Lukas Giner

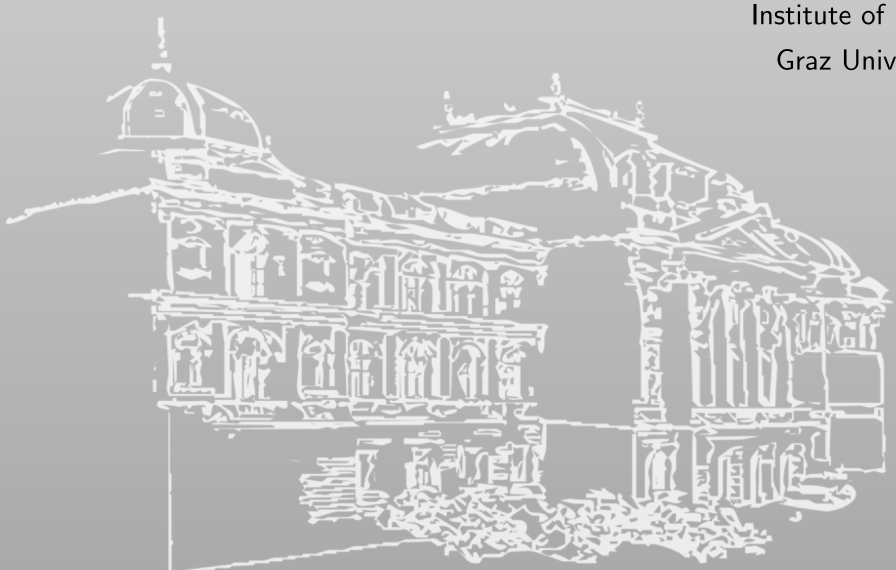
Microarchitectural Attacks and Defenses for Isolated Domains

PhD Thesis

Assessors: Daniel Gruss, Yuval Yarom

July 2025

Institute of Information Security
Graz University of Technology



SCIENCE · PASSION · TECHNOLOGY

Abstract

Modern computer systems use code and data from many different and mutually distrusting contexts. Consequently, they are isolated. However, as long as underlying hardware components are shared, information leakage is still a threat. Side channels can turn minute differences in timing, energy consumption, or other hardware behaviors into security breaches. Among them, cache side channels have proven a persistent threat, leaking everything from encryption keys to user behavior. When used with transient execution vulnerabilities, they have even enabled attackers to bypass memory isolation barriers completely.

In this thesis, we present attacks and defenses that contribute to the understanding of how we can reinforce isolation boundaries for the microarchitecture. We demonstrate cache attacks on GPU caches launched from the restrictive WebGPU JavaScript environment, showing that even without access to memory addresses or timers, malicious websites can use this generic interface to mount attacks on a range of GPUs. Our cache attacks exploiting the AMD SEV-SNP encryption coherence protocol demonstrate that even low-resolution leakage can be turned into highly accurate cache attacks. These two attacks illustrate that even when some side channels are restricted, reduced attack surfaces can still lead to powerful attacks. Spurred by ever-evolving cache attacks, we developed two state-of-the-art secure cache designs that impede attacks at different cache levels. Our LLC design, SassCache, employs a novel cryptographic scheme that combines cache-line randomization with randomization-based partitioning to overcome even occupancy attacks. For the L1 cache, we devise a partition-based scheme that enforces strong domain isolation, increases cache size, and maintains performance and compatibility. Lastly, we develop a low-overhead software mitigation for LVI-NULL attacks on SGX that stops data leakage before it can even reach a side channel like the cache.

This thesis is split into two parts. The first part positions this work within the context of the state of the art. The second part presents the unmodified[1] contributions to the field as they were peer-reviewed and accepted at international security conferences.

[1] The content of the papers is unmodified and as submitted, though colors and layout have been adapted to fit the thesis format.

Acknowledgements

First, I want to thank Daniel Gruss for giving me the opportunity to pursue this doctorate. It's been a long road since I started my master's thesis with you, and I appreciate all the guidance you've given me to get here.

Special thanks go to my work husbands Claudio Canella and Andreas Kogler. The two of you shaped my time at the institute more than anyone else, and I cannot imagine my time at the office without you there. From when I first started as a student researcher until you finished your PhD, Claudio, our daily rants and conversations improved my day, every day, and I appreciate you introducing me to the PhD with your example. When I moved into an office with you, Andi, the conversations turned slightly sillier – but even with the occasional "brrt" I appreciated them, and your friendship, just the same.

Thank you to Michael Schwarz and Moritz Lipp, the senior PhDs in the group when I joined. Whenever I had a question, I knew that one of you would have the answer and that I could always come to ask. The help you provided in that time, and in all conversations since, was invaluable.

Thank you, Jonas Juffinger, for the time we shared during our travels together. I had a blast with you and Andi, and those trips are some of the most memorable times of my PhD. I also appreciate our short time together in an office, where you stoically listened to my complaints (mostly about latex).

Thank you, Martin Schwarzl, Stefan Gast, and Fabian Rauscher. You each shared a significant part of this journey with me, and when I was wandering through the hallways, unsure about what to try next, you were always up for a productive chat. A *special* thanks also to my junior colleagues Sudheendra Raghav Neela, Roland Czerny, Hannes Weissteiner, and Carina Fiedler. You gave me a glimpse into your zoomer ways, and I begrudgingly admit that your memes made me smile every day.

Thank you to Toon Purnal. You were a fantastic lead author for my first paper during the PhD and I enjoyed working with you very much. Hanging out with you was always a highlight at conferences, though I want to note that I am still very suspicious about your varied language skills.

Thank you, Lukas Maar, for a fruitful collaboration and a fantastic adventure in the Indonesian jungle. I look forward to our next trip together!

Thank you, Lena Heimberger, for the same shared adventure and the numerous rants we shared in the kitchen.

Thank you to everyone else at the institute with whom I shared an office, a conversation, or just a coffee over the years!

To my parents, thank you so much. You got me here through my kicking-and-screaming resistance to homework as a kid and the mere reluctance during my bachelor's and master's, and without your support none of it would have been possible.

Thank you also to Yuval Yarom for assessing my thesis and providing a valuable external view.

Big thanks to all my other co-authors, your contributions made this work possible and I am grateful for them.

And finally, thank you to everyone I met along the way, from the hallway track to conference dinners (shout-out to the CISPA crew!), you all made it worth it.

Contents

Part I

Microarchitectural Attacks and Defenses for Isolated Domains

1 Introduction and Contributions

In modern life, computers in many forms are ubiquitous. Whether on personal computers, mobile devices, or cloud services, we expect our data, privacy, and intellectual property to be protected. Therefore, isolation techniques between different domains are at the heart of modern computing, and threat models are based on their efficacy. When these barriers fail or are circumvented, the threat models break down, and unexpected data leakage can occur. Encrypted memory, for instance, quickly loses its protective power when a transient execution attack can make a protected application reveal its own information [171], or a side-channel attack can infer what secrets a victim processed [131] without ever directly interacting with them.

Moving from abstract notions of isolated applications to the real implementations of CPUs, we find that many components are shared between different domains for performance and economic reasons. Whenever there are shared components, there is the risk of information leakage; if not directly, then via side channels [99]. Side-channel attacks are an indirect way of inferring information about a system by observing its externally visible behavior. This may be sound [49, 161], timing [26, 39, 92], electromagnetic emissions [48, 62, 106, 115, 175], power consumption [94, 95, 108], or other observable properties [90, 110, 181].

One of those shared components is the cache hierarchy. Virtual memory provides an isolating barrier between the data of different processes. Cache attacks, however, can reveal memory access patterns through this barrier. Cache side channels were first demonstrated in the 1990s in the form of a covert channel [64] and soon thereafter identified as a contributing factor to timing attacks on cryptographic implementations like RSA [92]. These and many following attacks [15, 21, 131, 133, 168] use the simple fact that data in a cache is returned faster than data that has to be retrieved from memory. This timing difference in a program's run time can be measured and can leak information about the program's data. The Prime+Probe attack [131, 134] improved on this cache attack's resolution using the specific hardware layout of caches to extract not just timing information, but approximate information about which data was accessed. The Flush+Reload attack [193] improved that accuracy to the level of a single cache line. While research continued to improve upon cache attacks [25, 58, 86, 136], transient execution attacks [19, 27, 29, 31, 63, 93, 97, 109, 119, 122, 151, 153, 154, 162, 170, 179, 184] heightened the impact of this side channel by making cache attacks an integral stepping stone to this devastating new class of attacks.

Transient execution attacks like Meltdown [109] and Spectre [93] revealed that while CPUs architecturally isolate domains correctly, the underlying microarchitectural execution of instructions may not follow this isolation as strictly. Using cache side channels to convert microarchitectural states into architecturally observable states, these attacks demonstrated that CPUs can be made to transiently cross permission boundaries and leak information from one domain to another. Where Meltdown-type attacks, *i.e.*, attacks using a very similar mechanism to Meltdown [27, 29, 109, 122, 151, 153, 162, 170, 184], were clear flaws in the design of CPUs and addressed in later hardware generations, Spectre-type attacks [19, 31, 63, 97, 119, 154, 179] are primarily based on intended behavior and are therefore much harder to mitigate.

Given the danger of cache attacks, not just on their own but also as a component of other attacks, it is unsurprising that much research has gone into developing secure cache designs. Secure caches modify current designs, which are optimized to be fast and efficient, to prevent cache attacks or make them infeasible, ideally without excessively reducing speed and efficiency. They try to isolate different security domains, mostly by partitioning the cache into different sections or randomizing the locations data can take in the cache. Research in this area has shown that both methods have advantages and disadvantages. Works analyzing the security of these designs [17, 20, 22, 30, 35, 47, 135, 139, 159] have repeatedly challenged prior assumptions, contributing to a cycle of new designs, new attacks targeting them, and improved follow-up designs.

Another effort towards protecting user data has been the introduction of Trusted Execution Environments (TEEs) [8, 12, 73, 74] TEEs want to eliminate the need for trust in the maker of an operating system or the owner of cloud infrastructure by anchoring trust in the hardware alone. They provide a secure environment for applications or entire virtual machines to run in, isolated from the rest of the system. This isolation includes architectural protections, e.g., data encryption, against malicious operating systems, hypervisors, or other attackers, but increasingly also protections against side-channel attacks. Some attacks, like transient execution attacks, however, proved far outside the scope of these protections, and a myriad of attacks have been demonstrated on TEEs over the years [31, 46, 66, 104, 121, 151, 153, 170, 171, 172, 173, 174, 188]. In the absence of fixes in hardware or microcode, software mitigations are sometimes the only available remedy [117].

In this thesis, we investigate hardware isolation schemes from both sides: as attackers and defenders. We examine AMD SEV-SNP and find that while Flush+Reload is impossible, cache coherence creates a similar side channel. Investigating GPUs, we find that the new WebGPU standard, even though it includes measures against some side-channel attacks, still cannot prevent attackers from conducting powerful GPU cache attacks from the browser. To defend against cache attacks, we examine two different methods on two different levels of the cache hierarchy. With our randomized cache design for the last-level cache, we tackle not only contention attacks like Prime+Probe but also the much harder-to-mitigate occupancy channel. Our partitioned design for the L1 cache demonstrates that it is possible to securely isolate domains while also increasing the effective cache size without increasing the associativity and, thus, the energy consumption. Finally, we build a software-only mitigation for a transient execution vulnerability in the Intel TEE SGX. In the following, we discuss the main contributions of this thesis as well as other contributions made in the same time frame.

1.1 Main Contributions

In this section, I summarize the contributions of the first-author papers written during my PhD. In these five papers, I covered novel cache attacks on SEV and GPU workloads, two very different secure cache designs for the first and last-level cache, as well as a software mitigation for a transient execution vulnerability affecting SGX.

Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP (Chapter 5). AMD Secure Encrypted Virtualization (SEV) [8] is a technology to secure virtual machines by encrypting their memory and isolating them from the host. As the host has no direct access to the memory of the guest, the standard Flush+Reload attack does not work. As AMD's documentation notes [10], this encrypted memory can also be read by the host in encrypted form, and on some machines, *coherence is maintained automatically.* This raises the question of what happens when both the host and guest read from encrypted memory, as the physical address is the same for the purposes of the cache set and tag. We found that the coherence mechanism not only evicts the ciphertext when loading the plaintext and vice versa, but it also evicts up to 31 other cache lines located on the same 4 KiB page. We investigated the coherence effect in detail and discovered that despite the lower spatial resolution compared to Flush+Reload, Cohere+Reload is a powerful attack primitive. In particular, we discovered that because the eviction mechanism is automatic and guaranteed, Cohere+Reload has an even higher temporal resolution than Flush+Reload. This enabled us to do high-resolution single trace attacks on RSA and AES in AMD SEV-SNP. The paper was accepted at DIMVA 2025 and done in collaboration with Sudheendra Raghav Neela and Daniel Gruss.

Generic and Automated Drive-by GPU Cache Attacks from the Browser (Chapter 6). The WebGPU standard [177] specifies an interface for interacting with a system's GPU from the browser. As an attractive target for attacks, the standard explicitly includes considerations for side-channel attacks, including timing-based attacks [178]. We investigated the effectiveness of these mitigations in preventing cache attacks. As prior work demonstrated cache attacks on GPUs from native code and on specific devices or vendors [3, 44, 45, 83, 128], we built WebGPU-based cache attacks that stealthily execute in browsers. We demonstrated that WebGPU compute shaders can be used to profile all cache sets in a matter of minutes, record keystroke timings, and even break an AES encryption running at the same time as our attacker. The paper was published at AsiaCCS 2024 and won a Best Paper award. It was a collaboration with Roland Czerny, Christoph Gruber, Fabian Rauscher, Daniel De Almeida Braga, and Daniel Gruss.

Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks (Chapter 7). Given the wide range of targets of cache attacks and the conclusions from our work on "Systematic Analysis of Randomization-based Protected Cache Architectures" [135] (see Section 1.2), we created a new secure cache design for the last-level cache (LLC) which can resist the Prime+Prune+Probe eviction set creation method, contrary to randomized designs like ScatterCache [186], CEASER-S [139] and PhantomCache [163]. Based on the conclusion that even advanced algorithms like Prime+Prune+Probe rely on re-accessing the same address several times, we developed the design for SassCache. In addition to the Index Generation Layer (IGL), which provides strong profiling resistance like ScatterCache, we introduced a second cryptographic layer, the Index Spacing Layer (ISL). This layer creates a unique subset of the total cache capacity for each security domain, creating a new property we call “hiding”. Each time an address is evicted from and reloaded into the cache, it has a chance to land in a portion of the cache that is unobservable by the attacker, *i.e.*, it is hiding. Excepting systematic self-eviction, once an address is hidden from attackers, it will remain so during their profiling attempts. This also protects against more generic occupancy attacks, as repeated measurements are unlikely to yield meaningful results. Our performance evaluation revealed that while SassCache is mostly slightly slower than ScatterCache and CEASER-S, it performs better in certain circumstances where partitioning is advantageous. The security evaluation confirmed that profiling with Prime+Prune+Probe is no longer feasible. The paper was published at the IEEE Symposium on Security & Privacy 2023 and was a collaboration with Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss.

Fast and Efficient Secure L1 Caches for SMT (Chapter 8). Where LLCs have some leeway to introduce additional delays, e.g., because of cryptographic functions for randomization, most cache hits originate in L1 caches, and any delays, therefore, have a much larger impact on overall system performance. For this reason, secure cache designs for the L1 cache focus mostly on partitioning. Inspired by the increased cache size of the Apple M1, we reexamined the idea of partitioning under the lens of a new set of constraints: page size, energy consumption, performance, and security. Where the M1 increased its page size to 16 KiB, which in part allows it to use a larger L1 cache, we maintained backward compatibility for software by keeping the current page size of 4 KiB while increasing the L1 cache size without simply increasing the associativity and, thus, energy. With SMTCache, we introduced a design that balances energy, performance, and security by creating copies of the traditional L1 cache and switching between them depending on the current security domain. With this, we can double the effective capacity for Simultaneous Multithreading (SMT) by providing a private L1 cache for each thread. As the individual caches are the same size as current standard L1 caches, we can keep the same associativity, and the energy consumption (assuming the same total number of loads) increases only marginally. We found that while switching between caches for different threads does not behave as if the total cache was larger, SMTCache still performs similarly to caches with the same total size, though at a much smaller energy cost. The paper was accepted at ARES 2025 and was a collaboration with Roland Czerny, Simon Lammer, Aaron Giner, Paul Gollob, Jonas Juffinger, and Daniel Gruss.

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX (Chapter 9). After the first wave of transient execution vulnerabilities [27, 93, 109, 151, 153, 170, 171, 184], Intel began rolling out CPUs with mitigations against the major vulnerabilities [68], such as Meltdown, Foreshadow (L1TF), and Microarchitectural Data Sampling (MDS) variants like Fallout [27], RIDL [151], and ZombieLoad [153]. The Intel Comet Lake architecture includes mitigations for MDS, which also stop the value injection of LVI [171]. However, the LVI variant LVI-NULL (or “Load Value Injection: Zero Data”) remained unmitigated in silicon or microcode. LVI-NULL is a special case of load value injection where only ‘0’ (but no other values) can be transiently injected into the victim data flow. The investigation of Van Bulck et al. [171] and our preliminary work suggested that while this was a less flexible attack, it still opened many attack vectors on Intel Software Guard Extensions (SGX) enclaves in particular. The available software mitigation in LLVM targeted LVI as a whole [24] by fencing all loads and had enormous overheads. Even an optimized version by Intel [69] that significantly reduced fences still left much to be desired regarding performance. We also discovered that prior proposed mitigations, such as modification of the null page [171], would not be enough to protect against LVI-NULL. Unsure of when this remaining attack surface might be fixed, we developed LVI-NULLify as a software-only mitigation for SGX enclaves. Through modifications to the LLVM compiler, SGX-SDK, and Intel Platform Software (PSW), we offset all loads and stores in an enclave using the segmentation feature. This allows us to create a “null page” for each enclave. More specifically, transiently injecting null into any pointers would no longer point them at the actual null page in the virtual memory space but at the beginning of the enclave that is under the mitigation’s control. Our mitigation shows modest performance overheads of less than 10 %, compared to other mitigations that can reach 1 000 % overhead or more. This work was published at the USENIX Security Symposium 2022 and was a collaboration with Andreas Kogler, Claudio Canella, Michael Schwarz, and Daniel Gruss.

1.2 Other Contributions

In addition to the first-author papers, I contributed to several other papers during my PhD, 3 of which were accepted at tier 1 conferences.

Following the publication of ScatterCache [186], Antoon Purnal discovered an improved eviction set creation method he called Prime+Prune+Probe [137]. Based on this, we created a **Systematic Analysis of Randomization-based Protected Cache Architectures** [135] that compares the security of several cache designs against Prime+Prune+Probe and other attacks. Our analysis revealed that most current randomized secure cache designs could be expressed and analyzed with a common model, and that those we examined were less secure than their authors had assumed. This work was published at the IEEE Symposium on Security & Privacy 2021 and was a collaboration with Antoon Purnal, Daniel Gruss, and Ingrid Verbauwhede.

After the publication of several software-based power attacks on commodity CPUs [108, 110, 181] that targeted specific algorithms, we investigated the feasibility of applying the methods of traditional correlation power attacks to generalize this attack and leak data values directly from the memory hierarchy. In **Collide+Power** [95], we demonstrated that this is indeed possible. We showed that wherever data *collides* in the microarchitecture, the power consumption of the CPU leaks recoverable information about the difference between the two colliding pieces of data. This paper was published at the USENIX Security Symposium 2023 and was joint work with Andreas Kogler, Jonas Juffinger, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard.

As prior work [28, 55, 65, 98, 107, 114, 165] has made clear, the Translation Lookaside Buffer (TLB) is structurally very similar to data caches and, therefore, susceptible to the same kinds of attacks. Consequently, any kernel changes to memory allocation must be carefully considered, as any differential treatment between different types of memory allocations is reflected in the TLB in one way or another. In **When Good Kernel Defenses Go Bad** [118], we examined a wide range of changes to the Linux kernel aimed at *increasing security*. We found that three of the twelve defenses that change page mapping open up a new attack surface in the TLB. These changes allowed us to leak the position of kernel objects, which can be used as an important stepping stone in kernel exploitation.

This work was published at the USENIX Security Symposium 2025 and was a collaboration with Lukas Maar, Daniel Gruss, and Stefan Mangard.

2 Background

This chapter gives general background that applies to later chapters. As this thesis mostly discusses x86-64 architectures from Intel and AMD, many implementation details in the following reflect this.

2.1 Virtual Memory and Segmentation

In modern computer systems virtual memory fulfills two important roles. Firstly, it enables domain isolation, e.g., between processes or virtual machines, by abstracting applications' address spaces away from the underlying physical memory. Secondly, it provides the illusion of near infinite and uninterrupted memory. This is supported by hardware through *paging* and *segmentation*.

Paging maps *virtual* addresses that instructions operate on to *physical* addresses with page granularity, where a page on x86-64 architectures is typically 4 KiB, 2 MiB, or 1 GiB large. This translation is achieved through several levels of *page tables*. Figure 2.1 shows how 5-level paging resolves an virtual address. In 4 or 5-level paging, virtual addresses may be 48 bit or 57 bit long and are resolved from the highest to the lowest level. Each level resolves 9 bit of the virtual address and points to the location of the next level's table. These page tables are set up by the operating system and contain not only the locations of the next table or the physical page location but also additional information like access, write, and execute permissions. Every process has its own set of page tables, with the `CR3` register always pointing to the root of the current top-level paging structure. CPUs accelerate this translation by caching recently used translations in Translation Lookaside Buffers (TLBs). The structure of TLBs is very similar to data caches, cf. Section 2.2.

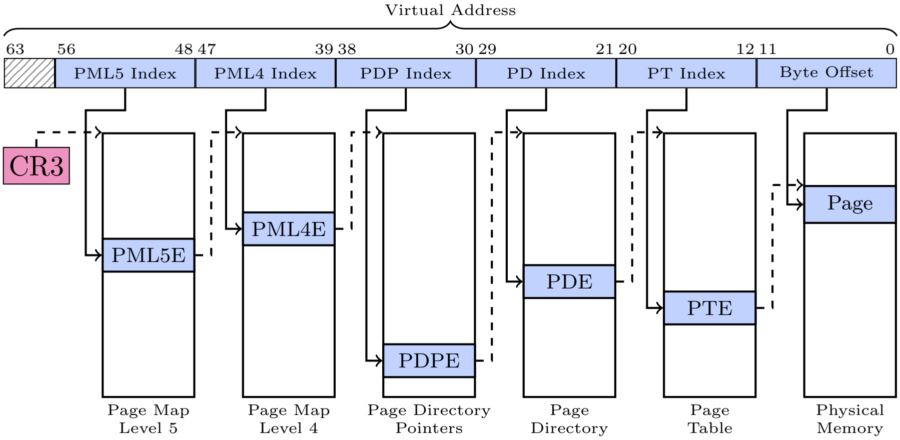


Figure 2.1: Virtual memory with 5-level paging. The virtual address is split into 9 bit segments that index the different page table levels to find the final physical memory address.

Segmentation is an older mechanism that is only used in limited ways in x86-64 architectures. Segmentation partitions memory into different *segments* for, e.g., data (DS), code (CS) or the stack (SS) based on a segment's base linear address, limit (size) and permissions. With segmentation, programs operate on *logical* addresses, which are then translated into *linear* addresses. When paging is used, these linear addresses are the virtual addresses. Though technically still enabled on x86-64 in 64 bit mode, all but two segments, FS and GS, have a forced base address of 0 and no limit. This effectively flattens all logical addresses into the same virtual address space by default, unless they specifically use the FS or GS registers. The FS and GS registers can still be used as offsets to instructions or data loads that specify their use, e.g., for thread-local storage [42].

2.2 Caches

Caches are small buffers of memory between the DRAM and the execution core that accelerate accesses to often-used data based on the principle of locality. Firstly, consecutive memory accesses are often close to each other, and secondly, memory that was recently accessed is likely to be accessed again in the near future. Caches store data in blocks, typically,

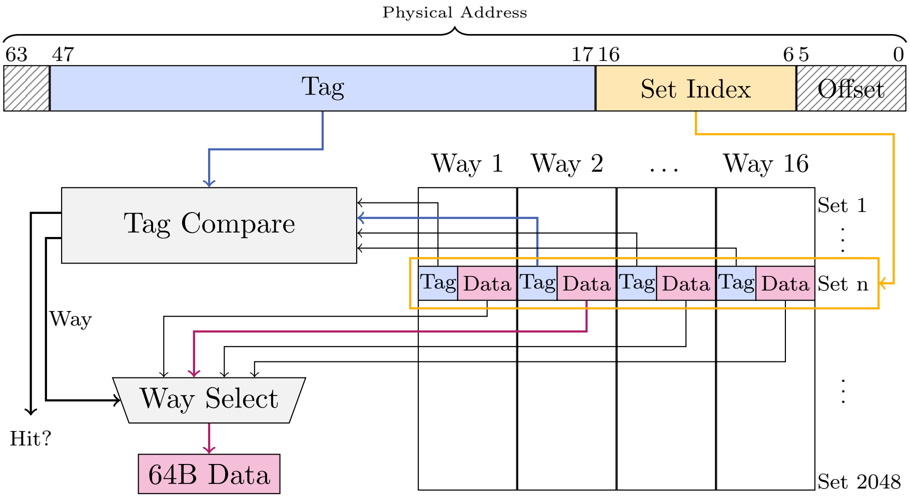


Figure 2.2: Cache addressing in a 16-way set-associative cache.

64 B, called *cache lines*, and each cache line is identified by its *tag*. In the following, we refer to the cache-line-sized and aligned data as *blocks*, while *cache lines* are the actual storage units in the cache. In modern caches, cache lines are grouped together in cache *sets*, forming a *set-associative* cache. The size of a set is called its *associativity*, while the cache lines in a set are called *ways*. Common associativities of caches are between 4 and 32 ways, depending on the use case of the cache.

Figure 2.2 shows how a block's location in the cache is determined. In standard designs, the block's address is simply split into three parts: *tag*, *set index*, and cache line *offset*. The set index determines the set that a block may be located in. Within the set, a way corresponding to a block may be found by comparing the tags. Addresses that map to the same set are sometimes called *congruent*. If a requested block is in the cache, it is referred to as a *cache hit*. When no match is found (a *cache miss*) and the requested block is to be stored in the cache, a *replacement policy* (see Section 2.2.1) decides which way the block should replace. In the example, the set index starts after the offset with bit 6 and ends with bit 16, while the tag consists of the upper part of the address. This implies a cache with 2 048 sets and 64 B cache line size. Together with the number of ways $n_{ways} = 16$, we can infer that this cache is 2 MiB in size.

Set-associative caches are a mixture of fully associative caches and direct-mapped caches [123]. In direct-mapped caches, the address alone determines the exact cache line its data may be stored it. For this reason, they feature the fastest lookup of the three types. The downside is that two congruent addresses will always evict each other. In fully associative caches on the other hand, addresses may map to any line in the cache. This makes theirs the most expensive lookup, as all tags have to be searched to determine a cache hit or miss. Their advantage is improved cache utilization and no fixed collisions between addresses, since the replacement policy can choose to place new data anywhere in the cache. Set-associative caches are a middle-ground that feature fast lookup while allowing many congruent addresses in the cache before evicting them (see Section 2.2.3).

For *physically indexed, physically tagged (PIPT)* caches, the set index and cache line tag are derived from the physical address. Each physical address is associated with exactly one set, in which it can be stored in any one way. The advantage PIPT caches is therefore that the set index and tag together uniquely identify a location in memory, which greatly simplifies their access logic. A *virtually indexed, physically tagged (VIPT)* or *virtually indexed, virtually tagged (VIVT)* cache on the other hand base either the set lookup (VIPT) or the set lookup and tag comparison (VIVT) on the virtual address. This has the advantage that no TLB query is necessary and responses can be faster and require less energy. On the other hand, these designs introduce a new set of problems: synonyms and homonyms. Synonyms are different virtual addresses pointing to the same physical address, while homonyms share same virtual address but refer to different physical addresses. VIVT caches are rarely employed because these problems require tradeoffs (such as flushing the entire cache on context switches) or more complex handling (e.g., page coloring [23]). In small caches, the VIPT scheme may be employed without downsides if only the bits within a page (which are the same as the physical bits) are used as a set index.

2.2.1 Replacement Policies

When a new line enters a cache and no way in its set is marked as invalid, a way must be evicted to make space for the incoming data block. The replacement policy determines which way is the next eviction candidate. Most replacement policies are derivations of Least Recently Used (LRU). In ideal LRU, all ways within a set store their “ages” in addition to the tag and data. When a cache line is accessed, its age is set to 0, and the age of all ways whose age was previously lower than the accessed one are increase by 1. As the policy name implies, the eviction candidate is then the way with the highest age, *i.e.*, the least recently used cache line in the set. This achieves good performance for programs with high temporal locality, *i.e.*, programs that often reaccess the memory locations. In practice, using ideal LRU is often too costly in hardware (e.g., storing 4 bit per cache line for a 16-way cache and modifying them all) or does not have all the desired properties. In L1 caches (see Section 2.2.2), a pseudo-LRU variant called tree-PLRU is often used [1, 2] because of its efficiency. Instead of $log(n)$ bits per way it only uses $n - 1$ bits in total. Despite this reduction, tree-PLRU still approximates LRU for many access patterns. L2 or L3 caches use more complex replacement algorithms than PLRU [1], like Quad-Age LRU (QLRU) [80], which stores a line's approximate age in only 2 bit, or policies similar to a Bimodal Insertion Policy (BIP) or Dynamic Insertion Policy (DIP) [140], like BRRIP or DRRIP [81]. These policies try to mitigate misses caused by streaming large amounts of single-use data by inserting new data in (or close to) the LRU position. Depending on the exact policy, this can cause a newly inserted line to be evicted by another new line either right away or very soon, unless it is accessed in the meantime. In this way, an access pattern generating large amounts of accesses will only occupy a small part of each cache set, while older but reused cache lines can persist. On the other end of the spectrum is the simple *random replacement* policy. As the name suggests, it simply inserts new lines in a random position. Its performance is generally worse than LRU.

