 8.2a) and a TDX host attacker targeting a guest VM (Figure 8.2b). The miss (not cached) timings for both scenarios are in a similar range. On average, the hit (cached) timings for the TDX attack are 30 cycles higher, falling into the higher end of measured hits in the native scenario, increasing the hit-miss margin significantly. The increased hit timings could be the result of overhead introduced with the memory encryption used for TDX guest memory.

Not only is the timing increased for `clflush` if the private memory is cached, but the cache line is also evicted, as shown in Figure 8.3. The L1 hit timings in the TD are at 52.4 cycles, with the miss timings if the guest flushes its own memory at 252.5 cycles. After the host evicts the guest memory with a different HKID, the average access time for the guest is 250.6 cycles, which is almost identical to the timings of a guest flush. Therefore, it is likely that guest memory was evicted by the host flush.

4.2. Evaluation

In this section, we evaluate the effectiveness of Flush+Flush on TDs and compare it to a native attack. To determine the potential leakage rate

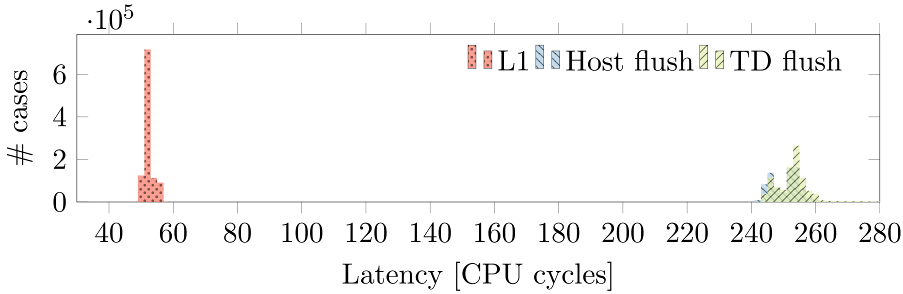


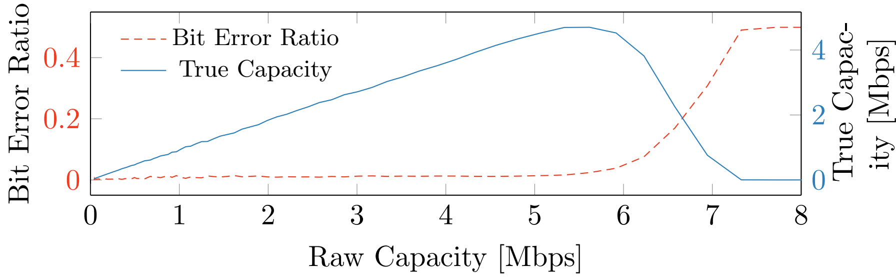
Figure 8.3.: TDX private memory access timings when the host flushes the memory using a different HKID, when the guest flushes the memory before the access, and when only L1 accesses are performed. The access timings after a host flush and after a TD flush are almost identical.

of Flush+Flush on TDs, we first build a covert channel and compare it to the native case. For further comparison, we perform an AES T-Table first-round attack on a TD and compare it to the native attack using the OpenSSL AES implementation. Finally, we perform a last-round attack on a TD and recover the full AES key. We assume the threat model described in Section 3.2.

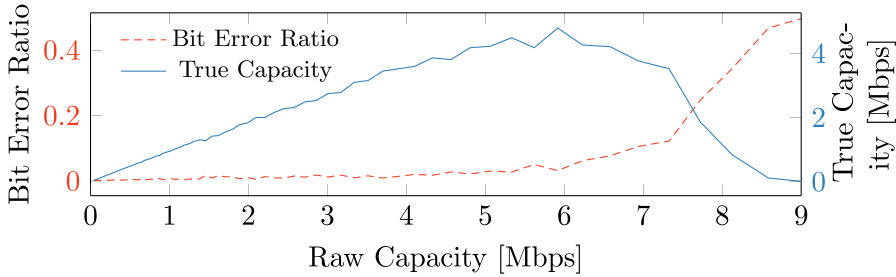
Covert Channel

Our covert channel is based on our observations discussed in Section 4.1. The TD sends data through memory accesses and the host receives data through flushing the same memory location. We use a time-sliced approach for data transmission, with one bit transmitted in each time slice. For a clock, we use the TSC, which can be accessed through `rdtsc`. While the absolute TSC value of the host and the guest may differ by a fixed offset, they still increment at the same rate [25]. We assume that the sender and receiver are perfectly synchronized at the start of the transmission as this is a separate challenge that has already been discussed and solved in previous work [34].

To find the optimal time-slice length for our native and TDX Flush+Flush attack, we perform our attack with decreasing time-slice length. With a decreasing time-slice length, the raw capacity of the transmission increases due to more bits sent in a given time frame. At the same time, the error ratio increases as time slices become too short to correctly transmit bits.



(a) TDX



(b) Native

Figure 8.4.: Covert-channel true capacity and error ratio with different raw capacities for a Flush+Flush covert channel, executed natively (Figure 8.4b) and with Intel TDX with the TD sending and the host receiving (Figure 8.4a). The channel on TDX reaches its maximum true capacity of 4.6 Mbit/s at a raw capacity of 5.6 Mbit/s and the native channel reaches its maximum of 4.8 Mbit/s at a raw capacity of 5.9 Mbit/s.

We use the binary symmetric channel model to compute the true capacity based on the error ratio and the raw capacity and determine the optimal time-slice length based on this.

The results of these measurements are provided in Figure 8.4. Our native Flush+Flush attack has a true capacity of 4.8 Mbit/s ($\sigma_{\bar{x}} = 0.01$ Mbit/s, $n = 20$) with an error ratio of 3.2 % ($\sigma_{\bar{x}} = 0.7$ %, $n = 20$) at a raw capacity of 5.9 Mbit/s. Our Flush+Flush attack with a host receiver and a TDX guest sender has a true capacity of 4.6 Mbit/s ($\sigma_{\bar{x}} = 0.01$ Mbit/s, $n = 20$) with an error ratio of 2.6 % ($\sigma_{\bar{x}} = 0.06$ %, $n = 20$) at a raw capacity of 5.6 Mbit/s. The Flush+Flush on TDX is only marginally slower than the native version, most likely due to the slightly higher average hit-timings,

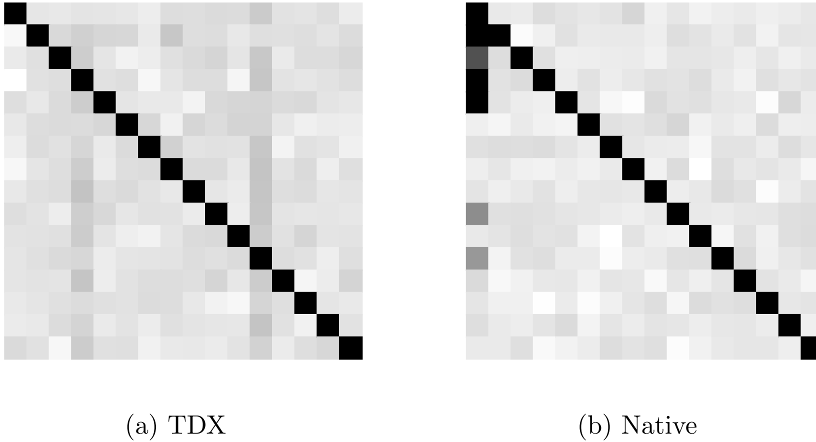
as discussed in Section 4.1 and slightly worse synchronization, due to the lack of a shared TSC in the TDX-based attack.

AES T-Table Attack

To demonstrate the effectiveness of Flush+Flush on TDX private memory, we perform an AES T-Table attack on OpenSSL. Contrary to a native Flush+Flush, a malicious VMM can not rely on shared memory for an attack, as shared libraries used inside a TD would normally be loaded into private memory, inaccessible by the host. First, the attacker has to find the page containing the T-Tables of the guest's AES implementation. As the guest's private memory is not readable to the host, we use Flush+Flush to search through the guest's physical address space for the T-Tables. During a typical AES encryption, almost all cache lines in a T-Table are accessed. To find the correct memory locations, we use the page offset for the T-Tables in the AES library and search for a large amount of cached memory that has the size of the T-Tables at these offsets after the AES encryption has been executed by the guest. The execution of the AES code can be detected through different ways, e.g., network transmissions or page fault tracking. To filter out false positives due to other applications, we repeat our measurements several times. After only 10 encryptions, we can determine the correct memory location with 99.2 % accuracy on our system. This search method is similar to the one proposed by Gruss et al. [19] for cache template attacks.

To visualize the accuracy of the tested attacks, we perform a first-round attack with known plaintext on one T-Table with a zero key. The access number heatmaps for the T-Table after 10 000 encryptions for both native Flush+Flush and the TDX scenario are shown in Figure 8.5. The visibility of the diagonal indicates the amount of noise each attack has. A strongly visible diagonal means more correctly detected accesses and, therefore, less noise. As expected, both the native Flush+Flush and the TDX scenario perform almost identically, indicating no significant additional noise in the TDX-based scenario.

In addition to the first-round attack, we perform a last-round attack on a TD, assuming known ciphertext. We are able to recover the full AES key after 8 986 ($\sigma_{\bar{x}} = 119$, $n = 100$) encryptions when attacking a TD. This number is similar to the results of a native attack reported in previous work [41].



(a) TDX (b) Native

Figure 8.5.: Comparison of a first-round AES T-Table key recovery between native and TDX Flush+Flush over 10 000 encryptions with a zero key. A highly visible diagonal indicates a low amount of noise. The attack on the TD (Figure 8.5a) performs similarly well as the native attack (Figure 8.5b).

OTP Recovery

To showcase the combined capabilities of TDXploit and Flush+Flush, we use the two primitives concurrently to attack a vulnerable TOTP library [46]. Gast et al. [15] demonstrated that on AMD SEV, it is possible to leak the TOTP secret of the same TOTP library by tracking loop iterations in the base32 decoder function. They tracked the guest by monitoring performance counters while single-stepping. Although TDX does not expose guest performance counter data to the host, we can mount a similar attack using Flush+Flush.

Whenever the TOTP library verifies or generates a TOTP token, the secret is base32-decoded. To decode the secret, the algorithm iterates over each character and compares it with the entries of an internal lookup table called `OTP_DEFAULT_BASE32_CHARS` as shown in Listing 8.3. Whenever the character matches the table entry, the comparison loop breaks, and the index of the matching entry is saved. Afterwards, the algorithm continues with the next character.

We can attack this decoding algorithm by single-stepping and using Flush+Flush to monitor the physical memory location of `OTP_DEFAULT_BASE32_CHARS`. The size of the lookup table is 32 B, which is small enough

```
1     for (int k = 0; k < 32; k++) {
2       if (c == OTP_DEFAULT_BASE32_CHARS[k]) {
3         // c is the current character
4         block_values[j] = k;
5         found = 1;
6         break;
7       }
8     }
9
```

Listing 8.3: TOTP base32 decoding loop, executed for each character in the input [46].

to fit into a single cache line. Thus, we only need to monitor a single address to detect all accesses. For our evaluation, we assume the memory location to be known to the attacker as it can be found through profiling the guest's physical memory similar to our attack described in Section 4.2. While the library is comparing a character with entries in the lookup table, we detect accesses every 5 instructions. The processing of correct characters causes another branch to execute, leading to a delay of 6 instructions between accesses. By distinguishing the two access patterns, we can follow the execution of the decoding algorithm. In addition, the data is decoded in chunks of 8 characters, after which we measure an overhead of 58 instructions before the algorithm continues processing secret characters, which we need to consider when extracting the secret key.

When executing the attack, we experience additional cache hits for `OTP_DEFAULT_BASE32_CHARS`. We assume that this is caused by branch prediction, prefetching of the lookup table, and out-of-order execution. We experimentally verified this assumption by adding the `serialize` instruction before the comparison, which reduced the additional cache hits significantly. However, the detected extra accesses mostly do not interfere with the decoding process. Even without the added `serialize` instruction, we are able to correctly extract the TOTP secret key **from a single trace** in 79.2 % of cases. We achieve a 99.0 % success rate in key recovery by evaluating three traces and deriving the key using a majority vote. The single-stepping process takes approximately 9.1 s to generate a single trace. Thus, the total attack time to recover the TOTP secret is below 30 s.

This attack is only possible due to the combination of a reliable single-stepping technique and Flush+Flush. When only using single-stepping, it is only possible to determine the total number of instructions executed for

the translation of all characters, as there is no way for the host to determine which character is currently translated. When only using Flush+Flush, the temporal resolution is not enough to differentiate between memory access after 5 or 6 instructions, making it not possible to determine the exact characters of the secret or which character is currently being translated. Consequently, this is an example of how reliable single-stepping in combination with existing primitives can be used to perform attacks that would otherwise not be impossible.

5. Systematic Evaluation

While our single-stepping attack discussed in Section 3 can by itself used to mount attacks, the main benefit of single-stepping to an attacker lies in strengthening other attacks, in particular, side-channel attacks. In this section, we look at different side-channel attacks and their applicability and efficacy in the context of Intel TDX and TDXploit. We discuss the viability of the attacks in different scenarios with Intel TDX, including the traditional host-to-guest (host attacking a guest) scenario, as well as the malicious guest scenarios guest-to-host (guest attacking the host) and guest-to-guest (guest attacking another guest). We verify the viability with a covert channel for each attack in each possible scenario.

All experiments and assumptions are based on an Intel Xeon Silver 4514Y running a stock Ubuntu 24.04 with the TDX changes provided by Canonical [10]. The CPU runs the most recent version of the TDX module at the time of writing (TDX module 1.5.06 [25]). The victim guest runs an Ubuntu 24.04 created through Canonical's TDX support scripts [10].

For each covert channel, we provide the true capacity based on the error rate and raw capacity, similar to previous work [34]. The covert channels are based on a time-sliced approach using the `rdtsc` instruction as the clock. While host and guest do not share the same TSC, the TSC in TDX guests increments at the same rate as the TSC of the host [25], which suffices for synchronization. As we only build the covert channels to show that a given attack works, we do not further optimize each channel beyond the point of showing that it can correctly transmit data, *i.e.*, the goal is not to outperform channels from prior work but only demonstrate the practicality. The results of our analysis are summarized in Table 8.1.

Table 8.1.: All analyzed attacks and their applicability to Host-to-Guest, Guest-to-Host, and Guest-to-Guest attack scenarios.

Attack	Host-to-Guest	Guest-to-Host	Guest-to-Guest
Analyzed in prior Work			
HKID Coherence [1]	✓	✗	✗
Analyzed in our Work			
Flush+Reload	✗	✗	✗
Flush+Flush	✓	✗	✗
PortSmash	✓	✓	✓
Prime+Probe (L1)	✓	✓	✓
Evict+Reload	✗	✗	✗

✗ denotes that this attack is not possible. ✓ denotes that this attack works. We are the first to demonstrate all of these attacks on TDX, except for HKID coherence, which was demonstrated in prior work [1].

Flush+Reload. A typical Flush+Reload [58] attack requires shared memory between the attacker and the victim. In the guest-to-guest scenario, there is no shared memory between the victim and the attacker. With regular VMs, guest-to-guest attacks with Flush+Reload can be possible through page deduplication by the host. As page deduplication between different TDs is not possible, this eliminates the possibility for Flush+Reload between guests. Similar to the guest-to-guest scenario, in the guest-to-host scenario, there is also no shared memory between host and guest that holds relevant information. While the host can provide shared memory pages with TDX, these pages are intended to transfer data between the guest and the host. In the host-to-guest scenario, the host has access to all physical pages used by the guest. Despite this, the host cannot use the private HKID used for the guest memory for an attack. As accessing memory with a different HKID results in a separate cache line being loaded, it is technically not possible to perform Flush+Reload on a specific cache line. Instead, as we discuss later in this section, such accesses trigger a HKID coherence side channel. Hence, **Flush+Reload is not applicable in any of the three scenarios**.

Evict+Reload. Evict+Reload [19] is based on the same principle as Flush+Reload but replaces the flush-step with an eviction using an eviction set. Therefore, Evict+Reload also requires shared memory between attacker and victim, which is not available as discussed for Flush+Reload.

Hence, **Evict+Reload is not applicable in any of the three scenarios**.

Flush+Flush. Similar to Flush+Reload, a traditional Flush+Flush [18] attack is not possible in the guest-to-host and guest-to-guest scenarios due to a lack of shared memory. Contrary to Flush+Reload, Flush+Flush is possible in the host-to-guest scenario, despite the lack of a shared cache line. As outlined in Section 4, the `clflush` instruction ignores the HKID when performing the flush, allowing the host to flush cache lines belonging to private guest memory. This enables Flush+Flush attacks on arbitrary guest physical memory by the host, without requiring any shared memory. We confirmed that **host-to-guest Flush+Flush is possible** and measured its capacity in a covert channel. We achieved a true capacity of 4.6 Mbit/s, as shown in Section 4.2. Note that our Flush+Flush attack does not trigger the HKID coherence side channel and, hence, is not mitigated by any measures taken against the HKID coherence side channel. As we showed in Section 4.2, there is also no interference between Flush+Flush and TDXploit.

PortSmash. The PortSmash [3] attack relies on contention in the execution ports of a physical CPU core. Consequently, PortSmash requires the attacker and the victim to run on the same physical core at the same time, albeit on separate logical cores. The attacker detects throughput changes on the target execution port through victim code execution. In principle, a TD can mitigate this attack by enforcing that SMT has to be disabled for it to run. Similarly, the host can mitigate guest-to-host attacks by either disabling SMT or not scheduling sensitive code on the same physical core as TDs while they are active. Still, we practically confirmed that **host-to-guest, guest-to-host, and guest-to-guest PortSmash attacks are possible** and built a covert channel for each scenario. For all three attack scenarios, we use the `divsd` instruction, which is handled by execution port 1 on our CPU. The true capacities for our channels are 356.0 kbit/s (guest-to-guest), 395.9 kbit/s (host-to-guest), and 396.0 kbit/s (guest-to-host). We observed no interference between TDXploit and PortSmash when using TDXploit to target a specific part of the code.

Prime+Probe (L1). An attacker can use Prime+Probe [40] to target different caches. Initial works target the L1 cache [40], whereas later works also target inclusive last-level caches [34]. However, recent Intel server processors abandoned this design in favor of non-inclusive last-level caches, which cannot be attacked by the known Prime+Probe attacks, *i.e.*, only Prime+Probe on the L1 cache has been demonstrated on these CPUs so far [41]. Similarly, as we target an Intel Xeon Silver 4514Y, which has a non-inclusive last-level cache, like all CPUs supporting TDX, we can only target the L1 cache. We fill the L1 cache sets with attacker cache lines and detect any evictions caused by the victim. We find that **host-to-guest, guest-to-host, and guest-to-guest Prime+Probe attacks against the L1 cache are possible**, as the L1 remains a hardware component shared across security contexts and Prime+Probe does not require any shared memory. To measure the capacity of the Prime+Probe attacks, we use covert channels, achieving transmission rates of 620.1 kbit/s (guest-to-guest), 561.0 kbit/s (host-to-guest), and 526.8 kbit/s (guest-to-host). We observed no interference between TDXploit and Prime+Probe when using TDXploit to target a specific part of the code.

HKID Coherence side channel. In their security report on TDX, Google mentions a possible attack based on a coherence mechanism related to HKIDs [1]. When a memory location is loaded with one HKID, all cache lines with the same physical address and a different HKID are evicted from the cache.[1] The HKID coherence side channel can be used by the host to detect guest memory accesses, which lead to coherence-induced changes in the cache state. The HKID coherence side channel requires access to the victim's memory with different HKIDs, which is not possible from inside a TD. Hence, we find that only the **host-to-guest HKID coherence side channel is possible**. We measured the capacity of the HKID coherence side channel in a host-to-guest scenario covert channel and achieve a capacity of 3.0 Mbit/s.

[1]While the authors mention that their attack could be seen as a Flush+Reload or Flush+Flush attack, this naming is not consistent with the published literature: Loading a cache line triggers coherence and, thereby, evicts one or more other cache lines. This is not the same as cache eviction through cache or cache-set contention, *i.e.*, it is different from the eviction in an Evict+Reload attack. It is also not a flush operation which specifically removes a single specified cache line from the cache, *i.e.*, it is also different from the flush in Flush+Reload or Flush+Flush attacks. Hence, we believe the attacks by Aktas et al. [1] should rather be referred to as HKID coherence side channel to avoid confusion with other distinct and already known attack techniques like Evict+Reload, Flush+Reload, or Flush+Flush.

6. Discussion & Related Work

In this section, we discuss the impact of our findings. TDXploit, unlike previous single-stepping approaches, exploits the possibility of spawning attacker-controlled TDs. With this we can leak information from the TDX module without noise.

The possibility of malicious code inside a TEE has previously been explored by Schwarz et al. [45]. They assume that a malicious or buggy piece of code is signed by Intel and, therefore, allowed to run inside Intel SGX and investigate the possibility of attacking the host through Prime+Probe from inside the TEE. Consequently, the attack is significantly more challenging for the host to detect, as it is protected by SGX. More recent Intel CPUs no longer require enclaves to be signed by Intel, making such attacks even more realistic [21]. Van Bulck et al. [50] analyzed 8 major open-source shielding frameworks for SGX enclaves and found 35 vulnerabilities. Schwarz et al. [44] show that the unrestricted access of SGX enclaves to the memory of their host application can be abused to manipulate the host to execute arbitrary code. This makes malware almost invisible to existing detection methods, as the actual attacker code is hidden inside the enclave. Jang et al. [26] use Rowhammer to flip SGX memory from inside the enclave, leading to a processor lockdown. Gruss et al. [17] flip bits in host memory from within an SGX enclave using Rowhammer. Contrary to existing Rowhammer attacks, their attack is almost invisible to the host.

The closest work to TDXploit is TDXdown [55]. TDXdown uses a different flaw in the TDX single-stepping mitigation to bypass it. In the initial version of the single-stepping mitigation, an attack was detected only through heuristics, most notably through the TSC. The TSC-based heuristic can be bypassed by fixing the CPU to its minimal frequency, tricking the mitigation into thinking a large number of instructions were executed in the guest. They then applied the APIC timer-based single-stepping technique already used in previous work, such as SGX-Step [52] and SEV-Step [56]. This bug is mitigated in the current TDX module version through the introduction of ICSSD [24]. Our TDXploit attack does not bypass the TDX single-stepping mitigation but exploits a flaw to abuse the mitigation and guarantee reliable single-stepping. Furthermore, unlike the single-stepping introduced in TDXdown, TDXploit works on the current version of the TDX module with ICSSD enabled.

To mitigate the Flush+Flush attack on TD private memory, software changes are not enough. As long as the physical memory used for TDs can be mapped by the host, Flush+Flush can be executed on them. While it is possible to avoid Flush+Flush for specific cases by only executing code not vulnerable to Flush+Flush inside TDs, this does not properly mitigate the vulnerability. The ability of TDX to run regular applications inside TDs leads to users unaware of this issue executing vulnerable code regardless of the existence of Flush+Flush resilient code. Furthermore, as it is possible to run general-purpose operating systems inside TDs, this would require all software that could potentially leak sensitive information to be resilient against Flush+Flush, which is impractical. Alternatively, this issue can be mitigated through a simple hardware change that makes `clflush` aware of HKIDs, as it seems to be the case on 4th generation Xeon Scalable CPUs [1, 55]. We are unaware of a valid use case for `clflush` ignoring HKIDs. In any case, a dedicated, page granular flushing mechanism, as e.g. implemented by AMD SEV [6], should always be sufficient. Pages that are returned to the host are already written back to main memory by the TDX module. Pages given to the TDX module for private memory can be written back to main memory by the host before they are provided to the module.