2.2.2 Cache Hierarchy, Inclusivity and Coherence

Modern CPUs employ a hierarchy of caches, with a small and fast level 1 (L1) cache nearest to the core and a large and comparatively slow last-level cache (LLC), usually the L3 cache, furthest from the core (Figure 2.3). Latencies for the different levels are typically 3 cycles to 5 cycles for L1, 10 cycles to 14 cycles for L2, and wildly varying (26 cycles to 70 cycles) for L3 caches [71]. The size of a VIPT L1 cache is determined by the page size, which limits the number of available bits for the cache line size and set index, and the number of ways, which is limited by energy and performance constraints [126, 164]. Common L1 cache sizes range from 32 KiB to 128 KiB [71]. Additionally, there is usually an L1 instruction cache (L1I) that exclusively stores data loaded as instructions in addition to the general L1 data cache (L1D). L2 caches and up are not constrained

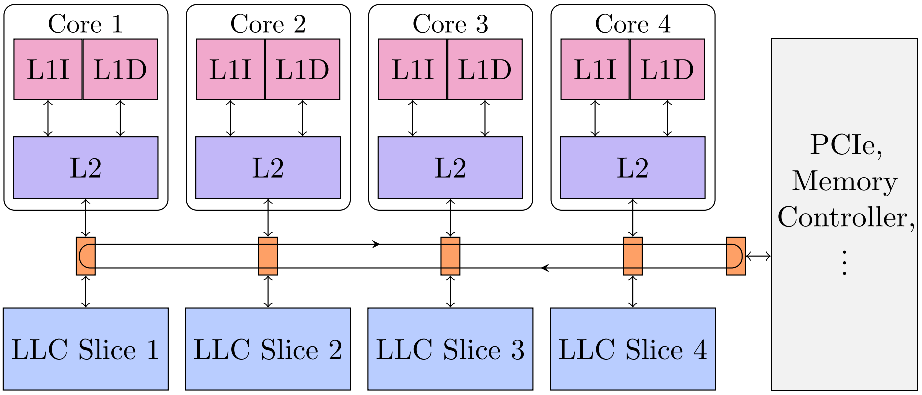


Figure 2.3: An example cache hierarchy as might be found in some Intel architectures. The first and second level caches are private to each of the four cores, while the shared last-level cache is split into slices. The depicted slices are connected to the cores and each other by a ring bus, and are typically physically close to one of the cores.

by the page size and feature sizes from 256 KiB per-core L2 cache to over 1 GiB in the case of L3 caches that can be distributed over many cores [7, 71]. For practical purposes, LLCs are often partitioned into cache *slices*, where each slice functions as its own autonomous cache (Figure 2.3). Though there are often as many slices as there are physical (or logical) cores, the *slice index* of an address is independent of the core an access is performed on. This slice index is determined by a slice addressing function (or hash function) that is often a linear combination of exclusive-or operations on physical address bits [50, 65, 67, 79, 113, 120, 125, 141, 156, 192, 194].

Within a core's cache hierarchy, caches can be *inclusive*, *non-inclusive* (sometimes called non-inclusive, non-exclusive, NINE) or *exclusive* w.r.t. each other [127]. A larger cache that is inclusive of a smaller cache guarantees that any cache line that is stored in the smaller cache can also be found in the larger cache. Conversely, this means that an eviction from the larger cache must also trigger and eviction from the smaller cache. Exclusive caches, as the name implies, ensure that none of their cache lines are duplicated in caches they are exclusive of. Non-inclusive caches give no such guarantees and are implemented with different strategies as to when lines will be filled or evicted. While inclusive last-level caches are common, non-inclusive LLCs with inclusive L2 caches are increasingly popular with a growing number of cores and increasingly large L2 caches [76].

In CPUs with more than one core, caches below the LLC are private to each core, while the LLC itself is shared accross all (or at least several [6]) cores. With more than one cache, it becomes necessary to keep all cached data coherent. This means ensuring that an address is never accessible with different data through different caches at the same time. A *cache coherence protocol* addresses this. It specifies states that a cache block may be in, and governs how blocks can transition from one state to another. A simple protocol is MSI, where blocks may be marked as *modified*, *shared*, or *invalid*. For example, when a cache controller requests to modify a cache block that it does not currently have, all other copies need to be invalidated to ensure consistency [127].

There are two main implementations of coherence protocols, *snooping* and *directory* coherence protocols. Snooping is a collaborative protocol where each cache controller broadcasts coherence requests that are observed by all cache controllers. As this type of coherence protocol does not scale well with an increasing number of cache controllers (e.g., cores), directories are more popular in modern CPUs. A directory coherence protocol is based on a central *directory* structure that maintains coherence information for all blocks in the cache hierarchy. Here, coherence requests only need to be sent to the directory, instead of broadcasting them. The directory may then forward subsequent updates only to the cache controllers that hold the affected blocks. In designs with an inclusive LLC, this scheme is greatly simplified, as the cache directory contains the same blocks as the LLC.

2.2.3 Side-Channel Attacks on Caches

Cache attacks are a class of *side-channel attacks* and as such do not leak data from the cache, but reveal meta information, such as access patterns or occupancy levels. Since a program's memory accesses are determined by its control- and data-flow, sensitive information can often be recovered from observing the cache. We can broadly split cache attacks into five categories: Occupancy Attacks, Evict+Time attacks, contention attacks (Prime+Probe), shared-memory attacks (Flush+Reload), and internal-collision attacks.

Occupancy attacks measure the overall usage of the cache by filling it either totally or partly and measuring the share of attacker controlled lines that were evicted by a victim program's execution. Measuring eviction is as simple as timing a load, e.g., with the `rdtsc` instruction, and comparing it to a threshold above which it was likely served from DRAM. The information gained is limited, but they can still be used for website fingerprinting [158] or even attacks on cryptography implementations [30]. **Internal-collision attacks** do not interact with the victim directly, but observe their execution time based on their own cache usage. Some versions of this attack may clear the entire cache before an attack or rely on consistent but non-attacker-controlled outside behavior to cause caches evictions. This type of attack potentially yields less granular information than later attacks, but has been used to great effect nonetheless [15, 21, 133, 168]. Closely related are **Evict+Time** [131] attacks. They use the layout of set-associative caches to target specific sets for eviction and then measure a victim algorithm's execution time.

Contention-based attacks (sometimes "*interference attacks*") are much more accurate improvement over Evict+Time attacks, with the *Prime+Probe* [131, 134] attack being the most prominent example. These attacks manipulate the state of one or more targeted cache sets and later observe the victim's accesses in them directly or indirectly [16]. Depending on the targeted cache level, this can leak around 11 bit of address information (for a cache with 2048 sets, bits 6-16). Prime+Probe first *primes* a cache set by filling it with attacker controlled lines and then *probes* those same lines again to see if any have been evicted in the meantime. Later variants of Prime+Probe extend it with knowledge of the replacement policy or adapt it to specialized cache designs (see Section 3.2). Set-based attacks that target the LLC require either knowledge of the slice function (see Section 2.2) to construct eviction sets, or a general algorithm for eviction-set construction [113, 130, 139, 176]. On non-inclusive caches (cf. Section 2.2.2), attacks on the cache directory instead of the LLC have been proposed [192].

The first and most well-known exemplar of **shared-memory attacks** is *Flush+Reload* [193]. Instead of indirectly measuring a target cache line's state via its set contention, these attacks target shared memory to directly cache lines of interest. In Flush+Reload, the targeted line is directly evicted from the cache with the `clflush` instruction and later measured (*reloaded*) to determine if the victim accessed it. While this has the additional requirement of shared memory with the victim, the benefit is cache-line accuracy and significantly reduced noise. Well-known variants if Flush+Reload are *Evict+Reload* [59] and *Flush+Flush* [58]. Flush+Flush uses the different duration of the `clflush` instruction itself to determine whether an address was cached, while Evict+Reload replaces the flush instruction with a set eviction like Prime+Probe for devices or contexts (e.g., JavaScript) that do not support `clflush`.

2.3 Trusted Execution Environments

A *Trusted Execution Environment (TEE)* is a hardware-supported mode of execution that provides code running on trusted hardware that is managed by untrusted third parties with certain security guarantees, such as code and data confidentiality, and integrity. These properties are intended to hold even in the face of a malicious hardware owner.Earlier examples of TEEs are Intel Software Guard Extensions (SGX) [73] or ARM TrustZone [12] which target mainly private devices. These extensions allow specialized software to execute security-critical code in the TEE *enclave*, while most code runs outside with no access to an enclave's memory or registers.

Recent TEEs, such as Intel Trust Domain Extensions (TDX) [74], AMD Secure Encrypted Virtualization (SEV) [8, 9] and ARM Confidential Compute Architecture (CCA) [11] have adopted a new paradigm of securing an entire virtual machine in an untrusted cloud environment [33], creating Confidential Virtual Machines (CVMs).

2.3.1 Intel SGX

Intel's Software Guard Extensions (SGX) provide confidentiality and integrity through hardware mechanisms under the assumption that only the hardware is trusted [34, 73] and an attacker may have control over the host operating system. The code and data that comprises the secure unit of memory is called an *enclave*. Confidentiality and integrity are enforced by several mechanisms, the first among which is hardware encryption of all enclave memory. This memory is allocated and managed by the operating system, but still protected from being read or written to by the OS. To prevent the operating system from simply remapping arbitrary pages into the enclave, SGX keeps a separate record of its expected pages and their properties in its enclave page cache map (EPCM). Building on this protected memory area are a limited number of well-defined entry and exit points for the enclave which automatically save and restore the confidential state of the enclave to its encrypted memory region, even if interrupted by a malicious operating system. Enclaves are executed in the virtual memory space of a host application. This means that they can read and write to data in the normal user space application, though enclaves cannot execute code outside their allocated enclave pages.

2.3.2 AMD SEV

AMD Secure Encrypted Virtualization (SEV) was introduced in 2016 together with Secure Memory Encryption (SME) in the Zen microarchitecture [85]. While SME simply provides encryption of selected pages in memory, SEV builds on this mechanism to provide a Confidential Virtual Machine solution. Each CVM is assigned a unique encryption key that is used to transparently encrypt and decrypt any data between the CPU and DRAM. This initial version of AMD SEV only provided confidentiality and was quickly shown to be vulnerable to several attacks [43, 61, 124, 185, 187]. By now, SEV is in its third iteration with SEV-SNP [8, 9]. SEV-ES adds encrypted state (ES) to the SEV design, which encrypts all register contents when the CVM is not paused. SEV-SNP adds Secure Nested Paging (SNP), which, among other things, protects against memory replay and remapping attacks, thereby adding integrity. Prior to SNP, the hypervisor had full control over the pages, which meant that even if they were encrypted, the hypervisor could overwrite a page with old encrypted data, or map other pages into the guest's address space. With SNP, the hypervisor can no longer write to guest memory, and the paging structure is secured by a Reverse Map Table (RMP). The RMP is a hardware-protected table in memory that tracks the owner of each page, and SEV guests need to validate each page when it is assigned to them.

On some CPUs, SME and SEV provide "coherency across encryption domains" [10], which ensures that any reads from any encryption domain always return the most recent value, even if it was not written back to memory yet.

2.4 Transient Execution

Transient execution is an effect that arises from the difference between the Instruction Set Architecture (ISA) and the microarchitecture. Where the ISA defines how a CPU that implements it behaves from the perspective of the programmer, the microarchitecture as an instance that implements the ISA may do so in a variety of ways.

Out-of-Order Execution. Diverging from the simple architecture model of in-order execution, where assembly instructions are executed one after another, modern out-of-order processors may execute instructions (or parts thereof) in parallel or in a different order entirely if their dependencies are met. The only guarantee is that at instruction *retirement* (or *commit*), the effects of the instructions become *architecturally visible* in program order. This is possible because a CPU has multiple *execution units* dedicated to different operations. Out-of-order operation enables the CPU to schedule operations that do not depend on each other on as many execution units as possible, thereby drastically improving throughput and latency.

Speculative Execution. While executing a linear instruction stream out of order can already speed up execution significantly, the CPU may still sometimes stall when it encounters a dependency on an earlier instruction. The most common example of this is a branch that depends on a load. Until the condition for the branch is resolved, it is unknown where the instruction stream will continue. Modern CPUs employ *predictors* to avoid stalling in these situations. In the simplest form, a "predictor" may simply guess a branch target and continue execution there. In practice, predictors make an initial guess when no information is available and a guess informed by the predictors history otherwise. When a misprediction occurs, the wrongly executed (or scheduled) instructions are "*squashed*", *i.e.*, removed from the scheduler and all other microarchitectural elements they may currently be processed in, and their effects are undone. The totality of situations where the CPU executes instructions based on guesses is called *speculative execution.*

Transient Execution. *Transient execution* (or *transient instructions*) as a term was coined in the Spectre [93] and Meltdown [109] papers. It refers to the natural consequences of processors that execute instructions out of order and speculatively; instructions that were already executed but need to be discarded before retirement. This may be because of a misprediction, a fault in a prior instruction, or any other microarchitectural event (e.g., an interrupt). While these transient instruction are not allowed to leave any trace in the architectural state of the machine, they can influence microarchitectural state. This can be exploited for transient execution attacks, which we discuss in Section 2.4.1.

2.4.1 Meltdown-Type Transient Execution Attacks

Transient execution attacks [29] are a class of attacks that leak information from within transient execution through the microarchitectural state into the architectural state of the attacker. While the transient instructions are squashed and do not affect the architectural state, some microarchitectural elements are still modified by their execution. This microarchitectural state can later be recovered via a side channel. An often-used example of such an element is the cache. Once an address has been loaded into the cache (and another evicted), the prior state cannot simply be restored by the squashing of the transient instructions that caused the change. During a transient window (a group of instructions that are executed but squashed together for the same reason), an attacker can intentionally encode sensitive information into the cache by accessing a certain offset in memory, thereby bringing this address into the cache. Other elements, such as the AVX engine [154] have also been shown to be exploitable for data extraction.

Meltdown-type attacks [29] are a subclass of transient execution attacks that exploit deferred permission handling or faulty behaviors during transient execution to leak otherwise inaccessible data. In the case of Meltdown [109], a user space attacker can transiently access kernel memory, as affected CPUs defer the permission check until retirement of the offending instruction. Until then, the data is still served from the L1 cache (if present) and can be used by later instructions. If the transient window is long enough, *i.e.*, the oldest instruction that has yet to retire takes long enough to do so, the value of an illegal access can be encoded into a cache access and later recovered. Other variants of Meltdown exploit data forwarding from different buffers or through different permission checks, leaking not only from the L1D [109, 170] but also the line-fill buffer [109, 151, 153], store buffers [27], load ports [151], the FPU register file [162], and SIMD register buffers [122].

Load Value Injection [77, 171] (LVI) inverts Meltdown by transiently *injecting* data into the victim. An attacker places data into a Meltdown-vulnerable microarchitectural buffer and then creates the conditions (e.g., a fault) for the victim to 'leak' this data into its own control or data flow. Data can be injected into *any* load in the victim, even implicit loads in instructions like `ret`. With the right gadget in place, the victim can be made to leak its own sensitive information into a microarchitectural element, e.g., the cache state. This aspect is similar to Spectre [93], which also depends on victim gadgets [171].

We discuss defenses against Meltdown-type attacks in Section 3.3.

3 State of the Art

This chapter gives an overview of the state of the art in the fields relevant to the publications in this thesis. We discuss secure cache designs in Section 3.1 and analyses on them in Section 3.1.4. Section 3.2 describes the cache attacks that motivate secure cache designs and finally Section 3.3 reviews defenses against Meltdown-type attacks.

3.1 Secure Cache Designs

Secure caches aim to address the problem of cache attacks in hardware. Over the years since publication of the first thoughts on secure caches [132], the understanding of cache attacks and secure cache designs have both evolved significantly, and the multitude of designs published since the first Prime+Probe cache attacks (see Section 2.2.3) reflect this. In this section, we overview two decades of secure cache designs and how our research contributed. We give a brief overview of a number of designs, how they relate to each other and what types of cache attacks they mitigate.

Though partitioned or skewed caches, for increased performance in specific applications, have a long history in academia (e.g., skewed caches first proposed in 1993 [157] and dynamic cache splitting in 1995 [84]), we limit our overview to cache designs that focus on security against attacks described in Section 2.2.3. One of the earliest secure cache designs is a partitioned cache design by Page [132]. He offered an exploration of what a cache partitioned by domain identifiers might look like, and even suggested an element of randomization with a mask to the address that changes the mapping of addresses to cache lines. Shortly thereafter, Wang and Lee [183] proposed both the *Random Permutation Cache* (*RPcache*) and *Partition-Locked Cache*(*PLcache*) in the same publication, embodying in a single paper the start of a field of research that is largely split between randomization- and partition-based designs.

3.1.1 Partition-based Secure Caches

Partition-based designs split the cache between different security domains, such that contention-based attacks are ideally completely ruled out. The simplest example of such a design is static partitioning. To support n security domains, the cache is split into n partitions. The drawback of such designs is a performance penalty for many workloads, as the available cache for each application effectively shrinks. An alternative are dynamically partitioned caches, where partitioning is done in the moment according to particular performance and security needs.

The most common strategy for partitioning is to split the cache within the sets, though there are many different approaches, such as modifying the cache line replacement policy [150, 183]. **PLcache** [183] employs new instructions for dynamic, on-demand locking of single cache lines or memory regions for security domains. These lines are locked into the cache when they are first loaded and the replacement policy may not select them until they are unlocked. Denial of service through locking too much of the cache is prevented by the operating system, which oversees all locking and unlocking. **Vantage** [150] is a design that allows for dynamic creation, removal and resizing of partitions with cache line granularity. It uses a modified LRU replacement policy to insert lines into standard sets, which allows it to keep the associativity of the cache. The size of a partition is not statically assigned, but instead controlled by monitoring of replacement rates and allowing occasional spill-over into a reserved cache region. The number of partitions is limited by the number of tag bits, which are used to identify the partitions. The **NoMo** (non-monopolizeable) cache [41] is a way-split design that proposes to reserve some ways in each set in the L1 per SMT thread. Reserved ways cannot be evicted by the other thread, but are still observable in the case of shared memory. When the two reserved partitions do not fill the entire set, the remaining ways are shared between the threads. **CATalyst** [111] is a similarly way-split design, however one that builds on top of Intel's existing Cache Allocation Technology (CAT) [70] for the LLC. CAT allows a hypervisor to assign different levels of classes of service, which can be given exclusive write access to parts of each cache set. *CATalyst* uses this in a hybrid software-hardware-managed scheme where VMs can request a certain number of secure pages that are the guaranteed to always reside in the LLC and be unevictable. These pages are not shareable between VMs, and their total number is limited not by the number of service classes, but by the portion of the LLC dedicated to the secure partition. Based on extending Verilog with information flow semantics, Zhang et al. present **SecVerilog** [196] and evaluate a cache where sets are statically partitioned into two different security levels. The design ensures that high security operations do not interfere with low-security cache state, and low security operations cannot generate hits from lines that are in the high-security partition. Instead, low-security operations simulate a miss, and the cache line is moved from the high security partition to the low-security partition. This creates a one-way flow of side-channel information. **SecDCP** [180] builds on this asymmetric information-flow model and extends it with more security levels and dynamic partitioning according to runtime demand of the low-security application. **DAWG** [91] (Dynamically Allocated Way Guard) partitions sets into domains with per-domain bit masks, allowing for replacement only in the same domain and hits only in a defined set of ways. This also enables duplicate read-only cache lines, which protects against shared memory attacks. **HybCache** [37] uses a hybrid design that provides a standard set-associative cache for most applications, but allows for a fully-associative *subcache* cache for security-critical applications. The subcache is made up of a fixed amount of reserved ways in each standard set. Isolated domains then only operate within this subcache, which does not leak address location information because it is fully-associative with random replacement. The fully-associative nature however also increases power overheads, making it impractical for large caches.

More recent publications [38, 145] argue that way-based partitioning is often not fine-grained enough, does not scale well with security domains or large caches, or does not support shared memory. Instead, these designs allocate entire sets to domains instead of ways. Saileshwar et al. propose **Bespoke Cache Enclaves** [145], which allocates dynamic LLC partitions of multiple *clusters* of sets which are of a fixed size, e.g., 64 KiB. A configurable cache indexing function is used to map a domains addresses to their own private cache enclave. Similarly, **ChunkedCache** [38] reserves chunks of full sets for TEEs, which are then fully isolated from each other. Domains can configure the size of their chunks and specific memory regions they apply to. **Composable Cachelets** [167] is another LLC design that provides dynamic and fine-grained cache isolation for enclaves. Contrary to the previous designs, cache lets may partition sets and ways, such that enclave memory is remapped to certain sets that belong to one of its assigned cachelets. Inside the set, a modified replacement policy ensures that an access can only evict ways belonging to that enclave. Enclaves can be assigned a variable number of cachelets, and non-enclave programs may use all ways that are not assigned to cachelets. The **Untangle** [199] framework makes the case that many dynamic partitioning designs still leak information through their resizing decisions. The framework is then applied to present a new set-partitioned LLC design with 9 pre-defined partition sizes. Partitions are resized based on active monitoring of LLC utilization in the recently retired instructions.

In **Jumanji** [155], Schwedock and Beckmann take a different approach to partitioning. Their proposal, based on the earlier partitioned design **Jigsaw** [14], splits the LLC by cache banks that are assigned to individual VMs, creating an isolating barrier between them. The banks are allocated such that they are closest to their respective assignees, which improves latencies over the design's predecessor.

With **SMTCache** (Chapter 8), our contribution to the state of the art is an approach that considers different constraints for L1 caches in particular. We increased the overall size of the cache while keeping latency, power consumption and page sizes the same as standard VIPT caches, at the cost of increase chip area. Our design does not modify the functions of current L1 caches, but instead uses two of them, so-called *slices*, in parallel within a new overarching structure. *SMTCache* is specifically tailored to the SMT use case, where two concurrent workloads can make use of an effectively doubled cache capacity while being isolated from each other. An entire slice is assigned to a workload at a time, and the slice is flushed when switching between security domains. As set sizes and cache addressing are not modified, hits are served with the same latency as standard caches.

3.1.2 Randomization-based Secure Caches

Designs based on randomization either seek to make the link between an address and its location in the cache infeasible to find, or at least hard enough that an attacker will be thwarted by an eventual rekeying that changes the mapping. They usually offer probabilistic security, but may have higher hit rates or be more flexible than partitioned designs that reduce the available cache size. They inherently try to prevent contention-based attacks that may leak address information, while typically leaving the cache occupancy channel open. Randomization-based designs are

Cache Design	Year	Level	Type	Predecessor
PLcache [183]	2007	any	P	
RPcache [183]	2007	any	R	
NewCache [182]	2008	L1	R	RPCache
Vantage [150]	2011	LLC	P	
NoMo [41]	2012	L1*	P	
Random Fill Cache [112]	2014	L1	R	
SecVerilog Cache [196]	2015	L1	P	
CATalyst [111]	2016	LLC	P	
SHARP [191]	2017	LLC	O	
RIC [87]	2017	LLC	O	
CEASER [138]	2018	LLC	R	
DAWG [91]	2018	LLC*	P	
ScatterCache [186]	2019	LLC	R	
CEASER-S [139]	2019	LLC	R	CEASER
PhantomCache [163]	2020	LLC	R	
HybCache [37]	2020	L1,L2	P	
Jumanji [155]	2020	LLC	P	Jigsaw
SecDCP [180]	2020	LLC	P	SecVerilog
FTM [142]	2020	LLC	O	
Mirage [147]	2021	LLC	R	
TimeCache [129]	2021	any	O	FTM
BCE [145]	2021	LLC	P	
Comp. Cachelets [167]	2022	LLC	P	
Chunked-Cache [38]	2022	LLC	P	
Chameleon Cache [169]	2022	LLC	P	CEASER-S, ScatterCache
ClepsydraCache [166]	2023	LLC*	R	
SassCache [52]	2023	LLC	R+P	ScatterCache
Recast [198]	2024	LLC	R	
Maya Cache [18]	2024	LLC	R	Mirage
Song et al. [160]	2024	LLC	R	CEASER
SMTCache [51]	2025	L1	P	

Table 3.1: A selection of secure cache designs since the publication of the Prime+Probe attack. The type is ‘R’andomized, ‘P’artitioned, or ‘O’ther. ‘Predecessor’ is loosely defined as prior work that the follow-up is either built on, or related to. Works where the targeted level is marked with * were evaluated on that level but claim broader applicability.

frequently accompanied by a rekeying mechanism to further diminish the predictability of the mapping. Many of them build on the idea of a skewed cache [157], where each way in a set is determined from the address with a different mapping function, such that two addresses that share a way in one bank do not necessarily share ways in other banks.

RPcache [183] uses pre-computed permutation tables that serve as a layer of indirection between the address and the cache set. Whenever a miss in a protected domain would evict a different domain's cache line, a random different set is selected instead and the table is updated. This removes predictable interference between victim and attacker lines. **NewCache** [182] extends the *RPcache* idea with a remapping table that can map an address to any line in the cache, creating a fully associative cache with random replacement. **MIRAGE** [147] picks up the idea of a fully associative cache design many years later and uses a randomization-based directory to achieve this for the LLC, arguing the impracticability of prior designs for very large caches. The design keeps the notion of sets for its tag storage that provides the indirection to the data, but uses a randomization-based skew to address them. On a miss, victims are chosen from all available cache lines, hiding the association between the evicted and evicting line's addresses. The **Maya Cache** [18] similarly argues that at 20 %, *MIRAGE's* design requires too much storage overhead for tags. They assert that most lines that enter the LLC are never reused, and that it is therefore more efficient to reduce the overall data storage in favor of the tag store and lower power consumption. **RECAST** [198] also approximates a fully associative cache, but does so by storing a per-cache-line secret in the L1 cache that is used to derive a private LLC set index. Whenever a line is evicted from the L1 cache and later reloaded, the secret is recalculated, which changes the mapping of the address to the LLC set.

In 2018, Qureshi proposed **CEASER** [138], a last-level cache design that encrypts the address with a low-latency block cipher and uses this encrypted address to derive the set index. While this alone breaks the predictable association between addresses and their cache sets, it is also combined with periodic rekeying, intended to make any information an attacker may have learned about the mapping useless. In 2019, **Scatter Cache** and **CEASER-S**, a direct follow-up to *CEASER*, concurrently proposed two very similar desings based on the idea of cryptographically constructing cache sets on the fly for each security domain. Where *CEASER-S* proposes a skew [157] with a certain number of set partitions (e.g., $P = 2$), such that a set is made up of P static partitions, *Scatter Cache* skews all individual ways (e.g., $P = 16$ for a 16-way configuration). Both designs add rekeying to deny attackers the time to build functional eviction sets. They are closely followed by **PhantomCache** [163], which randomizes sets in a similar manner. Following the publication of new attacks on randomized caches (see Section 3.1.4), we developed our own follow-up to *ScatterCache*. **SassCache** (Chapter 7) contributes a novel mixture of set randomization and randomization-based partitioning to the state of the art. By adding a second low-latency encryption layer to the index derivation function, *SassCache* creates unique partitions for each security domain. Like in *ScatterCache*, each address is mapped to a virtually unique set of ways, of which a cryptographically random subset will be extremely unlikely to fully overlap with any other security domain. This allows security critical accesses to quickly 'hide' in one of those ways after only one or two targeted evictions by an attacker. Contrary to earlier designs, *SassCache* forgoes rekeying in favor of an appropriately chosen security parameter. **Chameleon Cache** [169] also builds on earlier skewed randomized caches, but tries to overcome their weaknesses by adding a small, fully associative victim cache to the randomized LLC. This hides conflict evictions in the randomized cache, as evictions are still in the victim cache. **ClepsydraCache** [166] is a hybrid design that combines index randomization with a time-to-live (TTL) function for each cache line. This time component introduces 'decay', meant to counter state-of-the-art attacks on randomized caches (see Section 3.1.4). Whenever a new line is loaded, it is assigned a random TTL within certain bounds. On access, the TTL is refreshed to a new random value, and the line is evicted when the TTL expires. The TTL is regularly reduced by a dynamic amount related to the current rate of conflict misses in the cache. In contrast to the trend of skewed randomized designs, Song et al. [160] argue that all contemporary algorithms for eviction set generation can be overcome by returning to simple randomization of the cache set index with a secure single-cycle hash function, frequent rekeying, and an attack detector that triggers further rekeyings.

Instead of contention-based attacks like Prime+Probe, **Random Fill Cache** [112] targets internal-collision attacks and shared-memory attacks. It introduces new instructions that allow a secure application to request loads that do not cache the target address, but a random address in its vicinity. This largely preserves performance, while decorrelating address loads from the resulting cache patterns.

3.1.3 Other Secure Cache Designs

Some designs approach the problem from entirely different directions, choosing neither randomization nor partitioning.

SHARP [191] and **RIC** [87] both follow a similar idea for preventing cross-core cache attacks via the LLC. They identify inclusion victims, *i.e.*, cache lines that are evicted from private caches because the are evicted from the LLC, as a necessity for both Prime+Probe- and Flush+Reload-style attacks on inclusive cache architectures. *SHARP* modifies the replacement policy to prioritize the replacement lines that do not have copies in any private cache, and if none exist, those that only have a copy in the core causing the eviction. *RIC*, instead *relaxes* the inclusion property of the LLC for read-only lines, with the reasoning that those need not be considered for coherence, as they cannot change anyway. Thus, all read-only memory can be evicted from the LLC without being evicted from private caches.