The closest work to our Flush+Flush attack on TDX guests is the HKID coherence-based attack briefly mentioned in the Google security report on TDX [1]. The coherence protocol allows a memory location to be in the cache with only one HKID at a time. This can be abused to detect guest memory accesses, as the attacker can load the memory with a public HKID and detect memory accesses that use a private HKID on the same memory location through the cache line evictions enforced by the coherence mechanism. Contrary to our Flush+Flush attack, which exploits that `clflush` ignores HKIDs, the coherence-based attack exploits the current implementation of the coherence protocol regarding HKIDs, making them inherently different. Additionally, the Flush+Flush-based approach does not trigger cache misses on the host side, allowing for significantly faster resets of the cache state in case of a victim access. Finally, Intel implies in the TDX base specification that they, with the introduction of the introduction of the `TDX_FEATURES.CLFLUSH_BEFORE_ALLOC` feature flag, plan to mitigate the HKID coherence-based channel [24].

Similar to our work, Wang et al. [53] analyze multiple side channels, including Prime+Probe, and their effectiveness on Intel SGX. Additionally, they analyzed how the tested attacks, in combination with the Intel

SGX threat model, can be used to build significantly stronger attacks. Rauscher et al. [41] analyzed and compared a wide range of cache side-channel attacks using 9 metrics on Intel Sapphire Rapids and Emerald Rapids CPUs. Contrary to our work, which analyzes the viability of various side-channel attacks on Intel TDX, their work focuses only on cache side channels in a native scenario. Nilsson et al. [39] created a survey of published attacks on Intel SGX, which includes if they are SGX specific, the attack target of each attack, and possible mitigations for each attack.

7. Conclusion

We introduced a novel technique for single-stepping attacks on Intel TDX, named TDXploit. TDXploit exploits that an attacker can control and predict Intel's single-stepping mitigation. TDXploit achieves a higher (>99.99 %) single-stepping accuracy than if there were no mitigation in the first place. Furthermore, TDXploit is the first technique for reliable multi-stepping. While TDXploit does not rely on any side channels, we show that it can be combined with various side channels to mount powerful attacks: We discover a previously unknown microarchitectural behavior with Flush+Flush on TDX guest physical memory, allowing Flush+Flush attacks on TDX guests without shared memory. We demonstrate the impact of this finding by performing a full key recovery on the OpenSSL AES T-Table implementation using Flush+Flush, requiring only 8 986 encryption traces. We also systematically evaluated 6 different state-of-the-art side-channel attacks in the context of Intel TDX and TDXploit and found that only PortSmash and Prime+Probe (on the L1 cache) work in the more dangerous malicious guest scenario. However, Flush+Flush, PortSmash, Prime+Probe, and the HKID coherence side channel all work to attack TDX guests from a malicious host. Finally, we demonstrated the real-world impact of TDXploit by re-creating a previously mitigated attack on the ECSDA implementation of OpenSSL as well as an end-to-end attack on TOTP secret keys, previously only demonstrated with a different side channel on AMD SEV. We conclude that further mitigations are necessary, in particular mitigating single-stepping, which often acts as an amplifier for side-channel attacks.

Acknowledgements

This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), and the BMBF (projects SASVI and AnoMed). Additional funding was provided by generous gifts from Red Hat, and Intel.

Ethics Considerations

We responsibly disclosed our findings to Intel on November 27, 2024. Intel confirmed the issue on January 22, 2025. They plan to publish the vulnerability in August 2025. All our experiments were done on our own machines with no code from other users running on them.

Open Science

We plan to publish all our code used in this paper on Zenodo (`https://doi.org/10.5281/zenodo.15536636`) and GitHub (`https://github.com/isec-tugraz/TDXploit`).

References

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (TDX) security review. Tech. rep. Google, 2023 (pp. 228, 243, 251, 253, 255).

[2] Martin R. Albrecht and Nadia Heninger. On Bounded Distance Decoding with Predicate: Breaking the ”Lattice Barrier” for the Hidden Number Problem. 2021 (p. 239).

[3] 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. 252).

[4] AMD. AMD Secure Encrypted Virtualization (SEV). 2024. URL: `https://developer.amd.com/sev/` (p. 229).

[5] 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. 229).

[6] AMD. AMD64 Architecture Programmer's Manual. 2023 (p. 255).

[7] ARM. Arm Confidential Compute Architecture. 2024. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture (p. 229).

[8] ARM. TrustZone for Arm Cortex-M Processors. 2024. URL: https://www.arm.com/technologies/trustzone-for-cortex-a (p. 229).

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (pp. 226, 231).

[10] Canonical. Intel® Trust Domain Extensions (TDX) on Ubuntu. 2025. URL: https://github.com/canonical/tdx (pp. 238, 250).

[11] 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. 231).

[12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. {AEX-Notify}: Thwarting Precise {Single-Stepping} Attacks through Interrupt Awareness for Intel {SGX} Enclaves. In: USENIX Security. 2023 (p. 234).

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (p. 226).

[14] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (p. 231).

[15] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In: NDSS. 2025 (pp. 226, 228, 231, 248).

[16] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In: EuroSec. 2017 (pp. 226, 231).

[17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (pp. 231, 254).

[18] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 242, 252).

[19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (pp. 247, 251).

[20] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In: CHES. 2020 (p. 231).

[21] Intel. An update on 3rd Party Attestation. 2018. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/an-update-on-3rd-party-attestation.html` (p. 254).

[22] Intel. Intel Software Guard Extensions (Intel SGX). 2024. URL: `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html` (p. 229).

[23] Intel. Intel Trust Domain Extensions. 2021. URL: `https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf` (pp. 229, 234).

[24] 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. 227–231, 243, 254, 255).

[25] Intel. TDX Module 1.5.06 Source Code. 2024. URL: `%7Bhttps://github.com/intel/tdx-module%7D` (pp. 232–234, 245, 250).

[26] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (pp. 231, 254).

[27] 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. 231).

[28] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security. 2017 (p. 231).

[29] 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 (p. 239).

[30] 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. 239).

[31] 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. 239, 243).

[32] 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. 226, 231).

[33] Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Guo Shangwei, and Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In: DSN. 2024 (p. 231).

[34] 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. 245, 250, 253).

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (pp. 226, 231).

[36] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction. In: USENIX Security. 2020 (p. 226).

[37] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In: EuroSec. 2018 (p. 239).

[38] 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. 231, 234).

[39] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A Survey of Published Attacks on Intel SGX. 2020 (p. 256).

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 253).

[41] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025 (pp. 242, 247, 253, 256).

[42] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 226, 231).

[43] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 231, 234).

[44] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (pp. 231, 254).

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 254).

[46] Cody Tilkins. GitHub – tilkinsc/COTP: A simple One Time Password (OTP) library in C, supports C++. 2023. URL: `https://github.com/tilkinsc/COTP` (pp. 248, 249).

[47] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In: USENIX ATC. 2017 (p. 226).

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security. 2018 (pp. 231, 234).

[49] 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. 231).

[50] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In: CCS. 2019 (p. 254).

[51] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 226).

[52] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (pp. 226, 231–234, 238, 254).

[53] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: CCS. 2017 (pp. 226, 231, 255).

[54] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In: AsiaCCS. 2018 (pp. 226, 231).

[55] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In: CCS. 2024 (pp. 226–228, 231–234, 238–241, 243, 254, 255).

[56] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In: CHES. 2024 (pp. 226, 231, 234, 238, 254).

[57] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (p. 231).

[58] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (p. 251).

[59] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In: USENIX Security. 2024 (pp. 231, 234).

9

TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters

Publication Data

Fabian Rauscher, Hannes Weissteiner, and Daniel Gruss. TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters. In: AsiaCCS. 2026

Contributions

Main Author.

TELESCOPE: TDX Exploit Leaking Encrypted Data using Sibling Core Performance Counters

Fabian Rauscher, Hannes Weissteiner, Daniel Gruss

Graz University of Technology

Abstract

Trusted execution environments (TEEs) protect applications running inside of them from untrusted host systems. The host can not access or modify the memory of applications protected by a TEE. Intel TDX is a recently introduced TEE that allows for the execution of arbitrary code, including entire operating systems, inside a protected environment. Prior work has attacked AMD SEV-SNP, AMD's counterpart to Intel TDX, using performance counters, by leaking sensitive information through them, e.g., which branches are taken. Intel TDX is thought not to be affected by such attacks, as it disables performance counters when entering the protected guests (TDs) to mitigate these attacks.

In this paper, we bypass the protections of Intel TDX, allowing us to not only recover secrets, such as private keys, but also expand on what is thought possible by leaking arbitrary memory using performance counters. We analyze available performance counters on the recent Intel Emerald Rapids microarchitecture, finding 8 that track events for the whole physical core and not just the current logical core. We use this to bypass the Intel TDX mitigation against performance counter-based attacks by monitoring TDs from the sibling logical core. One of these counters tracks uOPs executed, allowing an attacker to gain valuable insight into victim TDs. Using this information, we recover an RSA-2048 private key from a TD running MbedTLS with an average edit distance of only 0.92 bits. Furthermore, this performance counter includes speculatively executed uOPs, enabling the use of a large number of previously unusable Spectre compare gadgets in Spectre attacks. With this discovery, we leak TD memory at a rate of 52.6 bit/s and break KASLR in less than 2 s. Finally, we break the recently introduced inter-keystroke timing defense of OpenSSH, allowing us to detect real keystrokes with an F_1 score of 99.6 %.

Keywords: Intel TDX, Side Channel, Performance Counter, TEE.

1. Introduction

To reduce hardware and maintenance costs, an increasing number of companies are moving their infrastructure to the cloud. This change comes with additional confidentiality concerns, as sensitive data, e.g., customer data, company secrets, and cryptographic keys, are now stored and processed on hardware owned and operated by a third party. With traditional virtualization technologies, the cloud provider has full access to all data in the virtual machine (VM). Thus, a data breach or malicious insider at the cloud provider can have severe consequences for their customers. To mitigate these concerns, trusted execution environments (TEEs) have been introduced by hardware vendors. TEEs provide isolated environments that protect the confidentiality and integrity of applications and data in the TEE from the host system by protecting the memory and register state of the TEE. Traditional TEEs, e.g., Intel SGX or ARM TrustZone, require applications to be written specifically for the specific TEE implementation [15, 5], limiting their applicability and portability. Prior work demonstrated that generic programs can be ported to Intel SGX using a library OS approach [75], but resulting programs are still limited due to the design details of SGX. Recently, vendors introduced TEEs that allow entire VMs to run inside of them, such as AMD SEV-SNP [2] and Intel TDX [33]. These confidential VMs (CVMs) allow users to run unmodified applications on general-purpose operating systems in TEEs.

The TEE threat model includes a malicious machine owner, with control over the host operating system and physical access to the machine [15, 2, 33]. As such, the attack surface of this threat model is significantly larger than the traditional unprivileged user-space attacker model. Many prior works have demonstrated powerful attacks on SGX. Early works demonstrated that the fact that the host and the enclave share the same hardware allows for cache side-channel attacks [79, 7, 25, 68, 53, 80]. Many attacks on TEEs rely on functionality that is only available as a malicious hypervisor, e.g., page faults [85, 82], single-stepping [78, 84, 82, 64], performance counters [50, 21], or firmware modifications [17].

Recently, multiple works were released on performance counter based attacks on AMD SEV-SNP, as SEV did not protect against these type of attacks. Lou et al. [50] were the first to demonstrate a website fingerprinting attack on AMD SEV-SNP using performance counters. Gast et al. [21] demonstrated more fine-grained performance-counter-based attacks on AMD SEV-SNP, including the recovery of RSA keys, TOTP secrets, and

breaking the HQC post-quantum signature scheme. Both works mention that Intel TDX is not affected by their attacks, as Intel TDX disables performance counters when entering the TEE. Weissteiner et al. [81] proposed a method to decorrelate performance counter values between the host and the TEE, mitigating fine-grained performance counter leakage from TEEs. However, they only mitigate leakage from performance counters on the same logical core, without considering potential leakage across hyperthreads.

In this paper, we break Intel's mitigation against performance counter attacks and show how it is still possible to leak fine-grained information through them, allowing us to leak RSA keys and perform Spectre attacks on compare gadgets regardless of the operations they perform. We exploit performance counters that monitor the whole physical core to monitor TDX CVMs, called trust domains (TDs), from a sibling logical core. This is possible as Intel TDX only disables performance counters on the logical core the TD is running on, but not the entire physical core. We analyze available performance counters on a recent Intel processor and find 8 that count events for the whole physical core. One of these performance counters, `UOPSEXECUTED.CORE`, monitors the precise micro OP (uOP) throughput of the TD, allowing us to gain detailed information on the TD's execution flow. We leverage `UOPSEXECUTED.CORE` to fingerprint code passages inside the TD, identifying secret-dependent code execution. Using this information, we recover a full RSA-2048 private key from a TD running MbedTLS by only monitoring 400 encryptions happening in the TD. The recovered key has an average Levenshtein distance of only 0.92 bits, meaning on average *less than one bit* has to be changed in the recovered key to get the real key, showing the high accuracy of our attack. Additionally, `UOPSEXECUTED.CORE` also counts speculatively executed uOPs. Hence, our attack is a new way to leak information for speculative execution attacks on Intel TDX using `UOPSEXECUTED.CORE` that is universal across Spectre transmission channels.

Our discovery introduces an asymmetry in favor of the attacker when considering the four gadget types in Spectre attacks [9], namely prefetch, compare, index, and execute gadgets. Index gadgets have been explored the most so far, as they are easy to find and use [40, 73]. However, our attack provides a substantial benefit for all other gadget types: More specifically, the attacker can infer the outcome directly from the `UOPSEXECUTED.CORE` performance counter. Hence, there is no need for any other specific transmission channel, such as a cache side channel, to leak the secret-dependent

outcome of the gadget. Thus, we can use previously unusable gadgets for Spectre attacks. We demonstrate this by abusing the Linux kernel's implementation of the `memchr` function to leak arbitrary memory from a TD at a rate of 52.6 bit/s with an error rate of only 0.6 %, breaking the confidentiality guarantees of Intel TDX. We also demonstrate a KASLR break using our generic Spectre attack on TDs, leaking the KASLR offset in less than 2 s. Finally, we discover that the recently introduced inter-keystroke timing defense introduced to OpenSSH [58] is insufficient for a CVM scenario. By performing precise timing measurements and taking advantage of the high amount of control an attacker has in the traditional TEE threat model, we are able to detect real keystrokes with an F_1 score of 99.6 % and a temporal standard deviation of only 5.64 ms, despite the presence of a large number of fake keystrokes generated by the mitigation. Furthermore, we show that, even if this timing side channel is mitigated, the real keystrokes can still be determined using `UOPSEXECUTED.CORE`.

In summary, our work makes the following contributions:

- We systematically analyze available performance counters on a recent Intel CPU for cross-core leakage, identifying 8 counters that count events for the whole physical core and 1 counter that is a particularly high threat to Intel TDX.
- We mount an attack on MbedTLS `3.5.2` running inside of a TD using this dangerous performance counter to bypass Intel's mitigation to recover an RSA-2048 key with an average Levenshtein distance from the real key of **only** 0.92 bits.
- We demonstrate Spectre attacks exploiting the inherent asymmetry of the `UOPSEXECUTED.CORE` performance counter channel, *i.e.*, any Spectre gadget encoding information can be used; and use it to leak arbitrary memory at a rate of 52.6 bit/s and break KASLR in <2 s.
- We demonstrate that OpenSSH's recent inter-keystroke timing defense is insufficient for CVMs, allowing us to distinguish between real and fake keystrokes with an F_1 score of 99.6 %, thus, re-enabling inter-keystroke timing attacks.

Outline. We provide background in Section 2. In Section 3, we analyze available performance counters for cross-core leakage. In Section 4, we recover a full RSA-2048 private key from a TD running MbedTLS, using performance-counter leakage. In Section 5, we demonstrate Spectre

attacks using the new `UOPSEXECUTED.CORE` channel for data leakage from speculative execution on Intel TDX. In Section 6, we break openSSH's recent inter-keystroke timing defense. We discuss possible mitigations and related work in Section 7. We conclude in Section 8.

Responsible Disclosure. We responsibly disclosed our findings to Intel on August 7, 2025 and to the OpenSSH team on August 25, 2025. Intel recommends developers to follow their security best practices for side-channel resistance. The OpenSSH team does not consider confidential virtual machines as part of their threat model for the inter-keystroke timing attack mitigation and will therefore not mitigate our attack.

2. Background

In this section, we first discuss Trusted Execution Environments (TEEs) and, in more detail, Intel TDX. We also provide a brief overview of hardware performance counters (on Intel processors) and how they can be used both for malicious and benign purposes.

2.1. Trusted Execution Environments

Trusted Execution Environments (TEEs) are an emerging technology that processor vendors introduce to offer increased confidentiality and integrity guarantees compared to what traditional user-kernel isolation can provide [32, 2, 34, 3]. The first generation of TEEs mainly protected specific application components, e.g., Intel Software Guard Extensions (SGX) [32] and ARM TrustZone [5], focused on a scenario where a trusted application must be protected from a malicious user or compromised system. Since this design limits the potential usage scenarios to small applications handling small amounts of highly sensitive data e.g., fingerprints or cryptographic material, or for digital rights management (DRM), the scientific community quickly realized the need for TEEs that allow users to run any system and application as a TEE [75]. As a consequence, a new type of TEEs emerged that focuses on the scenario where an entire virtual machine must be protected from a malicious or compromised host. Since similar confidentiality guarantees (but not necessarily integrity guarantees [83]) are provided, these virtual machines are also called confidential virtual

machines (CVMs). Both AMD and Intel realized their respective CVM implementation: AMD Secure Encrypted Virtualization (SEV) [1] and Intel Trust Domain Extensions (TDX) [33].

2.2. Intel TDX

Intel calls CVMs running on Intel TDX trust domains (TDs) [34]. The open-source TDX module is designed to handle the encryption and management of guest memory and saved guest state within the trust domain virtual processor state area (TDVPS). The TDX module operates in SEAM root execution mode, which is protected from the host, and interfaces between the TEE and the host. Internally, the TDX module uses existing Intel Virtual Machine Extensions (VMX) for virtualization. To protect against malicious modifications, the TDX module is signed by Intel and can only be loaded if the processor can verify the signature. TDX splits the guest's physical memory into a shared section, accessible by both host and guest, and a private, encrypted section, only accessible to the guest and the TDX module. This design allows for fast communication with the host via the shared section, while still protecting the guest's private memory. The TDX module handles page table management for the private guest-physical memory, while the host manages the shared memory's page tables. A dedicated bit in the guest physical address allows the guest to distinguish between shared and private memory, avoiding accidental interactions with shared memory. Additionally, shared memory is not executable to mitigate certain bugs and attack vectors. To encrypt the memory of TDs, TDX employs Intel's Total Memory Encryption - Multi Key (TME-MK). TDX splits TME-MK's key ID (HKID) range into a public and a private part, reserving the private range for the TDX module and TDs. Each 64-bit region is validated against a cryptographic MAC on every memory access to protect against memory corruption, e.g., due to physical interference. However, the CPU is still shared between all TDX guests and the host, allowing for potential information leakage through shared hardware components.

Information leakage from TEEs is a major concern. While first side-channel attacks have been demonstrated, in the context of Intel SGX [7, 25, 68, 80, 53, 79, 20, 29, 42] and Arm TrustZone [47, 66], transient-execution attacks have been found to be an even more powerful information leakage primitive [40, 56, 12, 76, 67, 77, 73]. Thereby, the TEE threat model often allows for a stronger attacker, that can precisely control the execution of

the TEE, e.g., through single-stepping [85, 78, 53, 84, 64]. Alternatively, attackers can also use power side channels [48] and fault attacks [60, 61, 55, 13].

Unlike AMD's counterpart to TDX called AMD SEV-SNP, Intel TDX employs active mitigations against some controlled channels, such as single-stepping through interrupts and zero-stepping through page faults, and performance counter-based attacks, that are built directly into the TDX module. The single-stepping defense detects an attack through the number of instructions executed and introduces noise in case an attack is detected. The defense against page fault-based zero-stepping simply checks whether the instruction pointer changes between two consecutive page faults in the guest physical to host physical translations. To mitigate performance counter-based attacks, the TDX module disables performance counters when entering the TDX module through the `GLOBAL_PERF_CNT` MSR and context switches performance counter MSRs in case the TD is allowed to use them for themselves.

2.3. Hardware Performance Counters

Modern processors are highly complex and provide numerous ways to debug, profile, and monitor the processor's execution of software. Hardware performance counters are particularly useful when investigating how a piece of code exercises the CPU hardware. Performance counters are provided to the user via registers that count certain hardware events, e.g., cache hits and misses in a specific cache level. Developers can use this information to debug performance bottlenecks and optimize the software [59], or to monitor software execution and detect anomalies [88, 44, 43, 16, 14, 11]. Performance counters are typically only reachable from kernel space, e.g., via Model Specific Registers (MSRs). The potential information leakage through performance counters was already known when Intel SGX was introduced. Intel excluded any SGX activity from all performance counters [35]. AMD did not exclude SEV activity from performance counters until recently in response to performance-counter-based attacks on AMD SEV-SNP [50, 21]. Gast et al. [21] demonstrated several end-to-end attacks, e.g., the recovery of RSA keys from a SEV-SNP CVM. Consequently, performance-counter-based attacks are currently considered mitigated on Intel SGX, Intel TDX, and recent versions of AMD SEV-SNP. However, some performance counters only count global

information, which so far has not been considered relevant in terms of leakage from TEEs.

3. Performance Counter Analysis

In this section, we analyze performance counters for the core domain and determine possible data leakage through them. We analyze the available core performance counters on our Intel Xeon Silver 4514Y Emerald Rapids CPU and determine which performance counters allow for leakage across logical cores. From the found performance counters, we discuss one promising counter for attacks, `UOPSEXECUTED.CORE`.

Unlike with AMD SEV-SNP, most performance counters cannot be used to monitor guest activity [34, 32]. This is due to the TDX module context switching the `GLOBAL_PERF_CNT` MSR. The `GLOBAL_PERF_CNT` MSR contains a bit for each hardware performance counter through which they can be enabled or disabled. When entering the TDX module, the `GLOBAL_PERF_CNT` is set to disable all performance counters. If a TD is allowed to use performance counters, a `GLOBAL_PERF_CNT` value, as well as the values for the other performance counter MSRs, are maintained for each virtual core of the TD and loaded by the TDX module. As these values are stored inside of the encrypted TD state, the host can neither read nor manipulate these values. Therefore, all performance counters that are set up by the host on the logical core on which the TD is run on do not count while executing the TD.

While the context switching of `GLOBAL_PERF_CNT` mitigates attacks on the same logical core, it does not prevent a malicious host from tracking performance counters that count events for the whole physical core through the use of the target's sibling logical core. In general, Intel differentiates between 3 types of performance counters: core, uncore, and offcore. Core performance counters track events occurring inside of the CPU core, e.g., instructions executed and cycles stalled. Uncore counters track events that are outside of the core, in the uncore (including the LLC and memory controller), e.g., DRAM interactions and data requests to MMIO, and are for the whole CPU. Finally, offcore counters track per-core events related to interactions between the core and the uncore, e.g., prefetches to the L2. In this work, we focus on core domain performance counters, as information that can be gained from uncore and offcore performance counters is limited, e.g., memory requests that go to the L3. Core performance counters, on

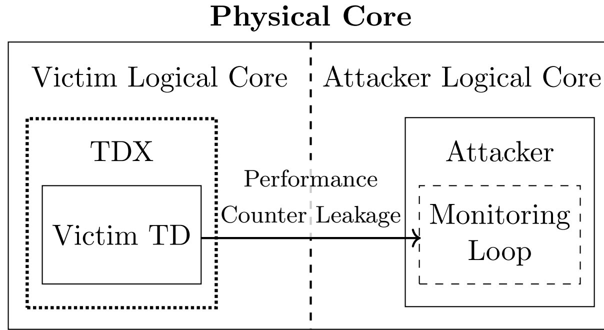


Figure 9.1.: Attack overview. TDX protects from performance counter leakage on the same logical core. However, the victim TD affects performance counters that count for the whole physical core, which can be monitored by an attacker on the sibling logical core.

the other hand, provide very specific information regarding the execution flow of a core, making them interesting tools for leaking data from a TD.

As core performance counters provide much more fine-grained information on the current execution flow, e.g., branches taken, instructions retired, and mispredictions, we want to analyze them to determine whether there are any that can be used to attack TDs, despite the existing defenses against such attacks. Performance counters of particular interest are the ones that count for the whole physical core, as they can be set up on one attacker-controlled logical core, while the sibling logical core is inside a TD, as shown in Figure 9.1. Intel's documentation regarding what exactly some performance counters cover can be very vague. For example, while some of the documented events explicitly mention whether they count for the current logical core or the whole physical core, most do not [31]. Despite this, a majority of the events only the current logical core that monitors them. This includes events such as `BR_INST_RETIRED.*`, `INST_RETIRED.ANY_P`, and `UOPS_DISPATCHED.PORT_*`. We found that performance counters only target the whole physical core when it is explicitly mentioned in the description of the counter, when it is part of the performance counter name, or when it targets offcore events.

We analyzed all the available performance counter events published by Intel for our Emerald Rapids CPU Intel Xeon Silver 4514Y [31] and listed all performance counters that we found that target the whole core in Table 9.1. From the found performance counters, `UOPSEXECUTED.CORE` is a very promising tool for attacks, as the uOP throughput of a core highly

Table 9.1.: List of performance core counters monitoring the whole core

Performance Counter Name
CORE_SNOOP_RESPONSE.*
CPU_CLK_UNHALTED.REF_DISTRIBUTED
CPU_CLK_UNHALTED.REF_TSC*
IDQ_UOPS_NOT_DELIVERED.CORE
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE
OFFCORE_REQUESTS.*
OFFCORE_REQUESTS_OUTSTANDING.*
UOPS_EXECUTED.CORE

```
1 size_t last = rdmsr(IA32_PMC0), i = 0:
2 while (!atomic_read(&done) {
3   size_t cur = rdmsr(IA32_PMC0);
4   measurements[i++] = cur-last;
5   last = cur;
6 }
```

Listing 9.1: Code of the performance counter measurement hot loop.

depends on the instructions executed. Assuming secret-dependent branches in an application, this can leak sensitive information. Additionally, we show that UOPSEXECUTED.CORE also counts speculatively executed uOPs that are never committed. This can introduce additional noise to some attacks, but enables the use of UOPSEXECUTED.CORE to broaden the scope of exploitable Spectre gadgets.

UOPSEXECUTED.CORE counts uOPs executed by the whole core. When the CPU executes instructions, they are first translated into simpler operations, so-called uOPs, which are in turn executed by different execution units of the core. The number of uOPs an instruction is translated to depends on the instructions and the system it is running on. This information is similar to the information leaked by single-stepping, which is actively mitigated by the TDX module [34], making it a relevant attack vector.

We now confirm whether UOPSEXECUTED.CORE actually counts the uOPs executed inside of a TD, and that we do not just measure contention or other side effects between the two logical cores. We let the TD execute either IMUL

```
1 test rax, rax;
2 je 1f;
3 serialize;
4 .rept 32;
5 add rbx, 1;
6 .endr;
7 serialize;
8 1: nop;
```

Listing 9.2: Code to determine whether UOPSEXECUTED.CORE counts specultively executed uOPs.

or ADD in a loop and continuously measure the uOPs executed on the sibling logical core through UOPSEXECUTED.CORE and UOPS_EXECUTED.THREAD. The UOPS_EXECUTED.THREAD performance counter only accounts for the current logical core, in contrast to UOPSEXECUTED.CORE, which targets the whole physical core. To minimize any unnecessary overhead, we do not track time and only record the value of the performance counters (see Listing 9.1). As the measurement loop consists of a fixed number of instructions that are executed, the uOPs executed should also stay constant with slight variations, e.g., due to misspeculation of the measurement loop. Therefore, if IMUL and ADD can be clearly distinguished using UOPSEXECUTED.CORE, but not with UOPS_EXECUTED.THREAD, we are able to track uOPs executed inside the TD with UOPSEXECUTED.CORE, and we are not observing a different side effect. In case IMUL and ADD can also be distinguished from each other with just UOPS_EXECUTED.THREAD, then we are likely observing a different side effect, e.g., the code in the TD affects the measurement codes branch prediction.

The results of these measurements are provided in Figure 9.2. With UOPS_EXECUTED.THREAD (Figure 9.2a), we measure ∼62 uOPs for the IMUL and ADD loops, making them indistinguishable from each other. The uOPs executed do not reach 0, in this case, as the measurement still includes the uOPs executed by the measurement code itself. With UOPSEXECUTED .CORE (Figure 9.2b), we measure ∼90 uOPs for IMUL and ∼150 uOPs for ADD, making them clearly distinguishable, showing that we are able to infer information about code executed inside of a TD. Furthermore, these measurements show that uOP throughput can also leak information on the instructions executed.

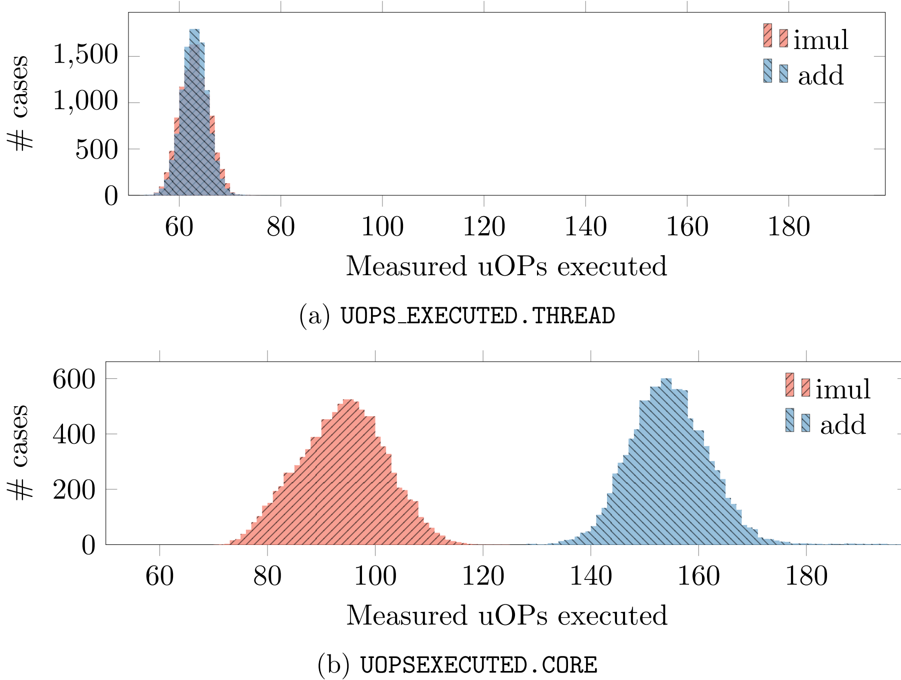


Figure 9.2.: Measured uOPs executed using `UOPSEXECUTED.CORE` (Figure 9.2b) and `UOPS_EXECUTED.THREAD` (Figure 9.2a) while the other logical core is inside of a TD and executing either `IMUL` or `ADD` instructions in a loop. With `UOPSEXECUTED.CORE`, the two instructions can be clearly differentiated from each other. With `UOPS_EXECUTED.THREAD`, the two instructions are indistinguishable, showing that the differences observed by `UOPSEXECUTED.CORE` are from the direct influence of the TD on the counter and not from other side effects, e.g., contention.

Intel defines `UOPSEXECUTED.CORE` as counting all uOPs executed, including speculatively executed instructions [31]. To confirm whether `UOPSEXECUTED.CORE` actually includes speculatively executed uOPs, we run a short test snippet shown in Listing 9.2 on our Intel Xeon Silver 4514Y. For each measurement, we first train the branch in line 2 not to be taken and then use `UOPSEXECUTED.CORE` to determine the uOPs executed when the branch is taken. Due to the training, the branch will misspeculate and execute `serialize` in line 3 only speculatively, which in turn stops the speculation. We then repeat the experiment without the `serialize` in line 3. Without `serialize`, the additions in line 5 should be executed speculatively. While with and without the `serialize` instruction, the exact same number of

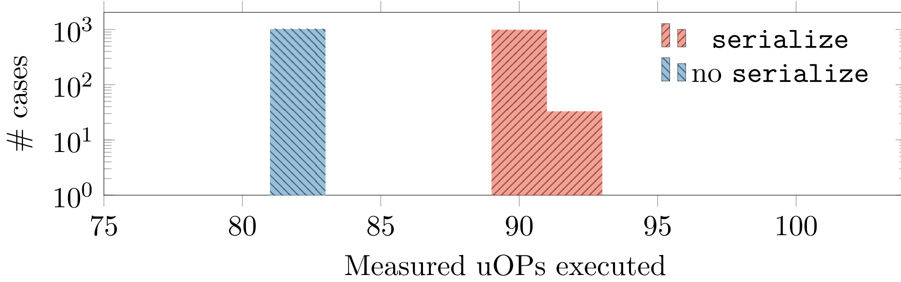


Figure 9.3.: Measured uOPs executed using `UOPSEXECUTED.CORE` of the code listed in Listing 9.2 when the conditional branch always misspeculates, with the `serialize` in line 2 and without it. As both cases commit the same instructions, the difference in measured uOPs executed (without the serialize higher due to more instructions being speculatively executed) confirms that `UOPSEXECUTED.CORE` also counts only speculatively executed uOPs.

uOPs are committed, there are more uOPs speculatively executed without `serialize`. Therefore, if the uOPS executed in the two cases differ, `UOPS EXECUTED.CORE` includes speculatively executed uOPs, even when they are never committed.

The results of our measurements are shown in Figure 9.3. Without `serialize`, the uOPs executed are 90.1 (n=1024, $\sigma_{\bar{x}}$=0.01). With `serialize`, the uOPs executed are 82 (n=1024, $\sigma_{\bar{x}}$=0). This confirms that `UOPS EXECUTED.CORE` includes speculatively executed uOPS.

4. RSA Key Recovery

In this section, we leverage `UOPSEXECUTED.CORE` to perform a full RSA-2048 private key recovery from a TD running MbedTLS 3.5.2 [4]. The MbedTLS RSA implementation is not constant-time and uses a windowed square-and-multiply approach to perform RSA encryption, enabling side-channel attacks. While both the square- and the multiply-operations by themselves are implemented in constant time, *i.e.*, they always take the same amount of time irrespective of the ciphertext, for each bit in the exponent (private key), the encryption either performs a square- or a square- and a multiply-operation. Therefore, if we can determine which operations were performed for each key bit, we can recover the private key. Similar to prior work [21, 23, 72, 48], we configure MbedTLS to use

a window size of 1, as a working attack on a window size of 1 can be extended to arbitrary window lengths [49].

4.1. Threat Model

We follow the typical TEE threat model of a compromised host [70, 55, 76, 82, 84, 78, 87, 64]. Our host system contains a Intel Xeon Silver 4514Y running TDX-enabled Ubuntu 24.04 [10] with TDX module 1.5.16, the most recent version at the time of writing [36]. Our guests run Ubuntu 24.04, which was created according to the TDX guide published by Canonical and has one virtual core. This setup is in line with previous work [84, 82, 64] and the official TDX threat model of a compromised cloud provider, published by Intel [33]. We run standard MbedTLS 3.5.2 [4] and did not modify it in any form to perform our attack.

4.2. Overview

With square-and-multiply, for each bit in the exponent, the value to be encrypted is either squared and multiplied by the initial value if the bit is 1 or only squared if the bit is 0. The code for processing a single bit from the MbedTLS RSA implementation is shown in Listing 9.3. MbedTLS uses the same constant time multiplication function (`mpi_montmul`) for both the square and the multiply steps, making them indistinguishable from each other. The `mpi_select` is a constant-time conditional copy. While the `mpi_montmul` and `mpi_select` functions have a relatively constant uOP throughput, the code in between them, where the exponent bits are checked, does not. Lines 2 to 7 perform the square operation when the exponent bit is 0, and lines 9 to 26 are only executed to perform square and multiply when the exponent bit is 1. By constantly monitoring uOPs executed through `UOPSEXECUTED.CORE` on the sibling logical core of the victim, it is possible to determine when which operation is executed (`mpi_montmul`, `mpi_select`, or the code in between) and which code path in the processing loop was taken.

The trace of uOPs executed during two executions of `mpi_select` and `mpi_montmul` during a regular encryption running inside a TD is shown in Figure 9.4. To record the trace, we use the same measurement setup as described in Section 3 with a measurement point consisting of the difference of two performance counter reads. Executions of `mpi_select`

```
1 //...
2 if (ei == 0 && state == 1) {
3   mpi_select(&WW, W, w_table_used_size,
4               x_index);
5   mpi_montmul(&W[x_index], &WW, N,
6                mm, &T);
7   continue;
8 }
9 //...
10 nbits++;
11 exponent_bits_in_window |=
12   (ei << (window_bitsize - nbits));
13 if (nbits == window_bitsize) {
14   for (i = 0; i < window_bitsize; i++) {
15       mpi_select(&WW, W,
16           w_table_used_size, x_index);
17       mpi_montmul(&W[x_index], &WW, N, mm,
    -> &T);
18   }
19   mpi_select(&WW, W, w_table_used_size,
20               exponent_bits_in_window);
21   mpi_montmul(&W[x_index], &WW, N, mm, &T);
22   //...
23 }
24 //...
```

Listing 9.3: Single loop iteration of the MbetTLS RSA implementation [4]. When the exponent bit is 0 (square), lines 2 to 7 are executed. When the exponent bit is 1 (square and multiply), lines 9 to 26 are executed.

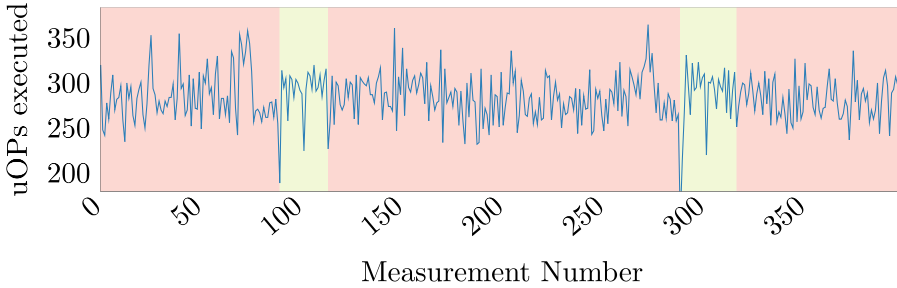


Figure 9.4.: Part of an RSA encryption using MbedTLS containing two executions of the `mpi_select` and `mpi_montmul` functions. `mpi_select` executions start at measurements ∼100 and ∼300 highlighted in green with executions of `mpi_montmul` highlighted in red.

are highlighted in green between ∼100 and ∼125 as well as between ∼300 and ∼325. Almost all of the other execution time is taken up by `mpi_montmul` highlighted in red. At the end of each multiplication, the uOP throughput spikes for a few measurements and then stabilizes, before finally having a 1 to 2 measurement drop (at ∼100 and ∼300). This short drop is the transition between `mpi_montmul` and `mpi_select`. When executing `mpi_select`, the throughput stabilizes with a short drop in the middle and at the very end when transitioning to the next `mpi_montmul` call. The measurements around the transitions from `mpi_montmul` to the next `mpi_select` hold the conditional code shown in Listing 9.3. With these measurements extracted, it is possible to determine which parts of the function were executed when, therefore, leaking the key.

As it is not possible for an attacker to know at which exact point in code a measurement point starts and stops, we require multiple traces for a full key recovery. To minimize implementation effort, we take advantage of a controlled channel in addition to our `UOPSEXECUTED.CORE`. We use the Intel TDX feature to block and unblock TD pages [34] to force a VM exit between the multiplications, allowing us to only measure the target code (Listing 9.3) together with the `mpi_select` calls, similar to prior work [82, 21]. Intel TDX allows the host to block the access to private pages of TDs as an initial step for hypervisor management functions, such as merging or splitting pages. This functionality is required for the host to perform the TLB invalidation sequence before actual changes to the TD mappings are made. While Intel actively tries to mitigate zero-stepping attacks that are done using this feature, as long as the instruction

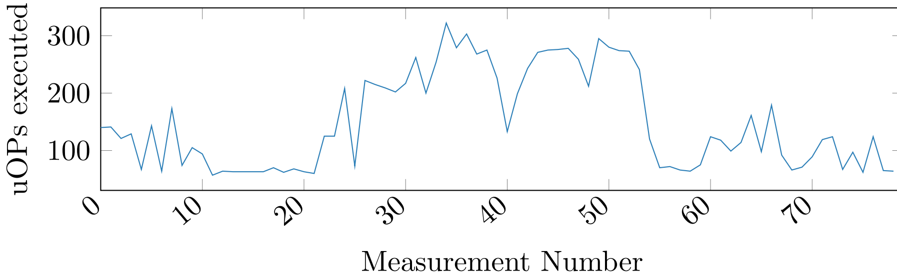


Figure 9.5.: Trace of a single `mpi_select` together with part of the code from Listing 9.3 recorded with the help of a controlled channel. The two plateaus with a drop in the middle are the `mpi_select` function, with the very short Listing 9.3 code being executed directly before it.

pointer makes progress between VM exits resulting from blocked pages, the mitigation does not intervene. During our testing, `mpi_montmul` and the MbedTLS RSA function were never on the same page, allowing us to use this controlled channel for synchronization. The guest physical addresses for these functions can be determined by the malicious host through page tracking proposed by Li et al. [45]. Despite this, we confirmed that our attack works even without the use of this controlled channel, by collecting full traces of encryptions, searching for the `mpi_select` executions through pattern matching, and recovering a significant part of the private key using this information. We did not implement the full key recovery using pattern matching, as the `mpi_select` execution is clearly visible (Figure 9.4). Therefore, implementing reliable pattern matching would only be an engineering challenge, providing no additional scientific value.

To set up the controlled channel, we block access to the page containing `mpi_montmul` and the page containing the rest of the code provided in Listing 9.3 which can be found by profiling the TDs physical memory. Whenever the TD tries to execute the regular RSA logic it will result in a VM exit, returning the control to the host. In this case we unblock the page and block the access to the `mpi_montmul` page. Whenever the TD gets to the next multiplication, we again receive a VM exit where we unblock the `mpi_montmul` page and block the access to the rest of the RSA implementation. Through this mechanism, we are able to precisely track when the code between two multiplications is executed. The trace

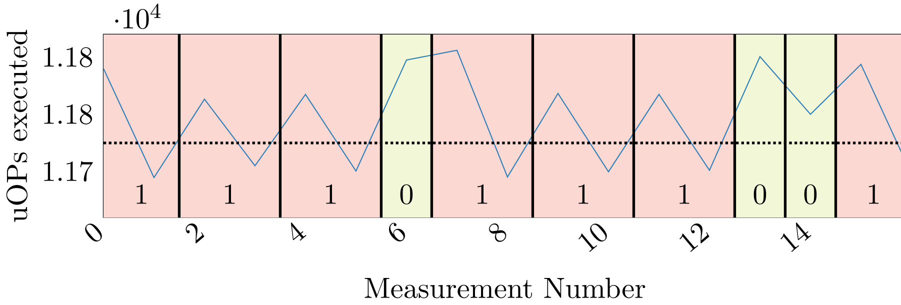


Figure 9.6.: Trace of uOPs executed. Each datapoint is the sum of uOPs measured of the code between two `mpi_montmul` calls of MbetTLS RSA averaged for 400 measurements. When a data point falls below the threshold (dotted line), the TD executed the short code between a square and a multiply, indicating a 1 (together with the previous measurement point, marked in red). Otherwise, the code executed is between two squares indicating a 0 (marked in green).

of an `mpi_select` and the logic provided in Listing 9.3, *i.e.*, for the code executed between two `mpi_montmul` calls, recorded with the help of the controlled channel is shown in Figure 9.5. We use the same measurement approach as in Figure 9.4. The relatively constant parts of the trace at the beginning and end of the trace are part of the VM entry and VM exit, respectively. The two plateaus with a drop in between them are caused by the `mpi_select` function, which performs a copy operation and takes up most of the execution time between two `mpi_montmul` calls. This same pattern can also be observed in Figure 9.4. Our target code from Figure 9.5 is executed right before `mpi_select` and is most likely contained in 1 to 2 measurement points, due to its short length.

Figure 9.6 shows the average of 400 uOP traces collected from a small part of an MbedTLS RSA encryption with the `mpi_montmul` calls filtered out. Each measurement point in Figure 9.6 is the sum of all uOPs executed for a call to `mpi_select` and the code in between function calls shown in Listing 9.3. We collected these traces with the controlled channel by blocking access to the `mpi_montmul` page and the page containing the rest of the RSA implementation, always unblocking the one the TD wants to execute and blocking the other one. Due to this, we are able to synchronize our measurements with the victim without requiring a change in the MbedTLS source code. The lowest points in the trace (below the dotted line) are the executions of the code between line 17 and line 22, shown in Listing 9.3 (the code between a square and a multiply). This

part of the code only contains a call to `mpi_select` and no other logic leading to the low number of uOPs executed. Such a low measurement is a clear indication that the TD just processed a 1, as this code part is only executed in this case. As processing a 1 results in two multiplications, we can group the current and previous point into a single operation. When a measurement of a high number of uOPs executed (above the threshold) is not followed by a low number of uOPs executed (below the threshold), this is an indication of two square operations after each other. Therefore, we can conclude that a 0 has been processed. We set the threshold at "$0.2 \cdot (max - min) + min$" where min is the lowest measurement point during the encryption and max is the highest. This resulted in the most stable results for our experiments.

4.3. Evaluation

We evaluated our attack on an Intel Xeon Silver 4514Y with MbedTLS 3.5.2 and Ubuntu 24.04 running on both the host and the guest. We followed the threat model outlined in Section 4.1. The victim TD is executing RSA-2048 encryptions using MbedTLS, while the host is monitoring the guest on a sibling logical core using `UOPSEXECUTED.CORE`. Using 400 encryptions for each key extraction, we are able to recover the key with an average Levenshtein distance of 0.92 over 25 key extractions with a standard error of 0.37, meaning on average 0.92 bits of the recovered key need to be changed to derive the correct key. We report the Levenshtein instead of the hamming distance, as the processing of a 1 has double the measurement points as the processing of a 0. Therefore, when a 1 is misclassified as a 0, a second 0 (the second part of the square and multiply operation) would automatically be detected, consequently shifting the rest of the recovered key by 1 bit. This shift would lead to a high hamming distance that does not properly reflect the correct number of key bits recovered. Additionally, 72 % of our attack runs were able to recover 100 % of the correct private key, while the rest only had a small number of bit errors.

Similarly to our work, Gast et al. [21] performed a single trace RSA key recovery on AMD SEV-SNP using branch-related performance counters on the same logical core. In contrast, our attack does not require single-stepping, which is actively mitigated on Intel TDX, but can be performed on a normally running guest. We use `UOPSEXECUTED.CORE` for data leakage, which was not highlighted by them as a possible attack target. In contrast, they use branch-related counters, which do not account for the other

logical core, making them unusable on TDX. Furthermore, we bypass the TDX mitigation against performance counter-based attacks, which, as Gast et al. [21] acknowledged, mitigates their attacks.

5. Any Gadget Spectre Attacks

In this section, we introduce Spectre attacks that have a much broader range of suitable gadgets. We leverage that the performance counter channel using `UOPSEXECUTED.CORE` turns almost every secret dependent operation, e.g., a conditional branch, into a data leaking gadget when targeting a TD. We first explain how Spectre attacks using the `UOPSEXECUTED.CORE` channel work and how they drastically increase the possible gadgets available. We then demonstrate a KASLR break by combining Spectre with the `UOPSEXECUTED.CORE` performance counter channel and show how it can be used in Spectre attacks to leak arbitrary memory from a victim TD. For all attacks, we follow the threat model outlined in Section 4.1.