First Time Miss [142] (FTM) specifically prevents re-accesses in shared memory, e.g., during Flush+Reload, from revealing information about the LLC state of the victim. For the first time that a core loads a line, *FTM* guarantees that it behaves like a miss for that core, even it was already cached in the LLC by a different core. Thus, an attacker cannot tell whether an address was loaded earlier by the victim. This requires that different security domains are isolated to their own cores. **TimeCache** [129] builds on this idea and extends the granularity by removing the limitation that only cores are tracked as domains. Instead, any process with a different address space is treated as a different domain. Hyperthreading and context-switching are supported by implementing a software-assisted scheme that associates the timestamp of context-switches with the processes cache state.

3.1.4 Analyses of Secure Caches

As the number of secure cache designs has steadily grown over the years, so has the number of publications analyzing the security of the proposed designs. Already in 2008, Kong et al. [96] analyze *PLCache* and *RPcache*. They point out that the designs are at least still vulnerable to shared-memory and cache-collision attacks. He and Lee [60] confirm their findings and examine five more secure caches, including *NoMo*, *NewCache* and *Random Fill Cache*. They find that all are vulnerable to at least one of their four types of attacks (Evict+Time, Prime+Probe, Flush+Reload, and internal collisions), with *Random Fill Cache* being the only design to resist both Flush+Reload and internal-collision attacks. In a large survey [35], Deng et al. test the leakage of 17 secure cache designs in the face of known cache attack primitives and new variants they propose. While virtually all tested designs are vulnerable to *some* variant of an attack, *Catalyst* is a notable standout. This is because it essentially turns parts of the cache into software-controlled memory regions that never leave the cache, at the cost of flexibility and generalized protection of all memory. Targeting only CEASER(S), *Brutus* [20] effectively demonstrates the dangers of custom cryptographic functions on the example of *CEASER* and *CEASER-S*. It shows that the low-latency block cipher in *CEASER* does not change the relationship between the locations of congruent addresses across rekeying periods. With *CaSA* [22], Bourgeat et al. present an analysis framework specifically for randomization-based secure caches. They argue that some prior designs considered a threat model that was too narrow, and specifically that *CEASER-S* and *ScatterCache* are vulnerable to attacks that do not rely on building high-probability eviction sets, but instead repeatedly use low-probability sets to accumulate information. Of the four tested designs, only *NewCache* is not vulnerable to their attacks due to its conflict-based address remapping. Almost concurrently, Purnal et al. [135] presented a unified model for analyzing the security of *ScatterCache*, *CEASER-S* and standard set-associative caches. Based on this, they show that the state-of-the-art eviction-set-finding algorithms assumed by *ScatterCache* and *CEASER-S* can be improved by orders of magnitude (see Prime+Prune+Probe, Section 3.2), significantly weakening the security assumptions of both designs. Song et al. [159] also analyze *ScatterCache* and *CEASER-S*, coming to similar conclusions. They suggest improvements to the remapping period for designs these designs or active attack detection methods for simpler designs like *CEASER*. Genkin et al. propose *CacheFX* [47], a cache attack analysis framework to measure the entropy introduced by each memory access, the difficulty of building eviction sets and protection against cryptographic attacks. They evaluate *PLcache*, *CEASER*, *CEASER-S*, *ScatterCache*, *PhantomCache* and *NewCache* and find that all designs are susceptible to some attacks. Ramkrishnan et al. [143] identify a potential problem with randomized designs that implement domain-fusion for shared memory without full operating system support. When the OS and hardware design do not support creating new domains for shared, writeable memory, the domains have to be fused to maintain cache coherence, thereby greatly reducing the security the design provides. Hence, they propose a new coherence protocol that avoids the need for domain fusion. In a recent analysis [30] of randomized cache designs Chakraborty et al. examine their resistance against the often overlooked occupancy channel. They find that among the analyzed designs (*CEASER*, *CEASER-S*, *ScatterCache*, *MIRAGE* and *SassCache*), *SassCache* is the only design that consistently resists all occupancy attacks, while *MIRAGE* proves to be quite vulnerable to a novel AES key recovery attack. In a preprint SoK [17] on the contributing security attributes of secure cache designs, Bhatla et al. come to a similar conclusion w.r.t. occupancy attacks. Further, they find that skewing is one of the most promising attributes secure caches can have, and that high associativity of 64 or even 128 offers similar security against conflict-based attacks as designs like ScatterCache or SassCache.

3.2 Cache Attacks

Beyond the basic cache attack types (see Section 2.2.3), new and refined attacks are still being actively researched. For set-contention based attacks like Prime+Probe, one avenue of improvement is the generation of eviction sets. When physical (and sometimes even virtual) addresses are not available, or the set indexing or slicing function of a cache are unknown, eviction sets can only be constructed by testing addresses for eviction against each other.

An early method for constructing eviction sets without knowledge of cache indexing is described by several works [113, 130]. It starts with an array large enough to fill the cache and then iteratively removes one address at a time, testing if the remaining addresses combined can still evict a target address. If an address is necessary to evict the target, it is added to the eviction set. This method finds an eviction set in $\mathcal{O}(n^2)$, where n is the number of starting addresses (typically enough to cover the entire cache). Concurrently, Qureshi [139] as well as Vila et al. [176] propose a method (named *Group Elimination Method* (*GEM*) by Qureshi) that removes the quadratic dependency on the initial set size, by eliminating entire groups of addresses at once. They reason that in a w-way associative cache, it must always be possible to divide a set of addresses $> w$ into $w+1$ groups, such that at least one group can be removed while the rest still evicts a target address. Removing $1/w+1$ of the addresses at once, *GEM* improves set-finding to $\mathcal{O}(wn)$ [139]. In our work on cache attacks in WebGPU (Chapter 6), we develop a parallelized version of *GEM*, suitable for GPUs with LRU replacement. In a pre-processing step, we first divide a large chunk of memory into smaller buckets that do not interfere with each other in the cache. We then run an optimized version of *GEM* on each bucket in parallel. Using the LRU replacement properties, we can extract many complete eviction sets from a single run of *GEM* on a bucket, significantly speeding up the process. Our method results in eviction sets for *all* cache sets, recovering a set every 28 ms on average on an Nvidia RTX 3080. With *Prime+Prune+Probe* [135] we improve on prior work by introducing a pruning step that ensures that the entire starting set of addresses fits into the cache and ideally fills it. By accessing the target address next and then measuring the access times of all following accesses to the pruned set, a cache with LRU replacement will generate a cascade of exactly w misses in ideal conditions, reducing the runtime to $\mathcal{O}(n)$. *Conflict Testing with Probe+Prune* ()*CTPP*) [189] finds that while Prime+Prune+Probe is fast, its success rate is not very high. They combine it with a pre-processing step that produces an optimal starting set for Prime+Prune+Probe. Similar to our work on GPUs, *Prune+PlumTree* [88] finds eviction sets for a large part of the cache in bulk instead of one at a time. Also building on Prime+ Prune+Probe, their method reduces the factor of the number of sets s in the runtime to $log(s)$, resulting in $\mathcal{O}(n \log(s))$. Combined with other optimizations, they are able to find 98 % of eviction sets in 40 ms to 60 ms.

Another avenue of improvement for cache attacks is to tailor attacks to the replacement policy of the cache. *Reload+Refresh* [25] is a shared-memory LLC attack that uses detailed knowledge of the QLRU [1] replacement policy in 4th-8th generation Intel processors to decrease the detectability of the attack. Before each measurement, a target address is placed in the cache deliberately made the eviction candidate by filling its set with other addresses. When the attacker later adds another address into the target address' set, they can measure if it was evicted or not, and thereby learn if the victim accessed the target. Contrary to Flush+Reload, the victim will not experience a miss when accessing it. *Prime+Scope* [136] is an improvement for cross-core Prime+Probe. It reduces the attack to a single cache line access per measurement, vastly improving the time resolution over Prime+Probe. A targeted cache set is primed such that a chosen *scope* line is made the eviction candidate, *and* accesses to it do not change the LLC set state because they are served by the L1 cache. Accesses to this line can then be performed quickly and repeatedly without a prime period in between that would constitute a blind spot for the attacker. A victim access is detected when the scope line is evicted through the cache's coherence mechanism.

Lastly, some publications have found new attack vectors in different hardware mechanisms. *Prime+Abort* [40] leverages Intel's transactional synchronization extensions (TSX) [75] to be architecturally notified when cache lines of interest leave the cache, creating timer-free Prime+Probe variants. TSX creates sections of code that do not architecturally commit their results until the entire transaction is completed. When a transaction performs data writes that are later evicted from the L1 cache, or data reads that are later evicted from the L3 cache, the transaction is aborted. *Prime+Abort* uses this to *prime* sets within a transaction and be notified by TSX with an abort when a victim program accesses the same set. *Synchronization Storage Channels* [195] similarly shows timer-less cache attacks on the Apple M1 using its hardware synchronization instructions. They use the `ldrex` and `strex` instructions to monitor a single address for eviction in the attacker's L1 cache, which they leverage into a Prime+ Probe-style attack by building a LLC eviction set around it that makes it the eviction candidate. Yan et al. [192] find that the cache directory is an often overlooked attack vector for caches, as it needs to contain information on all lines in the cache hierarchy to maintain coherence. They demonstrate that Prime+Probe attacks can also be mounted on the directory in non-inclusive caches. Another (ab-)usable hardware feature is the `cldemote` instruction, currently available on Intel server CPUs, which *demotes* targeted cache lines from the core towards the LLC, without flushing them. Rauscher et al. [144] demonstrate that this enables two new primitives (*Demote+Reload*, *Demote+Demote*) that are similar to Flush+Reload and Flush+Flush, but are faster and have lower blind spots than early attacks due to not evicting data to the slow DRAM. With *Cohere+Reload* (Chapter 5), we find a coherence based attack that targets AMD SEV. On recent Zen Server architectures, AMD provides automatic coherence between plaintext and ciphertext for its memory encryption feature. We find that this is implemented with a half-page granularity, though the 32 affected cache lines are not contiguous, but spread out configurable patterns. This means that *any* access to an encrypted cache line will check for the presence of any of 32 unencrypted cache lines on the same page and evict them. The same holds for accesses to the unencrypted lines evicting encrypted lines. We find that this provides an extremely high temporal resolution with virtually no blind spot, enabling high-resolution access traces for victim CVMs across cores or even sockets. In concurrent work, the same attack is presented under the name *Reload+Reload* [32]. A similar attack was hypothesized to affect Intel TDX [5], though with single-line granularity.

3.3 Defenses Against Meltdown-Type Attacks

Contrary to speculation-related attacks like Spectre, Meltdown-type attacks (see Section 2.4.1) are not caused by intended behavior, and thus typically considered ‘bugs’ in the microarchitecture, even though they may arise because of performance optimizations. Consequently, definitive mitigations often involve hardware changes to the CPU that are not available until long after the initial discovery. Though microcode mitigations are sometimes possible, software mitigations are often the only immediate remedy, even if the performance impact is sometimes significant [100, 101, 103].

The first mitigation against the original Meltdown attack was *kernel page-table isolation* (KPTI). It was initially proposed under the name *KAISER* [53, 117] as a mitigation against kernel address leakage, a year before the Meltdown attack was known. It separates the kernel and user page table structure and exchanges them during context switches, such that the kernel-exclusive address space is not mapped into the user page tables at all. This prevents Meltdown, as the requested kernel address cannot be resolved in user space. On CPUs that support *process-context identifiers* (PCID), a feature that enables entries in the TLB to be tagged, speeds up this process [56] by obviating the need to flush the entire TLB when the page tables are swapped out. Later CPU generations were shipped with hardware or microcode mitigations to Meltdown and related attacks [68], making KPTI once again an optional feature. In a security bulletin [13] published in 2025, ARM announced a vulnerability related to prefetching that may lead to kernel data leakage, making KPTI the default mitigation on affected ARM CPUs [102].

KPTI, however, is ineffective against an attack like Foreshadow [170] (L1TF-SGX in Intel terminology) that targets SGX. In this threat model, the adversary controls the operating system, and thus the page tables. As a pure software solution, KPTI could therefore not be enforced, as this would be the role of the operating system. Foreshadow was initially addressed with a microcode update [78] that flushes the L1 cache whenever the enclave is exited. This does not prevent attacks on a sibling hyperthread however, as the L1 cache is shared between the two threads. The status of hyperthreading was therefore added to the SGX attestation process, allowing enclave software to ensure that hyperthreading is disabled. In the case of Foreshadow-NG [184] (L1TF-VMM), new microcode added a mechanism to flush the L1 cache, which a hypervisor can use on each VM entry to prevent the attack. Similar to SGX, the hypervisor also needs to ensure that only mutually trusted workloads are scheduled on the same physical core at a time to prevent attacks from a sibling thread.

LVI is also not mitigated by KPTI, since attackers can either be privileged (e.g., for attacks against SGX) and cause faults in the victim program to trigger a value injection, or use OS-caused faults to attack other user space programs. As values can be injected through all the same buffers that Meltdown-type attacks can leak from (see Section 2.4.1), simply clearing an L1 cache is not enough. The initially proposed mitigations against LVI were therefore to insert fencing instructions after every (vulnerable) load that stop any out-of-order execution past the fence. A simple, effective, yet also very costly [100] mitigation is the “speculative execution side effect suppression” compiler pass for LLVM [24] that fences *all* loads. Intel provided their own optimized compiler option that finds loads that are followed by gadgets that can leak their value and inserts fences only for those loads [69, 77]. In the next generation of CPUs after the publication of LVI, most value injection attacks were mitigated in hardware [68], and software mitigations were not necessary. The only edge case still unmitigated in the “Comet Lake” architecture was the forwarding of the value ‘0’ instead of any buffered value, termed “LVI-NULL” or “LVI zero data”. Aside from particular code gadgets that are vulnerable to null injection specifically, LVI-NULL is not generally exploitable outside of SGX enclaves, as it requires access to the victim’s memory. Within SGX, however, attackers can still inject arbitrary values with one level of indirection, the null page, which is outside of the enclave. Mitigating LVI-NULL particular attack without the overhead of existing mitigations is the goal of LVI-NULLify (Chapter 9). Since all null value injections into pointers now redirect loads to the null page, our mitigation brings the null page into the enclave. We achieve this by using segmentation (Section 2.1) to make all loads in an enclave relative to the GS segment start, which we place at the beginning of the enclave. By marking the first pages in the enclave non-executable and non-readable, any transiently redirected load stalls. With this mechanism, our mitigation provides strong protection against LVI-NULL while only incurring a small performance overhead compared to general LVI mitigations.

Aside from direct mitigations for concrete vulnerabilities, both vendors and academia have presented more general defenses in depth against transient execution attacks. The new Intel feature *linear address space separation* (LASS) [72] is on such defense against transient attacks that try to cross the boundary into kernel space. When enabled, it enforces the already existing convention that all linear addresses with a most-significant bit of 1 are kernel space addresses. Contrary to permissions in the page tables entries, this permission is encoded in the linear address and can thus be enforced earlier in an operation, preventing data leakage like Meltdown and other microarchitectural attacks like KASLR breaks [27, 28, 57, 65, 82, 98, 152] from the outset. As another defense in depth, our own work, *SMTCache* (see Chapter 8), can also thwart the Meltdown attacks loading from L1, as the kernel resides in a separate cache slice. Other academic works investigate the feasibility of making information of transient execution architecturally available to programs [116], completely blocking all side effects of speculative execution until commit [89], or more narrowly targeting the cache as an exfiltration mechanism by only allowing some (safe) loads [105, 148, 149], buffering speculative accesses to the cache hierarchy until after instruction commit [4, 54, 190] or undoing cache changes when an instruction is squashed [146].

4 Conclusion

In this thesis, we presented novel attacks on caches and hardware cache coherence mechanisms. The constant stream of new attacks on caches and their surroundings motivates research into mitigations against these attacks. In this direction, we developed secure cache designs and a software mitigation that prevents side-channel leakage. Based on the insights from this work, we draw the following two conclusions.

The field of secure caches is maturing, and the remaining open research questions are moving from specific security properties towards the practicality and compatibility with the existing real-world hardware-software ecosystem. Since the publication of CEASER-S [139] and ScatterCache [186] in 2019, about as many secure cache designs have been published as in the 15 years prior. In this time, the field has undergone large changes, spurred by many works critically evaluating prior designs [20, 22, 30, 35, 47, 135, 139, 159]. The state of the art has advanced to a point where for a given set of security criteria, building blocks to create a fitting design are almost certainly known. In our work on secure last-level caches (Chapter 7), for instance, we found that combining randomization with elements of partitioning can be a viable way to achieve strong security for Prime+Probe-style attacks as well as occupancy attacks, and recent work has confirmed these findings [30]. However, this design comes at an increased performance cost compared to other, less secure works. This tradeoff is one of the areas that will need to be investigated more concretely. In particular, we need to investigate what the most efficient ways to achieve different security guarantees are, and whether some design elements provide a significantly better security-to-performance ratio than others. While the most obvious and indeed the most frequent path is to trade security for performance, our secure L1 design (Chapter 8), for example, suggests that chip area is another possible tradeoff. In either case, it remains an open question how close performance, security and efficiency can ultimately be aligned, and, more importantly, how big a cost any particular use case can bear. For example, the interaction of secure cache designs with high-performance ecosystems that are reliant on specific hardware behaviors to optimize performance, e.g., structure alignment for optimal cache utilization, is currently unclear for most designs. Finally, the complexity of modern CPU designs has long been so high that formal verification of correctness in cache coherence is a hard problem [36], and newer work [197] shows that current CPUs can still be affected by coherence errors. While many proposed secure cache designs will undoubtedly be more difficult to verify, it is unclear if this is universally the case, or if some designs' coherence is only as complex as current cache designs, or even less.

Layered defenses create flexibility against yet unknown side-channel attacks. As discussed above, verification of correctness alone is a monumental task in today's highly complex processors. Precluding all side-channels in the design step is currently not feasible. However, as our software defense against LVI-NULL (Chapter 9) demonstrates, having a broad selection of hardware features to draw from can allow the creation of impromptu defenses when they are needed. Besides segmentation, a new feature that we will most likely provide significant defense in depth is Intel LASS [72], which cuts off many memory-related side channels, e.g., KASLR breaks, at the root. Our design, SMTCache (Chapter 8), similarly provides such defense in depth and can prevent Meltdown-like leakage from the L1. Intel CAT [70] further strengthens this point. Though not initially intended as a security feature, research [111] has demonstrated that it can be a powerful measure against cache attacks. If applied to GPUS, a either partitioning solution could also protect high-value GPU workloads from the Prime+Probe attacks demonstrated in Chapter 6.

References

[1] Andreas Abel and Jan Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In: ISPASS. 2020 (pp. 15, 35).

[2] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS. 2019 (p. 15).

[3] Jaeguk Ahn, Cheolgyu Jin, Jiho Kim, Minsoo Rhu, Yunsi Fei, David Kaeli, and John Kim. Trident: A Hybrid Correlation-Collision GPU Cache Timing Attack for AES Key Recovery. In: HPCA. 2021 (p. 6).

[4] Sam Ainsworth and Timothy M Jones. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In: arXiv:1911.08384 (2019) (p. 39).

[5] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Security Review. 2023. URL: https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf (p. 36).

[6] AMD. 5TH GEN AMD EPYC™ PROCESSOR ARCHITECTURE. 2025. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf (p. 17).

[7] AMD. AMD EPYC™ 9004 Series Architecture Overview. 2023. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/white-papers/58015-epyc-9004-tg-architecture-overview.pdf (p. 16).

[8] AMD. AMD Secure Encrypted Virtualization (SEV). 2024. URL: https://developer.amd.com/sev/ (pp. 4, 5, 19, 20).

[9] 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. 19, 20).

[10] AMD. AMD64 Architecture Programmer's Manual. 2024 (pp. 6, 20).

[11] ARM. Arm Confidential Compute Architecture. 2024. URL: `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture` (p. 19).

[12] ARM. TrustZone for Arm Cortex-M Processors. 2024. URL: `https://www.arm.com/technologies/trustzone-for-cortex-a` (pp. 4, 19).

[13] Arm. Arm CPU Security Bulletin: CVE-2024-7881. 2025. URL: `https://developer.arm.com/documentation/110326/latest` (p. 37).

[14] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In: PACT. 2013 (p. 28).

[15] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 3, 18).

[16] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In: International Conference on Information Technology: Coding and Computing (ITCC) (2005) (p. 18).

[17] Anubhav Bhatla, Hari Rohit Bhavsar, Sayandeep Saha, and Biswabandan Panda. SoK: So, You Think You Know All About Secure Randomized Caches? In: (2025). URL: `https://anubhavbhatla.github.io/assets/pdf/SoK.pdf` (pp. 4, 34).

[18] Anubhav Bhatla, Biswabandan Panda, et al. The Maya Cache: A Storage-efficient and Secure Fully-associative Last-level Cache. In: ISCA. 2024 (pp. 29, 30).

[19] 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. 4).

[20] Rahul Bodduna, Vinod Ganesan, Patanjali Slpsk, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. In: IEEE Computer Architecture Letters 19.1 (2020), pp. 9–12 (pp. 4, 33, 41).

[21] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In: CHES. 2006 (pp. 3, 18).

[22] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In: MICRO. 2020 (pp. 4, 33, 41).

[23] Bramley, Jacob. Page Colouring on ARMv6 (and a bit on ARMv7). 2013. URL: `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/page-colouring-on-armv6-and-a-bit-on-armv7` (p. 14).

[24] Zola Bridges. LLVM SESES pass for LVI. 2020. URL: `https://reviews.llvm.org/D75939` (pp. 8, 38).

[25] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In: USENIX Security. 2020 (pp. 4, 35).

[26] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In: USENIX Security. 2003 (p. 3).

[27] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (pp. 4, 8, 22, 39).

[28] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 9, 39).

[29] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security. 2019 (pp. 4, 22).

[30] Anirban Chakraborty, Nimish Mishra, Sayandeep Saha, Sarani Bhattacharya, and Debdeep Mukhopadhyay. Systematic Evaluation of Randomized Cache Designs against Cache Occupancy. In: USENIX Security. 2025 (pp. 4, 18, 33, 41).

[31] 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 (pp. 4, 5).

[32] Li-Chung Chiang and Shih-Wei Li. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In: ASPLOS. 2025 (p. 36).

[33] Confidential Computing Consortium. A Technical Analysis of Confidential Computing. 2022 (p. 19).

[34] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (p. 19).

[35] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Analysis of Secure Caches using a Three-Step Model for Timing-Based Attacks. In: Journal of Hardware and Systems Security 3.4 (2019), pp. 397–425 (pp. 4, 32, 41).

[36] Andrew DeOrio, Adam Bauserman, and Valeria Bertacco. Post-Silicon Verification for Cache Coherence. In: IEEE International Conference on Computer Design. 2008 (p. 42).

[37] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In: USENIX Security. 2019 (pp. 27, 29).

[38] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Chunked-cache: On-demand and scalable cache isolation for security architectures. In: NDSS. 2022 (pp. 27, 29).

[39] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In: International Conference on Smart Card Research and Advanced Applications. Springer. 1998 (p. 3).

[40] 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. 35).

[41] 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) 8.4 (2011) (pp. 26, 29).

[42] Ulrich Drepper. Elf Handling for Thread-Local Storage. Tech. rep. 2013. URL: `https://www.akkadia.org/drepper/tls.pdf` (p. 12).

[43] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. In: arXiv:1712.05090 (2017) (p. 20).

[44] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems. In: ISCA. 2021 (p. 6).

[45] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael B. Abu-Ghazaleh, Andres Marquez, and Kevin J. Barker. Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems. In: ISCA. 2022 (p. 6).

[46] 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. 5).

[47] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. CacheFX: A Framework for Evaluating Cache Security. In: arXiv:2201.11377 (2022) (pp. 4, 33, 41).

[48] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In: CHES. 2015 (p. 3).

[49] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In: CRYPTO. 2014 (p. 3).

[50] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. Efficient and generic microarchitectural hash-function recovery. In: S&P. 2024 (p. 16).

[51] Lukas Giner, Roland Czerny, Simon Lammer, Aaron Giner, Paul Gollob, Jonas Juffinger, and Daniel Gruss. Fast and Efficient Secure L1 Caches for SMT. In: ARES. 2025 (p. 29).

[52] 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. 29).

[53] Thomas Gleixner. x86/kpti: Kernel Page Table Isolation (was KAISER). 2017. URL: `https://lkml.org/lkml/2017/12/4/709` (p. 37).

[54] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture. In: Third Workshop on Computer Architecture Research with RISC-V (CARRV). 2019 (p. 39).

[55] 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. 9).

[56] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (p. 37).

[57] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (p. 39).

[58] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 4, 19).

[59] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (p. 19).

[60] Zecheng He and Ruby B Lee. How secure is your cache against side-channel attacks? In: MICRO. 2017 (p. 32).

[61] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In: ACM SIGPLAN Notices 52.7 (2017), pp. 129–142 (p. 20).

[62] Zhang Hongxin, Huang Yuewang, Wang Jianxin, Lu Yinghua, and Zhang Jinling. Recognition of electro-magnetic leakage information from computer radiation with SVM. In: Computers & Security 28.1-2 (2009), pp. 72–76 (p. 3).

[63] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (p. 4).

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

[65] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (pp. 9, 16, 39).

[66] 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. 5).

[67] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. In: Cryptology ePrint Archive, Report 2015/898 (2015) (p. 16).

[68] Intel. Affected Processors: Transient Execution Attacks. 2020. URL: `https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model` (pp. 8, 37, 38).

[69] Intel. An Optimized Mitigation Approach for Load Value Injection. 2020. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/optimized-mitigation-approach-load-value-injection.html` (pp. 8, 38).

[70] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Cache. 2015. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf` (pp. 26, 42).

[71] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2023 (pp. 15, 16).

[72] Intel. Intel Architecture Instruction Set Extensions and Future Features. 2022 (pp. 38, 42).

[73] 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. 4, 19).

[74] Intel. Intel Trust Domain Extensions. 2021. URL: `https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf` (pp. 4, 19).

[75] Intel. Intel® Transactional Synchronization Extensions (Intel® TSX) Memory and Performance Monitoring Update for Intel® Processors. 2024. URL: `https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html` (p. 36).

[76] Intel. Intel® Xeon® Processor Scalable Family Technical Overview. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html (p. 17).

[77] Intel. Load Value Injection. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/load-value-injection.html (pp. 22, 38).

[78] Intel. Q3 2018 Speculative Execution Side Channel Update. 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html (p. 37).

[79] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In: Euromicro Conference on Digital System Design. 2015 (p. 16).

[80] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge. In: IEEE Hot Chips Symposium (HCS). 2012 (p. 15).

[81] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In: ACM SIGARCH Computer Architecture News 38.3 (2010), pp. 60–71 (p. 15).

[82] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (p. 39).

[83] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In: HPCA. 2016 (p. 6).

[84] Toni Juan, Dolors Royo, and Juan J Navarro. Dynamic Cache Splitting. In: International Conference of the Chilean Computer Science Society (1995) (p. 25).

[85] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. 2016 (p. 20).

[86] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In: USENIX Security. 2023 (p. 4).

[87] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In: Design Automation Conference. 2017 (pp. 29, 32).

[88] Tom Kessous and Niv Gilboa. Prune+PlumTree - Finding Eviction Sets at Scale. In: S&P. 2024 (p. 35).

[89] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: DAC. 2019 (p. 39).

[90] Taehun Kim and Youngjoo Shin. ThermalBleed: A Practical Thermal Side-Channel Attack. In: IEEE Access 10 (2022), pp. 25718–25731 (p. 3).

[91] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: MICRO. 2018 (pp. 27, 29).

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

[93] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 4, 8, 21, 23).

[94] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (p. 3).

[95] 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. 3, 9).

[96] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: Computer Security Architectures Workshop (CSAW) (2008), p. 25 (p. 32).

[97] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (p. 4).

[98] 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. 9, 39).

[99] Butler W Lampson. A note on the confinement problem. In: Communications of the ACM (1973) (p. 3).

[100] Michael Larabel. Google Engineer Shows “SESES” For Mitigating LVI + Side-Channel Attacks. 2020. URL: `https://www.phoronix.com/scan.php?page=news_item&px=LLVM-SESES-Mitigating-LVI-More` (pp. 37, 38).

[101] Michael Larabel. The Brutal Performance Impact From Mitigating The LVI Vulnerability. 2020. URL: `https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf` (p. 37).

[102] Larabel, Michael. Arm Changing Linux Default To Costly ”KPTI” Mitigation For Some Newer CPUs. 2025. URL: `https://www.phoronix.com/news/Arm-Linux-CVE-2024-7881-KPTI` (p. 37).

[103] Larabel, Michael. The Current Spectre / Meltdown Mitigation Overhead Benchmarks On Linux 5.0. 2019. URL: `https://www.phoronix.com/review/linux50-spectre-meltdown` (p. 37).

[104] 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. 5).

[105] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In: HPCA. 2019 (p. 39).

[106] Qianru Liao, Yongzhi Huang, Yandao Huang, Yuheng Zhong, Huitong Jin, and Kaishun Wu. MagEar: eavesdropping via audio recovery using magnetic side channel. In: MobiSys. 2022 (p. 3).

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

[108] 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. 3, 9).

[109] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security. 2018 (pp. 4, 8, 21, 22).

[110] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency Throttling Side-Channel Attack. In: CCS. 2022 (pp. 3, 9).

[111] 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 (pp. 26, 29, 42).

[112] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In: MICRO. 2014 (pp. 29, 31).

[113] 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. 16, 18, 34).

[114] EntryBleed: A Universal KASLR Bypass against KPTI on Linux. 2023 (p. 9).