5.1. Overview

Traditional Spectre attacks rely on specific secret-dependent operations to leak information. Kocher et al. [40] showed that memory accesses are an effective means to encode arbitrary data into the cache state during speculative execution. The attacker can then infer the secret information through the cache state after the execution. This greatly limits the number of gadgets available to an attacker. Canella et al. [9] distinguished between 4 gadget types: prefetch, compare, index, and execute gadgets. Most prior works rely on index gadgets that perform two memory accesses: one to load the secret and one to access an array based on the secret value [9]. However, such gadgets can be difficult to find in real-world software [24]. Furthermore, branches are known targets for Spectre attacks, and memory fences are a viable solution against these attacks. While other works explored further covert channels beyond the cache to exfiltrate data from transient execution [6, 71], they still require the attacker to have control over the corresponding covert channel and to find gadgets that encode the data accordingly into this covert channel. Instead, the `UOPSEXECUTED.CORE` performance counter tracks the number of **any** uOPs executed, even speculatively. Consequently, we can use the `UOPSEXECUTED.CORE` performance counter as a covert channel receiver that is influenced by

```
1 if (expression) {
2   //...
3   function_ptr(...);
4   //...
5 }
```

Listing 9.4: Example victim code, similar to Göktaş et al. [24]. The attacker can control the function pointer during a speculative execution, but never during actual execution.

any secret-dependent code execution. The change in uOP throughput can be the result of a difference in the microarchitectural state, e.g., if cached memory is accessed in contrast to non-cached memory, or simply different instructions are executed due to a conditional branch, or any other microarchitectural difference.

For our Spectre attack, we focus on comparing gadgets, which are abundant in the Linux kernel [9] and not considered in Spectre mitigation efforts so far that focused on index gadgets [30]. Unlike index gadgets that rely on a secret-dependent memory access, compare gadgets only leak a single bit per speculative branch, *i.e.*, branch taken or not taken. While the `UOPS EXECUTED.CORE` channel is a very generic Spectre receiver, uOPs executed can also be a noisy channel. Still, we believe the significantly broadened scope of possible gadgets makes it an interesting channel for attackers.

For our experiments, we use the threat model described in Section 4.1 and assume the existence of victim code similar to the one shown in Listing 9.4, in line with prior work [24]. The attacker can control the value of the function pointer that is being called in speculation and two arguments passed to it. However, importantly, the attacker's chosen function pointer is **only** executed in speculation, never during actual execution. Furthermore, we note that this is just one gadget that is realistic based on prior work [24], whereas there is an abundance of further compare gadgets e.g., in the Linux kernel that may also leak information [9].

5.2. KASLR Break

KASLR randomizes the virtual addresses of memory mapped in the kernel space. Given that it has virtually no performance overhead, it is widely

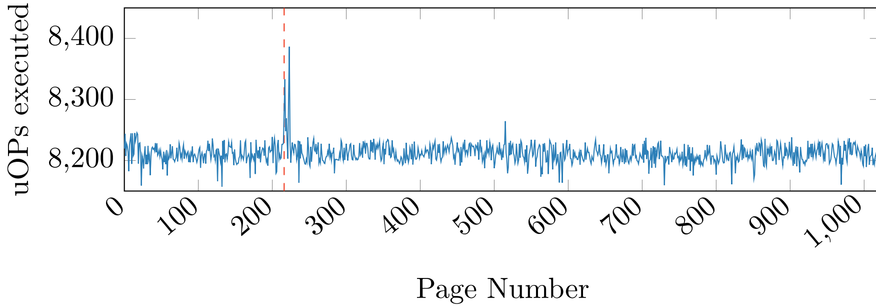


Figure 9.7.: KASLR break using our Spectre attack with the `UOPSEXECUTED.CORE` channel. Each point is the median of 30 measurements on a single 2 MB page starting at the start of the Linux kernel code ASLR region (`0xffffffff80000000` or page number 0 in the plot). Kernel code starts at page number 216 (marked by the dashed line), which is clearly visible through a spike in uOPs executed in the plot.

deployed in modern operating systems. Practically, it is often the first line of defense for an attacker to cross: The attacker often needs to know the address of specific code or data in the kernel, e.g., to leak data from this specific location, modify memory, or reuse code for the attack [38, 27]. Guessing the KASLR offset is often not feasible due to the severe consequences if the attacker guesses wrong [19, 28], e.g., crashing the system.

To circumvent this challenge, prior work used various microarchitectural attacks [38, 27, 8, 41] and Spectre attacks [51, 39, 37, 24]. Similarly, we also use a Spectre attack to break KASLR on Intel TDX, exploiting the `UOPS EXECUTED.CORE` performance counter channel. We target the kernel code region, which is mapped using 2 MB pages above `0xffffffff80000000` on `x86_64` Linux, with its exact location randomized every boot. To leak the KASLR offset, we exploit that `UOPSEXECUTED.CORE` counts speculatively executed uOPs. To probe a virtual address of a victim TD, we let the TD speculatively execute it using the gadget shown in Section 5.1. For addresses with executable pages mapped, the processor can execute the code on them speculatively, leading to an increased amount of uOPs executed. Addresses without executable memory mapped lead to a stall until the CPU determines that a misprediction occurred.

We performed measurements every 2 MB (30 times per page) starting at the beginning of the Linux kernel code ASLR range (`0xffffffff80000000`),

```
1 add rdx, rdi
2 jmp 0xe
3 lea rax, [rdi + 1]
4 cmp byte [rdi], sil
5 je  0x13
6 mov rdi, rax
7 cmp rdi, rdx
```

Listing 9.5: Beginning of the disassembled Linux kernel `memchr` function. The compare operation in line 4 is a Spectre compare gadget leading to different instructions executed and leaking the result of the comparison.

on our Intel Xeon Silver 4514Y system and provide the results in Figure 9.7. The baseline is visible at ∼8 200 uOPs, which corresponds to no code page being mapped at these locations. The first increase in uOPs is at page number 216, corresponding to virtual address `0xffffffff9b000000`. This aligns exactly with the virtual address provided by `/proc/kallsyms` in the victim TD. We can determine the correct KASLR offset in <2 seconds, requiring 30 measurements per page. This is the same performance range as prior KASLR breaks [38, 27, 8, 41, 51, 39, 37, 24] albeit on Intel TDX.

5.3. Leaking Arbitrary TD Memory

Being able to leak arbitrary memory provides the host with all secrets located inside the TD, such as encryption keys and other sensitive information, breaking the confidentiality of TDX. To leak memory with a Spectre attack based on the `UOPSEXECUTED.CORE` channel, we require nothing more than a gadget that varies the number of uOPs executed depending on some input, e.g., any compare and index gadgets. Such gadgets are common in the Linux kernel [9], e.g., every time a flag in memory is checked, memory locations are compared. Which gadgets exactly are exploitable depends on the exact registers the attacker can control. For our attack with the vulnerable code described in Section 5.1, we use a compare gadget in the `memchr` function.

The first few instructions of the `memchr` in our Linux `6.8` kernel are provided in Listing 9.5. Line 4 compares the byte stored at the memory location held in the `rdi` register and the `sil` register, which is the lowest

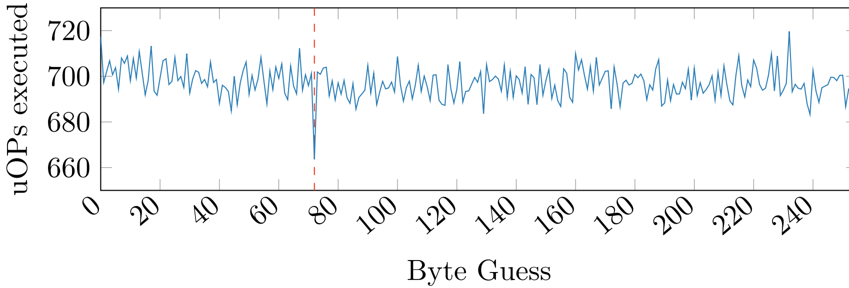


Figure 9.8.: Example of leaking bytes using `memchr` (Listing 9.5) as a Spectre gadget. Each data point is the average of 10 executions for a byte guess. When the guessed byte and the target byte are the same at byte 71 (dashed line), the compare is true, leading to a jump and a lower number of uOPs executed due to the different speculative execution path.

significant byte of the `rsi` register on x86. This comparison is followed by a conditional branch, which leads to different code being executed depending on whether the branch is taken. To leak specific values, we let the TD trigger our gadget for each possible value for `sil`. We repeat this 10 times and compute the mean to eliminate noise.

The results of leaking a byte by using the `memchr` compare gadget are shown in Figure 9.8. The average uOPs executed hover around 700 cycles for all byte guesses except for byte guess 72. This is the result of the comparison in line 4 returning false for most byte guesses, therefore, not taking the branch in line 5. With byte guess 72, the number of executed uOPs drops to ~660 cycles. As this is the correct value, the branch is taken during speculation, leading to different instructions and, consequently, a different number of uOPs being executed. With our attack, we are able to leak memory at a rate of 52.6 bit/s (n=800, $\sigma_{\bar{x}}$=0.03), with an error rate of 0.6 % (n=800, $\sigma_{\bar{x}}$=0.02). This is slower than prior Spectre attacks [40, 24], but still completely breaks the confidentiality of Intel TDX.

6. SSH Keystroke Timing Attack

Inter-keystroke timing attacks have been demonstrated from various environments using different techniques [63, 26, 65, 86, 74, 54]. Song et al.

[74] showed that attackers can detect keystrokes by observing the encrypted network traffic of an SSH session. All network traffic to a TD is forwarded through the host, making it possible for the host to observe the network traffic. To mitigate against this attack, OpenSSH implements the `ObscureKeystrokeTiming` feature [57], which hides the real keystroke packets by sending additional fake interactive packets in short intervals. The fake packets cannot be distinguished from real keystroke packets by observing the network traffic, and are treated as ping packets by the SSH server. However, we can distinguish the fake and real packets by observing the CVM through precise timing measurements.

6.1. Attack

Because the attacker controls the host, they can observe the network traffic to and from the TD. Thus, the attacker knows exactly when the TD receives an SSH packet and when it responds to it. To distinguish fake and real packets, we use the precise timing between when the network package is delivered and when the TD sends a response. In OpenSSH `10.0p2`, when `ObscureKeystrokeTiming` is enabled, the client periodically (default: every 20 ms) sends a `SSH2_MSG_PING` packet. To an attacker, the encrypted network packages stemming from real keystrokes and `SSH2_MSG_PING` packets are indistinguishable from each other. When the server receives a `SSH2_MSG_PING` packet, it responds with a `SSH2_MSG_PONG`. This response is sent immediately in the `ssh_packet_read_poll_seqnr` function. In contrast, normal keystroke packets cause the function to return and look up the appropriate handler function in the `*ssh->dispatch` table, which eventually forwards the keystroke to the application. As both execution paths are not identical and require a different amount of work, it should be possible to distinguish them through precise timing measurements.

To determine the processing time of a package, we measure the time between the interrupt for the network packet being injected into the TD and the MMIO TDcall by the guest to send the response packet. In the TDX threat model, interrupts that the TD receives are untrusted, as the host can decide when or even if an interrupt is injected into the guest. We, therefore, do not inject any interrupts into the guest in between the network packet being delivered and the guest responding. As this includes the guest timer interrupts, this essentially disables preemption for the TD, eliminating the threat of unwanted threads being scheduled in the TD.

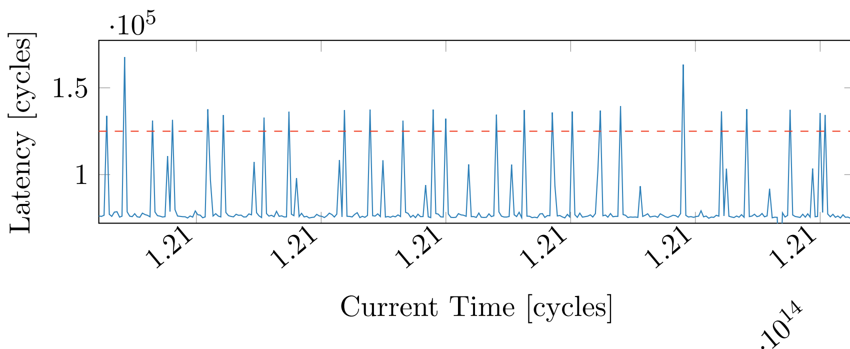
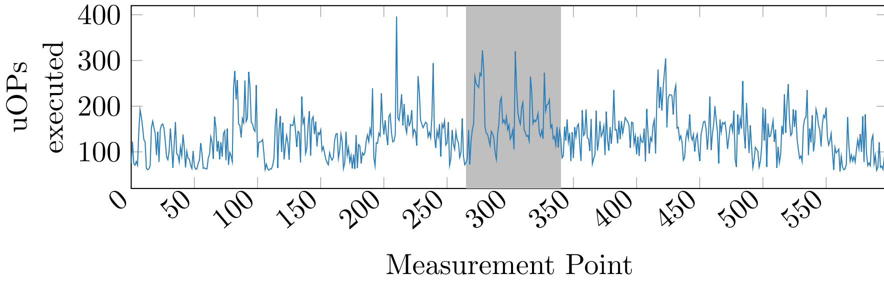


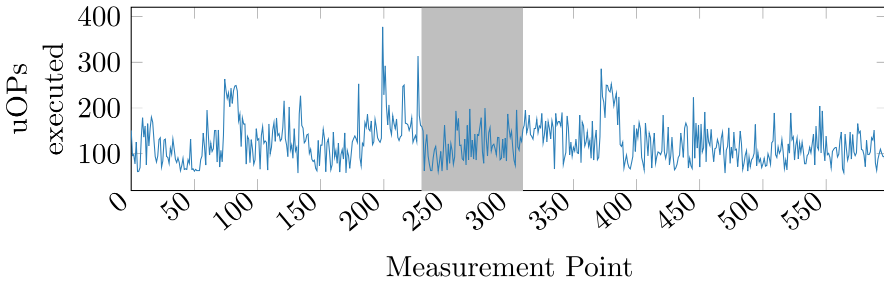
Figure 9.9.: Latency between forwarding an SSH package to the TD and the network response of real and fake OpenSSH keystroke packages. Real keystroke packages take more than 125 000 cycles (marked by a dashed line) to process on our system, while fake keystrokes are processed significantly faster. This makes keystrokes easily detectable for a malicious host.

A precise timing trace of network packet response times of an SSH connection with `ObscureKeystrokeTiming` enabled is shown in Figure 9.9. Each data point is a single timing measurement done using the host's timestamp counter (TSC). There is a clear baseline visible at ∼75 000 cycles or 37.5 µs, which corresponds to `SSH2_MSG_PING` used by the mitigation. A large number of spikes stand out from this baseline. The spikes above ∼125 000 cycles (62.5 µs), marked with a dotted vertical line, are keystroke packages that are being processed. This small timing difference would be extremely difficult to measure over a regular network, as it is small enough to completely vanish through regular OS operation, such as scheduling. Despite this, these timing differences can be easily and reliably measured under the regular CVM threat model.

Performance Counter While we focus our evaluation on precise timings as they are currently enough to differentiate real keystrokes from fake keystrokes, `UOPSEXECUTED.CORE` can also be used to distinguish between the two packet types. A part of the uOP traces of the processing for real keystrokes and fake keystrokes generated by OpenSSH are shown in Figure 9.10. Despite two traces having multiple similar features, such as a plateau between measurement points 80 and 100, as well as a slight increase in uOPs executed around measurement point 200, there are multiple differences, making them reliably distinguishable. One of these



(a) Fake Keystroke



(b) Real Keystroke

Figure 9.10.: `UOPSEXECUTED.CORE` traces of a fake keystroke (Figure 9.10a) and a real keystroke (Figure 9.10b) for OpenSSH. The main difference between the two traces is highlighted in gray. The fake keystroke trace has multiple spikes of high uOP throughput, while the real keystroke trace lacks these spikes.

differences is highlighted in gray in Figure 9.10a and Figure 9.10b. For fake keystrokes, there are large spikes in uOPs executed in this area, which are missing for real keystrokes. As this is a very distinct difference, performing inter-keystroke timing attacks on OpenSSH would be possible through `UOPSEXECUTED.CORE` if the OpenSSH team mitigates the currently existing timing side channel.

6.2. Evaluation

We performed our attack on a TD running OpenSSH version `10.0p2` on Ubuntu 24.04, following the threat model outlined in Section 4.1. We enabled `ObscureKeystrokeTiming` with the default settings, resulting in fake packets being sent approximately every 20 ms. Our experiments were

performed with one of the authors typing at their regular speed inside of a text file edited using VIM inside of the TD through an SSH connection. Overall, 418 keys were typed. Of the 418 keys, all were detected correctly with 3 false positives. This results in an F_1 score of 99.6 %, making it an extremely reliable attack. The high F_1 stems from the high amount of control the host has over the environment in this threat model, allowing for the elimination of noise sources such as interrupts and scheduling.

While correctly detecting a majority of the keystrokes is important for an inter-keystroke timing attack, a low variation in the detection latency is vital to recover words from the recorded information. For our experiment, we measured an average latency of 60 ms between the key being pressed and the interrupt for the key being injected into the guest. The standard deviation of the latencies is 5.64 ms. While the latency itself seems high, it itself is irrelevant for an inter-keystroke timing attack. As long as all packages have a similar delay, the timings of the key presses relative to each other (which is holding the information) remain the same, making the standard deviation the relevant metric for this kind of attack. Our standard deviation of 5.64 ms is significantly lower than the average inter-keystroke interval of 120 ms for fast typists [18] and similar to existing inter-keystroke timing attacks [62, 69], making it a viable channel for this attack.

7. Discussion & Related Work

In this section, we discuss related work, as well as possible mitigations for our attacks. We first discuss the performance counter-based attacks and finish with the attack on the OpenSSH keystroke detection mitigation.

7.1. Performance Counter Attacks

Unlike the mitigations against single-stepping and performance-counter-based attacks on the same logical core that Intel employs, our attacks can not be mitigated through a change in the TDX module. The TDX module can not hinder sibling logical cores from using certain performance counters. The best way to mitigate this issue is to not count performance counter events of sibling logical cores across TD boundaries. While we do not believe that this can be done on existing hardware, it is the best long-term solution.

For existing hardware, the best option is to disable hyperthreading. With hyperthreading disabled, there is no sibling logical core available to collect data from performance counters that monitor the whole core. Whether hyperthreading is enabled is attestable, making this enforceable by the TD and very easy to implement. The main disadvantage of this mitigation is the loss in performance.

Alternatively, the TDX module could enforce that all logical cores of a physical core have to be either inside of one TD or in the host. With both logical cores inside of the same TD, the host can not monitor the performance counters. To enforce this, the TDX module could block when entering a TD until both logical cores are ready to enter it. For VM exits, whenever one logical core performs a VM exit, the TDX module could send an IPI to the other logical core, forcing a VM exit. This would lead to performance loss due to unnecessary VM exits and VM entries blocking until both cores are ready, but this loss in performance should be significantly smaller than disabling hyperthreading.

To mitigate the impact of `UOPSEXECUTED.CORE` on Spectre attacks, compilers can insert fences between compares and their subsequent branch [30]. We note that this is not the same as simply disabling branch prediction. Since this would lead to a significant loss in performance [9], other mitigation options may be more desirable. Additionally, this approach would only mitigate the use of `UOPSEXECUTED.CORE` in Spectre attacks and not any other attack vectors stemming from the discussed performance counters. Our new side channel is most relevant when using a gadget that, itself, does not lead to a branch prediction, but instead is executed during speculation. It is necessary for our side channel that the branch is correctly evaluated, as it leaks the outcome of the branch.

Closest to our work is CounterSEVeillance by Gast et al. [21]. They use performance counters on the same logical core to attack a TOTP implementation, the MbedTLS RSA implementation, and perform a divide-and-surrender-style attack on a HQC-KEM implementation running in an AMD SEV-SNP CVM. To perform these attacks, they mainly take advantage of performance counters tracking branches. While CounterSEVeillance [21] and our attacks are similar on the surface, there are key differences: First, the attack style of CounterSEVeillance monitors the performance counters on the same logical core and assumes that performance-counter-based attacks are mitigated if performance counters are context switched when entering a TEE, explicitly mentioning that Intel TDX implements this mitigation. Our attacks take advantage of a

performance counter that can monitor the victim from the sibling logical core, bypassing Intel's current defense. Second, in addition to traditional attacks with performance counters, we explore Spectre-type attacks using the information gained by performance counters, allowing for the use of a wide range of potential new gadgets. Third, we do not rely on single-stepping for any of our attacks, instead leveraging the highly detailed information `UOPSEXECUTED.CORE` provides us to leak secret information. Lastly, we show a completely novel attack that bypasses the recently introduced mitigation against inter-keystroke timing attacks in OpenSSH.

Lou et al. [50] also attack AMD SEV-SNP with performance counters and perform website fingerprinting and keystroke detection. The performance counters are monitored on the same logical core as the victim, which is not possible on Intel TDX. Similar to Gast et al. [21], Lou et al. [50] mention that Intel TDX already protects against performance counter-based attacks and recommend a similar mitigation for AMD SEV-SNP. We show that this defense is insufficient and still makes leakage through some performance counters possible.

Cho et al. [14] and Li et al. [43] use performance counters to detect malicious applications, which could also be applied to TEEs, making a case for providing some performance counter information available to the host. Weissteiner et al. [81] try to strike a balance between confidentiality and thread detection by decorrelating reported performance counter values from the actual hardware events in TEE environments, allowing performance counters to be enabled while protecting against fine-grained information leakage. However, this mitigation does not protect against information leakage through shared performance counters, as it only applies the decorrelation when context switching from the host to the TEE. Thus, the performance counters from a sibling logical core are not decorrelated and still leak information.

Mandal et al. [52] use performance counters to monitor the host application managing the TD and detect contention effects through them. By monitoring performance counters such as instructions executed, L1 dcache misses, and branch misses, they can determine whether the throughput on the sibling logical core of the TD changes due to contention. Their attack does **not** directly leak information from the TD through the performance counters, as all performance counters listed only monitor the current logical core and are context switched whenever entering and exiting a TD. Due to the limited information, Mandal et al. [52] only perform application fingerprinting. In contrast, our work found a small subset of

performance counters that allow for direct monitoring of the TD workload, as they capture data for the whole physical core, even when one of the two logical cores is currently running a TD. This allows us to perform much more fine-grained attacks, such as an RSA private key recovery and Spectre-type attacks.

7.2. OpenSSH

To mitigate the OpenSSH inter-keystroke timing attack, the most effective approach would be to implement one processing path for real and fake keystroke packages to avoid a deviation in timing. Due to the difference in how real keystrokes have to be processed, e.g., , there might be a response package with new information that is displayed, but this might not be fully possible. An approximation of this would be to delay the fake keystroke responses of the mitigation into a similar timing range as the last real keystrokes being responded to. The fake keystrokes do not have to be able to mimic the timings of real keystrokes perfectly. Inter-keystroke timing attacks rely on precise timing differences between individual keystrokes. Therefore, as long as large parts of the keystrokes can not be differentiated from fake keystrokes, the timing differences are not valuable for the recovery of the typed words. Another possible angle to defend against these types of attacks would be to delay keystrokes from being sent for a random amount of time. If the random delay range is large enough, e.g., in the range of a few 100 ms, the inter-keystroke timings are also no longer useful for recovering words. This approach has the disadvantage that it does not hide the number of characters typed, but only obfuscates the timings between them. Additionally, this random delay can make the SSH connection less responsive, which might discourage some users from activating this mitigation.

Giavridis [22] discovered that the OpenSSH inter-keystroke timing defense in version `9.7` can be easily bypassed by measuring the response times of packages. Keystroke packages take 3 times longer (60 ms) to process than fake keystrokes (20 ms). This attack was mitigated starting OpenSSH `9.8p1`. Despite the mitigation, we show that there is still a significant timing difference between response times for SSH packages, which can easily be exploited in a CVM scenario running the most recent OpenSSH version (`10.0p2`). Lipp et al. [46] leaked keystrokes from sandboxed JavaScript with an identification rate of 81.75 %. Rauscher et al. [62] measure the

timings of inter-processor interrupts to detect other interrupts and performed an inter-keystroke timing attack with a standard deviation of 6.15 ms and an F_1 score of 97.9 %. Schwarz et al. [69] propose generating random keystrokes to mitigate inter-keystroke timing attacks, similar to the OpenSSH mitigation.

8. Conclusion

Intel's recent CVM extension, Intel TDX, actively mitigates performance counter attacks by context switching relevant registers. Despite these mitigations, we uncovered a fatal flaw in this mitigation, as it does not account for leakage through counters that monitor the whole physical core. We analyzed the available performance counters on our recent Intel CPU and found 8 that allow for information leakage from the sibling logical core, even when the other core is inside of a TD. We uncovered one particular counter, `UOPSEXECUTED.CORE`, that provides information on uOPs executed, which can be exploited to leak sensitive information on the execution flow inside of a TD. With `UOPSEXECUTED.CORE`, we attack the MbedTLS RSA-2048 implementation running inside of a TD and leak the full private key with an average Levenshtein distance of only 0.92 bits. While this attack already shows how dangerous this exposed information can be, we further abuse that `UOPSEXECUTED.CORE` does not only count uOPs of instructions that are committed, but also of speculatively executed instructions. This enables the use of a wide range of new gadgets for Spectre attacks by relying on uOPs executed during speculation instead of other side effects, such as memory accesses. We demonstrate how dangerous this can be by using the Linux kernel's `memchr` implementation to leak arbitrary TD memory at a rate of 52.6 bit/s and by breaking KASLR in less than 2 s. In addition to performance counter-based attacks, we discovered that the novel inter-keystroke timing defense of OpenSSH is insufficient for a CVM scenario, allowing us to detect real keystrokes with an F_1 score of 99.6 %, despite the active mitigation. We conclude that the current mitigations against performance counter attacks in Intel TDX are incomplete and require further refinement to protect against this threat.

Ethics Considerations

We responsibly disclosed our findings to Intel on August 7, 2025, and to the OpenSSH team on August 25, 2025. Both Intel and the OpenSSH team have products that are impacted by our attacks. We conducted our research to highlight the existence of the found vulnerabilities and provide the stakeholders with the opportunity to mitigate them before they are published. While the existence of new attacks can be seen as a negative, it is vital that researchers uncover these vulnerabilities and adequately disclose them to the vendors before a malicious third party can find and exploit them. The publication of such attacks is also essential, as it is a reference for future developers not to introduce similar vulnerabilities into their products. We have proposed mitigations for both existing as well as future platforms to minimize any negative side effects of our research. Lastly, all our experiments were done on our own machines with no code from other users running on them.

Acknowledgements

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 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. AMD Secure Encrypted Virtualization (SEV). 2024. URL: https://developer.amd.com/sev/ (p. 272).

[2] 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 (pp. 268, 271).

[3] ARM. Arm Confidential Compute Architecture. 2024. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture (p. 271).

[4] ARM. mbed TLS. 2020. URL: https:///tls.mbed.org (pp. 279–281).

[5] ARM. TrustZone for Arm Cortex-M Processors. 2024. URL: https://www.arm.com/technologies/trustzone-for-cortex-a (pp. 268, 271).

[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. 286).

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (pp. 268, 272).

[8] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 288, 289).

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security. 2019 (pp. 269, 286, 287, 289, 295).

[10] Canonical. Intel® Trust Domain Extensions (TDX) on Ubuntu. 2025. URL: https://github.com/canonical/tdx (p. 280).

[11] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. Fight Hardware with Hardware: Systemwide Detection and Mitigation of Side-channel Attacks Using Performance Counters. In: Digital Threats: Research and Practice (DTRAP) 4.1 (2023), pp. 1–24 (p. 273).

[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. 272).

[13] 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 (p. 273).

[14] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. Real-time detection for cache side channel attack using performance counter monitor. In: Applied Sciences 10.3 (2020), p. 984 (pp. 273, 296).

[15] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (p. 268).

[16] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In: S&P. 2019 (p. 273).

[17] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In: S&P. 2025 (p. 268).

[18] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. Observations on typing from 136 million keystrokes. In: CHI Conference on Human Factors in Computing Systems. 2018 (p. 294).

[19] Jake Edge. Kernel address space layout randomization. 2013. URL: `https://lwn.net/Articles/569635/` (p. 288).

[20] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (p. 272).

[21] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In: NDSS. 2025 (pp. 268, 273, 279, 282, 285, 286, 295, 296).

[22] Philippos Maximos Giavridis. SSH Keystroke Obfuscation Bypass. 2024. URL: `https://crzphil.github.io/` (p. 297).

[23] Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In: DIMVA. 2025 (p. 279).

[24] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In: CCS. 2020 (pp. 286–290).

[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In: EuroSec. 2017 (pp. 268, 272).

[26] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (p. 290).

[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 288, 289).

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 288).

[29] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In: CHES. 2020 (p. 272).

[30] Intel. Intel Analysis of Speculative Execution Side Channels. 2018. URL: `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf` (pp. 287, 295).

[31] Intel. Intel Performance Monitoring Events. 2024. URL: `https://perfmon-events.intel.com/` (pp. 275, 278).

[32] Intel. Intel Software Guard Extensions (Intel SGX). 2024. URL: `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html` (pp. 271, 274).

[33] Intel. Intel Trust Domain Extensions. 2021. URL: `https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf` (pp. 268, 272, 280).

[34] 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. 271, 272, 274, 276, 282).

[35] Intel. Intel(R) Software Guard Extensions Developer Guide. 2025. URL: https://cdrdv2-public.intel.com/671334/intel-sgx-developer-guide.pdf (p. 273).

[36] Intel. TDX Module 1.5.16 Source Code. 2025. URL: %7Bhttps://github.com/intel/tdx-module%7D (p. 280).

[37] Hyerean Jang, Taehun Kim, and Youngjoo Shin. SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon. In: CCS. 2024 (pp. 288, 289).

[38] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (pp. 288, 289).

[39] Yu Jin, Chunlu Wang, Pengfei Qiu, Chang Liu, Yihao Yang, Hongpei Zheng, Yongqiang Lyu, Xiaoyong Li, Gang Qu, and Dongsheng Wang. Whisper: Timing the Transient Execution to Leak Secrets and Break KASLR. In: DAC. 2024 (pp. 288, 289).

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 269, 272, 286, 290).

[41] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In: EuroS&P. 2020 (pp. 288, 289).

[42] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security. 2017 (p. 272).

[43] Congmiao Li and Jean-Luc Gaudiot. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In: COMPSAC. 2019 (pp. 273, 296).

[44] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In: Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). 2018 (p. 273).

[45] 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. 283).

[46] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (p. 297).

[47] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (p. 272).

[48] 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. 273, 279).

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 280).

[50] Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Guo Shangwei, and Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In: DSN. 2024 (pp. 268, 273, 296).

[51] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. In: arXiv:1801.04084 (2018) (pp. 288, 289).

[52] Upasana Mandal, Shubhi Shukla, Nimish Mishra, Sarani Bhattacharya, Paritosh Saxena, and Debdeep Mukhopadhyay. Exploring side-channels in Intel Trust Domain Extensions. In: Cryptology ePrint Archive (2025) (p. 296).

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (pp. 268, 272, 273).

[54] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 290).

[55] 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. 273, 280).

[56] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. Spectre attack against SGX enclave. 2018 (p. 272).

[57] OpenBSD Journal. Keystroke timing obfuscation added to ssh(1). 2023. URL: https://undeadly.org/cgi?action=article;sid=20230829051257 (p. 291).

[58] OpenSSH. 2025. URL: https://www.openssh.com (p. 270).

[59] Perf Wiki. Main Page. 2020. URL: https://perf.wiki.kernel.org/index.php/Main_Page (p. 273).

[60] 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. 273).

[61] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In: AsianHOST. 2019 (p. 273).

[62] Fabian Rauscher and Daniel Gruss. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In: CCS. 2024 (pp. 294, 297).

[63] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In: NDSS. 2024 (p. 290).

[64] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. TDXploit: Novel Techniques for Single-Stepping and Cache Attacks on Intel TDX. In: USENIX Security. 2025 (pp. 268, 273, 280).

[65] 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. 290).

[66] Keegan Ryan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In: CCS. 2019 (p. 272).

[67] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. 2020 (p. 272).

[68] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 268, 272).

[69] 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 (pp. 294, 298).

[70] 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. 280).

[71] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 286).

[72] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. In: Cybersecurity 3.1 (2020), p. 2 (p. 279).

[73] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In: FC. 2021 (pp. 269, 272).

[74] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security. 2001 (pp. 290, 291).

[75] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In: USENIX ATC. 2017 (pp. 268, 271).

[76] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security. 2018 (pp. 272, 280).

[77] 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. 272).

[78] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (pp. 268, 273, 280).

[79] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: CCS. 2017 (pp. 268, 272).

[80] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In: AsiaCCS. 2018 (pp. 268, 272).

[81] 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. 269, 296).

[82] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In: CCS. 2024 (pp. 268, 280, 282).

[83] 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. 271).

[84] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In: CHES. 2024 (pp. 268, 273, 280).

[85] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (pp. 268, 273).

[86] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security. 2009 (p. 290).

[87] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In: USENIX Security. 2024 (p. 280).

[88] Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. HDROP: Detecting ROP Attacks Using Performance Monitoring Counters. In: ISPEC. 2014 (p. 273).

10

Systematic Analysis of Kernel Security Performance and Energy Costs

Publication Data

Fabian Rauscher, Benedict Herzog, Timo Hönig, and Daniel Gruss. Systematic Analysis of Kernel Security Performance and Energy Costs. In: AsiaCCS. 2025

Contributions

Main Author.

Systematic Analysis of Kernel Security Performance and Energy Costs

Fabian Rauscher[1], Benedict Herzog[2], Timo Hönig[2], Daniel Gruss[1]

[1]Graz University of Technology, [2]Ruhr-Universität Bochum

Abstract

Security measures and patches typically come with a performance cost. While performance is a common metric to assess the practicality of security measures and patches, energy cost is typically ignored. However, it is unclear to what extent performance and energy costs of security correlate and to what extent they diverge.

In this paper, we present the first systematic analysis of the energy costs of CVE fixes and mitigations. We use the Linux kernel as a case study. We perform energy-performance delta benchmarks using Intel RAPL on the two software versions or configurations under test on our i7-6700K. We automatically attribute CVEs from an 8-year time frame to patch sets, automatically compile the corresponding source code versions, *i.e.*, pre-patch and post-patch, and automatically benchmark performance and energy consumption. Furthermore, we perform an evaluation of all kernel mitigations. We show that energy and performance costs diverge very clearly in some cases. One of these cases is the `retbleed IBPB` mitigation, with runtime increases of 0.4 % and energy consumption decreases of 7.1 % for the `apache` Phoronix benchmark. While not a fully indicative experiment, our work underscores the need for future security research to evaluate energy cost in addition to performance.

1. Introduction

System security and efficiency are often seen in conflict. In particular, security measures and patches usually reduce performance. Security mechanisms and patches can be found on any layer of the system, *i.e.*, hardware, operating system, and application level. The standard metric to assess the cost of security mechanisms is measuring performance overhead over

benchmarks. Well-known examples of the overheads of security patches include the patch against Meltdown [20, 44], with −5 % to 800 % overhead [19], and the mitigations for various Spectre attack variants, also with a wide range of reported overheads [7]. Some of these are implemented on an application level (e.g., site isolation [57]), on the system level (e.g., KPTI [19]), or on the hardware level (e.g., eIBRS [7]).

Performance overheads of new security mechanisms and mitigations are studied extensively across most corresponding publications. However, performance only indirectly covers energy overheads. Herzog et al. [24] found that for KPTI [19], some benchmarks show different overheads for energy than for performance. The reason is that the energy consumption is influenced significantly by the runtime and the performance mode the processor runs in. For instance, earlier stalling of the out-of-order execution can reduce energy consumption without affecting performance significantly. Conversely, adding memory traffic or on-core activity can increase energy consumption significantly without a significant effect on the performance. This lack of information about the energy overhead of security mechanisms, however, is in stark contrast to the importance of energy efficiency, especially in data centers [17].

A series of works on software energy costs [8, 10, 54], mostly focused on user-level software, consistently showed that software performance and energy costs are only weakly correlated and performance (*i.e.*, execution time) cannot be used to estimate energy costs accurately. Very few works analyzed the energy costs of security mechanisms besides Herzog et al. [24], e.g., Siavvas et al. [60] analyzed the costs of application-level security checks. This is a stark contrast across scientific communities, as our analysis of 1 257 publications from 2023 from 7 top-tier systems and system security conferences reveals. In some systems conferences, more focused on novel functionality and performance gains than security, about 70 % of the published papers mention energy and power consumption. However, only ∼1 % of published papers in top-tier security conferences provide energy and power consumption. This is particularly concerning as there is a clear relation between power consumption and security, e.g., Rowhammer faults related to newer DRAM operating with lower power [35] and undervolting-related faults [51, 30]. Given frequent new mitigations and the performance of some recent CPUs being degraded to that of 3 years older predecessors [25], *i.e.*, about 15 % slower, we also have to ask:

What are the energy costs of system security? How far do energy and performance costs of system security diverge?

In this paper, we present a systematic analysis of the energy costs of CVE fixes and mitigations in the Linux kernel. The idea behind our systematic analysis is a differential energy measurement using Intel RAPL, a processor interface accurate enough to mount power analysis attacks [43]. By benchmarking the two corner cases of the software under test, *i.e.*, either the source code commits pre-patch and post-patch or the mitigation enabled and disabled, we can determine precise performance and energy overheads. For this purpose, our analysis framework automatically attributes CVEs to patch sets and locates them in the source-code versioning repository.

We evaluate kernel CVE fixes in a case study starting from Linux 4.0 over an 8-year time frame on an i7-6700K. From these, we were able to automatically identify 1 616 CVEs that we can automatically map to patch sets present in the source-code repository. We automatically compile the Linux kernel pre- and post-patch and benchmark it using the Stress-NG and Phoronix benchmark suits. We automatically analyze all fixes through automatic debugging, allowing us to filter for fixes that affect code executed by our benchmarks. We then collected performance and energy data for the 108 CVEs, for which we determined through this debugging that they affect our system. This reduces the number of CVEs for which a large amount of benchmark runs are necessary for statistically significant results while still analyzing all CVEs.

Overall, we obtain energy and performance data for 108 Linux CVEs that affect our system configuration and benchmarks. In this case study, we observe that the energy and performance costs are largely correlated with a few exceptions. We discuss these exceptions, as well as corner cases that reach the limitations of our approach. This work is **not** a cost-benefit analysis. Instead, we introduce a framework for automatically filtering through a large number of security patches and analyzing patches that potentially have large effects on energy consumption or runtime. Additionally, we analyze the correlation of energy and runtime overhead.

Beyond the automated quantitative analysis, we also qualitatively evaluate the performance and energy overheads of mitigations in the Linux kernel. While energy and performance costs are still correlated, we can see more clear corner cases here with clear energy overheads while the performance is largely unchanged and vice versa. This shows that energy cost and performance cost should both be evaluated in future security works.

Contributions. We make the following main contributions:

1. We automatically run combined energy-performance benchmarks for security patches related to CVEs on our i7-6700K.
2. Based on our approach, we provide the first large-scale systematic analysis of energy and performance costs of security patches and mitigations in the Linux kernel.
3. We perform a large-scale analysis of energy and performance costs of kernel patches over an 8-year time frame, revealing significant outliers and divergences.
4. We perform 14 qualitative case studies of energy and performance costs and use performance counters to reason why energy and performance counters diverge significantly.

Outline. In Section 2, we provide background. In Section 3, we describe our automated performance-energy benchmarking of security patches and mitigations. In Section 4, we present our large-scale analysis of Linux patches over an 8-year time frame. In Section 5, we present our analysis of Linux mitigations. We discuss limitations in Section 6 and related work in Section 7. We conclude in Section 8. **Ethics Considerations.** There are no new vulnerabilities discovered or published in this work. We analyzed public CVE patches and mitigations present in the Linux kernel.

2. Background

In this section, we provide background on OS kernel security, the performance overheads of security measures, and CPU energy measurement interfaces for benchmarking and side-channel analysis.

2.1. Kernel Security

Since modern systems run code from arbitrary sources, containerization and process isolation using operating system kernel support have become key elements of system security. The operating system kernel is typically considered part of the trusted computing base (TCB). For decades, the academic community has explored pathways to minimize a system's TCB, e.g., by moving components out of the kernel or only allowing verified

code to run in the kernel [48, 11], or entirely switching to a micro-kernel approach [36].

The most widely used open-source kernel today is the Linux kernel, which is also integrated into the Android operating system [1]. Some parts of the kernel source code are continuously improved and extended [23], especially drivers or modules. Furthermore, there are more developers involved in driver or module code, potentially increasing the risk of bugs in these parts of the kernel [61]. Given the large code base of the Linux kernel, it is natural that numerous bugs are introduced and discovered [59]. Furthermore, as the Linux kernel is part of the TCB, any exploitable bug must be considered a security issue that requires a patch [62]. Prior work studied Linux kernel vulnerabilities and found typical C program bug classes to be most prevalent, e.g., buffer errors and invalid dereferences [59]. Consequently, there is a constant stream of security patches for the kernel [31]. Not patching a security issue in the kernel (in time) is a significant security risk [9]. These severe consequences imply that patching security issues is of utmost importance, and the attached costs are an unavoidable side effect.

2.2. Performance Overheads of Security

The performance costs of security patches and security mechanisms are well-studied. Kernel developers are required to keep the performance costs for new security mechanisms to a minimum and justify any overheads [4, 13, 14]. Consequently, the performance of the core functionality of Linux remains consistently fast and does not continuously slow down due to new functionality [58]. An exception to this rule can be mitigations for known vulnerabilities to ensure the integrity and security of the operating system. Nevertheless, such mitigations are continuously analyzed and improved to further reduce the overhead where possible [26, 3, 24, 46, 5]. This applies, in particular, to hardware vulnerabilities, where the vulnerability cannot be remedied directly, but software mitigations must remain permanently active. Besides analyzing the performance costs, additional studies analyze the timeliness and effectiveness of security patches, e.g., in a large study consisting of 4 000 security patches [42]. The energy overhead of security patches, however, has received only little attention so far, with only few works analyzing the overhead of specific mitigations [24, 46].

As a consequence of these considerable performance overheads, the Linux kernel offers many boot-time configuration options to enable or disable specific security measures. Hence, system administrators can select which measures are required in their specific setup. However, also as a consequence of the performance overheads, users are disabling mitigations regardless of their individual exposure to attacks to restore the performance without the patches or mitigations [41, 12]. Even though mitigations can even improve the performance for specific workloads [12]. This already shows that users do not necessarily share the view of the scientific community that the mitigation of security issues is of utmost importance.

2.3. CPU Energy Measurement Interfaces

Both Intel and AMD introduced a mechanism for controlling thermal and power constraints from software [18]. On Intel processors, the Running Average Power Limit (RAPL) mechanism allows software to adjust the CPU frequency and voltage, as well as the power limits. RAPL provides different energy domains for different parts of the processor [21]. AMD provides an energy measurement interface that is essentially compatible with RAPL [2]. The accuracy of energy measurements is typically on the scale of milliseconds or microseconds. Because of its accuracy and precision, RAPL has been used for benchmarking works in the past [34, 24]. RAPL can be accessed through a set of model-specific registers (MSRs), which are available to the kernel. To make them accessible to regular software, Linux provides a driver that allows user-space software to directly read the information provided by RAPL [53]. These integrated energy measurement interfaces significantly lower the barrier for energy measurements compared to measurements with external devices. In particular, they are an enabler for systematic analyses, such as the one presented in this work, as well as for energy overhead measurements for new security mechanisms.

3. Methodology

In this section, we discuss our measurement methodology and test setup. Furthermore, we propose a prefiltering method allowing us to evaluate a large number of CVEs in a short amount of time.

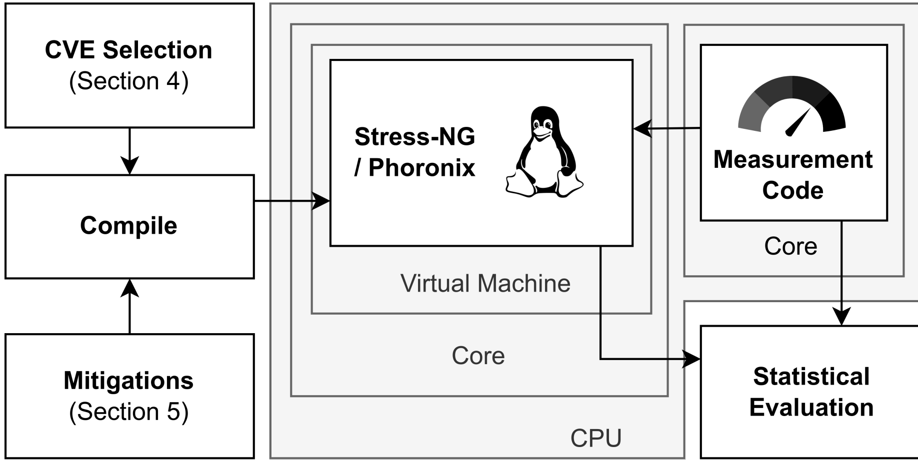


Figure 10.1.: Overview of our analysis framework and experimental setup. The kernels are first compiled. The tested kernel is launched through KVM on its own core and executes the benchmarks. The measurement code runs on its own core, tracking energy consumption through RAPL and the runtime of each benchmark. Once a sufficient number of measurements are completed, the results are evaluated.

3.1. Measurement Methodology

All our measurements are conducted on an i7-6700K CPU. The benchmarks are executed on an isolated core inside a virtual machine running under KVM with an unmodified Debian 11 using the ext4 filesystem on an Ubuntu 20.04 LTS. An overview of our setup is provided in Figure 10.1. We run the tested kernels inside a virtual machine similar to the way they would be used in the cloud. All Linux kernel versions are compiled with the default KVM configuration. The energy measurements are done on the host using Intel RAPL. We use the PKG RAPL domain, which provides the energy consumption of the whole CPU. As RAPL does not distinguish between code executed inside a VM and code on the host, energy measurements include the power consumption of the benchmarks running in the VM. We sample the RAPL model-specific register (MSR) in 1 ms intervals throughout all benchmark executions. For runtime measurements, we use a millisecond-accurate timer.

Our setup offers fast drop-in replacement of the kernel without requiring a system restart, adapting the measurement code for different kernel versions, relying on the stability of the tested kernels and driver availability in

the case of older kernels. This setup allows us to test a wide range of kernel versions and CVE fixes in a short amount of time. We evaluate all Linux CVEs for which an explicit fix commit exists from Linux 4.0 to Linux 6.2, covering an 8 year range of Linux releases. We further evaluate existing command line controlled mitigations in Linux 6.2 and their configurations. To avoid confusion, we refer to mitigations or fixes to software bugs in the Linux kernel as `fixes` or `patches` and mitigations to hardware vulnerabilities, which can be enabled or disabled at boot time as `mitigations`. To find the commits, we use a publicly available database that logs Linux CVEs and fixes [47].