[115] Zhuoran Liu, Niels Samwel, Leo Weissbart, Zhengyu Zhao, Dirk Lauret, Lejla Batina, and Martha Larson. Screen gleaning: A screen reading TEMPEST attack on mobile devices exploiting an electromagnetic side channel. In: NDSS. 2021 (p. 3).

[116] Jason Lowe-Power, Venkatesh Akella, Matthew K Farrens, Samuel T King, and Christopher J Nitta. Position Paper: A case for exposing extra-architectural state in the ISA. In: HASP. 2018 (p. 39).

[117] LWN. The current state of kernel page-table isolation. 2017. URL: `https://lwn.net/Articles/741878/` (pp. 5, 37).

[118] Lukas Maar, Lukas Giner, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025 (p. 9).

[119] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (p. 4).

[120] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015 (p. 16).

[121] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (p. 5).

[122] Daniel Moghimi. Downfall: Exploiting Speculative Data Gathering. In: USENIX Security. 2023 (pp. 4, 22).

[123] Baker Mohammad. Embedded Memory Design for Multi-Core and Systems on Chip. Vol. 116. Analog Circuits and Signal Processing. Springer, 2014 (p. 13).

[124] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In: EuroSec. 2018 (p. 20).

[125] Bradley Morgan, Gal Horowitz, Sioli O'Connell, Stephan van Schaik, Chitchanok Chuengsatiansup, Daniel Genkin, Olaf Maennel, Paul Montague, Eyal Ronen, and Yuval Yarom. Slice+Slice Baby: Generating Last-Level Cache Eviction Sets in the Blink of an Eye. In: S&P. 2025 (p. 16).

[126] 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. 15).

[127] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. Springer Nature, 2020 (pp. 16, 17).

[128] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael B. Abu-Ghazaleh. Constructing and characterizing covert channels on GPGPUs. In: MICRO. 2017 (p. 6).

[129] Divya Ojha and Sandhya Dwarkadas. TimeCache: Using Time to Eliminate Cache Side Channels when Sharing Software. In: ISCA. 2021 (pp. 29, 32).

[130] 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 (pp. 18, 34).

[131] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 3, 4, 18).

[132] Dan Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. In: Cryptology ePrint Archive, Report 2005/280 (2005) (p. 25).

[133] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. In: Cryptology ePrint Archive, Report 2002/169 (2002) (pp. 3, 18).

[134] Colin Percival. Cache Missing for Fun and Profit. In: BSDCan. 2005 (pp. 4, 18).

[135] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In: S&P. 2021 (pp. 4, 6, 9, 33, 35, 41).

[136] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (pp. 4, 35).

[137] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. In: arXiv:1908.03383 (2019) (p. 9).

[138] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: MICRO. 2018 (pp. 29, 30).

[139] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In: ISCA. 2019 (pp. 4, 7, 18, 29, 34, 41).

[140] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In: ACM SIGARCH Computer Architecture News 35.2 (2007), p. 381 (p. 15).

[141] Mikka Rainer, Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, Leon Trampert, Lukas Gerlach, and Michael Schwarz. Rapid Reversing of Non-Linear CPU Cache Slice Functions: Unlocking Physical Address Leakage. In: S&P. 2025 (p. 16).

[142] Kartik Ramkrishnan, Stephen McCamant, Pen Chung Yew, and Antonia Zhai. First Time Miss: Low Overhead Mitigation for Shared Memory Cache Side Channels. In: Conference on Parallel Processing. 2020 (pp. 29, 32).

[143] Kartik Ramkrishnan, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Non-Fusion Based Coherent Cache Randomization Using Cross-Domain Accesses. In: Asia CCS. 2024 (p. 33).

[144] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025 (p. 36).

[145] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin Qureshi. Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. In: SEED. IEEE. 2021 (pp. 27, 29).

[146] Gururaj Saileshwar and Moinuddin K Qureshi. CleanupSpec: An "Undo" Approach to Safe Speculation. In: MICRO. 2019 (p. 39).

[147] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security. 2021 (pp. 29, 30).

[148] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. Ghost loads: what is the cost of invisible speculation? In: International Conference on Computing Frontiers. 2019 (p. 39).

[149] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In: ISCA. 2019 (p. 39).

[150] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In: ISCA. 2011 (pp. 26, 29).

[151] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (pp. 4, 5, 8, 22).

[152] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725. 2019 (p. 39).

[153] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 4, 5, 8, 22).

[154] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (pp. 4, 22).

[155] Brian C. Schwedock and Nathan Beckmann. Jumanji: The Case for Dynamic NUCA in the Datacenter. In: MICRO. 2020 (pp. 28, 29).

[156] Mark Seaborn. L3 cache mapping on Sandy Bridge CPUs. Apr. 2015. URL: `http://lackingrhoticity.blogspot.com/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html` (p. 16).

[157] André Seznec. A case for two-way skewed-associative caches. In: ACM Computer Architecture News (1993) (pp. 25, 29, 30).

[158] 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. 18).

[159] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In: S&P. 2021 (pp. 4, 33, 41).

[160] Wei Song, Zihan Xue, Jinchi Han, Zhenzhen Li, and Peng Liu. Randomizing Set-Associative Caches Against Conflict-Based Cache Side-Channel Attacks. In: IEEE Transactions on Computers 73.4 (2024), pp. 1019–1033 (pp. 29, 31).

[161] Gerson de Souza Faria and Hae Yong Kim. Differential audio analysis: a new side-channel attack on PIN pads. In: International Journal of Information Security 18 (2019), pp. 73–84 (p. 3).

[162] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (pp. 4, 22).

[163] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In: NDSS. 2020 (pp. 7, 29, 30).

[164] Mutaz Al-Tarawneh. An Investigation of the Impact of Instruction Cache (I-Cache) Organization on Power-Performance Trade-Offs in the Design of Scalar Processors. In: European Journal of Scientific Research 115 (Nov. 2013), pp. 7–26 (p. 15).

[165] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In: USENIX Security. 2022 (p. 9).

[166] Jan Philipp Thoma, Christian Niesler, Dominic Funke, Gregor Leander, Pierre Mayr, Nils Pohl, Lucas Davi, and Tim Güneysu. ClepsydraCache – Preventing Cache Attacks with Time-Based Evictions. In: USENIX Security. 2023 (pp. 29, 31).

[167] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oğuz Ergin. Composable Cachelets: Protecting Enclaves from Cache {Side-Channel} Attacks. In: USENIX Security. 2022, pp. 2839–2856 (pp. 27, 29).

[168] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In: CHES. 2003 (pp. 3, 18).

[169] Thomas Unterluggauer, Austin Harris, Scott Constable, Fangfei Liu, and Carlos Rozas. Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks. In: IEEE International Symposium on Secure and Private Execution Environment Design (SEED). IEEE. 2022 (pp. 29, 31).

[170] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security. 2018 (pp. 4, 5, 8, 22, 37).

[171] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (pp. 3, 5, 8, 22, 23).

[172] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 5).

[173] 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. 5).

[174] 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. 5).

[175] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? In: Computers & Security 4.4 (1985), pp. 269–286 (p. 3).

[176] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In: S&P. 2019 (pp. 18, 34).

[177] W3C. WebGPU. 2023. URL: `https://www.w3.org/TR/webgpu` (p. 6).

[178] W3C. WebGPU - W3C Working Draft - Timing attacks. 2023. URL: `https://www.w3.org/TR/webgpu/#security-timing` (p. 6).

[179] Jack Wampler, Ian Martiny, and Eric Wustrow. ExSpectre: Hiding Malware in Speculative Execution. In: NDSS. 2019 (p. 4).

[180] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In: Design Automation Conference (DAC). 2016 (pp. 27, 29).

[181] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In: USENIX Security. 2022 (pp. 3, 9).

[182] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In: MICRO. 2008 (pp. 29, 30).

[183] 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 (pp. 25, 26, 29, 30).

[184] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf` (pp. 4, 8, 37).

[185] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In: AsiaCCS. 2019 (p. 20).

[186] 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 (pp. 6, 9, 29, 41).

[187] 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. 20).

[188] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (p. 5).

[189] Zihan Xue, Jinchi Han, and Wei Song. CTPP: A fast and Stealth Algorithm for Searching Eviction Sets on Intel Processors. In: RAID. 2023 (p. 35).

[190] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: MICRO. 2018 (p. 39).

[191] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In: ISCA. 2017 (pp. 29, 32).

[192] 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 (pp. 16, 18, 36).

[193] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 4, 18).

[194] 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. 16).

[195] 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 (p. 36).

[196] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. In: ACM SIGPLAN Notices 50.4 (2015), pp. 503–516 (pp. 27, 29).

[197] 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. 42).

[198] Xingjian Zhang, Haochen Gong, Rui Chang, and Yajin Zhou. RECAST: Mitigating Conflict-Based Cache Attacks Through Fine-Grained Dynamic Mapping. In: IEEE Transactions on Information Forensics and Security (2024) (pp. 29, 30).

[199] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes. In: ASPLOS. 2023 (p. 28).

Part II

Publications

List of Publications

During my PhD, I contributed to 9 publications, 8 of which appeared in conference proceedings, and 5 of which are included in this thesis as my first-author publications, as shown below.

Publications in this Thesis

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

2. **Lukas Giner**, Sudheendra Raghav Neela, and Daniel Gruss. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In: DIMVA. 2025.

3. **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.

4. **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.

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

Other Contributions

1. Lukas Maar, **Lukas Giner**, Daniel Gruss, and Stefan Mangard. When Good Kernel Defenses Go Bad: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks. In: USENIX Security. 2025.

2. 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.

3. Claudio Canella, Andreas Kogler, **Lukas Giner**, Daniel Gruss, and Michael Schwarz. Domain Page-Table Isolation. In: arXiv:2111.10876. 2021.

4. Antoon Purnal, **Lukas Giner**, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In: S&P. 2021.

5. 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.

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

7. 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.

8. 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.

5

Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP

Publication Data

Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In: DIMVA. 2025

Contributions

Main author.

Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP

Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss

Graz University of Technology

Abstract

Confidential computing platforms, e.g., AMD SEV-SNP, allow running mutually distrusting workloads on the same hardware with the protection of several isolation mechanisms: data is encrypted in RAM, and access to unencrypted data is architecturally prevented. Furthermore, access and cache line operations are restricted, mitigating attacks like Flush+Reload. The hypervisor can access the encrypted data of virtual machines, e.g., for migration purposes. This creates a coherency challenge around modifications between encrypted and decrypted cache lines. AMD enforces coherency between these two cache lines by removing one when the other is *accessed*.

In this paper, we present Cohere+Reload, a novel side-channel attack exploiting AMD's coherency for encrypted memory. We discover two types of leakage in the coherency mechanism: First, coherence conflicts leak victim operations on a spatial granularity of a 2 kB block. Second, the timing correlates with number and location of accesses the victim performed within the confidential virtual machine, allowing to infer how often or where within a coherence partition victim accesses were performed, with a maximum spatial resolution of 256 bytes. We evaluate Cohere+Reload in two synthetic and two real-world attacks: In synthetic attacks, we demonstrate that Cohere+Reload can observe the control flow and access locations in workloads within a confidential virtual machine. We present a real-world attack on mbedTLS RSA, leaking 4096 key bits in a single-trace attack, with 99.7 % of bits correct. We present another real-world attack on OpenSSL AES exploiting disalignments on a cache line granularity: In a first round T-table attack we achieve an accuracy of 100 % in only 1500 encryptions and with a novel correlation attack an accuracy of 92.81 % in 12000 encryptions. We conclude that the coherence approach for AMD SEV-SNP should be re-evaluated and discuss further potential mitigations.

1 Introduction

Modern processors have a multi-layered memory hierarchy for data, including code. Data can reside in registers, in cache lines in L1, L2, or L3 cache, or in the RAM. Some processors have even further cache layers, e.g., an L4 cache. While caches are crucial for the performance of modern computers, they also inherently introduce timing side channels that distinguish cached from non-cached data.

The most widely known attacks are Prime+Probe [47] and Flush+ Reload [42]. Flush+Reload [42] works by constantly flushing a cache line from the cache, using the processor's flush instruction, and measuring how long it takes to reload the cache line. Flushing a cache line requires read access to the memory, e.g., read-only shared memory with the victim, which is typically not available across virtual machines or in the context of confidential computing. Prime+Probe does not require shared memory. Prime+Probe [47] works by measuring how much time it takes to constantly re-fill a specific cache set. If a victim access falls into the same cache set, the timing increases.

Confidential computing is an emerging compute paradigm where a confidential workload, running isolated inside a virtual machine, is isolated from all other workloads and from the host. More specifically, it is part of the threat model that the hypervisor can be malicious or compromised but the confidential virtual machine remains secure. Confidential computing platforms, e.g., Intel TDX and AMD SEV-SNP, still share the underlying hardware across mutually distrusting workloads running in virtual machines. However, vendors introduced several isolation mechanisms to protect workloads: For instance, data is encrypted in RAM, decrypted on-the-fly when moved into the caches, and the processor prevents direct access to unencrypted data in caches or registers. The hypervisor cannot access unencrypted memory of the confidential virtual machine. Furthermore, cache line operations are restricted, mitigating attacks like Flush+ Reload.

Despite the strong isolation, some functionality requires access from the hypervisor to the encrypted data, e.g., migration of virtual machines in the cloud. Consequently, the hypervisor can access the encrypted data of virtual machines. However, this implies that data can be in the caches twice: once encrypted for the host, and once unencrypted for the confidential virtual machine. Clearly, this creates a coherency challenge as a virtual machine may modify a cache line, *i.e.*, the cache contains a modified unencrypted cache line and an outdated encrypted cache line. Google reported that there is a coherency mechanism on Intel TDX [11] for this purpose, where accesses with one key flush *all* other copies of the address with different keys from the cache. AMD pursued a similar approach by enforcing coherency between the unencrypted and encrypted cache lines by removing one when the other is *accessed*. Still, AMD does not operate on the granularity of a cache line, as we show in this work.

In this paper, we present Cohere+Reload, a novel attack exploiting that AMD's coherency approach introduces a surprisingly powerful side channel. We thoroughly analyze AMD's coherence mechanism for encrypted memory and discover two properties that form the basis of our Cohere+ Reload attack: First, there is a significant timing difference between cache misses and coherence conflicts on a spatial granularity of a 2 kB coherence partition, *i.e.*, half a page. This timing difference directly reveals whether a victim confidential virtual machine just accessed a specific memory location. Second, the amplitude of the timing coarsely correlates with the number of accesses the victim performed. It is also more finely correlated with the location of single victim accesses, allowing to infer which out of 8 alignments within a coherence partition the victim access had, *i.e.*, we have a maximum spatial resolution of 256 bytes, which is in the same order of magnitude as Flush+Reload with a spatial resolution of 64 bytes.

We evaluate Cohere+Reload in two synthetic and two real-world attacks: The two synthetic attacks demonstrate that Cohere+Reload can observe the control flow in workloads within a confidential virtual machine, *i.e.*, identify the target of a jump; and that Cohere+Reload can observe which data locations a workload within a confidential virtual machine accessed, naturally within the limits of the spatial resolution of Cohere+Reload. We present an attack on mbedTLS RSA-4096 and show that we can leak all 4096 key bits in a single-trace attack, with a Levenshtein distance of less than 11 bits on average. Finally, we present a novel attack on OpenSSL AES that exploits disalignments on a cache line granularity. Based on this insight, we mount a first round attack with and accuracy of 100 % in only 1500 encryptions. We recover all upper nibbles of AES with a novel correlation attack with an accuracy of 92.81 % in 12000 encryptions. We conclude that the coherence approach for AMD SEV-SNP should be re-evaluated and discuss further potential mitigations.

Disclosure. We disclosed our results to AMD, who addressed our findings in security bulletin AMD-SB-3010.

Contributions. In summary, our main contributions are:

- We introduce Cohere+Reload, a novel attack exploiting that AMD's coherency with a spatial granularity of a 2 kB per coherence partition, and 8 distinguishable alignments within a partition, yielding a maximum spatial resolution that is on par with Flush+Reload.
- We evaluate Cohere+Reload in two synthetic attacks demonstrating that we can leak control flow and data access from a confidential virtual machine on AMD SEV-SNP.
- We present an attack on mbedTLS RSA-4096 and show that we can leak all 4096 key bits in a single-trace attack, with a Levenshtein distance of less than 11 bits on average.
- We present a novel attack on OpenSSL AES that exploits disalignments on a cache line granularity, yielding an accuracy of 100 % in only 1500 encryptions in a first-round attack and an accuracy of 92.81 % in 12000 encryptions in a novel correlation attack.

Outline. We provide background in Section 2. We define our threat model in Section 3. We present our novel Cohere+Reload attack in Section 4. We template target pages in Section 5. We present an attack on mbedTLS RSA in Section 6 and an attack on OpenSSL AES T-Tables in Section 7. We present an attack on control flow and data accesses in Section 8. We discuss potential mitigations in Section 9 and conclude in Section 10.

2 Background

In this section, we provide background on trusted-execution environments, side-channel attacks, and coherence in the context of memory encryption.

2.1 Trusted-Execution Environments

The goal of trusted-execution environments (TEEs) is to provide confidentiality and integrity for code and data on a system even on a compromised system [6, 8, 9, 19]. Older TEEs often focus on personal and mobile computers, e.g., Intel Software Guard Extensions (SGX) [8]. The TEE runs a small trusted workload in a signed enclave [8]. These enclaves run on the same CPU as regular applications. To prevent access from a compromised host system, SGX prevents access to the encrypted enclave memory and register state.

More recent TEEs focus on cloud use cases and virtual machines (VMs). Instead of protecting a small workload, the idea is to move entire VMs into the TEE, which are then called confidential virtual machines (CVMs) and protect them from a malicious or compromised host [13], e.g., AMD Secure Encrypted Virtualization (SEV) [4] and Intel Trust Domain Extensions (TDX) [15].

AMD SEV protects memory contents of CVMs by encrypting any data moved out of the CPU, e.g., to DRAM or disk [37]. Still, there are many attacks on SEV. In particular the basic SEV design was demonstrated to provide too little protection for the guest state [26, 33] and memory [21, 28, 32, 33]. AMD addressed this issue with with the *Encrypted State (ES)* and *Secure Nested Paging (SNP)* SEV extensions, protecting guest state and memory integrity.

Like AMD SEV-SNP, Intel supports CVMs through their Trust Domain Extensions (TDX) [9]. Guest memory and state are encrypted and managed by the TEE. The host can only interact with the guest through well-defined secure interfaces. For fast inter-process communication, the memory has both private encrypted parts and shared parts that are equally accessible to the host.

2.2 Side-Channel Attacks

Side channels can be used to attack systems even if there are no software or hardware vulnerabilities or they are not known. Side channels instead exploit side effects of the implementation such as timing [51], power consumption [50], or radiation [49]. Older works focused on cryptographic primitives [30, 48, 51], leaking keys of vulnerable cryptographic implementations of e.g., AES [47, 48], RSA [30, 42], or ECDSA [41]. More recent works often focus on larger systems to leak information from one system component, e.g., kernel information [43], user input [27, 39, 45], and system activity [23].

Many side-channel attacks target caches as they can be probed without privileges at a high temporal (*i.e.*, nanosecond to microsecond range) and spatial resolution (*i.e.*, 64 B) while being comparably robust against noise. Most importantly, they allow for use generic attacks that are not tailored to specific applications and victim programs. Consequently, the community developed a set of generic attack techniques that follow a uniform naming pattern based on the attack components, e.g., Prime+

Probe and Flush+Reload. One of the first generic attack techniques was Evict+Time [47], in which an attacker runs and times a victim process twice, once with evicting a target cache line from the cache by performing a larger number of memory accesses that collide in the cache, e.g., due to set associativity, and once without. A statistically higher execution time means the cache line was used by the victim. Instead of timing the victim, Prime+Probe [47] times the (evicting) memory accesses, *i.e.*, they time how long it takes to (re-)prime a cache set. If it takes more time, more cache lines were replaced by the victim execution. Prime+Probe is one of the most widely used attack techniques besides Flush+Reload [42]. In the Flush+Reload attack, an attacker flushes a cache line using a dedicated flush instruction, and measures the time it takes to reload the memory location in order to decide whether or not the victim used it. Variations of Flush+Reload include Evict+Reload [38], which substitutes eviction for flushing, and Flush+Flush [36], which measures the timing of the flush instruction instead of the reload, thereby revealing a similar timing difference. Similarly, for Prime+Probe, several attack techniques and variations have been presented more recently, such as Prime+Abort [31], Prime+Scope [17], and Spec-o-Scope [7].

2.3 Coherence Between Ciphertext and Plaintext

Modern CPUs feature memory encryption technology, such as Intel's Total Memory Encryption (TME) [2] and AMD's Secure Memory Encryption (SME) [37]. Memory encryption is a crucial aspect of trusted execution environments, including Intel TDX and AMD SEV, designed to safeguard sensitive data. These built-in memory encryption systems encrypt data before it is written to the main memory and decrypt it when loaded into the CPU caches.

Given the nature of trusted computing, the untrusted hypervisor must interact with ciphertexts to facilitate operations such as migrating the guest machine to another server. To enable direct access to encrypted data, AMD's encryption unit includes a short-circuit path that forwards the data without decrypting it. Each data access's physical address encodes a so-called encrypted bit (C-bit), which indicates whether this short-circuit path should be utilized. This leads to an important question: how is coherence maintained when the hypervisor actively requests ciphertext while the guest is processing plaintext?

AMD states that coherency between the ciphertext and plaintext depends on the hardware [5] as some hardware enforce coherency while others do not. In the systems where coherence is not enforced, the hypervisor must flush the encrypted data from all CPU caches. In other systems, hardware supports coherency across encryption domains and software does not have to flush encrypted data, and the presence of this feature can be determined by the CPUID bit "CoherencyEnforced" (see AMD Architecture Programmer's Manual [5], 7.10.6).

In our systems, all AMD EPYC CPUs support SEV-SNP and include this coherence feature. Consumer Ryzen CPUs only support SME [24] without automatic hardware coherence between ciphertext and plaintext. For SEV systems without SNP, not having hardware coherence poses significant risks, allowing potential fault-attack-like exploitation during the write-back of ciphertext.

With the introduction of AMD SEV-SNP, the hypervisor can no longer directly write to an encrypted page [19]. Each guest page undergoes a procedure to assign it in a reverse page map, indicating its ownership to a given guest VM. Once a guest accepts a page, the hypervisor retains only read access, which is ciphertext. Despite these advancements, maintaining coherence between ciphertext and plaintext remains essential. On Intel TDX, it is known that accesses to a ciphertext flushes all other copies of the address from the cache [11]. In Section 4, we present our initial analysis of how coherence is managed on AMD systems.

3 Threat Model

Exploiting the Cohere+Reload mechanism requires a ciphertext view and a plaintext view on the same memory region. Outside of SME, this can only happen when a hypervisor maps an SEV guest page (ciphertext view), as guests have no option to map pages outside their allocated memory. Our threat model is therefore a malicious hypervisor trying to extract information from an encrypted SEV, SEV-ES or SEV-SNP guest. In this scenario, the hypervisor has control over all parts of the CPU that are not part of the attestation. This includes control over CPU frequency, disabling hardware prefetching and selecting a suitable DRAM interleaving setting at boot (see Section 4.1). While Cohere+Reload attacks can be performed even without stabilizing the frequency or disabling prefetching, like many

Table 5.1: Test systems.

CPU	Architecture	SME	HW Coherence	SEV	VM page flush MSR
2x AMD EPYC 7443	Zen 3	✓	✓	SNP	✓
AMD EPYC 7313P	Zen 3	✓	✓	SNP	✓
AMD EPYC 8024P	Zen 4c	✓	✓	SNP	X

cache attacks [10, 18, 22, 25, 34], it is simplified by these settings and they will be used throughout the paper.

4 Cohere+Reload

In this section we examine the behaviour of coherence for AMD memory encryption. In all of our tests, hardware cache coherence works the same in SME as it does in SEV, SEV-ES or SEV-SNP. Therefore we will conduct all basic experiments in SME for simplicity, unless specifically mentioned.

As a first step, we configure our systems (cf. Table 5.1) for transparent secure memory encryption (TSME). This means all pages will be encrypted by default, denoted by a bit in the physical address, e.g., bit 51. To get a ciphertext view of a page, we create a second mapping where this bit is not set. When we now measure access times to a cache line in the ciphertext mapping, we can clearly distinguish three cases: hits, misses and coherence conflicts (see Figure 5.1). We cause a normal miss by flushing the line with `clflush` before measuring it, and a coherence conflict by accessing the same line in the plaintext mapping. We attribute the latency increase to the fact that when there is a plaintext line to evict, this has to happen before the load is completed, to ensure coherency. We also observe, as expected, that this effect is entirely symmetrical; it does not matter which mapping is used as the observer. The coherence also holds for code pages that were cached through code execution.

This basic hit/conflict behaviour constitutes the first part of the Cohere+Reload primitive (see Section 4.2 for the second).

4.1 Eviction Pattern

Contrary to Intel TDX, AMD memory encryption does not enforce its coherence with single line granularity. Instead, we find that any access *always*

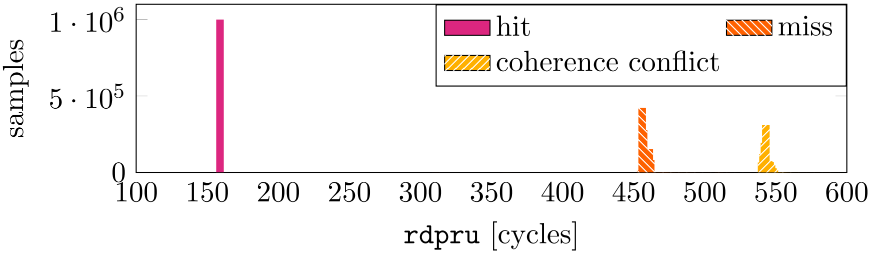


Figure 5.1: Access timing histogram for accesses that are hits, misses (flushed) or conflicts caused by SME coherence.

Table 5.2: DRAM interleaving size and coherence pattern block size on our two systems.

DRAM interleaving setting	off	256	512	1024	2048	4096
block size EPYC 7443	2048	256	512	1024	2048	256
block size EPYC 7313P	2048	256	512	2048	2048	256

triggers the eviction of 32 out of 64 cache lines on a 4 kB aligned section of memory. We notice that in all of our machines' default configurations, the page is not simple split into two contiguous 2 048 B halves, but instead shows an alternating pattern of 256 byte (4 cache lines) *coherence blocks* between the two *coherence partitions* (see Figure 5.2a). Concretely, this means that an access to one or multiple plaintext (or ciphertext) addresses in the first (or second) coherence partition of a page will always trigger an eviction of *all* ciphertext (or plaintext) addresses in the first (or second) partition of a page. This limits the channel's spatial resolution compared to Flush+Reload, though it speeds up page profiling (see Section 5). We run this experiment on different physical and virtual pages, different page sizes (4 kB and 2 MB) and between different cores. We find that the pictured pattern is always the same. However, two of our machines' (EPYC 7443 and EPYC 7313P) mainboard menus expose a boot setting for "*DRAM interleaving size*". When we change it from its default of 256 B to 512 B, 1 024 B or 2 048 B, we can see the coherence eviction pattern changing to match (see Figure 5.2), except for "off", 1 024 B in the case of the EPYC 7443 system, and 4 096 B (Table 5.2). While we do not know why the coherence mechanism is implemented as it is, we suspect some form of load balancing consideration w.r.t. DRAM.

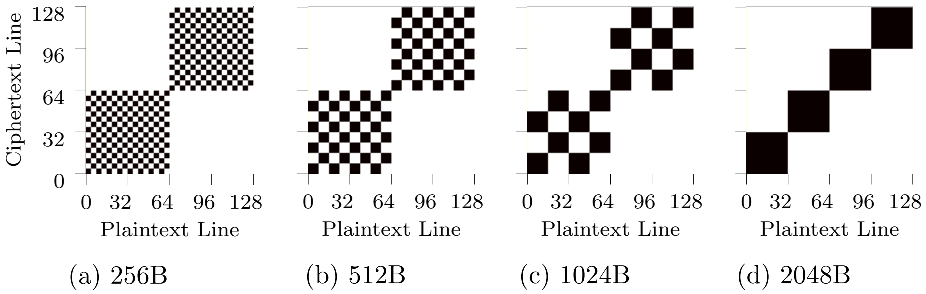
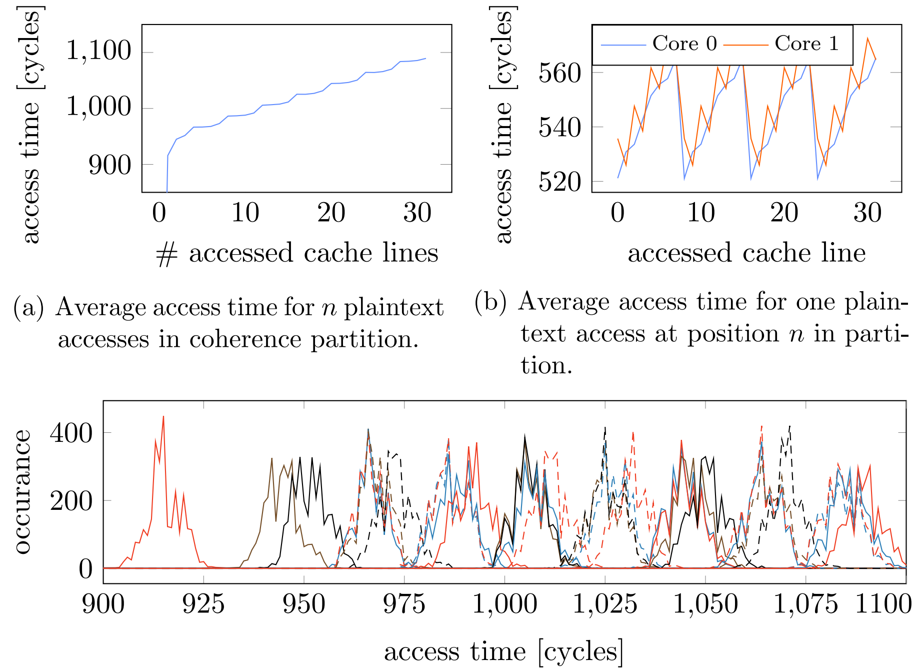


Figure 5.2: Eviction Pattern for different *DRAM interleaving size* setting over 8 kB physically contiguous memory. A plaintext cache line is evicted by (and evicts) all corresponding ciphertext cache lines in black.

4.2 Access Delay Time

Since we have seen in Section 4 that the presence of a single plaintext cache line increases the access time for a ciphertext line in the same coherence partition, it stands to reason that more cache lines in the same partition might take even longer to evict. Indeed, we find that the access delay on the evicting party's side is related to the number of accessed lines in the coherence partition. When we access from 0 to 32 lines of plaintext in the same coherence partition and measure a ciphertext access, we see a monotonically increasing access time (Figure 5.3a). But we do not observe a strictly monotonic increase, instead we see plateaus every 4 cache lines. When we look at the individual distribution of access times for each number of accesses (Figure 5.3c), these groupings are visible. The first three access groupings are somewhat separated (that is, measuring after 1,2 or 3 accessed lines), but from then on there are quartets of consecutive numbers of accesses that display very similar access times.

Investigating further, we can see that Figure 5.3a and Figure 5.3c are actually a special case of timings when the accessed cache lines are contiguous, *i.e.*, we do not skip lines within a coherence partition up to our chosen number of accesses. Measuring the ciphertext access times for single plaintext evictions of different plaintexts we find the cause of this behaviour: different offsets within a coherence block have distinct timings. As Figure 5.3b shows, when there is only one plaintext access in the coherence domain the access time of the ciphertext depends on the position of the plaintext access within the coherence block and repeats from block to block. This means that while *on average* a higher number



(a) Average access time for n plaintext accesses in coherence partition.

(b) Average access time for one plaintext access at position n in partition.

(c) 32 histograms of access times for $n = 1$ to 32 plaintext accesses. Certain numbers of accesses have very similar timing distributions.

Figure 5.3: Ciphertext conflict access times for different numbers of prior plaintext accesses on congurent adresses. Which addresses and how many where accessed changes the conflict eviction time.

of accessed cache lines within a partition will increase the conflict eviction time, some combinations of lines will lead to far higher delays than others. This pattern depends on the core number *within a core complex* that loaded the plaintext address, but is the same between core complexes. We believe the pattern comes from the topology of the core complexes. When the block size is increased, the timing pattern also expands, though only up to 8 lines. For pattern sizes of 1 024 B and larger, the pattern begins to repeat after 512 B, *i.e.*, 8 cache lines.

This timing behaviour is the second aspect of the Cohere+Reload primitive. We will further explore this effect in an attack in section Section 7.2.

Table 5.3: Comparison of Cohere+Reload with Flush+Reload and Flush+Flush.

	Flush+Reload			Flush+Flush			Cohere+Reload		
System	Hits [cycles]	Misses [cycles]	Blind Spots [%]	Hits [cycles]	Misses [cycles]	Blind Spots [%]	Conflict [cycles]	Hit [cycles]	Blind Spots [%]
EPYC 7443	122 σ=13	389 σ=130	65.1%	482 σ=96	367 σ=90	3.2%	428 σ=103	125 σ=130	0.06%
EPYC 7313P	150 σ=0	658 σ=173	73.2%	833 σ=124	632 σ=31	3.5%	800 σ=167	151 σ=0	0.53%
EPYC 8024P	145 σ=0	497 σ=117	74.8%	623 σ=92	484 σ=59	1.6%	623 σ=131	146 σ=1	0.02%

We compare Cohere+Reload with Flush+Reload and Flush+Flush across three metrics: hit time, miss/conflict time, and blind spots. The measurement for all three metrics is repeated 100 000 times on each system on different physical cores. Cohere+Reload has a much smaller blind-spot size and comparable hit and conflict times.

4.3 Cohere+Reload Compared to Other Cache Attacks

In this section, we compare Cohere+Reload with two cache attacks: Flush+Reload and Flush+Flush. Our results, presented in Table 5.3, show that Cohere+Reload is a fast attack, comparable to the two Flush-based cache attacks across three metrics: hit time, miss time (conflict time), and blind spots. Using the methodology presented in prior work [3], we measure each metric 100 000 times on three systems: 2x EPYC 7443 (Zen 3), EPYC 7313P (Zen 3), and EPYC 8024P (Zen 4c). We disabled the hardware prefetchers and fixed the frequency on all three machines. To measure the metrics, we spawn two threads on different physical cores: a victim thread which randomly accesses a predetermined memory location, and an attacking thread that mounts the attack, measuring the metric.

On all three systems, we notice that Flush+Reload has a large blind spot — between 65-75% of victim accesses were missed by the attacker. Flush+Flush has a much smaller blind spot, with only 1.5-3.5% of victim accesses being missed by the attacker. Cohere+Reload has a minuscule blind spot, with less than 0.5% of victim accesses being missed by the attacker.

To measure the attack time, we consider both the hit and miss (conflict) timings. We see that the hit timings of Cohere+Reload are comparable to the hits of Flush+Reload, and the conflict timings of Cohere+Reload are comparable to Flush+Flush. This shows that Cohere+Reload is as fast as comparable attacks with a much smaller blind spot size, making it a very reliable side-channel attack.

5 Target Page Templating

In a standard SEV-SNP scenario, the host has no knowledge about where the guest maps which data in its virtual memory range. For an attack, the first step is therefore to locate pages of interest in the guest. The same result could be achieved with page access flags, though this is an alternative approach that does not require the flushing of TLB entries. The only requirement for this step is that a victim page access can be reliably triggered (e.g., establishing a connection that causes an RSA encryption).

We implement this for RSA by using a network call to the guest that triggers an encryption. In our test, we filter 4 GB of VM virtual memory with a sieve of sorts. Starting with all pages, we repeatedly cause the guest to access or not access the page of interest while measuring each page from different core with Cohere+Reload. We access each page with a single split load to detect coherence eviction in both coherence partitions at once, as we find that the penalty for accessing both partitions is only ≈ 30 cycles. Pages that do not show the expected hits or conflicts are discarded from the list and the next step operates on the reduced list. After only 4 sieve steps, 1 048 510 pages can be reduced to an average of 7.36 pages in 10.6 s in 100 experiments, with the two RSA pages of interest always being among them. Finding the correct page from there is trivial, as only those two pages show the expected access pattern during an encryption (see Section 6).

6 High Frequency Code Attack - RSA

We attack the square-and-multiply `mbedtls_mpi_exp_mod` implementation of RSA in Mbed-TLS v3.0.0. For the purposes of this demonstration, we configure it to use a maximum window size of 1. While the specific version is not crucial as long as the algorithm is the same, note that because of the coherence pattern, Cohere+Reload requires a suitable code layout. That is, the code that can be attacked needs to be aligned suitable within a coherence partition, while code that would hinder the attack needs to fall into the other partition. In this example, we find that a 256 B pattern is unsuitable, but switching to a 512 B pattern with the DRAM interleaving setting results in a working attack.

Our victim is an RSA encryption service in an SEV-SNP guest triggered by the attacker, which runs on the host. For the attack, the host program

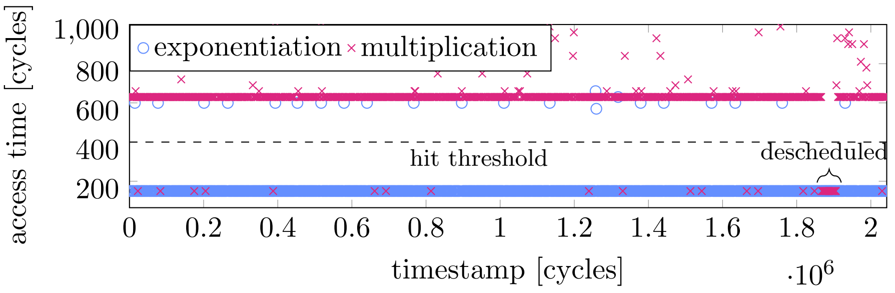


Figure 5.4: Part of a raw Cohere+Reload trace of an RSA encryption. When *exponentiation* shows a conflict, a new exponentiation loop was started. When *multiplication* shows many in a row, the victim was most likely descheduled.

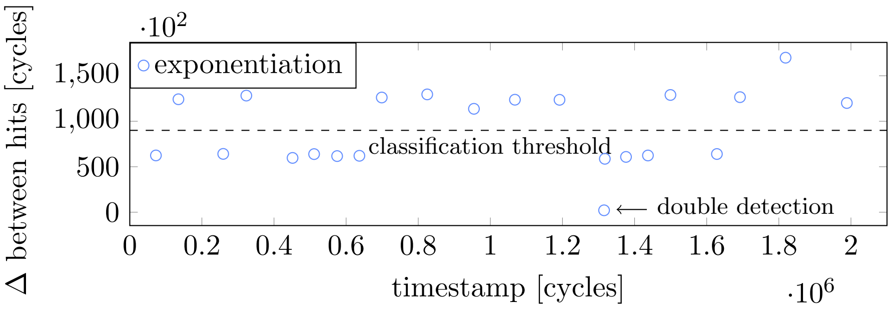


Figure 5.5: Time difference between hits on *exponentiation*. The two bands show where a multiplication was executed (large difference) and the key bit is 1 or where it was not and the bit is 0 (small difference).

records traces of two code locations. First, the second partition in the `mpi_exp_mod` function. This contains the beginning of the loop that iterates over each key bit. Second, we trace the `mpi_montmul` function that does the squaring *and* multiplication operations. Our attack traces starts when the `mpi_exp_mod` is called for the first time and records long enough to capture the entire encryption (240000 samples). Figure 5.4 shows a section of such a trace. The `mpi_exp_mod` signal (blue) carries most of the key information, as it ideally detects an eviction precisely once per processed bit. This lets us infer whether or not `mpi_montmul` was called in addition to the square function (indicating that the key bit was 1) by the time delay to the next detection. The time difference between two conflicts in this signal is about twice as long when a '1' bit is processed. While this alone allows us to recover most of the key, we can correct some

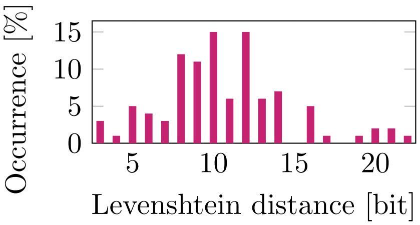


Figure 5.6: The Levenshtein distances in bit for 100 single-trace RSA 4 096 bit key recovery attacks.

mistakes with the signal in `mpi_montmul` (pink). As the algorithm spends most of its time in this function, we detect almost all conflicts. However, when the algorithm is paused for any reason (e.g., scheduling), we see periods of hits on this address that we can then use to correct the primary signal. Figure 5.4 shows one occurance of this near the end. In minor post-processing we also detect the precise start and end of the encryption and remove double detections for single bits that are too close together (see Figure 5.5). Over 100 runs, our attack recovers randomly generated 4 096 bit keys with a Levenshtein distance of 10.7 ± 3.94 (μ, σ) with a single trace (see Figure 5.6). In terms of attack performance, this is on par with related works attacking RSA [1, 12, 14, 20, 35].

7 AES T-Tables

In this section, we evaluate Cohere+Reload on the AES T-table implementation of OpenSSLv3.4 with 128 bit keys. Specifically, the `AES_encrypt` function which uses T-tables in lieu of hardware support (*i.e.*, AES-NI). Similar to the RSA attack (Section 6), we choose this implementation as it has been used extensively to evaluate prior side-channel attacks and is therefore a well-understood attack target [1, 12, 14, 16, 20, 29, 35].

The well-known first- and last-round attacks on AES T-tables [40, 46, 48] are both based on access probabilities. For a first-round cache attack, each of the four T-tables' cache lines (16 per table with 16 entries each) are measured as hits or conflicts after an entire encryption. As an encryption consists of 40 total accesses to each table (10 rounds with each 4 accesses), over a sufficient number of random plaintexts the probability for each line to be accessed at the end of an encryption is $1 - \frac{15}{16}^{40} = 92.43\%$. This can be distinguished from lines that are *always* accessed. In the first round of the encryption, the tables are accessed according to the result of $P_n \oplus K_n$, where K_n is byte n in the original key and P is the plaintext. Fixing one

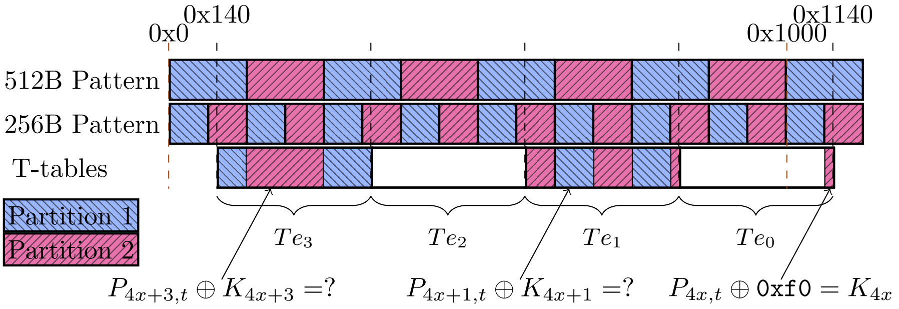

0x0
0x140
0x1000
0x1140
512B Pattern
256B Pattern
T-tables
Partition 1
Partition 2
Te_3
Te_2
Te_1
Te_0
$P_{4x+3,t} \oplus K_{4x+3} =?$
$P_{4x+1,t} \oplus K_{4x+1} =?$
$P_{4x,t} \oplus \texttt{0xf0} = K_{4x}$

Figure 5.7: AES T-table memory alignment in OpenSSL and memory coherence partition patterns for 256 B and 512 B. Annotations show where first round T-tables are accessed depending on the key- and plaintext bytes. Te_1 and Te_3 demonstrate the 256 B/512 B patterns respectively (only 1 pattern can be active for all memory for each system boot), and Te_0 shows the second partition on the next page.

plaintext byte therefore allows an attacker to control which line is always accessed. After measuring enough encryptions, only this line will show no conflicts, hence we can infer the upper nibble of each key byte.

The resolution of Cohere+Reload, however, is not a single cache line. The partition size of half a page is simply not enough to perform this attack merely by distinguishing hits from conflits, and even other tables on the same page influence each other. Even if one performed enough measurements to detect a non-accessed coherence partition (quite unlikely with $P_{accessed} = 1 - \frac{1}{2}^{160}$), this would only leak one bit per key byte.

However, we notice that while the default T-table placement in memory is aligned to cache lines with default compilation options, it is not necessarily aligned to a page. From this simple fact, we are able to conduct a limited standard first-round attack (Section 7.1) as well as a novel variation on the first-round attack based on a bias in the number of accessed cache lines instead of their location (Section 7.2).

7.1 Disaligned T-Table First-Round Attack

For this first attack, we only look at table Te_0, which deals with key/-plaintext bytes 0,4,8 and 12 in the first round. As we can see in Figure 5.7 bottom right, Te_0 spans 5 cache lines into a new page. For a Flush+Reload attack, this would not make a difference, as the attack either works on

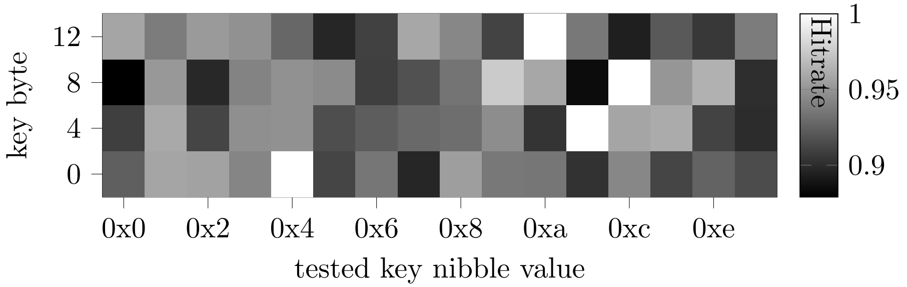


Figure 5.8: Heatmap of conflicts in an AES disaligned T-table attack on key bytes 0, 4, 8, 12 with a total of 1500 encryptions. Correct key nibbles are 0x4, 0xb, 0xc and 0xa.

all cache lines or it does not work at all (e.g., because the target cannot be accessed). For Cohere+Reload however, this enables an attack. With a coherence pattern size of 256 B, this means that the last line of Te_0 is the only one that accesses the second coherence partition on that page. With this disalignment we can convert the Cohere+Reload primitive into what is essentially a Flush+Reload primitive for this implementation, an improvement in the same vein as the attack described by Spreitzer et al. [44].

For random plaintexts, each cache line will be used by an encryption in 92 % of cases. In our case, the second coherence partition of the second page will measure this percentage for the last line of Te_0. The T-table access in the first round depends only on the XOR of the plaintext and key bytes, e.g., $Te_0 = P_{\{0,4,8,12\}} \oplus K_{\{0,4,8,12\}}$. As only an XOR product of `0xf` in the upper nibble accesses the measured partition, the correct key nibble can thus be derived with $(P_{x,t} \oplus \text{0xf0}) \wedge \text{0xf0} = K_x$ for the test plaintext byte $P_{x,t}$ for which Cohere+Reload shows a 100 % hit ratio.

We can recover all upper nibbles for these 4 key bytes in 1500 encryptions with 100 % accuracy. Figure 5.8 shows a heatmap of one such attack with a total of 1000 encryptions, clearly displaying the key correct key nibbles 0x4, 0xb, 0xc and 0xa.

7.2 First-Round Correlation Attack

For the standard T-table attack, we use the fact that there is a 92 % chance that any given cache line in a table will be accessed with a random plaintext. We have established above that the spacial resolution of Cohere+Reload is not enough to us this in a standard first-round cache attack. However, we know each key- and plaintext byte combination accesses a specific cache line with 100 % certainty. This in turn means either the first or second coherence partition will be accessed for each plaintext byte in the first round. Looking at a single key byte at a time, we can easily calculate a pattern of partition 1 and partition 2 accesses for each plaintext and each key. Concretely, we can create a template vector for each possibly key byte value of the form $\{0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0\}$. Each number denotes the partition that $P_n \oplus K_n$ will access, in this case for $K_n = 0$ and a pattern size of 256 B. We can now pick a fixed plaintext and change only one plaintext byte at a time and record the number of total accessed cache lines in the T-tables that fall within the one of the partitions. With this, we can now calculate the Pearson correlation coefficients between the templates and the access count vector. The highest correlation will show the template corresponding with the correct upper nibble of the tested key byte.

When we generate the template vectors, we find that they contain a different number of unique templates for the 256 B and 512 B coherence patterns. Depending on the cache line offset within a page, there are at most 8 unique templates for a 256 B pattern and 16 for the 512 B pattern. This means for all odd cache line offsets of the T-tables, we can recover 3 or 4 bits per key byte, depending on the chosen pattern size. The pattern size 256 B yields one bit less, since the disaligned coherence pattern within each table repeats once (see Figure 5.7) and we can therefore not distinguish the most significant bit.

While this attack benefits from chosen plaintexts (see above), a sufficient number of (mostly) random known plaintexts will work just the same. By adding the number of partition accesses to a bucket for each plaintext byte value and for every byte of a random plaintext, each encryption can contribute to the recovery of all 16 key bytes instead of just one. With many encryptions, this creates a 16x16 matrix with one correlation vector for each key byte. After enough encryptions, the bias in the average number of accessed cache lines in a coherence partition outweighs the initial noisiness of random plaintexts and the key bytes can be recovered. Therefore we consider this a known-plaintext attack.

Cohere+Reload provides for two methods of measuring the number of accesses. Firstly, the access time for a single read, as described in Section 4.2. In theory, over enough measurements with randomized plaintexts,

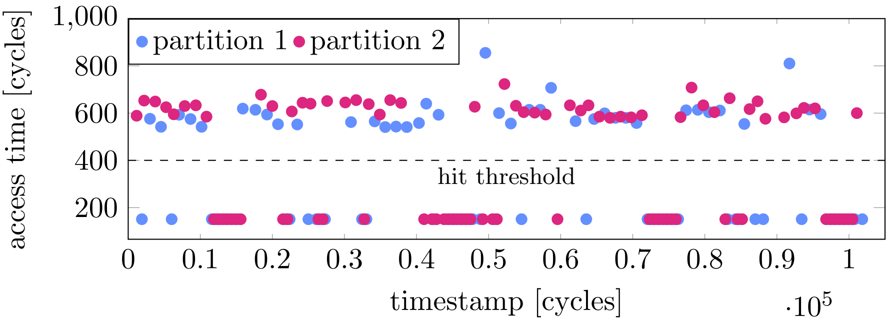


Figure 5.9: Cohere+Reload access trace for a single AES encryption.

the average access time should provide a proxy measure for the number of cache lines that were accessed within a partion. Unfortunately, we could not make this method work with AES. Fortunately, we can make use of the minimal blind spot and high frequency of Cohere+Reload and mount a trace attack.

Unlike in the case of RSA, there is very little time between accesses to the T-tables in the OpenSSL implementation. When we try to trace an encryption normally, we only observe 10-15 accesses per partition, for a total of $\approx 20 - 30$ accesses out of the ≈ 135 expected accesses (less than 160, as some accesses fall on the second page). Since in our threat model (cf. Section 3) we are the hypervisor, we can however employ a little trick to slow down our victim. Even in SEV-SNP, the hypervisor has control over the bits in the page table entries, including the uncacheable bit. We find that by making both the function code and the stash page uncacheable, we can slow down our victim considerably. This allows us to record around 100 total memory accesses for a single encryption, as we can see in Figure 5.9. Though still shy of 160, we cannot reliably see all 16 individual accesses in the first round and infer the table accesses directly. We can, however use the number of hits to the partitions as a proxy. Even though some hits will contain two or more accesses, on average the number of accesses in a partition will be biased by the first round. To reduce the noise, we only look at the first 16 accesses to both partitions, and extract the number of accesses to one of them as our signal. We choose 16 as it is the upper limit to how many hits we can see within the first round, and even if accesses from the second round are included, they only add noise. We can see this signal plotted together with the correct template in

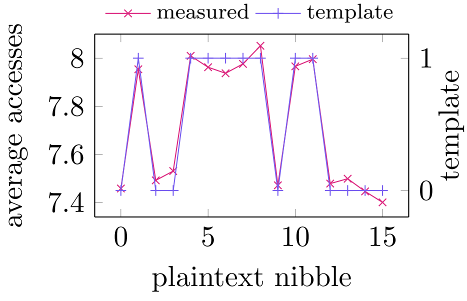


Figure 5.10: Correlation attack template vector for key nibble `0xa` vs. average access counts for correct key nibble guess with 8000 encryptions. $\rho = 0.99$.

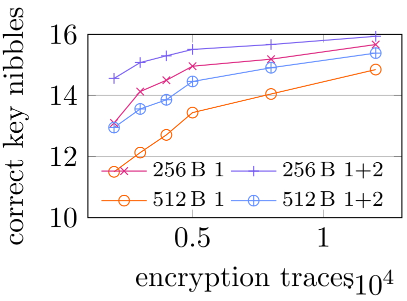


Figure 5.11: Average correct key nibbles for a given number of AES traces for the top guess (1) and the top 2 guesses combined (1+2). $n = 100$ per point.

Figure 5.10. In this case, we achieve a very high correlation coefficient of $\rho = 0.99$.

Figure 5.11 shows our results for both pattern sizes. We can see that recovering 3 bits is more robust, as the templates are more different to each other. With 4000 traces, we correctly recover 3 bit for an average of 14.5 key bytes in our first guess. Adding second guesses, this rises to 15.3. Using the 512 B pattern, we can recover an average of 14.85 nibbles per byte with 12000 encryptions with first guesses and 15.4 when we also include second guesses.

Since this is a correlation based attack, we can identify weakly correlating key nibbles, or those where several candidates are close, and record more traces only in these cases to minimize overall traces.

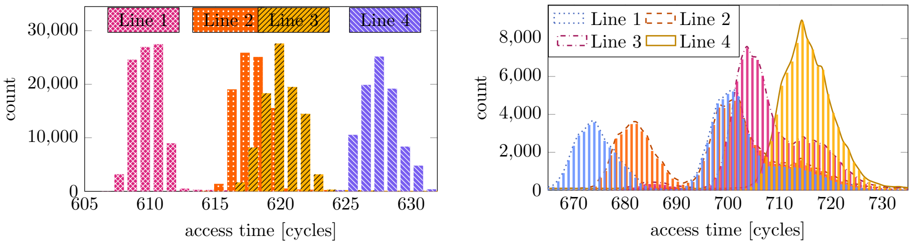
Tracing AES with this time resolution (without repeating encryptions) is something we do not believe can be easily achieved *across cores or even sockets* with other cache attacks like Flush+Reload or Flush+Flush, as Cohere+Reload can monitor an entire page with only two addresses (cf. Section 4.3). Though compared to the standard our attack takes longer (e.g., Flush+Reload can work with only a few hundred to low thousands of encryptions as shown in Section 7.1), it also has slight advantages. Firstly, as a trace-based attack it is not hindered by software prefetching, as each read resets the cache line. Secondly, this attack functions the same for 128 bit keys as it does for 192 bit or 256 bit keys, as the additional two or four rounds do not affect the beginning of the attack, whereas for other attacks the probabilities become less favorable.

8 Load-time based attacks

In Section 4.2, we observed that Cohere+Reload-timings to different cache lines in the same coherence block have discernible timing differences, *i.e.*, an attacker can conclude which cache line was accessed by measuring the time taken to access a cache block. In this section, we use this observation to mount two synthetic attacks: the first reveals which part of an array is accessed while the second detects which case of a `switch` statement is taken. We test these attacks on an AMD EPYC 8024P (Zen 4c) with hardware-prefetching disabled.

Our experimental setup consists of two processes, an attacker and a victim, which can access the same physical page as a ciphertext or plaintext mapping respectively. This page is either a dynamically allocated array storing values (the first attack), or it the victim's code (the second attack). We assume that the region of interest is one cache-block large (256 B) and assume that the physical address is aligned to this value. For the attack on code execution, we ensure that each case of the switch is one cache-line long and all four cases are within the same cache block. By measuring the access time with Cohere+Reload, we can now infer which line in a coherence block was accessed by the victim, which in turn can let us infer control- or data flow. With the attack on code, a source of noise is the (speculative) fetching of instructions from the next case by the instruction prefetcher. To overcome this, we use the `ret` instruction at the end of every case to indicate that the next set of instructions will not be executed.

For the purposes of the experiment, the attacker and victim alternately access the physical address. The attacker records the access times and the victim accesses only one random cache line. Each cache line is accessed 100 000 times, resulting in 400 000 measurements. Figure 5.12a shows the access time distribution to the coherence block when the monitored address corresponds to a dynamically allocated array, *i.e.*, *data*. In Figure 5.12b, we see the access times to the coherence block when the physical address corresponds to a switch and each case is on a different cache line, *i.e.*, *code*.



(a) Data

(b) Code

Figure 5.12: Cohere+Reload access times to a coherence block when it contains victim data (Figure 5.12a) and victim code (Figure 5.12b).

With these measurements we can make several observations. First, access times to code (Figure 5.12b) is noisier than data (Figure 5.12a), even in our example with a `ret` at the end of every case. In further tests we learn that the distributions become visually separable when the attacker can choose to repeat the same input. Since the `ret` instruction also improves the result but does not make it perfect, we believe this is a combination of misspeculation and a race condition between how fast the front end can fetch data vs. how soon the `ret` is decoded.

In a real attack, exploitability depends heavily on the control an attacker has over the target branch. If a single secret bit can be reliably repeated, it only takes a few samples to train predictors and receive a clean signal (cf. Section 4.2). If a sequence of bits can reliably be repeated in the same order (e.g., a key that is used bit by bit), Figure 5.12b shows that by combining repeated measurements we can produce distinct distributions, even when attacking instructions.

9 Mitigation

AMD disabling the coherence feature will undoubtedly mitigate Cohere+Reload, though `VMPAGE_FLUSH`, if present, may lead to similar leakage, as it may well have the same time dependence. Here, the fundamental question is if the host's ability to read the ciphertext at all is even necessary for SEV-SNP. The SEV-SNP attestation flag `CiphertextHidingDRAM` suggests it may not always be necessary, as it disallows reading of the ciphertext by the hypervisor when active. As we have no hardware that supports this feature, we can only speculate that the coherence mechanism could be disabled when this setting is enabled, depending on how it is implemented.

On the developer side, hardened implementations could change the memory type of critical sections (e.g., T-tables) to *uncacheable*. While this slows down execution, our experiments show that *loads* to uncacheable memory do not trigger the coherence mechanism.

10 Conclusion

In this paper, we introduced Cohere+Reload, a novel side-channel attack exploiting AMD's coherency for encrypted memory. We exploit two types of leakage in the coherency mechanism: First, we exploit coherence conflicts, leaking victim operations on a spatial granularity of a 2 kB block. Second, we exploit timing correlations with the number and location of accesses, reaching a maximum spatial resolution of 256 bytes. In our synthetic attacks, we showed that Cohere+Reload can observe control flow and access locations in workloads within a confidential virtual machine. As a benchmark we mounted an attack on mbedTLS RSA, leaking 99.7 % of the 4096 key bits in a single-trace attack. We also mounted an attack on OpenSSL AES exploiting disalignments on a cache line granularity, achieving an accuracy of 100 % in only 1500 encryptions in a first round T-table attack and an accuracy of 92.81 % in 12000 encryptions with a novel correlation attack. Our work shows that coherence mechanisms can undermine the confidentiality of confidential virtual machines. Consequently, we believe that vendors need to weigh the necessity of coherence, as discussed above, against the opening of this side channel for future implementations.