For CVEs, our baseline is the kernel version before the commit marked as the fix is applied. We compare this baseline with the kernel version after the fix is applied. In our evaluated 8 year time period, there are 1 616 CVEs according to [47]. We only evaluate CVEs where an explicit fix commit exists, leaving us with 1 604 CVEs. Of the 1 604 CVEs, 46 did not compile or boot on our system, resulting in 1 545 testable CVEs. Build problems for some commits are to be expected as we do not build Linux releases but specific commits. We run multiple benchmarks to evaluate the impact of CVE fixes on performance and energy consumption. Our benchmark set consists of 14 benchmarks from the Stress-NG suite and 5 benchmarks of the Phoronix Test Suite. The Stress-NG suite benchmarks used are `icache`, `fork`, `pthread`, `context`, `pipe`, `io`, `sock`, `udp`, `futex`, `aio`, `switch`, `sctp`, `signal`, and `cpu`. The Phoronix Test Suite benchmarks are `network-loopback`, `mutex`, `osbench`, `apache`, and `pmbench`. We chose our benchmarks to cover a wide range of use cases while keeping the number of benchmarks low. While more benchmarks would be beneficial, it would result in significantly longer measurement times, as many kernels have to be tested with each one. The low number of benchmarks allows us to test a wide range of Linux versions in a reasonable amount of time. We further prefilter benchmarks for each CVE to only test benchmarks that change code executed during a benchmark run. Furthermore, we do not explicitly enable or disable any kernel mitigations when testing CVE fixes.

For Linux kernel mitigations, we run all benchmarks with all available command line options for each mitigation on Linux 6.2. Our baseline for the mitigation measurements is the kernel with the tested mitigation disabled. At the time of testing the available mitigations according to the official documentation for our x86 system are `spectre_v2`, `spectre_v1`, `spec_store_bypass_disable`, `pti`, `l1tf`, `mds`, `tsx_async_abort`, `retb leed`, `mmio_stale_data`, `l1d_flush`, and `kvm.nx_huge_pages` [33]. We do

Table 10.1.: Discussions and Evaluations of Performance- and Energy-related Metrics in Top Publications.

Papers discussing	ISCA	MICRO	ASPLOS	ACM CCS	IEEE S&P	NDSS	USENIX Sec.
Overall	151	229	195	69	101	90	422
Performance	55	41	38	20	33	18	84
Energy	37	20	13	47	68	7	30
Perf. & Energy mentioned	25	8	10	17	29	5	19
Performance & Energy	13	5	8	15	21	2	9
Energy measured, estimated, or discussed	8	2	3	11	16	1	3

not apply prefiltering for the command line options, as the number of mitigations that can be enabled this way is manageable.

3.2. Methodologies across Communities

To understand the current situation of energy measurements in the security community, we performed a comparative study with the systems community. We want to determine to which extent energy costs are discussed in systems and system security publications and identify methodological discrepancies between the communities.

We selected the top system security venues (ACM CCS, IEEE S&P, NDSS, and USENIX Security) as well as the top systems venues (ASPLOS, ISCA, and MICRO). We used a semi-automated approach to retrieve the publications for all 7 conferences in 2023 from the publisher websites. While this worked perfectly for some conferences, one can notice that the number of publications is just slightly below the number of accepted papers for some conferences due to missing publications[1], or repeatedly failed download attempts from the publisher. Thus, overall, we base our evaluation on 1 257 unique papers published at top systems and system security venues. Most publications are in the same length range, given the tight and similar page limits. Despite the differences in their respective

[1]Instead of the paper PDF, the publisher provided a PDF stating that the corresponding paper is not available.

topics, both software and hardware mechanisms have been published in each of these 7 conferences. In systems conferences, both publications focused on novel functionality and performance gains, and publications proposing new security mechanisms can be found.

By-hand evaluating all 1 257 papers is a prohibitive amount of work, even when involving multiple experts for the evaluation. Therefore, we pre-filtered the papers based on an automated keyword search. To identify papers discussing performance-related aspects, we used the expression `(performance|run.?time|execution.time|CPU.time).(cost|consumption|overhead|increase)`. To identify papers discussing energy-related aspects, we used the expression `(energy|power)(.consumption)?.(cost|consumption|overhead|increase)`. It is clear that this approach may have missed some papers, but based on our manual checks, this filter is representative of the entire set of 1 257 papers. To filter papers discussing performance-related aspects further, we used the expression `(energy|power.consumption)` and refined it to also use the expression for energy-related aspects above. All resulting publications were manually evaluated by an expert.

As shown in Table 10.1, our automated keyword analysis yields that 152 out of 321 systems publications from 2023 mention energy efficiency or costs or power consumption. For ASPLOS, this number is around 25 %, whereas for ISCA and MICRO, it is closer to 70 %. We can also see that for those papers that mention performance costs or overheads, more than 85 % of ISCA and MICRO papers also mention energy, and about 45 % of ASPLOS papers. For the system security venues, the numbers look more devastating: Only 63 out of 846 system security publications in our evaluation mention energy or power consumption, corresponding to only 7 %. This is a significant discrepancy to the systems community.

This difference does not exist to the same level for performance metrics, where all conferences are in the range of 18 % to 36 %. This discrepancy can be explained by attack papers that do not present a mitigation in detail and, hence, also no performance evaluation.

To narrow down the set for manual analysis by experts further, we combined the two initial filters, resulting in a selection of 73 publications that possibly discuss performance- and energy-related aspects. We manually evaluated these 73 publications. The manual analysis should identify how many of these 73 publications provide energy costs through measurements, estimations, or at least a discussion of energy costs in a wider sense. The

result of the manual analysis shows that only 44 of the 73 publications (1 257 overall) our search identified actually discuss energy costs. For systems conferences, an overall number of 35 out of 49 publications (321 overall) were identified to discuss energy costs, whereas, for system security, it is only 9 out of 24 (936 overall). However, taking into account that the remaining publications already did not match the keywords in our automated search, our manual analysis provides estimates for the overall ratio of publications that discuss energy additionally to performance: For system security, we can estimate the ratio to ≈1 % that discuss energy and power consumption; for systems publications, we can estimate the ratio to ≈11 %.

This discrepancy indicates methodological differences that need to be addressed. During our manual analysis, we discovered numerous works that could have provided energy costs or estimates, for instance, by using CPU energy power measurements (e.g., using RAPL and equivalent features [18, 2]), or software-based estimates (e.g., using CACTI [50]).

4. Analysis of 8 Years of Kernel CVEs

In this section, we analyze the measurement results of our benchmarks on 1 616 CVEs, which were introduced to the Linux kernel between Linux 4.0 and Linux 6.2.

4.1. Benchmark Filtering

A high-level overview of our filtering process is provided in Figure 10.2. For our evaluation period of Linux 4.0 to Linux 6.2, we found 1 616 CVEs. For 12 of the CVEs, there is no explicit commit marked as a fix [47]. For further 46 CVEs, the unpatched or the patched kernel does not compile. Of the remaining 1 558 CVEs, 13 had at least one kernel that did not boot. This results in 1 545 testable CVEs.

With 1 545 CVEs, a complete run with all benchmarks for each CVE would take over two weeks on our test system. For a high enough sample size, our system would have to run for at least a year. Such a long runtime is not practical. As we do not want to reduce the number of CVEs or benchmarks, we prefilter the benchmarks for each CVE. While there are 1 545 testable CVEs, only a fraction affect the code that our benchmarks run. These

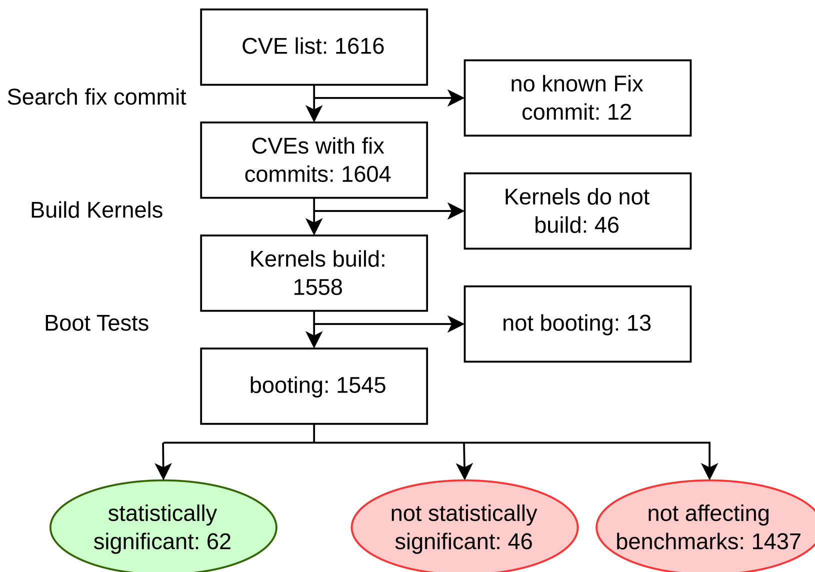


Figure 10.2.: Statistics on how many CVEs exist for the tested time frame, how they were processed in our benchmarking approach, and which led to statistically significant results.

CVEs might only affect Android, ChromeOS, other architectures, or drivers of devices and file systems not present in our test system. By filtering CVEs and benchmarks for CVEs that affect code that is not executed, we significantly reduce the number of overall benchmark runs.

For filtering, we leverage automatic debugging in combination with the ability to debug kernels in QEMU. First, we find all code lines changed by the CVE patch. Second, we set a breakpoint for each changed line. Third, we execute a benchmark and log each breakpoint hit. We use the resulting information to remove CVEs that do not affect any benchmarks and only run benchmarks that execute changed code. Furthermore, we do not execute benchmarks for CVEs if breakpoints were only hit less than 100 times, as the changes are not often executed. A low number of breakpoint hits can be the result of the changed code executed once during setup, which will only marginally impact the benchmark. By removing all CVEs that do not affect code executed by our benchmarks, we reduce the 1 545 testable CVEs to 108 CVEs. This number can be further increased through more benchmarks and testing on different architectures and hardware configurations. The prefiltering is automated and does not require manual intervention.

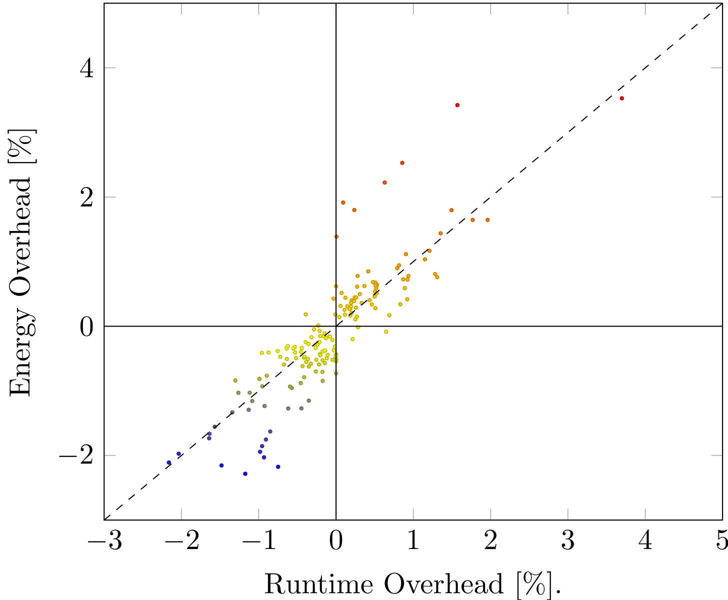


Figure 10.3.: Energy overhead and runtime overhead scatter plot of benchmark runs from 108 Linux kernel CVE fixes.

4.2. High-Level Analysis

Figure 10.3 provides an overview of the benchmark results for all 108 tested CVEs. Each dot in the scatter plot corresponds to the result of a single benchmark run. Each benchmark was executed 92 times, and the results are averaged. Black lines separate the four quadrants, and the expected runtime-to-energy correlation of 1:1 is shown as a dotted line. Of the 108 CVEs, 46 have no benchmark runs where either the runtime or energy change show statistical significance according to the Mann-Whitney-U-Test ($p \geq 5\%$). This means that 62 out of the 108 measured CVE fixes have a measurable impact on our benchmark suite's runtime or energy consumption. 42.2 % of benchmark runs are in the first quadrant, meaning positive runtime and energy overhead. 53.0 % of benchmark runs are in the third quadrant, meaning negative runtime and energy overhead. Results in the first and third quadrants follow the assumption that energy overhead can be roughly estimated by runtime overhead. 2.4 % of benchmark runs fall into the fourth quadrant, meaning positive runtime overhead and negative energy overhead. For these CVEs, the corresponding benchmarks took longer to execute but did so in an overall

more energy-efficient way than without the patch. 2.4 % of benchmark runs fall into the second quadrant, meaning negative runtime overhead and positive energy overhead. Therefore, energy and runtime are largely correlated. Contrary to what is expected, there is a large number of patches that improve energy consumption and runtime. This can be the result of code simplification but also more complex changes, such as fewer branch predictions and more energy-efficient stalls instead of mispredictions.

47.6 % of benchmark runs with statistically significant results have either a runtime or energy overhead change that is statistically significant, while the corresponding other metric is statistically insignificant. This means that the patch affected either energy consumption or runtime overhead, but not both. Following the assumption that energy and runtime follow a 1:1 relationship, faster code should be more energy-efficient. While energy and runtime seem to roughly correlate (Figure 10.3), the correlation is far from perfect, with the measured overheads scattered around the expected correlation. For optimizations of fixes, typically, only runtime overhead is considered, while energy overhead is ignored. This one-sided optimization results in an energy overhead that can vary wildly from the runtime overhead. The Pearson correlation coefficient of energy and runtime overhead is 0.84 with a coefficient of determination (r^2) of 0.71, where 1 would be a perfect linear correlation. Our measurements show that runtime is a rough estimate for energy consumption, but it is not precise and can vary significantly.

For all CVEs that we further analyzed, the performance and energy changes can be explained by code changes or by changes in performance counter values. Given that some results significantly deviate from the expected 1:1 correlation between energy and runtime shown in Figure 10.3, it is crucial to not only measure runtime overhead but also energy overhead when testing CVE patches.

4.3. Case Studies

In this section, we discuss interesting results of our measurements. We discuss a selection of CVEs and corresponding benchmarks with a significant change in runtime or energy consumption according to the Mann-Whitney-U-Test ($p \geq 5\%$). For each case study, we first compare the code changes with the benchmark results. Second, we look at the change of performance counter values between the unpatched and patched kernels to gain further

insights. In particular, we track stalled cycles, dTLB-load and dTLB-store misses, iTLB-load and iTLB-store misses, LLC-load and LLC-store misses, L1-dcache loads and stores, branch loads, branch misses, instructions executed, mispredicted branches retired, uOPs issued, stalled cycles to recover from a misprediction, and the number of times the front end is resteered, mainly when the branch predictor cannot provide a correct prediction (BACLEARS). We repeated the measurements for the CVEs discussed in this section to collect the performance counter-values. We executed the benchmarks 65 times while tracking the performance counters. The newly collected results are consistent with our first measurements, where we collected results from all CVEs. This further proves the repeatability and consistency of our results. For all results discussed, we provide the average overhead in percent, the standard error of the mean $\sigma_{\bar{x}}$ and the sample size n. To improve readability, numbers that increase with the patch applied are additionally colored green and numbers that decrease red.

CVE-2017-1000112. This CVE belongs to an exploitable memory corruption in the UDP Fragmentation Offload (UFO) implementation in the Linux kernel [47]. The bug was fixed by adding checks to four if conditions in the UDP, IPv4, and IPv6 implementations. The fix results in a runtime overhead of 1.5 % ($\sigma_{\bar{x}} = 0.25\,\%$, $n = 65$) and an energy overhead of 1.4 % ($\sigma_{\bar{x}} = 0.042\,\%$, $n = 65$) for the Stress-NG `udp` benchmark. Despite this significant change in runtime and energy consumption, the performance counters indicate no increase in the number of instructions executed. The increase stems from an increase in stalled cycles by 2.4 % ($\sigma_{\bar{x}} = 0.7\,\%$, $n = 65$) and an increase of stalled cycles used to recover from an earlier branch misprediction of machine clear event by 12.8 % ($\sigma_{\bar{x}} = 3.1\,\%$, $n = 65$). Furthermore, we detected an increase of L1-dcache loads and stores by 2.3 % ($\sigma_{\bar{x}} = 1.23\,\%$, $n = 65$) and 3.0 % ($\sigma_{\bar{x}} = 1.2\,\%$, $n = 65$) respectively. The `udp` benchmark is the only benchmark for this CVE with a statistically significant runtime or energy overhead, perfectly matching the fix that changes the UDP implementation.

CVE-2020-29534. This CVE belongs to a bug where `io_uring` takes a non-recounted reference to the file struct that submitted a request [47]. For Stress-NG `pipe` the runtime overhead decreases by 0.6 % ($\sigma_{\bar{x}} = 0.46\,\%$, $n = 65$) and the energy overhead by 0.9 % ($\sigma_{\bar{x}} = 0.10\,\%$, $n = 65$). The dTLB store misses decrease by 4.6 % ($\sigma_{\bar{x}} = 0.6\,\%$, $n = 65$), the branch misses by 0.92 % ($\sigma_{\bar{x}} = 0.68\,\%$, $n = 65$), and the number of cycles stalled due to recovery from a branch miss by 2.1 % ($\sigma_{\bar{x}} = 1.36\,\%$, $n = 65$).

Contrary to that, the BACLEARS increase by 3.6 % ($\sigma_{\bar{x}} = 1.54\,\%$, $n = 65$) and the LLC loads increase by 3.9 % ($\sigma_{\bar{x}} = 1.50\,\%$, $n = 65$). From these results, we assume that the decrease in branch misses and the cycles used to recover from branch misses are the reasons for the improvements.

For Stress-NG `sock` the runtime overhead decreases by 0.39 % ($\sigma_{\bar{x}} = 0.08\,\%$, $n = 65$) and the energy overhead decreases by 0.43 % ($\sigma_{\bar{x}} = 0.02\,\%$, $n = 65$). The branch misses decrease by 1.5 % ($\sigma_{\bar{x}} = 0.50\,\%$, $n = 65$), the L1-dcache loads decrease by 3.0 % ($\sigma_{\bar{x}} = 1.31\,\%$, $n = 65$). Similarly to the pipe benchmark, the improvements for the sock benchmark are most likely due to fewer branch misses.

Insight 1. By optimizing for branch prediction, patches can, even with more complex code, significantly decrease energy and runtime overhead.

For Stress-NG `udp`, runtime overhead decreases by 1.3 % ($\sigma_{\bar{x}} = 0.19\,\%$, $n = 65$) and energy overhead by 1.3 % ($\sigma_{\bar{x}} = 0.04\,\%$, $n = 65$). The dTLB store misses decrease by 7.5 % ($\sigma_{\bar{x}} = 0.63\,\%$, $n = 65$), L1-dcache stores by 1.5 % ($\sigma_{\bar{x}} = 0.47\,\%$, $n = 65$), LLC loads by 6.7 % ($\sigma_{\bar{x}} = 0.72\,\%$, $n = 65$), instructions executed by 1.2 % ($\sigma_{\bar{x}} = 0.45\,\%$, $n = 65$), and BACLEARS by 6.0 % ($\sigma_{\bar{x}} = 0.70\,\%$, $n = 65$). However, cycles stalled due to misprediction increase by 21.4 % ($\sigma_{\bar{x}} = 4.2\,\%$, $n = 65$). Energy consumption and runtime most likely improved due to fewer instructions executed and lower cache pressure.

CVE-2020-12114. This CVE is a race condition in `fs/namespace.c` allowing users to cause a denial of service [47]. The fix results in a runtime overhead of 1.9 % ($\sigma_{\bar{x}} = 0.35\,\%$, $n = 65$) and an energy overhead of 2.0 % ($\sigma_{\bar{x}} = 0.044\,\%$, $n = 65$) for Stress-NG `pipe`. The instructions executed increase by 2.1 % ($\sigma_{\bar{x}} = 0.5\,\%$, $n = 65$), uOPs issued by 3.0 % ($\sigma_{\bar{x}} = 1.03\,\%$, $n = 65$), dTLB load misses by 2.4 % ($\sigma_{\bar{x}} = 0.50\,\%$, $n = 65$), dTLB store misses by 3.3 % ($\sigma_{\bar{x}} = 0.83\,\%$, $n = 65$), and L1-dcache stores by 1.4 % ($\sigma_{\bar{x}} = 0.64\,\%$, $n = 65$). The stalled cycles from misprediction decreased by 5.9 % ($\sigma_{\bar{x}} = 1.50\,\%$, $n = 65$), while the overall stalled cycles stalled increased by 2.6 % ($\sigma_{\bar{x}} = 1.010\,\%$, $n = 65$). Therefore, the runtime and energy overhead changes are most likely due to more code being executed.

For Stress-NG `udp`, the runtime increases by 0.75 % ($\sigma_{\bar{x}} = 0.19\,\%$, $n = 65$) and the energy consumption by 0.87 % ($\sigma_{\bar{x}} = 0.02\,\%$, $n = 65$). The amount of stall cycles increased by 2.6 % ($\sigma_{\bar{x}} = 1.09\,\%$, $n = 65$) while the instructions executed only slightly increased by 0.68 % ($\sigma_{\bar{x}} = 0.48\,\%$,

$n = 65$). Similar to the `pipe` benchmark, the stalled cycles due to recovery from a misprediction decreased by 14.6 % ($\sigma_{\bar{x}} = 1.51\,\%$, $n = 65$). We did not observe an increase in TLB or LLC misses. We did observe an increase in BACLEARS by 1.4 % ($\sigma_{\bar{x}} = 0.70\,\%$, $n = 65$), which, together with other factors that we did not track, likely resulted in the overall increase in stalled cycles. Therefore, the change in runtime and energy overhead is likely due to the overall increase in stalls and the reason behind them.

For the `network-loopback` benchmark, the runtime increased by 0.25 % ($\sigma_{\bar{x}} = 0.06\,\%$, $n = 65$) with no change in energy consumption. This is contrary to benchmark runs form other CVEs, where runtime and energy overhead change at a similar rate. From all tracked performance counters, only stalled cycles due to misprediction recovery increase by 3.2 % ($\sigma_{\bar{x}} = 1.70\,\%$, $n = 65$). The change in runtime is likely due to very subtle changes in performance counter values, which would require a higher sample size to detect, or due to reasons that we do not track. The difference between energy overhead and runtime overhead could stem from a more efficient way of stalling than stalling from misprediction recovery.

For all affected benchmarks, the changed code is heavily executed. The `network-loopback` benchmark, in particular, executes changed code lines 159 040 times in its execution.

CVE-2018-1108. This CVE belongs to a weakness in the generation of random seed data. The weakness allows programs to use the random seed before it was sufficiently generated [47]. The fix results in a runtime overhead of 0.72 % ($\sigma_{\bar{x}} = 0.07\,\%$, $n = 65$) and an energy overhead of 1.7 % ($\sigma_{\bar{x}} = 0.24\,\%$, $n = 65$) for `network-loopback`. While this fix is intended for early boot-time random number generation by updating four if-conditions in the random number generator code, it does affect programs even after that. Some of the changed conditions are frequently executed during random number generation. The `network-loopback` benchmark, in particular, executed the changed code over 2 600 times per run. Due to this, Branch misses increased by 4.6 % ($\sigma_{\bar{x}} = 1.49\,\%$, $n = 65$) and dTLB load and store misses by 2.1 % ($\sigma_{\bar{x}} = 0.69\,\%$, $n = 65$) and 1.5 % ($\sigma_{\bar{x}} = 0.7\,\%$, $n = 65$) respectively. This fix results in an energy overhead that is significantly higher than the runtime overhead for this benchmark. The result of the overhead appears to stem from an increase in mispredictions due to changed if-conditions.

> **Insight 2.** Changes that should only be executed rarely, e.g., at boot time, can have a significant impact on regular runtime through misprediction of added branches that are rarely taken.