Acknowledgments

We want to thank Andreas Kogler. After tinkering with cache coherency with SEV for some time at TU Graz, Andreas found an effect and suggested that we investigate it. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85 and FWF project NeRAM 10.55776/I6054). Additional funding was provided by a generous gift from 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] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In: NDSS. 2025 (p. 82).

[2] Intel. Intel Total Memory Encryption White Paper. 2025. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/vpro/hardware-shield/total-memory-encrpytion.html` (p. 73).

[3] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In: NDSS. 2025 (p. 79).

[4] AMD. AMD Secure Encrypted Virtualization (SEV). 2024. URL: `https://developer.amd.com/sev/` (p. 72).

[5] AMD. AMD64 Architecture Programmer's Manual. 2024 (p. 74).

[6] ARM. Arm Confidential Compute Architecture. 2024. URL: `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture` (p. 71).

[7] Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-Scope: Cache Probing at Cache Speed. In: CCS. 2024 (p. 73).

[8] 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. 71).

[9] 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. 71, 72).

[10] Luyi Li, Jiayi Huang, Lang Feng, and Zhongfeng Wang. PREFENDER: A Prefetching Defender against Cache Side Channel Attacks as A Pretender. In: IEEE Transactions on Computers (2024) (p. 75).

[11] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Security Review. 2023. URL: https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf (pp. 70, 74).

[12] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: S&P. 2023 (p. 82).

[13] Confidential Computing Consortium. A Technical Analysis of Confidential Computing. 2022 (p. 72).

[14] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In: S&P. 2022 (p. 82).

[15] Intel. Intel Trust Domain Extensions. 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf (p. 72).

[16] 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. 82).

[17] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (p. 73).

[18] Zihao Wang, Shuanghe Peng, Wenbin Jiang, and Xinyue Guo. Defeating Hardware Prefetchers in Flush+Reload Side-Channel Attack. In: IEEE Access 9 (2021), pp. 21251–21257 (p. 75).

[19] 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. 71, 74).

[20] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA. In: Information Systems 92 (2020), p. 101524 (p. 82).

[21] 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. 72).

[22] C Ashokkumar, M Bhargav Sri Venkatesh, Ravi Prakash Giri, Bholanath Roy, and Bernard Menezes. An error-tolerant approach for efficient AES key retrieval in the presence of cacheprefetching–experiments, results, analysis. In: Sādhanā 44 (2019). DOI: `https://doi.org/10.1007/s12046-019-1070-8` (p. 75).

[23] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (p. 72).

[24] Tom Lendacky. What processors support SEV? #1. 2019. URL: `https://github.com/AMDESE/AMDSEV/issues/%5C#issuecomment-581426096` (p. 74).

[25] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack. In: DAC. 2019 (p. 75).

[26] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In: AsiaCCS. 2019 (p. 72).

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

[28] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In: EuroSec. 2018 (p. 72).

[29] Mark Zhao and G Edward Suh. FPGA-based Remote Power Side-Channel Attacks. In: S&P. 2018 (p. 82).

[30] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. Photonic Side Channel Attacks Against RSA. In: HOST. 2017 (p. 72).

[31] 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. 73).

[32] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. In: arXiv:1712.05090 (2017) (p. 72).

[33] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In: ACM SIGPLAN Notices 52.7 (2017), pp. 129–142 (p. 72).

[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 (p. 75).

[35] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 82).

[36] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 73).

[37] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. 2016 (pp. 72, 73).

[38] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In: ESORICS. 2015 (p. 73).

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

[40] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID. 2014 (p. 82).

[41] 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. 72).

[42] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (pp. 69, 72, 73).

[43] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 72).

[44] Raphael Spreitzer and Thomas Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In: COSADE. 2013 (p. 84).

[45] 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. 72).

[46] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In: CHES. 2006 (p. 82).

[47] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 69, 72, 73).

[48] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (pp. 72, 82).

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

[50] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In: CRYPTO. 1999 (p. 72).

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

6

Generic and Automated Drive-by GPU Cache Attacks from the Browser

Publication Data

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

Contributions

Main author.

Generic and Automated Drive-by GPU Cache Attacks from the Browser

Lukas Giner[1], Roland Czerny[1], Christoph Gruber[1], Fabian Rauscher[1], Andreas Kogler[1], Daniel De Almeida Braga[2], and Daniel Gruss[1]

[1]Graz University of Technology,
[2]University of Rennes, CNRS, IRISA,

Abstract

In recent years, the use of GPUs for general-purpose computations has steadily increased. As security-critical computations like AES are becoming more common on GPUs, the scrutiny must also increase. At the same time, new technologies like WebGPU put easy access to compute shaders in every web browser. Prior work has shown that GPU caches are vulnerable to the same eviction-based attacks as CPUs, e.g., Prime+Probe, from native code.

In this paper, we present the first GPU cache side-channel attack from within the browser, more specifically from the restricted WebGPU environment. The foundation for our generic and automated attacks are self-configuring primitives applicable to a wide variety of devices, which we demonstrate on a set of 11 desktop GPUs from 5 different generations and 2 vendors. We leverage features of the new WebGPU standard to create shaders that implement all building blocks needed for cache side-channel attacks, such as techniques to distinguish L2 cache hits from misses. Beyond the state of the art, we leverage the massive parallelism of modern GPUs to design the first parallelized eviction set construction algorithm. Based on our attack primitives, we present three case studies: First, we present an inter-keystroke timing attack with high F_1-scores, *i.e.*, 82 % to 98 % on NVIDIA. Second, we demonstrate a generic, set-agnostic, end-to-end attack on a GPU-based AES encryption service, leaking a full AES key in 6 minutes. Third, we evaluate a native-to-browser data-exfiltration scenario with a Prime+Probe covert channel that achieves transmission rates of up to 10.9 kB/s. Our attacks require no user interaction and work in a time frame that easily enables drive-by attacks while browsing the

Internet. Our work emphasizes that browser vendors need to treat access to the GPU similar to other security- and privacy-related resources.

Keywords: Side Channels, Cache Attacks, GPU computing.

1 Introduction

In the last decades, Graphics Processing Units (GPUs) have seen an important evolution. While they were initially designed for the specific purpose of graphic rendering, most modern discrete GPUs offer the possibility of general-purpose computing. With the introduction of NVIDIA's CUDA [7] in 2007 and OpenCL [4] in 2009, GPUs have become commonplace for workloads that benefit from the massive parallelism they can offer. While the individual execution speed is still slow compared to recent CPUs, current-generation cards offer thousands of cores, enabling a huge performance boost for parallelizable operations.

The increasing number of use cases of general-purpose GPU computing includes computations on potentially secret information, e.g., neural networks [25] or cryptographic applications [22, 54]. Thus, general-purpose GPU computing also becomes a more interesting attack target. Recent research confirms these security concerns, as GPUs have become a recurrent target of side-channel attacks, exploiting various shared components [6, 16, 25, 28, 34, 36, 39, 42]. Furthermore, attackers may also leverage the GPU to attack other system components [24, 31]. As on CPUs, the GPU cache is a particularly interesting resource for side channels. Consequently, prior work also replicated well-known CPU cache side-channel attacks on GPUs [14, 16, 17, 34, 39], albeit only in native code so far.

While native code has direct access to a large variety of GPU APIs, e.g., CUDA, Vulkan, Metal, and Direct3D, acquiring native code execution is a significant hurdle for any attacker. Instead, the browser has become a more interesting attack vector, as users routinely run untrusted third-party code on their devices within the browser. Since GPU computing can also offer advantages for computations within websites, browser vendors decided to expose the GPU to JavaScript through APIs like WebGL and the upcoming WebGPU standard. WebGPU is not only available on desktop browsers but is also already partially supported on mobile devices in Chrome Canary version 117. As the future standard for web-based general-purpose interaction with GPUs, WebGPU aims to lay solid foundations for performance and security. The standard already has explicit mitigations against timing side channels [11], e.g., disabling timer access (making it a trusted feature), and mimicking the JavaScript mitigation against malicious use of the `SharedArrayBuffer` [23, 35, 41, 52]. Previous work demonstrated native code side-channel attacks on GPUs, where the browser triggered L1 and L2 cache activity, e.g., through WebGL [34]. However, the feasibility of a browser-based GPU cache side-channel attack, targeting a victim running in native code or another browser window, nor the possibility of an attack with the upcoming WebGPU standard [12] have been demonstrated yet. Considering the ubiquitous attack surface browsers offer to attackers, we need to investigate the following questions:

Can GPU cache side-channel attacks also be mounted from within a restrictive browser environment using APIs like WebGPU? Can these attacks be made generic enough to work on the wide spectrum of GPU hardware? To what extent can an attacker leverage GPU parallelization to enhance attacks?

In this work, we answer these questions by presenting the first end-to-end cache side-channel attacks from within browsers, leveraging the new WebGPU standard. Despite the inherent restrictions of the JavaScript and WebGPU environment, we construct new attack primitives enabling cache side-channel attacks with an effectiveness comparable to traditional CPU-based attacks. Our attacks are generic and automated, in the sense that our 2 attack primitives automatically determine GPU-specific configuration parameters required for an attack, *i.e.*, the cache hit-miss threshold, the cache size, and the number of cache sets. Consequently, our attacks work on a wide variety of devices, which we demonstrate in our evaluation: We show that our 2 basic attack primitives work on 11 desktop GPUs from 5 different generations and 2 vendors, NVIDIA and AMD. We demonstrate that based on these, we can also identify cache sets and monitor cache set collisions directly from a browser on a variety of NVIDIA GPUs.

We introduce 3 techniques to exploit cache contention on the L2 cache of discrete GPUs from JavaScript via WebGPU compute shaders. First, we highlight that significant cache eviction, often induced by graphical rendering, can enable attackers to discern instances of re-rendering. Second, we implement a templating attack within the browser, designed to monitor memory access patterns. Lastly, we present the first Prime+Probe attack on discrete GPUs executed from a browser. For all 3 attacks, we evaluate whether using the GPU's parallelism improves the basic attacks. For the eviction set construction in particular, we extend the state of the art by leveraging the massive parallelism of modern GPUs with the first parallelized eviction set construction algorithm.

We evaluate our attacks in 3 distinct scenarios covering both low-frequency non-repeatable events, as well as repeatable and high-frequency events: an inter-keystroke timing attack, AES key extraction, and the establishment of a covert channel, all initiated from a browser, *i.e.*, through an attacker-controlled website. Our keystroke monitoring attack detects inter-keystroke timings with F_1-scores in the range of 82 % to 98 %, and a sampling time below 15 ms, fast enough to distinguish even very fast typing. We successfully extract AES keys in 6 min with a precision of 100 %. Our templating approach enables us to profile the T-tables in 13 s on average, with the remaining time (5.7 min) dedicated to the last round attack. We perform the attack on 2 recent GPUs, a NVIDIA RTX 3060 Mobile and a NVIDIA RTX 3060 Ti, with similar results. Lastly, we demonstrate a covert channel with true channel capacities between 7.3 kB/s and 10.9 kB/s on the NVIDIA RTX 2070 Super, NVIDIA RTX 3080 and NVIDIA RTX 3060 Ti.

Our attacks require no user interaction and work within a realistic time frame a user might spend on a website, e.g., in the range of multiple minutes. Therefore, they can easily be implemented as drive-by attacks, targeting arbitrary users while browsing the Internet. Furthermore, since our attacks are based on WebGPU, they are applicable to all operating systems and browsers implementing the WebGPU standard and, as we demonstrate, to a broad range of GPU devices. Consequently, it becomes clear that browser vendors need to reassess their approach to offer GPU access to untrusted websites without user consent. Instead, we recommend a security-centric interactive approach that is already applied to all other security- and privacy-related resources, such as the microphone and the camera.

In summary, our paper makes the following main contributions:

1. We present the first end-to-end cache attacks on GPU caches from the browser, using the restrictive WebGPU API.
2. We evaluate our attack primitives and attacks on a wide range of GPU architectures and explore where the massive parallelism of GPUs can improve attacks.
3. Based on our insights, we develop the first parallel eviction set construction algorithm and the first Prime+Probe attack on the L2 cache of a single, dedicated GPU.

4. We describe a novel templating approach that we use in an attack on an AES T-table GPU implementation. Using predictable LRU cache set eviction cascades on GPUs, our attack can skip the lengthy set-construction phase by exploiting only contention in sets that are actually used by the victim.

Outline. Section 2 provides the background and Section 3 our threat model. Section 4 presents the primitives for Prime+Probe on the GPU from the browser. Section 5 explores an inter-keystroke timing attack. Section 6 evaluates our attack for an AES key recovery and Section 7 in a covert channel scenario. Section 8 discusses limitations and mitigations. Section 9 concludes.

2 Background

2.1 GPU architecture

The architecture of discrete GPUs may vary by brand. Hereafter, we focus on giving an insight into discrete GPUs architecture, tackling both computation and memory management. We default to the concepts and notations adopted by NVIDIA, but similar concepts are used by other manufacturers, such as AMD.

A GPU consists of multiple Streaming Multiprocessors (SMs), called Compute Units (CUs) on AMD cards. Each SM has its dedicated memory subsystem, including shared memory (SM-local memory), caches, and functional units, to execute multiple threads in parallel, operating under the SIMD paradigm. On GPUs, threads are organized into *thread blocks* (also called *workgroups* on WebGPU) that are assigned their own SM when executed. SMs consist of multiple processing blocks (4 on recent NVIDIA and AMD GPUs). Each processing block is a separate SIMD execution unit with its own load and store units capable of running 32 threads in parallel. Thread blocks are divided into *warps*, groups of 32 threads that are scheduled on processing blocks. Processing blocks have warp schedulers, hardware schedulers that schedule *warps* in and out of the processing block. When a *warp* has to wait for a memory access or register dependencies, the warp scheduler schedules a different warp that is ready to execute to keep the SIMD units busy. The constant rescheduling of warps allows for latency hiding and, therefore, more efficient use of processing blocks [7].

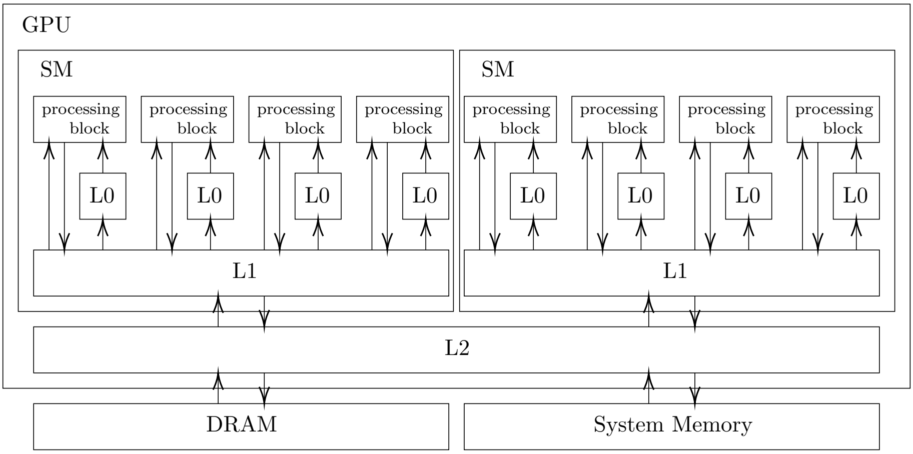


Figure 6.1: Modern GPUs (here NVIDIA) have an L0 cache per processing block, an L1 cache per SM, and a shared L2 cache.

Prior to the Volta architecture, all threads in a warp share the same instruction unit with a single program counter, *i.e.*, instructions execute in lockstep [40]. If threads in the same warp diverge, they are masked until they converge again. If some threads execute the if-branch and some execute the else-branch, the entire warp executes the if-branch and the else-branch with threads masked accordingly. Volta introduced independent thread scheduling, with a per-thread program counter and call stack, allowing the processing block to interleave execution of diverging branches [40].

Similar to CPUs, GPUs use caches to reduce the latency for memory accesses. Namely, each SM has a dedicated L1 cache that is shared between processing blocks, and each processing block has access to a smaller private L0 cache. Finally, GPUs share one global L2 cache (LLC) between the SMs. Figure 6.1 illustrates the cache hierarchy of Nvidia Turing GPUs. However, we note a few relevant peculiarities of GPU caches. First, there is no coherency protocol between caches, and maintaining coherency is the responsibility of developers. Second, unlike the classical 64-byte cache line in CPUs, GPUs' LLC commonly has 128-byte cache lines [1, 8].

2.2 GPU APIs

GPUs can be called through different APIs depending on the context. We distinguish two main API families: native APIs (e.g., OpenGL, Vulkan and CUDA), and web APIs (e.g., WebGL and WebGPU).

Native APIs. The most straightforward and efficient way to interact with a GPU is through dedicated native APIs. They enable the use of GPUs for either graphic rendering or generic computing. OpenGL was introduced in 1992 to support GPU-assisted rendering on Linux and Apple platforms, while Windows uses the Direct3D framework. For general-purpose computing, OpenCL, released in 2008, provides support for all major GPU vendors and is widely used. In 2007, NVIDIA released CUDA [7], a compute language specifically designed for NVIDIA GPUs. CUDA is supported by both consumer- and business-oriented NVIDIA GPUs. The high market share of NVIDIA GPUs and the ease of use of CUDA makes it the currently most widely used framework for general-purpose computation on GPUs. Apple recently dropped support for OpenGL in favor of Metal. Similarly, Vulkan was released in 2016 as a modern alternative to OpenGL. While these APIs have their differences, typical calls include operations on texture mappings, rasterization, and memory management on the GPU.

Web APIs. WebGL is the current baseline JavaScript API giving access to the GPUs rendering. As its name suggests, it was originally designed with a specific goal: graphic rendering in browsers. Hence, its API is limited and does not provide support for generic computations on the GPU [3]. This was the motivation behind the *WebGL 2.0 Compute* initiative [19]: "to bring compute shader support to the web via the WebGL rendering context". Due to the emergence of new native rendering APIs, the diminishing prominence of OpenGL, and the perceived constraints of WebGL, the project contributors decided to deprecate it [19] in favor of a more contemporary alternative, namely WebGPU.

Like WebGL, WebGPU provides access to the GPU graphics capabilities in the browser. It is, however, not a mere wrapper around OpenGL. More than that, it aims at being cross-platform and supporting modern graphic APIs, such as Vulkan, Metal, and DirectX, through JavaScript. Compared to WebGL, it offers a cleaner API, significantly better performance, and a more generic application range. At the time of writing, the standard is still under active deployment. However, the involvement of major browsers in this process, and the promising performance, foreshadow a widespread deployment in the next years. Chrome, Chromium, and Microsoft Edge already support WebGPU in their official release, and Firefox has it in its Nightly version [5]. Support of mobile GPUs is also in progress, with recent deployment on Android [2].

Developers can create rendering pipelines and manage GPU resources with WebGPU. WebGPU has its own shader language called WebGPU Shading Language (WGSL) to write custom shaders that are compiled at runtime. While WebGPU provides access to GPUs through native APIs, implementations of the standard may restrict the available GPU resources, e.g., memory and runtime, for security reasons. Without restrictions, big WebGPU workloads could significantly impact the useability of the host system [10], as most GPUs only allow for one active shader at a time.

2.3 Prime+Probe

In the last decades, microarchitectural attacks have been studied extensively. Prime+Probe [47, 56] is a cache-based attack that exposes the memory access patterns of a process by exploiting cache contention to leak the cache set accesses. This technique is particularly useful for attackers with limited control over the victim's machine, since it has low requirements and does not need shared memory or direct control over the cache with a flush instruction. Because of these weak assumptions, it is well-suited for browser-based attacks, where an attacker controls JavaScript on a web page [21, 49].

Assuming the attacker can execute code on the same processor as the victim, the attack works as follows. First, the attacker *primes* the cache by filling well-chosen cache sets with its own data. Then, they wait for the victim to make memory accesses. Finally, the attacker *probes* their data to access the same cache sets as before. If the victim accessed one of the sets monitored by the attacker, they will have evicted some of the attacker's data, causing a longer latency in the probing phase. In the context of a covert channel, the attacker would run both sender and receiver, and use the contention on the cache sets to build the channel.

2.4 Related Work

Covert and side channels on GPUs. Naghibijouybari et al. [6] describe multiple covert and side-channel attacks on GPUs. Wu et al. [15] exploit rendering contention to perform side-channel attacks on browsers. Many works consider a spy outside of the targeted GPU. Jiang et al. [28, 36, 42] present a cache-based attack, a shared memory attack, and a bank-conflict attack, all leading to a key recovery attack on AES. Similarly, Ahn et al. [16] exploit cache conflicts to recover an AES key from a GPU implementation. While they rely on cycle-accurate timers, our attack works from the browser without a timer. In addition, their spy uses the native API, while we perform our attack from the browser. More similar to our approach, Dutta et al. [18] perform Prime+Probe on Intel's integrated GPU through contention on the LLC shared between the CPU and GPU with native OpenCL. They also demonstrate ring-bus interconnect covert channel reaching the LLC. Dutta et al. [14] present a cross-multi-GPU Prime+Probe covert channel based on L2 contention. Our threat model is different, assuming a spy co-located on the same GPU in a drive-by attack from the web.

Naghibijouybari et al. [34, 39] present the first attacks in the co-located setting. Their first work [39] presents an in-depth study of General Purpose GPUs and highlights various ways to build covert channels on GPUs using caches and functional units. In their following work [34], they demonstrate the ability of an attacker to implement website fingerprinting based on GPU memory usage and performance counters. They also demonstrate the practical impact of their attack by tracking keystrokes from users and recovering some internal parameters of a neural network running on the GPU. Wei et al. [25] present a similar approach, using the GPU context-switching impact on performance counters to enhance the leakage and recover the complete structure of a neural network.

Despite the groundbreaking nature of these works, our contributions differ in key aspects. All aforementioned contributions rely on the attacker having access to the native APIs of the GPU through CUDA or OpenGL. This enables them to monitor high-precision performance counters. Our attack works entirely from the browser using JavaScript, with the corresponding API limitations (e.g., we do not have an accurate timer). This results in better portability but also a weaker attacker in our threat model.

Browser-based cache attacks on GPUs. The growth of web-based API usage to offer GPU-enhanced rendering inadvertently enables attackers to run GPU-based attacks through JavaScript, bypassing its existing limitations. To our knowledge, all existing works exploit the GPU through the WebGL API. Frigo et al. [31] leverage the integrated GPU to mount Rowhammer attacks from browsers on mobile devices, using the WebGL

```
if global_id.x != 0 {
  var time: u32 = 0;
  atomicStore(&timer, 0);
  while (atomicLoad(&stop) != stop_value) {
      for (var a: u32 = 0; a < 100000; a++) {
          time++;
          atomicStore(&timer, time);
} } }
else {
  start = atomicLoad(&timer);
  var c : u32 = atomicLoad(&buffer); //access
  if c != 0 { //prevent optimization
    return;
  }
  end = atomicLoad(&timer);
  atomicStore(&stop, stop_value);
}
```

Listing 6.1: Counting thread implementation in WGSL based on the global thread id and `atomic` operations.

timing APIs. In response, major browsers disabled this timer. Cronin et al. [17] presented a browser-based attack with assumptions similar to ours. They target SoC platforms and leverage system-level cache occupancy to build a covert channel and fingerprint websites. They differ from our work in multiple aspects. First, they focus on a SoC system and use contention on the system-level cache, which is shared between the CPU cores and its peripherals (namely the GPU) in ARM systems. This enables them to create contention from the CPU, whereas we consider spy and attacker to be co-located on the GPU. Second, the cache occupancy of the system-level cache is significantly different, resulting in different challenges to overcome. Finally, they exploit it using WebGL code, while we focus on its successor, WebGPU, which claims to consider and address the side-channels threat. Recently, Taneja et al. [9] demonstrated hybrid side channels on the CPU and GPU, based on how they adjust their frequency, power, and temperature depending on the workload. They demonstrate that GPUs exhibit instruction and data-dependent throttling. Their JavaScript attack assumes a victim in the browser but still relies on the ability of the attacker to access native APIs to monitor the power consumption and frequency of the GPU.

3 Threat Model

As we target WebGPU, our primary requirement is a browser with WebGPU support. As of writing, this includes Chrome releases since version 112, Chromium, Edge, and Firefox Nightly. By targeting web browsers, our threat model includes any scenario where a browser might run while sensitive information is being processed. Because the entire system usually shares the GPU, this can include anything rendered (such as websites or applications) and general-purpose computing operations. We show that our attack can be done in a drive-by manner, simply by visiting a website for a while. We assume that the victim will visit an attacker's page for several minutes, e.g., reading a blog with malicious WebGPU code. We do not assume that WebGPU provides any interface for hardware timers. To further constrain our attacker, we assume that WebGPU provides no workgroup memory in reaction to prior work [18]. In this paper, we attack dedicated NVIDIA and AMD GPUs, whereas some other works [31] have focused on integrated mobile GPUs.

4 WebGPU primitives

To build advanced cache attacks in WebGPU, we need several key primitives. The first is a timer accurate enough to reliably distinguish a cache hit from a miss. Using this timer, we can then detect cache size, cache activity, and build Prime+Probe eviction sets. While these primitives have been extensively studied on CPUs [30, 47, 49, 56], building them on GPUs involves some difficulties, especially from a browser. In this section, we detail these challenges and how to overcome them using WebGPU and minimal requirements.

4.1 Timing without clocks

Most previous works on GPUs are run natively and rely on high-precision timers or related performance counters for their measurements. However, WebGPU took explicit measures to preclude timing attacks, such as making `timestamp-query` optional and limiting interactions via shared buffers, similar to `SharedArrayBuffer` mitigations in browsers. To our knowledge, the shader language WGSL does not include any timers at this point. To present a generic primitive, we will construct our attacks without API-provided timers.

JavaScript encounters the challenge of imprecise timers, so prior works on the CPU had to consider similar constraints and employed counting threads [18, 35, 41, 43]. The idea is to set up a shared memory buffer and use a dedicated thread to constantly increment a shared variable. Another thread can then read this variable and interpret its value as a timer. When applying this concept to WebGPU, we face three challenges.

C1. Threads serialization. Different compute shaders cannot run at the same time. Therefore, the same shader needs to count on one thread and execute the attack code on another, as shown in Listing 6.1. While this would be straightforward on the CPU, GPU threads scheduled on the same processing block may run in *lockstep* on some architectures [13]. This means that if threads in the same warp need to execute different instructions (*warp divergence*), they would run sequentially, hindering the use of our counter as a timer.

C2. Memory coherency. Unlike for CPUs, GPUs have no automatic coherency guarantee in the memory hierarchy. Each SM manages a dedicated memory subsystem, so SMs may contain different copies of the same data in their L1 caches. Maintaining a coherent state by synchronizing the data is left to the developers. Therefore, without coherency, our counting thread would increment the timer in its private L1 cache, unobservable from the outside.

C3. Optimization. The WGSL compiler aggressively optimizes the code, such that a counting `while` loop may be replaced with the final result, and memory accesses may be replaced by registers.

Solutions. In their OpenCL implementation, Dutta et al. [18] solve the first challenge by executing enough counting threads to fill one or more warps. Then, they conduct the attack in a separate warp within the same SM, so each warp only executes the same branches, avoiding warp divergence. To address the other challenges, they simply store the counter in a shared memory region available to all threads in the same workgroup.

In line with our goal to get a portable and low-assumption attack, we suggest a more generic approach. To address **C1**, we set the workgroup size of the shader to 1, which prevents scheduling on the same processing block. Our solution demonstrates that even a strong security measure, like disabling shared memory, would not prevent timers in WGSL. We

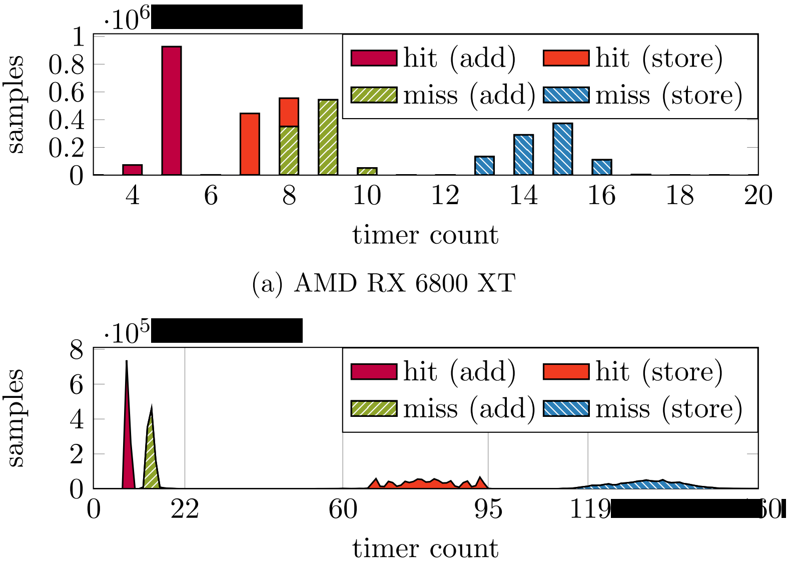
Table 6.1: Timing thread counter value for the 98th and 5th percentile for L2 cache hits and misses, respectively, for a variety of GPUs and the methods `add` and `store`. A good threshold can be found when the distributions are clearly separable. $n = 1\,000\,000$ hits and misses were recorded each.