CVE-2015-8839. This CVE belongs to multiple race conditions in the ext4 implementation [47]. For the `pipe` Stress-NG benchmark the runtime increases by 1.9 % ($\sigma_{\bar{x}} = 0.19\,\%$, $n = 65$) and the energy consumption by 1.5 % ($\sigma_{\bar{x}} = 0.03\,\%$, $n = 65$). For this benchmark the BACLEARS increase by 7.2 % ($\sigma_{\bar{x}} = 2.86\,\%$, $n = 65$), branch misses by 2.6 % ($\sigma_{\bar{x}} = 0.59\,\%$, $n = 65$), LLC loads by 8.4 % ($\sigma_{\bar{x}} = 2.7\,\%$, $n = 65$), L1-dcache stores by 2.9 % ($\sigma_{\bar{x}} = 1.74\,\%$, $n = 65$), and instructions executed by 1.4 % ($\sigma_{\bar{x}} = 0.54\,\%$, $n = 65$).

For Stress-NG `aio`, runtime increases by 2.9 % ($\sigma_{\bar{x}} = 0.58\,\%$, $n = 65$) and energy consumption by 2.4 % ($\sigma_{\bar{x}} = 0.13\,\%$, $n = 65$). For this benchmark, BACLEARS increase by 5.0 % ($\sigma_{\bar{x}} = 1.80\,\%$, $n = 65$), dTLB store misses by 21.4 % ($\sigma_{\bar{x}} = 4.23\,\%$, $n = 65$), dTLB load misses by 15.8 % ($\sigma_{\bar{x}} = 5.20\,\%$, $n = 65$), cycles stalled by 5.8 % ($\sigma_{\bar{x}} = 1.75\,\%$, $n = 65$), and instructions executed by 2.0 % ($\sigma_{\bar{x}} = 1.52\,\%$, $n = 65$).

For Stress-NG `udp` the runtime increases by 3.2 % ($\sigma_{\bar{x}} = 0.26\,\%$, $n = 65$) and the energy consumption by 2.7 % ($\sigma_{\bar{x}} = 0.15\,\%$, $n = 65$). For this benchmark BACLEARS increase by 2.0 % ($\sigma_{\bar{x}} = 0.64\,\%$, $n = 65$), branch misses by 4.1 % ($\sigma_{\bar{x}} = 0.95\,\%$, $n = 65$), LLC loads by 1.7 % ($\sigma_{\bar{x}} = 0.64\,\%$, $n = 65$), LLC stores by 6.9 % ($\sigma_{\bar{x}} = 1.37\,\%$, $n = 65$), L1-dcache stores by 4.5 % ($\sigma_{\bar{x}} = 0.67\,\%$, $n = 65$), and instructions executed by 3.0 % ($\sigma_{\bar{x}} = 0.91\,\%$, $n = 65$). Furthermore, branch misses increase by 3.4 % ($\sigma_{\bar{x}} = 1.14\,\%$, $n = 65$).

For the `network-loopback` benchmark the runtime increases by 2.4 % ($\sigma_{\bar{x}} = 0.15\,\%$, $n = 65$) and the energy consumption by 2.4 % ($\sigma_{\bar{x}} = 0.15\,\%$, $n = 65$). For this benchmark BACLEARS increased by 7.2 % ($\sigma_{\bar{x}} = 2.86\,\%$, $n = 65$), and LLC loads by 3.4 % ($\sigma_{\bar{x}} = 2.7\,\%$, $n = 65$). During a run of the `network-loopback` benchmark, the VM executes over 700 000 times code modified by the CVE fix.

As our VM uses ext4 as its filesystem, a change in the ext4 implementation affects a large number of benchmarks. These results show that minor changes in code frequently executed by most applications can significantly impact runtime and energy consumption.

Insight 3. Unoptimized branches in CVE patches in code important for regular OS operation can have drastic negative impacts on a wide range of applications.

CVE-2016-5696. This CVE belongs to a bug in the IPv4 TCP stack that results in improper ACK segment rate determination, simplifying TCP hijacking through a blind in-window attack [47]. The fix decreases the runtime and energy consumption of Stress-NG `sock` by 0.9 % ($\sigma_{\bar{x}} = 0.16\,\%$, $n = 65$) and 1.0 % ($\sigma_{\bar{x}} = 0.046\,\%$, $n = 65$) respectively. Furthermore, retired branches decrease by 4.8 % ($\sigma_{\bar{x}} = 1.78\,\%$, $n = 65$), and stall cycles to recover from earlier mispredictions by 16.0 % ($\sigma_{\bar{x}} = 2.53\,\%$, $n = 65$). This benchmark is affected by this fix, as it tests socket performance by setting up a server and client that transmit packages. The fix appears to decrease the mispredictions when executing Stress-NG `sock`, decreasing performance and energy consumption.

CVE-2017-7495. This CVE belongs to a bug in `fs/ext4/inode.c` of the ext4 implementation. The implementation mishandles a needs-flushing-before-commit list with data=ordered mode, allowing users to obtain sensitive information from other users' files [47]. For the `icache` benchmark runtime decreases by 2.5 % ($\sigma_{\bar{x}} = 0.25\,\%$, $n = 65$) and energy overhead by 2.4 % ($\sigma_{\bar{x}} = 0.03\,\%$, $n = 65$). The reason for these decreases is a decrease in instructions executed by 2.8 % ($\sigma_{\bar{x}} = 0.34\,\%$, $n = 65$).

For the `network-loopback` benchmark the runtime decreases by 0.5 % ($\sigma_{\bar{x}} = 0.05\,\%$, $n = 65$) and the energy overhead by 1.1 % ($\sigma_{\bar{x}} = 0.03\,\%$, $n = 65$). BACLEARS decrease by 2.7 % ($\sigma_{\bar{x}} = 0.93\,\%$, $n = 65$), branch misses by 3.2 % ($\sigma_{\bar{x}} = 1.49\,\%$, $n = 65$), and L1-dcache stores by 3.5 % ($\sigma_{\bar{x}} = 1.76\,\%$, $n = 65$). The lower runtime and energy overhead likely stems from fewer branch misses, resulting in efficient code execution.

5. Analysis of Mitigations

In this section, we discuss our results of all available Linux mitigations changeable by command line options. We run the same benchmarks as in Section 4. A wide variety of mitigations have been introduced over the past years. We benchmark all mitigation options available in Linux kernel 6.2 on an x86 system separately and evaluate the runtime and energy consumption changes introduced by them. We list the available

Table 10.2.: Tested mitigation options on Linux Kernel 6.2.

Mitigation	Options
spectre_v2	off; retpoline; retpoline,generic; retpoline,lfence; eibrs; eibrs,retpoline; eibrs,lfence; ibrs; on
spectre_v1	nospectre_v1; spectre_v1
pti	off; on
spec_store_bypass_disable	off; on
l1tf	off; flush; full
mds	off; full,nosmt; full
tsx_async_abort	off; full
retbleed	off; unret; ibpb
mmio_stale_data	off; full
l1d_flush	on
kvm.nx_huge_pages	off; force

mitigation options and their parameters in Table 10.2. For each mitigation, we benchmark all available options and compare them with the deactivated mitigation. While most mitigations can only be turned on or off, some can be configured to include a specific mitigation strategy. One of the most configurable mitigations are the Spectre V2 mitigations, with 8 different specific mitigation options.

5.1. Overview

In this section, we present the results of our measurements using 54 benchmark runs, with all not tested mitigation disabled, which are provided in Table 10.3 (runtime) and Table 10.4 (energy). All numbers are rounded to one decimal place if they are <10 and >−10 and to the next integer otherwise. There are multiple runs with overheads listed as 0, which result from very small changes in runtime or energy consumption (<0.1 %). Measurements with no statistically significant overhead according to the Mann-Whitney-U-Test ($p \geq 5\%$) are represented by $*$. `l1tf`, `l1d_flush`, and `kvm.nx_huge_pages` show (almost) no statistically significant change in runtime or energy consumption. This is as expected, as these mitigations target either SGX or the execution of VMs. While our tested kernel runs inside of a VM, it itself does not manage any VMs in any of our benchmarks.

The only exceptions for runtime are the `pthread` (p-value: 3.4 %) and `pipe` (p-value: 1.43 %) benchmarks for `l1tf=flush` and the `cpu` benchmark (p-value: 1.45 %) for `l1d_flush`. The only exceptions for energy overhead are the `pthread` (p-value: 3.76 %), `pipe` (p-value: 0.46 %), and `mutex` (p-value: 1.45 %) benchmark for `l1tf=flush` as well as the `pipe` (p-value: 1.84 %), and `mutex` (p-value: 4.86 %) benchmark for `l1tf=full`. We reran these benchmarks and could not find any statistical significance in the newly collected data. Therefore, we conclude that these were false positives. A small number of false positives are expected due to the high number of benchmarks executed and the low sample size.

Overall, energy consumption and runtime correlate well with a few exceptions discussed later, as shown in Figure 10.4 with a zoomed-in view in Figure 10.5. Each point corresponds to one benchmark execution where at least the energy overhead or runtime overhead is statistically significant. We highlight the cardinal axes using solid lines and the expected 1:1 correlation with a dashed line. When linearly approximating energy overhead through runtime, we computed the polynomial $e = 0.84r + 1.14$, where e is the energy overhead and r the runtime overhead, with an $r^2 = 0.992$ (1 would be a perfect representation), indicating that this is a very accurate approximation. The polynomial indicates that energy overhead is typically lower than runtime overhead, as shown by the 0.84 coefficient. Calculating the r^2 score for the samples with a small runtime and energy overhead in the range of $[-10, 10]$ shown in Figure 10.5, still containing roughly half of the measurements, results in $r^2 = 0.52$. While this linear model works well for large overheads, it does not provide accurate results for smaller overheads.

Table 10.3.: Runtime overhead of all measured software mitigations in percent. Values are rounded to one decimal point for changes <10 % and to the next integer for larger changes. Overheads that are not statistically significant are replaced by a ∗-symbol.

Benchmark \ Mitigation	spectre_v2=retpoline	spectre_v2=retpoline,generic	spectre_v2=retpoline,lfence	spectre_v2=eibrs	spectre_v2=eibrs,retpoline	spectre_v2=eibrs,lfence	spectre_v2=ibrs	spectre_v2=on	spec_store_bypass_disable=on	spectre_v1	pti=on	l1tf=flush	l1tf=full	mds=full	tsx_async_abort=full	retbleed=unret	retbleed=ibpb	mmio_stale_data=full	l1d_flush=on	kvm.nx_huge_pages=force
icache	4.3	4.3	*	3.9	4	4.3	71	4.7	*	*	17	*	*	16	16	6.4	238	16	*	*
fork	1.8	1.3	0.7	1.3	1.5	1.4	35	11	2.2	*	6.8	*	*	5	4.8	5.6	66	4.7	*	*
pthread	−7.4	*	*	*	*	*	36	*	−41	*	*	5.5	*	23	27	−9.9	379	25	*	*
context	3.6	3.6	4.3	3.6	3.6	3.6	322	3.6	0.7	*	87	*	*	96	96	5.6	972	96	*	*
pipe	21	21	13	21	21	21	81	55	1.5	*	19	0.5	*	19	20	10	261	19	*	*
io	8.8	8.8	7	10	8.6	10	40	8.6	13	*	6.7	*	*	7.3	6.4	7.6	75	9	*	*
sock	17	17	10	17	17	18	177	16	*	*	45	*	*	48	48	7.2	536	49	*	*
udp	26	26	14	26	26	26	109	313	1.4	*	15	*	*	18	18	15	347	17	*	*
futex	11	11	9.1	11	11	11	112	174	64	*	20	*	*	9.2	8.9	6.7	214	8.4	*	*
aio	17	18	8.4	17	17	18	354	18	1.1	*	84	*	*	94	94	19	1,147	94	*	*
switch	28	28	17	28	28	28	125	404	1.4	*	21	*	*	20	20	10	387	20	*	*
sctp	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	78	*	*	*
signal	4.1	4	3.2	4.2	4	3.9	233	4.1	1.3	*	57	*	*	62	62	7.6	734	62	*	*
cpu	*	0.1	*	0.1	*	*	1.7	0.1	6.7	*	*	*	*	0.1	*	*	3.2	*	−0.2	*
network-loopback	6.1	6	2.8	6.1	6	6	26	55	0.3	*	8.3	*	*	5.1	5	3.1	89	5	*	*
mutex	−0.1	−0.1	−0.1	−0.1	−0.1	0.1	0.5	−0.1	−0.1	*	0.2	*	*	0.2	−0.1	0.2	1.5	0.2	*	*
osbench files	3.8	3.8	2.3	3.8	3.8	3.9	4.2	3.8	0.4	*	1.5	*	*	1.6	1.7	3.7	12	1.5	*	*
osbench processes	0.1	0.1	0	0.1	0.1	0.1	1.3	0.2	0.1	*	0.3	*	*	0.3	0.3	0.2	4.2	0.3	*	*
osbench threads	0.1	0.1	0.1	0.1	0.1	0.1	1.3	0.2	0.1	*	0.3	*	*	0.3	0.3	0.2	4.2	0.3	*	*
osbench programs	0.1	0.1	0.1	0.2	0.1	0.1	1.4	0.2	0.1	*	0.3	*	*	0.3	0.3	0.2	4.2	0.3	*	*
osbench allocations	0.2	0.2	0.1	0.2	0.2	0.2	1.3	0.3	0.1	*	0.2	*	*	0.2	0.2	0.3	4.3	0.2	*	*
apache	0	0	*	0	0	0	0.1	0	*	*	0	*	*	0	0	0	0.4	0	*	*
pmbench	0	0	0	0	0	0	0.2	0	0	*	0	*	*	0	0.1	0	0.7	0	*	*

Table 10.4.: Energy overhead of all measured software mitigations in percent. Values are rounded to one decimal point for changes <10 % and to the next integer for larger changes. Overheads that are not statistically significant are replaced by a ∗-symbol.

Benchmark \ Mitigation	spectre_v2=retpoline	spectre_v2=retpoline,generic	spectre_v2=retpoline,lfence	spectre_v2=eibrs	spectre_v2=eibrs,retpoline	spectre_v2=eibrs,lfence	spectre_v2=ibrs	spectre_v2=on	spec_store_bypass_disable=on	spectre_v1	pti=on	l1tf=flush	l1tf=full	mds=full	tsx_async_abort=full	retbleed=unret	retbleed=ibpb	mmio_stale_data=full	l1d_flush=on	kvm.nx_huge_pages=force
icache	3.7	4.1	∗	3.1	3.7	3.9	64	4.5	∗	∗	15	∗	∗	18	17	5.6	211	17	∗	∗
fork	1.9	1.4	∗	0.7	1	1.1	30	10	2.1	∗	5.5	∗	∗	4.1	4.7	5.6	53	3.8	∗	∗
pthread	−8.8	∗	∗	∗	∗	∗	23	∗	−46	∗	∗	4.3	∗	21	25	−10	313	24	∗	∗
context	4.6	4.3	2.8	4.7	4.5	4.4	289	4.6	0.8	∗	79	∗	∗	117	118	8.2	841	118	∗	∗
pipe	19	19	11	19	20	19	69	42	1.7	∗	18	0.7	0.5	18	19	10	209	18	∗	∗
io	6.7	7.1	4.6	8.2	6.9	8.3	24	7.1	6.8	∗	3.6	∗	∗	6.2	2.9	6.4	41	5.1	∗	∗
sock	14	15	8	16	16	15	151	13	1.3	∗	40	∗	∗	44	43	6.3	425	43	∗	∗
udp	25	25	12	25	24	25	98	269	1.6	∗	13	∗	∗	17	17	14	282	16	∗	∗
futex	10	10	7.9	10	11	10	87	149	61	∗	12	∗	∗	15	14	7.3	169	14	∗	∗
aio	16	17	7.1	16	16	16	308	16	1	∗	79	∗	∗	92	91	19	959	92	∗	∗
switch	27	26	16	26	26	26	108	332	1.3	∗	20	∗	∗	19	20	10	316	20	∗	∗
sctp	∗	∗	∗	∗	∗	∗	0.4	∗	∗	∗	∗	∗	∗	∗	∗	∗	73	∗	∗	∗
signal	4.4	3.9	2.6	3.7	3.7	3.8	192	4.5	1.8	∗	49	∗	∗	61	61	8	600	59	∗	∗
cpu	∗	∗	∗	∗	∗	∗	2.1	0.2	5.6	∗	0.4	∗	∗	∗	∗	∗	4	0.2	∗	∗
network-loopback	13	14	6.2	13	12	12	53	109	0.9	∗	16	∗	∗	11	10	5.9	166	10	∗	∗
mutex	∗	∗	∗	∗	∗	∗	∗	∗	1.5	∗	∗	0.3	−0.1	∗	0.2	∗	1.3	∗	∗	∗
osbench files	6.1	5.4	3.1	5.6	5.6	5.7	5.5	6.1	∗	∗	1.6	∗	∗	2.2	2.4	6.3	14	3.2	∗	∗
osbench processes	0.8	0.3	∗	0.3	0.4	0.3	0.9	∗	∗	∗	∗	∗	∗	∗	0.2	0.7	2	∗	∗	∗
osbench threads	∗	∗	−0.8	∗	∗	∗	−0.4	∗	1.1	∗	−0.5	∗	∗	∗	∗	1.3	−0.9	∗	∗	∗
osbench programs	0.4	∗	∗	0.2	∗	∗	0.9	∗	0	∗	∗	∗	∗	0.7	0	0.2	1.3	0.1	∗	∗
osbench allocations	0.2	∗	∗	0.4	0.3	0.2	5.7	0.6	4.9	∗	1.7	∗	∗	2.3	1	0.5	13	2.2	∗	∗
apache	0.5	0.4	0.7	0.4	0.6	0.5	−2	−0.6	∗	∗	∗	∗	∗	0.8	1.1	0.4	−7.1	1.1	∗	∗
pmbench	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	∗	0.6	∗	∗	∗

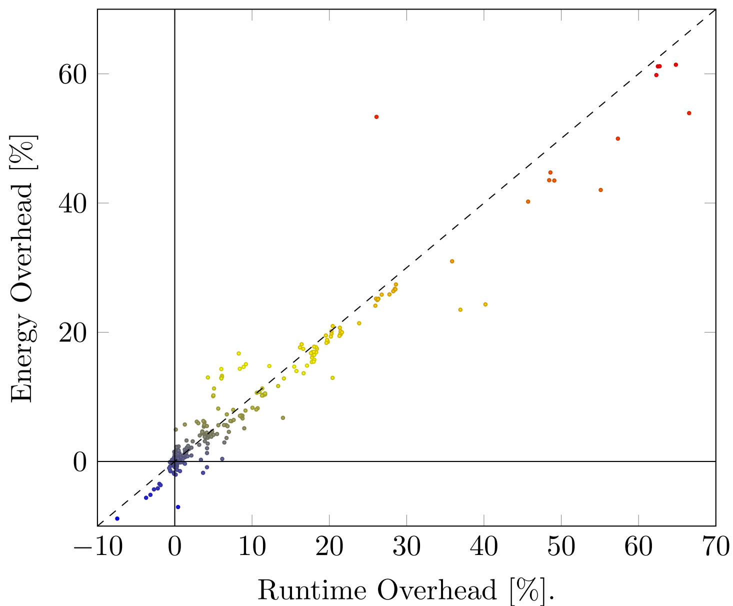


Figure 10.4.: Linux kernel hardware mitigation energy and runtime overhead.

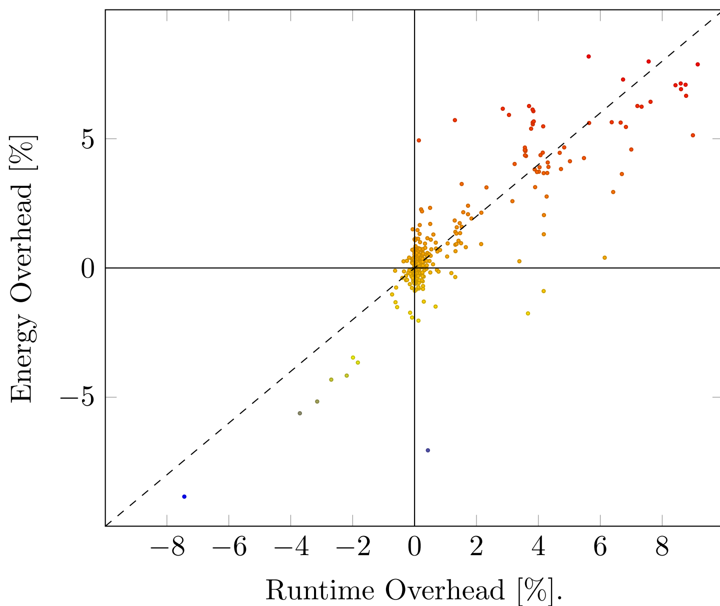


Figure 10.5.: Linux kernel hardware mitigation energy and runtime overhead zoomed-in with an x-range of $[-10, 10]$ and a y-range of $[-10, 10]$.

5.2. High-Level Analysis

While mitigations have a significant impact on the runtime and energy consumption of micro benchmarks, the impact on macro benchmarks is limited, as shown in Table 10.3 (runtime) and Table 10.4 (energy). For macro benchmarks, Phoronix `network-loopback` performs the worst, as the network stack is handled by the kernel, followed by Phoronix `osbench files`, which continuously creates files. Some benchmarks are only minimally affected by the mitigations, such as Stress-NG `cpu` and Phoronix `pmbench`. As the tested mitigations only change kernel behavior, minimizing kernel time through fewer syscalls minimizes the overhead. As the `cpu` and `pmbench` benchmarks focus on computations in user space and memory accesses, respectively, they are only minimally affected. Despite this, they are still affected by forced switches to kernel mode, e.g., through interrupts. The Spectre V1 mitigation shows that security does not have to come at a cost, as it is optimized enough that we did not detect any significant overhead.

Similar to CVE patches (Section 4), energy and runtime are largely correlated, especially for larger overheads as shown in Figure 10.4. Despite this, for smaller overheads, the correlation is significantly weaker (Figure 10.5), and in some cases, the two metrics are completely different. Due to this deviation, similar to CVE patches, it is crucial to not only measure runtime but also energy overhead when testing mitigations.

Our measurements also show an untapped way of optimizing energy consumption. Two examples of this are shown in the Phoronix `apache` benchmark for the Spectre V2 `ibrs` and the Retbleed `ibpb` mitigations. While the runtime only marginally increases with these mitigations, the energy consumed drastically decreases. This is the result of more cycles stalled instead of misspeculation. There seem to be code parts in the kernel where stalling saves a significant amount of energy. As energy consumption and runtime deviate in this case, forcing stalls instead of predictions could optimize energy consumption without impacting runtime.

5.3. Case Studies

In this section, we discuss interesting results for specific mitigations. We rerun the benchmarks for these mitigations while tracking performance counters. We track stalled cycles, dTLB-load and dTLB-store misses,

iTLB-load and iTLB-store misses, LLC-load and LLC-store misses, L1-dcache loads and stores, branch loads and misses, instructions executed, mispredicted branches retired, uOPs issued, and stalled cycles due to misprediction recovery. We use these performance counters to determine possible reasons for our results. For all results discussed, we provide the average overhead in percent, the standard error of the mean $\sigma_{\bar{x}}$ and the sample size n. To improve readability, numbers that increase with the mitigation active are colored green and numbers that decrease red.

Retbleed IBPB. The `retbleed` indirect branch prediction barrier (IBPB) mitigation results in the most significant performance hit. This mitigation introduces a barrier that prevents code executed before it from affecting future branches [29]. We observe overheads of 1 147 % ($\sigma_{\bar{x}} = 2.46\,\%$, $n = 54$) (runtime) and 959 % ($\sigma_{\bar{x}} = 0.62\,\%$, $n = 54$) (energy) in the case of `aio`, and 972 % ($\sigma_{\bar{x}} = 0.20\,\%$, $n = 54$) (runtime) and 841 % ($\sigma_{\bar{x}} = 0.15\,\%$, $n = 54$) (energy) in case of `context`. This is due to IBPB introducing barriers that reset the branch predictor, resulting in more stalls at conditional branches after the barrier [29].