		Add		Store	
	GPU	$\text{hit}_{>98\%}$	$\text{miss}_{<5\%}$	$\text{hit}_{>98\%}$	$\text{miss}_{<5\%}$
AMD	RX 6800 XT	6	7	9	11
	RX 6900 XT	5	7	9	11
NVIDIA	GTX 1070	5	8	62	95
	GTX 1650	7	13	75	94
	GTX 1660 Ti	7	11	74	94
	GTX 1660 Ti Lin	4	7	10	18
	RTX 2070 SUPER	6	8	80	106
	RTX 2070 SUPER Lin	4	8	11	18
	RTX 3060 Mobile Lin	5	8	11	18
	RTX 3060 Ti	8	13	90	124
	RTX 3060 Ti Lin	5	7	11	20
	RTX 3080	8	12	95	119
	RTX 4090	7	10	99	145
	Quadro P620	5	7	61	88

can solve **C2** and **C3** using `atomic` instructions (see Listing 6.1). First, they guarantee that memory accesses will not be turned into register accesses for optimization. Second, loads and stores performed by `atomic` instructions bypass the L1 cache to directly access the L2 cache, which enforces coherency. However, **C3** presents an additional challenge not solved by `atomic` operations. Sometimes the compiler will reorder or drop the measured load. We can prevent this by using the loaded value in a condition whose outcome the compiler does not know.

A minimal example of this approach is illustrated in Listing 6.1. One thread is chosen to be the timing thread via the global invocation id `global_id`, while the other can perform the attack. To spend as little time as possible reading the `stop` variable, the timing thread spends most of its time in a tight inner loop. All memory operations interacting with other threads are done with `atomic` instructions. The condition in Line 12 prevents compiler optimization of the load order or elimination of the target load.

Table 6.1 shows the hit-miss separation for both techniques on 11 different GPUs. We also find that on most cards, incrementing a local variable and



(a) AMD RX 6800 XT

(b) NVIDIA RTX 3080

Figure 6.2: WebGPU cache hit and cache miss histograms for different GPUs with counting thread for 1 million samples. Adding to a memory location provides less resolution than storing a register value. Higher counts show higher timer resolution.

updating it with `atomicStore` is significantly faster than using `atomicAdd`, since the latter is a blocking operation that requires waiting until the data is brought to the execution unit. However, this technique seems to be much less effective on AMD in general, but also on some Linux configurations (noted as Lin, as opposed to Windows default). Our testing revealed that this optimization may depend on multiple factors, such as the operating system and driver version (see Section 8). This optimization is of course only applicable to data accesses in L2 cache that are not meant to be timed. Figure 6.2 also highlights this difference but confirms that L2 cache hits and misses are clearly distinguishable in either case. In the end, our timer primitive cannot be prevented without removing the `atomic` operations, and its accuracy is only limited by the time it takes to load from and store to the L2 cache.

4.2 Cache-Size Detection

As our goal is to show that virtually all WebGPU-enabled devices are affected by these generic attacks, we try to hardcode as few parameters as possible. An important parameter for all further sections is the cache size. It determines how many sets we can expect (Section 4.3) and lets us derive suitable buffer sizes for cache eviction detections (Section 4.4).

We assume standard LRU, as suggested in previous work [26], and fill the cache with a large array of 10 MB. The buffer size choice is motivated by our observation that most GPUs have below 8 MB of L2 cache. In the same shader execution, we now iterate over the array forward and then backward, counting hits. Changing direction avoids self-eviction of an entire set after a single miss and allows us to accurately measure the number of cache lines that remain in the cache. If the hit rate is very high ($> 95\,\%$), we increase the test size in steps to 40 MB, 80 MB, and 100 MB. This keeps measurement times low for most cards, while allowing accurate detection even for larger caches. Finally, we match this approximate size to the closest larger size within a list of known sizes.

Table 6.2 shows that for most cards, we can reliably determine the cache size in less than 400 ms. Of interest among the outliers is the NVIDIA RTX 3060 Ti. It reliably returns a size of 3 MB, and indeed, we never see any hits more than exactly 3 MB, though the official L2 cache size is 4 MB. A simple explanation is that both our NVIDIA RTX 3060 Ti models, only have 3 MB of L2 cache. Another possibility is that these cards have a different mapping function, and some part of their cache is only reachable for much larger total VRAM allocations. We will encounter this again in Section 4.3.

4.3 L2 Cache Eviction Set Construction

The next step in building a Prime+Probe attack is to find a set of addresses that map to the same cache set. To make sure this set of addresses replaces all the current entries in the cache set, the cardinality of the set should at least match the cache associativity W. We call this an *eviction set*. On CPUs, much work has been done to reverse engineer the mapping from virtual-to-physical addresses to cache sets [46, 48, 50]. Comparable work on GPUs [14, 26, 27, 32, 44] however has shown that their cache set mapping can be much more complex. Jain et al. [26] used a modified

Table 6.2: Our WebGPU cache-size finding algorithm on a variety of GPUs, $n = 10$. With one exception, the correct size is almost always found on all cards.

		Size			Runtime	
		Actual	Detected	Correct μ	$\bar{x}$	σ
	GPU	MB	MB	%	ms	ms
AMD	RX 6800 XT	4.0	4.0	100	179.3	19.21
	RX 6900 XT	4.0	4.0	100	185.6	26.20
NVIDIA	GTX 1070	2.0	2.0	100	192.6	26.15
	GTX 1650	1.0	1.0	100	422.2	31.56
	GTX 1660 Ti	1.5	1.5	100	283.6	11.02
	RTX 2070 SUPER	4.0	4.0	100	189.9	6.15
	RTX 3060 Mobile	3.0	2.975	90	285.3	15.03
	RTX 3060 Ti	4.0	2.975	0	276.8	9.50
	RTX 3080	5.0	5.0	100	257.4	9.81
	RTX 4090	72.0	72.0	100	1 729.6	60.23
	Quadro P620	1.0	1.0	100	251.7	23.25

driver to reverse-engineer the hash functions for mapping addresses to both cache and VRAM on an NVIDIA GTX 1070 and 1080. However, our tests suggest these functions differ in newer generations of NVIDIA GPUs. In particular, many GPUs need to employ different non-linear (or linear, but different by address range) mapping functions due to their non-power-of-two cache and VRAM sizes. Additionally, AMD or even mobile GPUs may follow an entirely different scheme altogether. Like the work by Dutta et al. [14], we also do not have the advantage of relying on physically contiguous memory, or any specific page size.

In keeping with our generic approach, we do not attempt to rely on any known mapping functions or page sizes. Instead, we employ a generic set-finding algorithm based on prior work for CPU caches. Given a timer accurate enough to distinguish cache hits and misses, an attacker should be able to create eviction sets efficiently, similar to the methods presented by Qureshi and Purnal et al. [20, 29].

While this approach works well for CPUs, we encountered various challenges to efficiently port it to GPUs. Hereafter, we describe a novel approach to compute fast and reliable eviction sets on GPUs. In particu-

```
$\mathcal{S}$ ← {1.5x cacheSize, 128B steps}
Buckets ← {{}}
while $\mathcal{S}$ != {}
  $\mathcal{B}$ ← $\mathcal{S}$, $\mathcal{P}$ ← $\mathcal{B}$[0] ❶ //initialize B, select a pivot P
  while |$\mathcal{B}$| > targetSize
    $\mathcal{G}$ ← {}
    do
      shuffle($\mathcal{B}$)
      $\mathcal{B}$ ← $\mathcal{B} \cup \mathcal{G}$, $\mathcal{G}$ ← $\mathcal{B}$[0:$1/2W|\mathcal{B}|$], $\mathcal{B}$ ← $\mathcal{B}$\{$\mathcal{P} \cup \mathcal{G}$}
      access($\mathcal{P}$), parallelAccess($\mathcal{B}$) //access pivot, then B
    while isCached($\mathcal{P}$) ❷
    if optCondition() 2b
      hits ← accessAndMeasure({$\mathcal{B} \cup \mathcal{P}$})
      $\mathcal{B}$ ← $\mathcal{B}$\hits //remove addresses not part of eviction sets
  $\mathcal{B}$ ← $\mathcal{B} \cup \mathcal{P}$, $\mathcal{S}$ ← $\mathcal{S}$\$\mathcal{B}$
  Buckets ← Buckets ∪ {$\mathcal{B}$}
```

Listing 6.2: Parallel Set Construction. This simplified pseudo-code algorithm partitions an initial set of addresses S into several buckets B whose addresses do not share cache sets.

lar, we describe how to leverage the powerful parallelism that GPUs offer to speed up this process.

The basis of our implementation is the Group-Elimination Method (GEM) [29]. The goal of GEM is to find an eviction set for a target address. To this end, a large set of addresses $S >> W$ that evicts the target address is partitioned into $W + 1$ groups. As a full eviction set of W addresses must be contained in some combination of $\leq W$ out of the W groups, (at least) one group can be eliminated without affecting the eviction. GEM tries to remove each of the $W + 1$ groups from the set S until one is found that does not influence the eviction of the target. This is repeated until only W addresses remain in S, forming an eviction set for the target address.

Our implementation differs from GEM in two significant ways. First, we aim to find all sets, and we, therefore, try to find more than one eviction set at a time. Similar to Prime+Prune+Probe [20], we make use of the predictable behavior of LRU for this. Second, we parallelize parts of the algorithm to multiple threads. Many constants in the following are empirically determined values that work on a variety of GPUs, not optimal values.

Parallel Set Construction. See Listing 6.2. When we access many addresses in parallel on the GPU (or the CPU), ordering between them is not guaranteed. This means that when a set is split between different threads, we can no longer expect to observe effects stemming from LRU. In effect, eviction *measurements* that rely on access order become meaningless. We, therefore, add a preprocessing step to the eviction-set construction algorithm.

The goal of preprocessing is to separate an initially large set S of addresses $s_i = |S|$ (1.5x the cache size in 128 B steps) into buckets with no overlapping sets. This partitioning facilitates the independent examination of each bucket for sets, circumventing inter-thread interference. The process follows a similar approach as GEM and is delineated in two main steps.

Starting with $B = S$, the first (❶) step involves selecting a random element from the set as our *pivot*. This pivot address guarantees the presence of at least one complete set within the bucket, although it probably contains several more. The second (❷) step consists in removing a portion of the set, *i.e.*, a group, and verifying if the pivot element is still evicted by the residual B. If eviction is not observed, we reiterate with another group. Contrary to GEM, we find that eliminating $^1/_{2W}|B|$ rather than $^1/_W + 1$ better mitigates the excessive removal of elements in later steps (2b).

We also incorporate several optimizations not found in GEM. Until B diminishes to $^3/_4 s_i$, we exploit parallelism by accessing all set elements concurrently using 30 threads, measuring only the pivot at the end. When $|B\ | < {}^1/_6 s_i$ or on every fourth iteration when $|B\ | < {}^3/_4 s_i$ (`optCondition` is met, Line 12), we measure not just the pivot but all other elements . We only do this sparingly because measuring elements in addition to accessing them has a significant overhead, and, as mentioned, LRU-related observations can't be parallelized while sets are still unknown. However, this enables a crucial optimization: the removal of all set elements that register a cache hit (2b). Given a consistent access sequence and a cache replacement policy approximating LRU, all persisting elements in B are now part of full eviction sets.

This procedure can be iterated until B is below a predetermined threshold. Empirical evaluations suggest a bucket size of 3500 (equivalent to $145 - 206$ sets) works for the majority of GPUs. Upon completion, B is subtracted from S. We are left with a bucket of the desired size, exclusively containing eviction sets. The residual segment of S does not include overlapping sets with the bucket. Repeating the previous steps ensures that the final buckets

```
Buckets = ParallelSetConstruction()
originalBuckets ← Buckets, EvictionSets ← {}
Pivots ← {ℬ[0] | ℬ ∈ Buckets)} Ⓐ
Buckets ← {{ℬ\𝒫} | (ℬ,𝒫) ∈ (Buckets,Pivots)}
while ∃ℬ : ℬ ≠ {}
  Gs ← {{}}
  foreach ℬ ∈ Buckets : ℬ ≠ {}
    if |ℬ| > 1000
      𝒢 ← ℬ[0:1/2W|ℬ|] Ⓑ
    else
      𝒢 ← ℬ[0] B2
    Gs ← Gs ∪ {𝒢}, ℬ ← ℬ\𝒢
  AllHits = parallelMeasure(Pivots, Buckets) Ⓒ
  foreach (𝒫,Hits,ℬ,𝒢) ∈ (Pivots,AllHits,Buckets,Gs)
    if isCached(𝒫)
      ℬ ← ℬ ∪ 𝒢
      shuffle(B)
    else
      ℬ ← ℬ\Hits Ⓓ
      if |ℬ| == W //bucket has reduced down to one set
        EvictionSets ← EvictionSets ∪ {ℬ ∪ 𝒫}
        ℬ = originalBucket\{ℬ ∪ 𝒫} //refill Bucket
        shuffle(ℬ)
        𝒫 ← ℬ[0], ℬ ← ℬ\𝒫  Ⓐ
      else if |𝒢| == 1 && |Hits| == W Ⓔ
        EvictionSets ← EvictionSets ∪ {𝒢 ∪ ℬ} //free set!
```

Listing 6.3: Parallel Bucket Sifting. This algorithm sifts sets in parallel from the previously separated buckets.

consist only of non-overlapping eviction sets, collectively representing nearly all cache sets.

Parallel Bucket Sifting. With full eviction sets sorted into roughly equal buckets, we can now begin to extract single sets from them. Since there is no more overlap between the cache sets in the buckets, we can now run measurements on them in parallel. For the NVIDIA RTX 3080 and its 5 MB cache for example, the previously mentioned target bucket size produces around 16 buckets of 160 sets each, with up to 24 addresses per set. This means we can start a loop on our 16 input buckets with the following broad steps running in parallel for each bucket. First, we once again shuffle the elements in each bucket B and pick a pivot element to find an eviction set for (Ⓐ). Second, like before, we remove groups of $1/2W|B|$ elements until we find one that doesn't affect the pivot's eviction

(B). Third, to determine eviction, we measure the access latency for all addresses remaining in all buckets in parallel. (C) Fourth, addresses in buckets that show cache hits are also removed, such that all remaining addresses still form eviction sets within B (D). Buckets are shrunk in parallel this way until a bucket's size goes below 1000 elements. At this point, instead of $^1/_2W|B|$, we remove only a single element per loop (B2). This allows us to make use of the *cascading eviction effect* of the LRU replacement policy: when we remove only one address, that together with W other addresses in B forms an eviction set, those W addresses will now show up as cache hits (E). In effect, we have found an entire eviction set in a large bucket B by removing a single element. This allows us to *sift* out many sets for "free" while trimming the bucket to find the pivot element's eviction set. We continue decreasing the bucket size until either a complete eviction set for the pivot remains, or some false measurement has left us with an incomplete set. At this point, we refill the bucket with all discarded addresses that could not be attributed to a complete set and start again at step one. The algorithm terminates when either all buckets are empty, or no new sets have been found for too long.

This sifting method is so effective, in fact, that it finds significantly more sets than the number of pivots chosen. On our NVIDIA RTX 3080, for example, we might search 17 buckets for eviction sets with 90 chosen pivots (an average of 5.2 bucket "refills"), but sift out 2465 sets on the way. The change at 1000 elements represents an empirically found trade-off between fast bucket-shrinking and a high amount of sets found through sifting. When the number is too high, the time to find sets will needlessly increase, as most sets start with an average of 24 addresses, but can only be detected when just 16 are left in the bucket. When it is too low, many sets are lost to the sifting method through the removal of many elements.

Combining these optimizations, we can map most sets in the L2 cache of all NVIDIA GPUs in WebGPU in a reasonable time frame, as shown in Table 6.3. The notable exception is the NVIDIA RTX 4090, as the enormous cache size presented problems not found in other cards. Likewise, both AMD cards fail this important step to further attacks and are therefore not included in the more advanced attacks. One possible explanation is that the timing difference to other cards, which can already be seen in Table 6.1, causes more noise, as the hit and miss distributions are closer together. While we believe that from the basic timing difference, it is clear that all our attacks *could* run on these cards, we only had temporary and time-restricted remote access to these GPUs, which did not allow for

Table 6.3: Our WebGPU set-construction algorithm on a variety of GPUs, $n = 10$. All but one card reliably find $> 80\%$ of sets.

		Sets			Runtime	
		Overall	Found $\bar{x}$	Found σ	$\bar{x}$	σ
	GPU		%	%	min	min
NVIDIA	GTX 1070	1 024	96.0	2.1	11.8	4.8
	GTX 1650	512	82.9	2.2	4.2	3.9
	GTX 1660 Ti	768	96.4	2.1	12.1	3.8
	RTX 2070 SUPER	2 048	98.7	1.0	7.0	2.1
	RTX 3060 Mobile	1 536	99.9	0.1	2.3	0.4
	RTX 3060 Ti	1 536	94.5	5.3	2.6	1.5
	RTX 3080	2 560	99.3	1.9	2.8	1.2
	Quadro P620	512	50.8	24.5	13.7	9.0

analyzing the underlying problem. The NVIDIA RTX 3060 Ti also sticks out, as it consistently finds close to 1536 sets even when looking for 2048. This is consistent with 3 MB of L2 cache found in our experiments (see Section 4.2).

We see that the percentage of sets we find varies along with the time, though a majority of sets can almost always be found within 5 minutes. With this additional primitive, attackers can implement Prime+Probe to build a covert channel, as we show in Section 7, or execute some other cache attack, e.g., Rowhammer [31].

4.4 Full Cache Evictions

One of the first observations while measuring cache hit rates on GPUs is that some events evict a sizeable portion of the cache. Whenever an element on the screen is redrawn or the frame buffer is refreshed for some other reason, this occupies a large part of the cache. Depending on the total size of the L2 cache and what is being drawn, this may even evict the entire cache. On the one hand, this presents as noise during some attacks; each measurement that happens after a draw event is tainted. On the other hand, these evictions are indicators of activity on screen and can therefore be used as a side channel to user activity. We describe an inter-keystroke timing attack based on this primitive in Section 5.

5 Full-Eviction Keystroke Monitoring

Starting from the observation that drawing elements on screen evicts a significant part of the cache, we build an attack that records inter-keystroke timings by observing cache contention. As shown in prior work [33, 37, 45, 57], inter-keystroke timings carry significant information and can lead to password recovery.

While subsequent sections of this paper present conventional benchmarks for high-frequency side channels, this section focuses on low-frequency benchmarking. Despite the infrequent occurrence of events, achieving a high detection rate is crucial for accurately measuring inter-keystroke timings. In addition, keystroke profiling represents a practical application of our attack, as our setup mirrors the most prevalent end-user scenario: a computer equipped with a single discrete GPU engaged in internet browsing. As the full WebGPU standard becomes increasingly integrated into mobile devices, this scenario will gain further relevance in the future. Our approach for this attack is similar to Naghibijouybari et al. [34].

5.1 Construction

The attack is based on the following observation: for each character typed, the text box is re-rendered. We can measure this as the eviction of a certain amount of the cache, up to the entire cache, correlated with the size of the rendered area. To see this effect, we use a buffer that covers a part of the cache size and repeatedly measure its hit rate. Whenever we see a hit rate below a well-chosen threshold, e.g., 50 %, we record the timestamp as an event. The time resolution of this attack is determined by how fast our attacking shader can complete its measurement, which is determined by the total buffer size. Though on some GPUs we see that a small percentage of the cache is already enough to observe keystrokes, we find that for most, 35 % is a good tradeoff between detection and speed. The screen resolution, size of the text box and zoom level all contribute to the amount of evicted cache lines.

After recording raw traces, we filter based on two observations. First, very close measurements (<25 ms difference) are unlikely to be separate keystroke events. Second, after a short break in typing, the cursor starts blinking at a 530 ms interval on Windows. Filtering these sources of noise removes most false positives. Figure 6.3 shows the trace of an attacker

Table 6.4: Efficacy of WebGPU inter-keystroke timing detection on a variety of GPUs for 100 keystrokes.

		Performance Metrics			False		True	Interval Error		
		F_1 score	precision	recall	Positive	Negative	Positive	$\bar{x}$	σ	median
	GPU							ms	ms	ms
AMD	RX 6800 XT	0.27	0.16	0.99	533	1	99	133.58	101.18	121.50
	RX 6900 XT	0.29	0.17	0.99	490	1	99	126.05	119.22	86.00
NVIDIA	GTX 1070	0.97	0.99	0.96	1	4	96	−0.28	4.25	0.00
	GTX 1650	0.82	0.70	0.99	42	1	99	12.13	20.43	1.00
	GTX 1660 Ti	0.87	0.78	0.98	27	2	98	5.73	26.52	0.00
	RTX 2070 SUPER	0.86	0.78	0.97	28	3	97	19.81	57.81	0.00
	RTX 3060 Mobile	0.98	0.99	0.98	1	2	98	0.01	1.49	0.00
	RTX 3060 Ti	0.97	0.98	0.97	2	3	97	1.76	21.12	0.00
	RTX 3080	0.94	0.92	0.96	8	4	96	1.27	14.44	0.00
	Quadro P620	0.98	0.99	0.97	1	3	97	−5.57	54.90	0.00

typing at varying speeds compared to the ground truth on our NVIDIA RTX 3080. We can see that while we measure some spurious events, most timings are accurate.

5.2 Evaluation

We tested this attack with a small text box and generated input directly injected from javascript, randomly drawing inter-keystroke timings from distributions similar to the patterns observed by Song et al. [57]. Table 6.4 shows the tested GPUs and their F_1 scores and inter-keystroke timing errors. During this test, no other visual noise was present, similar to the static login pages of many websites. The consistently high recall shows that virtually no keystrokes are missed on most cards. However, even after filtering, the recall shows that there is a low average of false positives for most cards. AMD once again behaves differently. Despite the high recall, with the low precision, we can consider this attack mostly failed or severely degraded. The results suggest either a high level of noise or, more likely, frequent misclassification of hits as misses due to the close timing differences visible in Table 6.1.

An interesting example for the timing resolution is the NVIDIA RTX 4090. Because of its large L2 cache of 72 MB, simply measuring cache contention requires a disproportionately large measurement set. This is because the cache footprint of a text box does not increase with the cache size. While all other cards easily reach a sampling rate below 15 ms, the huge buffer means that each measurement takes more than 200 ms, making an inter-keystroke timing attack with this method impractical.

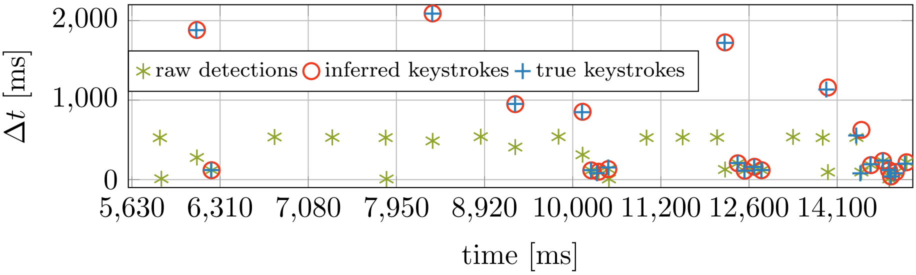


Figure 6.3: Inter-keystroke timing recovery on a NVIDIA RTX 3080. The raw activity detections (green) show prominent cursor blinking that can be filtered out very well, which leaves an accurate trace of inter-keystroke timings (red).

We also observe that on Windows, the blinking of the cursor causes slightly less eviction than a typed character. One possible explanation is that instead of re-rendering the entire text box, the cursor is drawn on top.

6 AES Key Recovery with Browser-based Templating

Recovering AES keys from vulnerable T-table implementations has become a benchmark for assessing how fine-grained side-channel attacks and microarchitectural attacks are. This case study has also been adopted in GPUs [16, 28, 36, 42]. Additionally, AES has been proposed as a use case for general-purpose GPU computing since 2007 [22, 54].

Unlike previous research, our method adopts a set-agnostic approach, eliminating the need to understand cache set sizes or mappings. Traditional set-based strategies would require extensive profiling of cache sets and mapping of T-table accesses. Our methodology bypasses this initial step, focusing on locating addresses congruent with T-table lines.

6.1 Threat Model

Like earlier, we assume our attacker embeds some malicious JavaScript in a webpage the victim is browsing for several minutes. The victim runs a GPU-based AES implementation that can be queried for encryption with a chosen plaintext and key. The attacker aims to recover the AES key used by a victim. This scenario could be found in the case of an SFTP server, where the chosen plaintext and key represent downloading our own file. In order to implement a last-round attack, we assume the attacker has access to the victim's ciphertexts, but not the plaintext or the key.

6.2 AES Implementation Details

The native encryption service is an AES CUDA implementation, which uses combined T-tables for all rounds. This increases the difficulty of the attack compared to implementations that use separate tables for the last round, as all other rounds influence cache hits on table entries. As GPU cache line width is usually 128B, and each table is composed of 256 4-byte entries, each table fits in exactly 8 cache lines, for a total of 32 cache lines filled with table entries.

6.3 Attack Methodology

Our strategy, akin to the keystroke attack (Section 5), involves allocating a large buffer to occupy a significant cache portion, executing an AES encryption, and then identifying evicted buffer offsets using Prime+Probe. In an ideal scenario with a minimalistic AES kernel, evicted offsets would correlate with the table or the encryption's inputs and outputs. This is because, from our observation, GPUs implement a deterministic LRU-like eviction policy. This means that when one address from a full set is evicted, measuring all others in the same order used to place them in the set will cause a cascade of cache misses, as each new access will result in a miss, overwriting the next address. In practice, we find that kernel loading introduces substantial cache occupancy, leading to measurement noise. Our primary challenge is discerning the tables amidst this noise, and profiling each table's cache lines to track their access. We employ chosen keys and specially crafted plaintexts to deduce the relationship between our offsets and table entries. With this mapping, we can execute the traditional last-round attack [51, 55]. Hereafter, we delve into each step and the optimizations we employed to achieve a reliable key recovery in a drive-by manner.

Profiling T-Tables. The initial *profiling* phase allocates a sizable array (optimal size varies across models, even with similar cache sizes) in the browser, ensuring kernel loading and encryption evict specific offsets, and templates the AES encryption's memory accesses. Using random plaintexts, we would expect a random distribution of the access to each entry of the tables, thereby causing a predictable eviction frequency of the set-congruent offsets (at a frequency of 0.995). On the contrary, the offsets that are set-congruent to the memory required for kernel loading would be evicted every time, and other noise artifacts should be sparse. Our differential access templating, using a fixed key and chosen plaintexts, enhances profiling reliability and efficiency.

The strategy involves pre-generating 32 plaintexts p_i with a fixed key, ensuring the encryption of each plaintext access all but one cache line within the tables, and a reference plaintext p_r, which encryption accesses all cache lines. Comparing memory accesses during the encryption of p_i and p_r reveals offsets congruent to cache line i. This process identifies offsets that are set congruent to each cache line, though some cache lines may remain undetected due to kernel loading *noise*. This refined offset list streamlines the attack, focusing on a reduced subset of offsets.

Last Round Attack. As we are performing a last-round attack, the only requirement is that we can make measurements during encryptions by the victim, and observe the ciphertext. The attack aligns with the non-elimination method presented by Neve and Seifert [53]. The idea is, given a collection of ciphertexts and the access to T-tables entries that happened during the encryptions, to remove possible values on the key bytes by looking for cache lines that were *not* accessed in the process. For each cache line not accessed during the encryption, we can remove all last-round key bytes that would have resulted in a memory access during the last round, based on the ciphertext value. Given the cache line size of GPUs, we get up to 2^4 bit of information on the last-round key every time a cache line is not accessed.

The more cache lines we can monitor, the more likely we are to reduce the search space for the last-round key. Once we get below 2^{40} candidates, we switch to an exhaustive search of the key.

6.4 Evaluation

For our evaluations, we focus on a CUDA-based target implementation, rendering evaluations on AMD cards infeasible. Somewhat breaking with

Table 6.5: Evaluation of the AES Last Round Attack (LRA) on two NVIDIA cards. All values are average across $n = 50$ runs.

GPU	Measurements	Time (min)		
	(x1000)	Profiling	LRA	Total
RTX 3060 Mobile	9.3±1.6	0.23±0.1	5.7±0.9	6.0±1.0
RTX 3060 Ti	9.8±3.4	0.23±0.1	5.9±1.9	6.1±2.0

the theme of this work, the nature of this attack necessitates some parameter adjustments, which adds complexity and extends the evaluation duration compared to other attacks. Therefore, we settle on evaluating our attack on two recent cards: NVIDIA RTX 3060 Ti and NVIDIA RTX 3060 Mobile. We run all our experiments on Ubuntu 22.04 and Chromium 117 but we also have observed consistent results Chrome versions 112 to 115.

Table 6.5 showcases our findings. It highlights the average duration of the attack's primary steps and the mean number of encryptions required for successful key recovery. Notably, both cards yield similar results, recovering the key in 6 min. The average encryptions needed are 9300 and 9800, respectively. The uniformity of results across GPUs, coupled with the low standard deviation, underscores the stability and reproducibility of our attack.

Profiling the T-table takes on average 13 s. The variability in this phase predominantly stems from the inconsistent repetition of profiling until an optimal buffer size is identified, enabling sufficient eviction observation. Typically, a single profiling session lasts 6 s. Once profiling is complete, the same session can be repurposed to divulge multiple AES keys, thereby reducing the attack duration to the sample collection time needed in the concluding step.

Profiling often does not provide a complete mapping for every cache line. The disparities in measurements and time allocated for the last-round attacks correlate directly with the number of cache lines we can monitor. On average, we can spy on $^{20}/_{32}$ cache lines. The NVIDIA RTX 3060 Ti exhibits marginally less consistent results, occasionally mapping fewer cache lines, leading to an elongated attack duration and increased standard deviation. Our evaluation on both cards consistently had a 100 % success rate.

7 A Prime+Probe Covert Channel

A covert channel is a channel that is constructed on top of some shared resource that is not meant for data transmission. This allows an attacker to transmit data between two domains that should be isolated or strictly monitored. Because both sender and receiver work together to transmit data, covert channels are a valuable benchmark for any side channel's bandwidth. In a traditional Prime+Probe cache covert channel, the sender transmits bits by *priming* (evicting) cache sets to transmit a binary 1, which the receiver can later detect by *probing* (measuring) its own lines in the same set. With our reliable timer and a method to find the required eviction sets (see Section 4), we can now construct a Prime+Probe cache covert channel for the L2 cache.