For all benchmarks, the IBPB mitigation increases runtime. The energy overhead is almost fully in line with the runtime, with two exceptions. For `apache`, the energy consumption decreases by 7.1 % ($\sigma_{\bar{x}} = 0.03\,\%$, $n = 54$), despite a runtime increase. This means that with IBPB enabled, the `apache` benchmark consumes less energy than with the mitigation disabled despite running slightly longer. Branch misses decrease by 6.8 % ($\sigma_{\bar{x}} = 1.45\,\%$, $n = 90$) and cycles stalled increase by 4.2 % ($\sigma_{\bar{x}} = 0.59\,\%$, $n = 90$). Due to IBPB, there are fewer mispredictions in the `apache` benchmark, counteracted by more stalls. This leads to a slight runtime increase. As stalls are more efficient than mispredictions and due to almost no runtime change, the energy consumption decreases.

> **Insight 4.** Stalling instead of speculating difficult to predict branches can lead to significant energy savings at almost no performance cost.

Speculative Store Bypass Disable (SSBD) & Retbleed Unret. Speculative store bypass allows the CPU to speculatively execute a load after a store with potentially overlapping addresses if the CPU predicts that they do not overlap. SSBD forces the CPU to wait until the addresses of all previous stores are known before executing any following loads [29]. Executing the `pthread` benchmark with SSBD enabled decreases runtime by 41 % ($\sigma_{\bar{x}} = 1.97\,\%$, $n = 54$) and the energy consumption by

46 % ($\sigma_{\bar{x}} = 0.99$ %, $n = 54$). The instructions executed decrease by 63 % ($\sigma_{\bar{x}} = 1.08$ %, $n = 90$), while the amount of stalled cycles increase by 46 % ($\sigma_{\bar{x}} = 0.82$ %, $n = 90$). Furthermore, L1-dcache stores and loads, branch loads, branch misses, and mispredicted retired branches decrease by 60 %. The decrease in runtime and energy overhead seem to stem from fewer instructions executed and the decrease in mispredictions and cache misses when spawning a large number of threads. The benchmark seems to regularly trigger speculative store bypass, leading to mispredictions. Enabling SSBD, therefore, reduces mispredictions and improves performance for this benchmark. The `retbleed unret` mitigation behaves similar to SSBD for Stress-NG `pthread` with an energy and runtime decrease of 10 % ($\sigma_{\bar{x}} = 0.85$ %, $n = 54$) while instructions executed, and retired mispredicted branches decreased by 11 %.

Spectre V1. Contrary to all other mitigations that target code tested, we did not observe any overheads for the `spectre_v1` mitigation. To rule out a problem with our setup, we validated that the mitigation is correctly applied by debugging the kernel. While unexpected, this result is not unreasonable as the `spectre_v1` mitigations only consist of `lfence` and `swapgs` barriers for selected user-copy functions, as well as explicit pointer sanitation, on a case-by-case basis. These defenses are lightweight and might not be encountered a significant number of times by our benchmarks.

> **Insight 5.** Some mitigations can be lightweight enough to not result in a statistically significant overhead, making it reasonable to always enabled them.

Spectre V2 IBRS. Indirect branch restricted speculation (IBRS) introduces a bit in the `IA32_SPEC_CTRL` MSR. When this bit is set after a switch to a higher privilege level, e.g., user mode to kernel mode, branches executed in the higher privilege mode can not be controlled by software executed in the lower privilege mode. IBRS is extremely expensive, which is why some CPUs received a more optimized version called enhanced IBRS (eIBRS) [29]. Contrary to the previously discussed IBPB `retbleed` mitigation, IBRS takes effect only when switching to a higher privilege level. The IBRS mitigation increases the runtime of `apache` by 0.14 % ($\sigma_{\bar{x}} = 0.01$ %, $n = 54$) and decreases energy consumption by 2.2 % ($\sigma_{\bar{x}} = 0.034$ %, $n = 54$). The cycles stalled increase by 28.4 % ($\sigma_{\bar{x}} = 0.47$ %, $n = 166$), presumably due to an increase in branch misses by 7.4 % ($\sigma_{\bar{x}} = 0.62$ %, $n = 166$). Despite this and the low runtime change, instructions executed

increased by 11.7 % ($\sigma_{\bar{x}} = 0.82\,\%$, $n = 166$). LLC-store misses decreased by 15.3 % ($\sigma_{\bar{x}} = 3.93\,\%$, $n = 166$), L1-dcache loads by 8.0 % ($\sigma_{\bar{x}} = 1.37\,\%$, $n = 166$), and L1-dcache stores by 3.6 % ($\sigma_{\bar{x}} = 1.40\,\%$, $n = 166$). The increase in branch misses and instructions executed seems to be counter-acted by the drastic decrease in cache misses, resulting in only a minor runtime change. We conclude from these results that the almost unchanged runtime, combined with the increase in stalls, resulted in this decrease in energy consumption.

PTI. Page table isolation (PTI) unmaps the whole kernel except for trampoline code and data structures, which always have to be accessible while in user space. While this mitigation was initially proposed to protect against a wide range of side-channel attacks that break kernel address space layout randomization (KASLR) [20], it was introduced into the kernel as a software mitigation against the Meltdown attack [44, 19]. PTI has the largest runtime and energy overheads with 87 % ($\sigma_{\bar{x}} = 0.06\,\%$, $n = 54$) and 79 % ($\sigma_{\bar{x}} = 1.06\,\%$, $n = 54$) respectively when running the `context` Stress-NG benchmark. The `context` benchmark continuously performs user-level context switches. While the user can manage the contexts and trigger context switches to them, each context switch triggers a syscall and, therefore, a switch to kernel mode. This results in a high number of switches between kernel and user mode, which is the worst case scenario for PTI. The stalled cycles increase by 91.8 % ($\sigma_{\bar{x}} = 14.71\,\%$, $n = 43$), the dTLB-store misses by 67.2 % ($\sigma_{\bar{x}} = 8.07\,\%$, $n = 43$), the iTLB-load misses by 35.3 % ($\sigma_{\bar{x}} = 6.90\,\%$, $n = 43$), the branch misses by 102.7 % ($\sigma_{\bar{x}} = 3.04\,\%$, $n = 43$), uOPs issued by 431.2 % ($\sigma_{\bar{x}} = 47.8\,\%$, $n = 43$). The increased pressure on the TLB is the result of the regular unmapping and remapping of the kernel memory. Benchmarks that do not induce a significant amount of privilege changes are only marginally affected by PTI. The `cpu` Stress-NG benchmark, in particular, which performs calculations purely in user space, has an energy consumption increase of 0.4 % ($\sigma_{\bar{x}} = 0.12\,\%$, $n = 54$) and no statistically significant runtime overhead. Contrary to the expected behavior, the OSBench `threads` benchmark from the Phoronix test suit has a runtime increase of 0.28 % ($\sigma_{\bar{x}} = 0.01\,\%$, $n = 54$) and an energy consumption decrease of 0.5 % ($\sigma_{\bar{x}} = 0.13\,\%$, $n = 54$). None of our tracked performance counters indicate why the energy consumption decreased for this benchmark, but the result is consistent even after multiple reruns.

MDS & TAA. Microarchitectural data sampling (MDS) and TSX async abort (TAA) mitigations behave almost identically. According to the

documentation, Linux uses the same mechanism to mitigate both MDS and TAA [32], which is also suggested by Intel in their advisory [27]. The slight differences in some of the measurements for the two can be attributed to noise and the low sample size.

6. Discussion of Limitations & Robustness

In this Section, we discuss the limitations to our work and the robustness of our results. While there are limitations to our work, we discuss why they do not affect our main insights.

Setup. CVEs affect specific systems in virtualized or native environments. In particular, we only focused on Linux kernels in a virtual machine. There may be vulnerabilities that affect only kernels in virtual machines or only kernels outside of virtual machines (home computer scenario), or other kernels than Linux. Future work has to determine whether the relation between performance and energy is similar for scenarios and kernels we did not study.

We measure the energy consumption with RAPL of the whole CPU package on a single system for comparability of the measurements. The measurements, thus, include the energy consumption of host programs and the hypervisor. The host and hypervisor are the same for all test runs and, hence, add a noisy baseline to the measurements but the absolute overheads stays the same. To compensate for the noise, we perform >60 measurements.

Patch Commit Choice. There is no generic way to find all CVE fix commits. We benchmark only the commit that marks the CVE as fixed. This strategy is sufficient for smaller and, therefore, most CVEs, as they are usually fixed by one commit. Larger CVEs might have multiple commits with other commits in between. For major vulnerabilities, such as Meltdown and Spectre, that were under embargo, the fix commit might only change the names of defines and does not directly fix the vulnerability. We cannot test these fixes using our automatic approach. To incorporate mitigations to major hardware vulnerabilities into our evaluation, we individually benchmark each mitigation that can be activated and deactivated through command line options on the recent Linux 6.2 kernel.

Prefiltering. The benchmark prefiltering (Section 4.1) is fundamental to efficiently test a wide range of CVEs on a wide range of benchmarks in

a short amount of time. It allows us to run benchmarks only on CVEs that use the affected code. Using breakpoints, while completely automatic, can lead to false positives and false negatives, e.g., if the code change is in a preprocessor macro. Furthermore, our prefiltering does not account for changes in the binary layout due to the fix. This may affect performance and energy consumption due to caches and other CPU internal buffers and optimizations. Accounting for such changes is infeasible as they can also occur from a compiler version or build system change and can be negated entirely by unrelated code changes.

Measurement Accuracy. Systematic energy analyses require accurate energy measurements from software, e.g., RAPL. However, in response to attacks via RAPL [43], Intel limited RAPL's accuracy in certain cases [28]. For this work, the mitigation is inactive, and unfiltered energy measurements are reported by RAPL.

Benchmark Choice. While we chose a broad set of benchmarks, covering a wide range of the Linux kernel's functionality, our results are inherently limited by our selection. This is an inherent issue with benchmarking. Therefore, it is not possible to determine the energy or runtime overhead of CVEs on kernel code parts that are not covered by our benchmarks, e.g., CVEs that affect other architectures, not used drivers, not tested kernel interfaces.

Hardware Choice & Measurement Interface. In this work, we execute all our measurements on an i7-6700K. Depending on factors such as microarchitecture and core frequency, these results can differ on other CPUs, limiting their general applicability. Despite this, our experiments can show a general trend between energy overhead and runtime overhead, as well as unexpected results such as energy and runtime overhead differing from each other. Furthermore, the framework described in this work can be easily applied to kernel versions running on a wide range of CPUs.

For our experiments, we use the Intel RAPL interface. As RAPL only measures CPU energy consumption, our determined overheads are CPU energy overheads and do not include other parts of the system, e.g., disk and network card. Therefore, additional overheads induced by a patch or mitigation outside of the CPU are not covered by our results. Most of these additional overheads would most likely be due to changes in runtime, as most CVE patches and mitigations target software or CPU hardware vulnerabilities and not devices. Additionally, while our experiments are executed on an Intel CPU relying on Intel RAPL, this interface can be

replaced by other available measurement interfaces, e.g., AMD RAPL [2] or even smart plugs that can cover the whole system energy consumption, making our framework applicable to a wide range of CPUs.

7. Discussion & Related Work

Our work highlights a fundamental problem in the security research community: Energy costs of security fixes and mitigations are systematically not measured and thus not understood. In the systems community, measuring energy costs has already become more common, as energy costs are highly relevant to assessing the value of newly proposed mechanisms (cf. Section 3.2). The two most closely related works are by Herzog et al. [24] and Siavvas et al. [60]. Herzog et al. [24] specifically focused on mitigations against Meltdown [44] and Spectre [37]. They made the surprising observation that for KPTI [19], some benchmarks show different overheads for energy than for performance, motivating our more comprehensive study of 1 616 Linux CVE fixes and all Linux security mitigations.

Performance and energy costs of mitigations reach far beyond the Linux kernel, and future work needs to investigate in both directions: On the level of applications and libraries, mitigations can have a substantial impact on performance and energy consumption, e.g., the Chrome site-isolation patch already had a reported overhead of 8.2 % on the CPU usage [57]. Similarly, on the hardware side, there are substantial costs attached to mitigations. Examples include the initial patches against Rowhammer that doubled the refresh rate. While the performance cost of the double refresh rate is reported as around 8 % [35, 40], it can be expected that the energy cost is higher as the performance is only affected when a DRAM access is deferred due to a refresh operation occurring exactly at the same time. However, the cost to charge and cells always applies. Another example of the hidden cost of security measures is Intel's mitigation to the Plundervolt attack on SGX [51] was to disable the corresponding DVFS interface via a microcode update, which was already only accessible with kernel privileges. As a consequence, system-specific optimization of voltage and frequency is largely impossible on updated machines and on more recent processors. Juffinger et al. [30] report efficiency gains of up to 20 % from voltage-frequency optimizations that are now made impossible due to the disabled DVFS interface. Thus, we see room for future work in many directions following up on our work.

While RAPL and similar interfaces can be used to profile a CPU's energy consumption for profiling [55, 34, 22], they are also actively used in a wide range of attacks. The first works exploring the security aspects of Intel RAPL focused on container co-location detection [16] and branch side-channel attacks [15]. Mantel et al. [49] demonstrated a side-channel attack distinguishing RSA keys. More recently, Lipp et al. [43] showed that RAPL-based energy measurements are precise enough to mount power analysis attacks [38, 6] purely from software. Surprisingly, their work showed that even the values of data operands can have a RAPL-measurable effect on the energy consumption. Subsequently, Wang et al. [63] showed that the energy-budget-induced throttling even creates remotely measurable timing differences. Liu et al. [45] demonstrated that the power side channel signal is also contained in the frequency due to the same throttling due to dynamic voltage and frequency scaling (DVFS). Kogler et al. [39] showed that specific workloads can amplify the leakage to speed up software-based power analysis attacks significantly. Qin et al. [56] and Yan et al. [64] used interfaces on mobile devices that report direct or indirect information on the power consumption (voltage, current, battery charge) and demonstrated website and application fingerprinting attacks. O'Flynn [52] exploited an onboard analog-to-digital converter to recover secrets processed in the secure world on a TrustZone-enabled device.

8. Conclusion

In this paper, we presented the first systematic analysis of the energy costs of CVE fixes and mitigations with the Linux kernel as a case study. We evaluated Linux kernel CVE fixes starting from Linux 4.0 over an 8-year time frame, covering 1 616 CVEs that we can automatically map to patch sets present in the source-code versioning repository. We automatically compiled Linux pre- and post-patch and benchmarked them using the Stress-NG and Phoronix benchmark suits. Overall, our work confirms benchmark-affecting code changes for 108 Linux CVE fixes, for which we collected energy and performance data. We also benchmarked all flag-controlled mitigations. While energy and performance cost is largely correlated, there are notable exceptions where energy and performance costs diverge significantly. It is important to note that this is **not** a cost-benefit analysis of CVE patches and whether they should be applied. Instead, our work underscores the need for future security research to evaluate both performance and energy cost.

Acknowledgements

This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 539710462 ("DOSS"), 502228341 ("Memento") and 465958100 ("NEON"), the Bundesministerium für Bildung and Forschung (BMBF, Federal Ministry of Education and Research) in Germany (project SUSTAINET-inNOvAte 16KIS2262), and the Austrian Science Fund (FWF project NeRAM 10.55776/I6054). Additional funding was provided by generous gifts from Red Hat, 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] Akinlolu Adekotujo, Adedoyin Odumabo, Ademola Adedokun, and Olukayode Aiyeniko. A Comparative Study of Operating Systems: Case of Windows, UNIX, Linux, Mac, Android and iOS. In: International Journal of Computer Applications 176.39 (2020), pp. 16–23 (p. 315).

[2] AMD uProf User Guide. 3.2. Advanced Micro Devices Inc. 2019 (pp. 316, 321, 341).

[3] Nadav Amit, Fred Jacobs, and Michael Wei. Jumpswitches: restoring the performance of indirect branches in the era of spectre. In: USENIX ATC. 2019 (p. 315).

[4] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In: USENIX Security. 2021, pp. 1073–1090 (p. 315).

[5] Lucy Bowen and Chris Lupo. The Performance Cost of Software-based Security Mitigations. In: International Conference on Performance Engineering. 2020, pp. 210–217. DOI: 10.1145/3358960.3379139 (p. 315).

[6] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In: CHES. 2004 (p. 342).

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security. 2019 (p. 312).

[8] Eugenio Capra, Chiara Francalanci, and Sandra A Slaughter. Measuring application software energy efficiency. In: IT Professional 14.2 (2012), pp. 54–61 (p. 312).

[9] Alexis Challande. Towards 1-day Vulnerability Detection using Semantic Patch Signatures. PhD thesis. nstitut Polytechnique de Paris, 2, 2022 (p. 315).

[10] Shaiful Alam Chowdhury and Abram Hindle. Greenoracle: Estimating software energy consumption with energy measurement corpora. In: Mining Software Repositories (MSR). IEEE. 2016 (p. 312).

[11] Jonathan Corbet. BPF: the universal in-kernel virtual machine. 2014. URL: `https://lwn.net/Articles/599755/` (p. 315).

[12] Jonathan Corbet. Many uses for Core scheduling. 2019. URL: `https://lwn.net/Articles/799454/` (p. 316).

[13] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. PIBE: Practical kernel control-flow hardening with profile-guided indirect branch elimination. In: Architectural Support for Programming Languages and Operating Systems. 2021, pp. 743–757. DOI: `10.1145/3445814.3446740` (p. 315).

[14] Christian Eichler, Jonas Röckl, Benedikt Jung, Ralph Schlenk, Tilo Müller, and Timo Hönig. Profiling with trust: system monitoring from trusted execution environments. In: Design Automation for Embedded Systems (2024). DOI: `10.1007/s10617-024-09283-1` (p. 315).

[15] Matteo Fusi. Information-Leakage Analysis Based on Hardware Performance Counters. MA thesis. Politecnico di Milano, 2017 (p. 342).

[16] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In: DSN. 2017 (p. 342).

[17] Alexander Gilgur, Brian Coutinho, Iyswarya Narayanan, and Parth Malani. Transitive Power Modeling for Improving Resource Efficiency in a Hyperscale Datacenter. In: Companion Proceedings of the Web Conference. 2021 (p. 312).

[18] 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. 316, 321).

[19] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (pp. 312, 338, 341).

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 312, 338).

[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 (p. 316).

[22] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. In: ACM SIGMETRICS Performance Evaluation Review 40 (2012), pp. 13–17 (p. 342).

[23] Ashif S Harji, Peter A Buhr, and Tim Brecht. Our troubles with Linux kernel upgrades and why you should care. In: ACM SIGOPS Operating Systems Review 47.2 (2013), pp. 66–72 (p. 315).

[24] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level. In: EuroSys. 2021 (pp. 312, 315, 316, 341).

[25] Joel Hruska. Intel Performance Hit 5x Harder Than AMD After Spectre, Meltdown Patches. 2019. URL: `https://www.extremetech.com/computing/291649-intel-performance-amd-spectre-meltdown-mds-patches` (p. 312).

[26] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: efficient defence against meltdown attack for unpatched VMs. In: USENIX ATC. 2018 (p. 315).

[27] Intel. Intel® Transactional Synchronization Extensions (Intel® TSX) Asynchronous Abort / CVE-2019-11135 / INTEL-SA-00270. 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/intel-tsx-asynchronous-abort.html (p. 339).

[28] Intel. Running Average Power Limit Energy Reporting. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html (p. 340).

[29] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (pp. 336, 337).

[30] Jonas Juffinger, Stepan Kalinin, Daniel Gruss, and Frank Mueller. SUIT: Secure Undervolting with Instruction Traps. In: ASPLOS. 2024 (pp. 312, 341).

[31] The Linux Kernel. Linux CVE-Announce Mailing List. 2024. URL: https://lore.kernel.org/linux-cve-announce/ (p. 315).

[32] The Linux Kernel. The kernel's command-line parameters. 2023. URL: https://www.kernel.org/doc/html/v6.2/admin-guide/kernel-parameters.html (p. 339).

[33] kernel.org. The kernel's command-line parameters. 2023. URL: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html (p. 318).

[34] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. In: ToMPECS 3 (2018), pp. 1–26 (pp. 316, 342).

[35] 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. 312, 341).

[36] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. In: ACM Transactions on Computer Systems 32.1 (2014), 2:1–2:70. DOI: 10.1145/2560537 (p. 315).

[37] 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. 341).

[38] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (p. 342).

[39] 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. 342).

[40] Changmin Lee, Wonjae Shin, Dae Jeong Kim, Yongjun Yu, Sung-Joon Kim, Taekyeong Ko, Deokho Seo, Jongmin Park, Kwanghee Lee, Seongho Choi, Namhyung Kim, Vishak G, Arun George, Vishwas V, Donghun Lee, Kangwoo Choi, Changbin Song, Dohan Kim, Insu Choi, Ilgyu Jung, Yong Ho Song, and Jinman Han. NVDIMM-C: A Byte-Addressable Non-Volatile Memory Module for Compatibility with Standard DDR Memory Interfaces. In: HPCA. 2020 (p. 341).

[41] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. In: Architectural Support for Programming Languages and Operating Systems. 2022 (p. 316).

[42] Frank Li and Vern Paxson. A Large-Scale Empirical Study of Security Patches. In: Conference on Computer and Communications Security. 2017, pp. 2201–2215. DOI: 10.1145/3133956.3134072 (p. 315).

[43] 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. 313, 340, 342).

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security. 2018 (pp. 312, 338, 341).

[45] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In: CCS. 2022 (p. 342).

[46] Michail Loukeris. Efficient computing in a safe environment. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019, pp. 1208–1210. DOI: 10.1145/3338906.3342491 (p. 315).

[47] Nicholas Luedtke. Linux Kernel CVEs. 2023. URL: https://www.linuxkernelcves.com (pp. 318, 321, 325–329).

[48] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. Efficient Scheduler Live Update for Linux Kernel with Modularization. In: Architectural Support for Programming Languages and Operating Systems. 2023, pp. 194–207. DOI: 10.1145/3582016.3582054 (p. 315).

[49] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. How Secure is Green IT? The Case of Software-Based Energy Side Channels. In: ESORICS. 2018 (p. 342).

[50] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A tool to model large caches. In: HP laboratories 27 (2009), p. 28 (p. 321).

[51] 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. 312, 341).

[52] Colin O'Flynn and Alex Dewar. On-Device Power Analysis Across Hardware Security Domains. In: CHES. 2019 (p. 342).

[53] Jacob Pan. RAPL (Running Average Power Limit) driver. 2013. URL: https://lwn.net/Articles/545745/ (p. 316).

[54] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In: ACM SLE. 2017 (p. 312).

[55] James Phung, Young Choon Lee, and Albert Y Zomaya. Modeling System-Level Power Consumption Profiles Using RAPL. In: NCA. IEEE. 2018 (p. 342).

[56] Yi Qin and Chuan Yue. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In: TrustCom/BigDataSE. 2018 (p. 342).

[57] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In: USENIX Security. 2019 (pp. 312, 341).

[58] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In: Symposium on Operating Systems Principles. 2019, pp. 554–569. DOI: 10.1145/3341301.3359640 (p. 315).

[59] Alireza Shameli-Sendi. Understanding Linux kernel vulnerabilities. In: Journal of Computer Virology and Hacking Techniques 17.4 (2021), pp. 265–278 (p. 315).

[60] Miltiadis Siavvas, Charalampos Marantos, Lazaros Papadopoulos, Dionysios Kehagias, Dimitrios Soudris, and Dimitrios Tzovaras. On the relationship between software security and energy consumption. In: China-Europe International Symposium on Software Engineering Education. 2019 (pp. 312, 341).

[61] Xin Tan, Minghui Zhou, and Brian Fitzgerald. Scaling open source communities: an empirical study of the linux kernel. In: Conference on Software Engineering. 2020 (p. 315).

[62] The Linux Kernel. Dealing with bugs: CVEs. 2024. URL: https://docs.kernel.org/process/cve.html (p. 315).

[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 (p. 342).

[64] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. In: Symposium on Internetware. 2015 (p. 342).

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.