The sender is a C++ application that uses CUDA. In this scenario, it is a malicious application without network privileges but with access to the GPU. The sender's goal is to exfiltrate sensitive data via a GPU covert channel. The receiver runs in a website the user visits at the same time. This may be a legitimate website with injected malicious JavaScript, or a website the user is led to in some way.

7.1 Construction

We write the **Sender** $\mathcal{S}$ in C++ and CUDA, making full use of the native high-resolution timer. The browser-based **Receiver** $\mathcal{R}$ uses a combination of JavaScript and WGSL. Using our eviction set construction (see Section 4.3), $\mathcal{R}$ starts by mapping all cache sets.

Setup - CJAG. As neither $\mathcal{R}$ nor $\mathcal{S}$ have absolute labels for their respective eviction sets, the first step is to communicate the shared sets from $\mathcal{S}$ to $\mathcal{R}$. For this, we implement a GPU-friendly version of the cache jamming agreement (CJAG) proposed by Maurice et al. [38]. In CJAG, $\mathcal{S}$ alternates between *jamming* a set, *i.e.*, evicting it continuously for some time, and *probing* the set for a slightly longer period. Meanwhile, $\mathcal{R}$ probes all sets continuously until the jammed set is detected. Then, $\mathcal{R}$ switches to a longer period of *jamming*, so that $\mathcal{S}$ knows the set has been received and moves on.

Unlike the CJAG approach on the CPU, distinct shaders do not execute concurrently on the GPU, making simultaneous detection and jamming infeasible. Rather than employing shaders that continuously loop through jamming or detection, we need to segment them into single invocations. Depending on the frequency of driver interruptions, we might otherwise see long shader executions that rarely interface with each other.

Additionally, we want to use a large number of sets (e.g., 1 024). Serial transmission, as implemented in CJAG, is, therefore, impractical. Instead, we enhance the CJAG framework by leveraging the inherent parallelism of our GPU, enabling both $\mathcal{S}$ and $\mathcal{R}$ to jam and detect all sets concurrently. Here, copying to and from shaders is the main bottleneck, with 3 ms on average. Thus, the time difference between measuring a single set versus 64 sets per shader invocation is marginal. So, we combine both as a trade-off and measure sets in parallel on 16 threads. At this stage, $\mathcal{S}$ also swaps out any sets that are not detected from $\mathcal{R}$'s jamming, thus ensuring that all sets are fully functioning for both parties. After a selection of 1 024 sets has been communicated, $\mathcal{S}$ switches to jamming on only half of all sets. The specific half is dictated by the current bit in the index number of its cache sets. In this way, $\mathcal{S}$ can transmit the order of all 1 024 sets in $\log_2(1024) = 10$ steps by jamming different 512 sets for each bit. After all sets have been communicated, data transmission can begin. Table 6.6 shows that it takes 14 s to 28 s to transmit 1024 sets.

Transmission. After the set jamming agreement has been completed, the transmission is entirely one-way. We opt for a channel design where sender and receiver are synchronized with the wall clock. In native C++, this provides at least µs accuracy, while in browsers, this is limited to 100 µs. We choose a default transmission window length of 5 ms for our packets. This length is limited not only by the accuracy of the timer but also by the time it takes for a shader to run. To compensate for the long packet duration, we use the GPU's parallelism to transmit on 1 024 sets concurrently.

While eviction for $\mathcal{S}$ is as simple as accessing many addresses in parallel in a loop, $\mathcal{R}$ still needs to measure time. Challenge **C1** (see Section 4.1) means that we need to separate each parallel thread into different workgroups to prevent lockstep execution. Additionally, individual sets always need to be measured by the same thread, as ordering between these accesses is crucial for the eviction policy.

As observed in Section 4.4, GUI-related events can introduce undesirable noise. Similarly, the operating system can deschedule $\mathcal{S}$ for periods of time. To reduce such noise, we adopt the following strategies. First, we

Table 6.6: Transmission results of our Prime+Probe covert channel from a native CUDA sender to a WebGPU receiver in the browser, $n = 10$. True channel capacity can vary widely either due to general noise, incorrect set transmission during CJAG or too few correct reads per window, *i.e.*, the number of correctly received pairs within the transmission window.

		Configuration			CJAG	Bandwidth			Bit Error Ratio	
		Sets$_{tx}$	Window	Reads/window $\bar{x}$	Set Tx	Raw	True $\bar{x}$	True σ	$\bar{x}$	σ
	GPU		ms		s	Byte/s	Byte/s	Byte/s	%	%
NVIDIA	RTX 2070 SUPER	1 024	6.0	3.3	14.6	10 666.7	8 963.0	360.8	2.4	0.7
		1 024	5.0	2.2	14.0	12 800.0	8 962.0	286.5	5.3	0.6
	RTX 3060 Ti	1 024	6.0	2.9	16.4	10 666.7	9 004.9	271.4	2.3	0.5
		1 024	5.0	1.9	15.7	12 800.0	7 272.0	1 252.5	9.1	3.1
	RTX 3080	1 024	5.0	2.7	27.8	12 800.0	10 897.5	698.3	2.2	1.0
		1 024	4.0	1.9	28.2	16 000.0	5 964.5	1 048.6	15.9	2.7

employ a *majority vote measurement* approach where each set is measured as often as possible within a transmission window. By counting evictions and non-evictions, we obtain the result through a majority vote. Second, we access the addresses within each set in an *alternating order*. This ensures a consistent read from the most to the least-recently-used cache line, precluding the cascade of self-evictions that would arise if the oldest cache line were evicted. This lets us determine how much of a set was evicted and easily identify low-level noise. Lastly, we use a *differential measurement scheme*. Here, a pair of sets transmit 1 bit, and measurements wherein neither or both sets are evicted are discarded. In a valid transmission, precisely one set is evicted for every pair, effectively halving our transmission rate and resulting in a total packet length of 64 B. Consequently, the raw transmission speed is fixed by the parameters to a default of 12.8 kB/s.

7.2 Evaluation

We evaluate the covert channel on 3 NVIDIA GPUs; the RTX 2070 SUPER, 3060 Ti and 3080. The GTX 1070 and Quadro P620's Pascal architecture does not support all the instructions used by the CUDA sender. AMD cards do not support CUDA, though as set-finding fails on AMD (see Section 4.3) the attack would not work either way. The GTX 1650 and GTX 1660 Ti both support the instructions as well as set-finding, but we could not reliably establish communication because of malfunctioning jamming detection in CUDA.

Table 6.6 shows the configuration and transmission details for all tested devices. We can see that as we shrink the transmission window, average reads in the window go down, and the error rate increases. At 4 ms, the NVIDIA RTX 3080 shows a decrease in true channel capacity compared to 5 ms for this reason. Because of it's higher clock speed, the 3080 supports faster transmission than the two other cards. Its average true bandwidth in its fastest configuration is, therefore, 10.9 kB/s, at a BER of 2.2 %. Though our channel is non-optimal and slower than prior work, it clearly demonstrates the viability of using WGSL code embedded in a website as a covert channel receiver.

8 Discussion

Supported Devices. Our research primarily targets recent NVIDIA GPUs, leading to worse results on AMD cards, as we only had very limited access. Despite these architectural differences, WebGPU clearly enables generic cache attacks from browsers. At the time of writing, WebGPU is already integrated into Android's Chrome Canary, though some features are not yet available. Once parity is achieved, the potential for browser-based GPU attacks could significantly increase.

Limitations. We evaluated our proof-of-concept on various operating systems using Chrome and Chromium versions 112-117. Despite identifying functional combinations for all devices, the WebGPU implementation remains inconsistent, as evidenced by our experiments. The same code might succeed in one version and unexpectedly fail in another, potentially due to variations in WebGPU's code compilation beyond user control. We observed notable differences between Linux and Windows (see Table 6.1). While the exact cause—whether driver, browser, or WebGPU's underlying framework (e.g., Vulkan vs. DirectX)—remains unclear, the fundamental time discrepancy supports the viability of these attacks.

Countermeasures. The attacks shown in this paper are generic and rely on only a few assumptions. Nevertheless, steps can be taken to limit the attack surface. As already suggested in the current WebGPU draft, timers can be made optional, very coarse, or ideally removed altogether [11]. However, as we have shown, as long as coherent memory is available between concurrent threads, it is possible to construct a timer. If, however, the coherency mechanism (in our case, `atomic` operations) were to be changed, such a timer would quickly fail. Of course, this could cause normal workloads to malfunction unless specifically redesigned.

The simplest and most effective solution against a drive-by attack scenario is, in our opinion, to treat GPU access in the browser as a sensitive resource, like microphone or camera access, that requires permission before use. For WebGL and WebGPU, this is not currently the case (Firefox 114, Chrome 115, Chromium 117). This would also prevent malicious parties from stealthily using local computing resources for, e.g., cryptomining.

Disclosure. We have disclosed our results to Mozilla, AMD, NVIDIA and the Chromium team.

9 Conclusion

GPUs have become a ubiquitous computation resource and as such require increased security scrutiny. We showed that it is possible to mount powerful GPU cache side-channel attacks directly from within the browser. We demonstrated that our basic attack primitives are generic and automated to the extent that we can run them without manual intervention on a set of 11 desktop GPUs from 5 different generations and 2 vendors, running in the browser through WebGPU. We showed that the massive parallelism of modern GPUs can be leveraged in parallelized eviction set construction algorithms. Our three case studies emphasized the relevance of our work: Our inter-keystroke timing attack, with F_1-scores between 82 % and 98 %, exposes sensitive user input to an attacker. Our set-agnostic end-to-end attack on GPU-based AES encryption leaks full AES keys in 6 min, showing that also cryptographic secrets are exposed to browser-based attackers. Our native-to-browser Prime+Probe covert channel shows that the bandwidth of this channel can reach average transmission rates of up to 10.9 kB/s. Since our attacks require no user interaction, they can be implemented as dangerous drive-by attacks. Thus, we conclude that GPU access should be treated as a similar security and privacy risk as other devices and resources that require explicit user consent.

Acknowledgments

This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM 10.55776/I6054). Additional funding was provided by a generous gift from Red Hat. Any opinions or recommendations expressed are those of the authors and do not necessarily reflect the views of the funding parties. We also thank Gregor Heindl for his generous donation of time and hardware.

References

[1] AMD. AMD RDNA Whitepaper. 2023. URL: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf (p. 103).

[2] Google. Chrome ships WebGPU. 2023. URL: https://developer.chrome.com/blog/webgpu-release/ (p. 105).

[3] Khronos Group. WebGL Specification. https://registry.khronos.org/webgl/specs/1.0.3/. 2023 (p. 104).

[4] Khronos. OpenCL. 2023. URL: https://www.khronos.org/opencl/ (p. 99).

[5] Mozilla. WebGPU API. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API (p. 105).

[6] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael B. Abu-Ghazaleh. Microarchitectural Attacks in Heterogeneous Systems: A Survey. In: ACM Comput. Surv. 55.7 (2023), 142:1–142:40 (pp. 99, 105).

[7] NVIDIA. CUDA C++ Programming Guide. 2023 (pp. 99, 102, 104).

[8] NVIDIA. Kernel Profiling Guide. 2023 (p. 103).

[9] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and ARM SoCs. In: USENIX Security. 2023 (p. 107).

[10] W3C. WebGPU. 2023. URL: https://www.w3.org/TR/webgpu (p. 105).

[11] W3C. WebGPU - W3C Working Draft - Timing attacks. 2023. URL: https://www.w3.org/TR/webgpu%5C#security-timing (pp. 100, 128).

[12] W3C. WebGPU Security Considerations. 2023. URL: https://www.w3.org/TR/webgpu%5C#security-considerations (p. 100).

[13] W3C. WebGPU Shading Language - Terminology and Concepts. 2023. URL: `https://www.w3.org/TR/WGSL%5C#uniformity-concepts` (p. 109).

[14] 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. 99, 106, 112, 113).

[15] Shujiang Wu, Jianjia Yu, Min Yang, and Yinzhi Cao. Rendering Contention Channel Made Practical in Web Browsers. In: USENIX Security. 2022 (p. 105).

[16] 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. 99, 106, 121).

[17] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. An Exploration of ARM System-Level Cache and GPU Side Channels. In: ACSAC. 2021 (pp. 99, 107).

[18] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In: ISCA. 2021 (pp. 106, 108, 109).

[19] Khronos WebGL Working Group. WebGL 2.0 Compute. `https://registry.khronos.org/webgl/specs/latest/2.0-compute/`. 2021 (p. 104).

[20] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In: S&P. 2021 (pp. 113, 114).

[21] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In: USENIX Security. 2021 (p. 105).

[22] Cihangir Tezcan. Optimization of advanced encryption standard on graphics processing units. In: IEEE Access 9 (2021), pp. 67315–67326 (pp. 99, 121).

[23] Anne van Kesteren. Safely reviving shared memory. 2020. URL: `https://hacks.mozilla.org/2020/07/safely-reviving-shared-memory/` (p. 100).

[24] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. 2020 (p. 99).

[25] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In: DSN. 2020 (pp. 99, 106).

[26] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). 2019 (p. 112).

[27] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVidia Turing T4 GPU via microbenchmarking. In: arXiv:1903.07486 (2019) (p. 112).

[28] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. Exploiting bank conflict-based side-channel timing leakage of gpus. In: ACM TACO (2019) (pp. 99, 106, 121).

[29] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In: ISCA. 2019 (pp. 113, 114).

[30] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In: S&P. 2019 (p. 108).

[31] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (pp. 99, 106, 108, 118).

[32] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. In: arXiv:1804.06826 (2018) (p. 112).

[33] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 119).

[34] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In: CCS. 2018 (pp. 99, 100, 106, 119).

[35] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (pp. 100, 109).

[36] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on GPUs. In: Proceedings of the on Great Lakes Symposium on VLSI. 2017, pp. 167–172 (pp. 99, 106, 121).

[37] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (p. 119).

[38] 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).

[39] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael B. Abu-Ghazaleh. Constructing and characterizing covert channels on GPGPUs. In: MICRO. 2017 (pp. 99, 106).

[40] NVIDIA. NVIDIA Tesla v100 GPU architecture. 2017. URL: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf` (p. 103).

[41] 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 (pp. 100, 109).

[42] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In: HPCA. 2016 (pp. 99, 106, 121).

[43] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (p. 109).

[44] Xinxin Mei and Xiaowen Chu. Dissecting GPU memory hierarchy through microbenchmarking. In: IEEE Transactions on Parallel and Distributed Systems 28.1 (2016) (p. 112).

[45] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (p. 119).

[46] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In: Euromicro Conference on Digital System Design. 2015 (p. 112).

[47] 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. 105, 108).

[48] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015 (p. 112).

[49] 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 (pp. 105, 108).

[50] 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. 112).

[51] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID. 2014 (p. 122).

[52] Mozilla. SharedArrayBuffer. 2012. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer` (p. 100).

[53] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In: SAC. Springer, 2007 (p. 123).

[54] Takeshi Yamanouchi. GPU Gems 3 - AES Encryption and Decryption on the GPU. 2007. URL: `https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu` (pp. 99, 121).

[55] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In: CHES. 2006 (p. 122).

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 105, 108).

[57] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security. 2001 (pp. 119, 120).

7

Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks

Publication Data

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

Contributions

Main author.

Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks

Lukas Giner[1], Stefan Steinegger[1,4], Antoon Purnal[2], Maria Eichlseder[1], Thomas Unterluggauer[3], Stefan Mangard[1], and Daniel Gruss[1]

[1]Graz University of Technology,
[2]imec-COSIC, KU Leuven,
[3]Intel Corporation,

Abstract

In this paper, we propose SassCache, a secure skewed associative cache with keyed index mapping. For this purpose, we design a new two-layered, low-latency cryptographic construction with configurable output coverage based on state-of-the-art cryptographic primitives. Based on this construction, SassCache is the first secure randomized cache with secure spacing. Victim cache lines automatically *hide* in locations the attacker cannot reach after *less than 1 access* on average. Consequently, attackers cannot evict the cache line, no matter which and how many memory accesses they perform. Our security analysis shows that all existing techniques for eviction set construction fail, and state-of-the-art attacks only apply to 1 in 3 million addresses, where SassCache is still as secure as ScatterCache. Compared to standard caches, SassCache has a single-threaded performance penalty of 1.75 % on the last-level cache hit rate in the SPEC2017 benchmark, and an average decrease of 11.7 p.p. in hit rate for MiBench, GAP and Scimark for our high-security settings.

1 Introduction

Caches hide the memory latency in modern CPUs. Modern Intel CPUs organize caches in slices, sets, and ways, which are selected based on the physical address. Slices are independent caches comprised of sets. Each set has multiple ways, i.e., the 64 B cache lines. Memory mapping to the

[4]Work done while affiliated with Graz University of Technology.

same set is called *congruent*. An attacker can use congruent addresses to measure contention or to *evict* cache lines of a victim process. The most notable cache attack exploiting set contention is Prime+Probe [34, 43, 46, 52, 58, 64]. But even without direct contention within one cache set, an attacker can still mount a cache-occupancy attack, where the attacker only observes aggregated cache usage. Cache-occupancy attacks have fewer requirements but contention-based attacks are more powerful and dangerous.

Contention-based attacks [5, 16, 22, 26, 29, 30, 40, 47, 57, 61, 63] are hindered by the scrambling of address-cache-line mappings in secure randomized caches. Recent designs [5, 16, 22, 26, 29] use cryptography to randomize mappings in hardware with a secret key. This is backward-compatible on the software level and maintains certain sharing capabilities.

While randomized caches are a promising solution to eviction-based attacks [3, 9], cache collisions still exist due to limited cache sizes. *Rekeying* alleviates the problem, but the best rekeying interval is difficult to determine for yet unknown attacks and known attacks require high rekeying intervals with a significant performance cost. Furthermore, some previous proposals suffer from cryptanalytic flaws [3, 8]. Hence, we ask the research questions:

Can a secure randomized cache prevent contention between any attacker cache line and a victim cache line in most cases? Which cryptographic constructions provide this property while maintaining high performance?

In this paper, we propose SassCache, a secure cache design with better security guarantees than previous randomized caches and better performance than static isolation. SassCache is a skewed set-associative cache with a keyed index-derivation function per security domain and a novel isolation property. Each security domain has access to a different and only partially overlapping part of the cache. Once a victim cache line is in a location the attacker cannot reach, there is no possibility for the attacker to evict the cache line. In our evaluation we see a victim cache line *hidden* from the attacker after less than 1 eviction on average.

At the core of SassCache is a new two-layered cryptographic construction with configurable output distribution determining reachable cache lines. Inspired by `QARMA` [31], we propose `QARTA`, tailored to our functional (i.e., uncommon bit sizes), latency, and security requirements.

In the recommended configuration, the attacker *cannot* build an eviction set for 99.999 97 % of the victim's addresses. For the remaining 0.000 03 %, SassCache maintains the same security as previous secure skewed caches. Even cache-occupancy attacks are not feasible anymore because the hiding effects affect this channel equally. We evaluate this property and discover that an attacker can observe the occupancy of OpenSSL AES T-Tables in less than 0.000 5 % of cases and the occupancy of secret-dependent cache lines in mbedTLS RSA-4096 in less than 0.000 3 % of cases.

The basic functionality of SassCache is fully compatible with standard caches and, hence, simple to integrate into existing CPU architectures. Sass Cache provides full compatibility with legacy software, but to fully harness its security, software has to configure and switch security domains. Our main use case for SassCache are multi-tenant cloud systems, as they offer high degrees of parallelism between clearly defined security domains (i.e., tenants). Here, SassCache provides inherent quality-of-service properties. We evaluate this in SPEC2017 with multiple other cache-intense tenants running in parallel, showing competitive performance.

We implemented SassCache in gem5, our own cache simulator[1], and in CacheFX [1]. We evaluate the impact on performance and cache-hit rate. Compared to a set-associative `LRU` cache, SassCache has a performance penalty of only 1.75 % on the cache hit rate in SPEC2017, similar to previous secure caches. In MiBench-small, the average cache hit rate of SassCache is 16 % lower than for a set-associative `LRU` cache; in scimark2, it is 23.6 % higher.

Contributions. In summary, our main contributions are:

- We design a novel low-latency cryptographic function with a configurable output distribution for secure caches.
- We propose SassCache, a secure cache that integrates this function, eliminating the attacker's capability of building an eviction set for 99.999 97 % of the addresses.
- We show that for the remaining 0.000 03 %, SassCache maintains the security level of previous secure caches.
- SassCache offers competitive performance, with only 1.75 % average overhead on the LLC hit rate in SPEC2017, compared to a set-associative LRU cache.

[1] `https://github.com/IAIK/CacheSim`

Outline. Section 2 provides background, Section 3 the threat model, Section 4 the design, Section 5 our cryptography, and Section 6 our implementation. Section 7 and Section 8 discuss security and performance. Section 9 concludes.

2 Background

In this section, we discuss non-secure and secure caches and their attack surface and mitigation strategies.

2.1 Caches

Caches hide memory latency by buffering data. In *set-associative caches*, a memory address can only be cached in a (fixed) subset of cache lines, a so-called cache set. Addresses that map to the same set are called *congruent*. When loading data from, the cache replacement policy determines which cache line in a set to evict. Each cache line has a tag that uniquely identifies a cached address. CPU caches can derive indices and tags from the virtual or physical address.

Modern CPUs have multiple cache levels and use physical tags throughout the cache hierarchy. The lower cache levels, e.g., the L1 caches, are small and fast caches that are private to each core. Other cores cannot access them directly but only via the coherency protocol. Most Intel and AMD CPUs also have private, larger L2 caches.

Modern CPUs have a large (multiple MB) shared LLC. It facilitates the use of code and data on multiple cores simultaneously and simplifies coherency and cache lookups. Typically, on Intel and AMD CPUs, this is the L3 cache, and on ARM CPUs, the L2 cache. The LLC is often *inclusive*, i.e., all cache lines in L1 or L2 caches are also in the LLC. Some CPUs split the LLCs into so-called *slices* [44], which are independent caches, e.g., per (logical) core. The slice is selected based on the physical address, not the core. Each core can access each slice via a ring bus that connects all cores and slices.

2.2 Index Derivation Function

Conventional caches map addresses to cache sets by simply using a part of the physical address as a set index, but more advanced functions can be implemented as well [75]. Werner et al. [26] introduced the term Index Derivation Function (IDF) for these mapping functions, which we will use going forward. An IDF generates a set of possile cache lines for each physical address. A more complex IDF can break the traditionally linear relationship between addresses and sets, and may also change the static sets into dynamic ones, such that sets are not fixed collections of cache lines anymore. For this, a cryptographic function that works on the physical address as well as some secret can be used. IDFs generally need to scale well, incur low overheads, and be fully transparent to the software level.

2.3 Cache Attacks

As the cache state depends on recent memory accesses, an attacker can learn interactions of other programs with memory (e.g., instruction and data accesses). Initial attacks focused on the execution time [65, 68, 70, 72]. More recent techniques interact directly with the shared cache.

Flush+Reload [49] relies on (read-only) shared memory between attacker and victim: It flushes an address and later determines whether a victim accessed it by measuring its load latency. Prime+Probe [64] has no such restriction. It measures contention on a portion of the cache (e.g., a cache set) by filling this portion (prime) and measuring the time this takes (probe). Victim accesses to congruent addresses evict the attacker's lines, influencing the probe time.

Contention-based attacks (e.g., Prime+Probe) on the LLC require profiling to identify *eviction sets*, i.e., sets of congruent addresses. The attacker starts with a large pool of lines and, by timing accesses, discards those that do not contribute to contention [43]. The profiling duration depends on knowledge of the mappings and the number of elements discarded per iteration [25]. Prime+Probe based attacks are still actively being improved [3, 4, 6, 9].

2.4 Secure Caches

We can roughly categorize secure caches into designs based on partitioning or randomization. Where partitioning designs reserve parts of the cache per security domain, randomized cache architectures usually make the entire cache accessible but obfuscate the mapping of addresses to cache lines.

Randomization-based designs aim to make contention attacks statistically hard by making the mapping of addresses to cache unpredictable and unobservable. This hinders the construction of eviction sets and the observability of targeted events (cf. Section 2.5).

Many of these designs require randomization mappings to be dynamic, i.e., renewed over time. This *rekeying* (or *remapping*) destroys any congruence information an attacker may have learned. More frequent rekeying is more secure but comes with a performance [29] and power penalty.

CEASER-S [22], **ScatterCache** [26] and **PhantomCache** [16] are examples of skewed designs [74] that compute indices on the fly. Scatter Cache computes indices to individual cache lines which together form a unique set with random replacement. CEASER-S and PhantomCache randomly select from computed indices to entire sets, which can then use standard replacement policies like LRU. Since computations are done on-the-fly, these designs are scalable and suited for large LLCs. However, they have been shown to be susceptible to recent attacks [3, 9, 15]. **MIRAGE** [5] moves the randomization to the cache directory and uses it to approximate a fully associative cache. So far, none of the randomization-based secure caches protect against cache occupancy attacks.

Random Permutation (RP) Cache [63] precomputes permutation tables per process. **Newcache** [61] proposes an entirely new cache design with a secure table of indirection that tries to approach a directly-mapped cache. These table-based designs require overhead proportional to their size, which can be prohibitve for large LLCs.

Partitioning splits the cache into (fixed) slices by its sets (e.g., cache coloring [38, 59]), or its ways (e.g., Intel Cache Allocation Technology (CAT)). The security depends on the strength of the isolation between domains and how much remains observable to attackers. However, static partitioning reduces performance and lacks flexibility and scalability.

Non-Monopolizeable (Nomo) [55] cache reserves some ways per set to be only writeable by one domain, but this leaves reserved ways observable. **Vantage** [56] partitions *most* of the cache while maintaining associativity. Partitions can outgrow their allocated size into a small, unpartitioned space. Additional cache tag bits determine the number of partitions. **AutoLock** [32] prevents cross-core evictions by locking cache lines on ARM CPUs. **Hybcache** [19] uses a hybrid approach between a set-associative and a fully-associative cache. Full associativity is realized only in a small number of ways used for security-critical applications, whereas the rest uses the cache set-associatively. This assumes secure and insecure domains, which differs from the usecase for SassCache. **Jigsaw** [51] and **Jumanji** [14] partition the cache dynamically by splitting it into shares. Software defines capacity and mapping by assigning identifiers to the Page Table Entries (PTEs). Jumanji has a lower latency, and higher performance and security than Jigsaw.

He et al. [33] analyzed static partitioning, Nomo, NewCache, RP Cache, and others, and found that all are, to some degree, vulnerable to at least 2 of 4 studied attacks.

In summary, later analyses [3, 9, 33] of randomized and partition-based caches that provide probabilistic security have shown that these can achieve relatively high performance, but are often not as secure as first thought. Static and total partitioning, on the other hand, provides strong security guarantees at the cost of flexibility and performance.

With SassCache, we combine these two strategies. We make security guarantees for *most* addresses that are the same as a statically partitioned cache (Section 7), with better performance for our target environment (Section 8.4).

2.5 Attacking Secure Caches

Profiling secret-dependent cache lines and finding addresses congruent to them, is an important prerequisite for successful exploitation. The first proposals [29, 30] were bypassed by optimized eviction set algorithms [22, 25], allowing the exploitation phase to proceed as in traditional caches. Consequently, they are practically broken since they require impractical rekeying periods to maintain security [22].

Randomized caches with a probabilistic component [16, 22, 26] preclude traditional eviction by design. Obtaining fully congruent addresses, mapped to the same set in every partition, is theoretically infeasible. In particular, the notion of eviction sets needs generalization (i.e., weakening) if the attacker is to succeed at all. Werner et al. [26] propose eviction sets with addresses congruent in at least one cache way, which was later generalized to partitions [3].

While finding generalized eviction sets is more difficult, a resourceful attacker can still find them by observing which lines are evicted by victim execution. To maximize the chance of observing such evictions, Purnal et al. [3, 21] propose Prime+Prune+Probe (PPP). It extends Prime+Probe with a *pruning* step, enabling occupying a large portion of the cache before transferring control to the victim. PPP was originally applied to CEASER-S and ScatterCache and reduced the complexity of finding eviction sets by orders of magnitude. However, it also applies to other randomized caches, e.g., those that skew across sets instead of ways [16]. Song et al. [6] propose to flush the attacker's own lines to speed up PPP by avoiding costly full cache evictions.

Given a rekeying period, the attacker needs to split resources between profiling (i.e., gaining spatial information) and exploitation (i.e., the actual attack). Bourgeat et al. [9] propose a methodology to navigate this tradeoff.

At one extreme, an attacker spends no time profiling and just measures *cache occupancy* [24], i.e., cache utilization. While less accurate than congruence-based channels (i.e., no spatial information), it is unaffected by rekeying. Current secure LLC designs, even those approximating fully-associative caches [5, 19], have the property that victim accesses are reflected in the observable cache utilization, leaving the cache occupancy channel unmitigated. Some designs are also vulnerable to so-called *shortcut* attacks that exploit weaknesses in randomization to bypass the protection altogether [3]. Thus, it is crucial that the randomization mapping uses well-designed cryptographic primitives.

3 Threat Model and Mitigation Goals

In this section, we describe our threat model and mitigation goals for secure caches, forming the basis of our secure cache design, SassCache.

As shown in Section 2.5, even modern secure caches are affected, e.g., by Prime+Prune+Probe [3] or due to weak cryptographic constructions [3].

3.1 Threat Model

Our threat model is in line with prior work [3, 16, 26] but takes more recent and advanced attack techniques into account (cf. Section 2.5). In this threat model, SassCache constitutes the cache level that is shared between attacker and victim. For our evaluation, we do not consider self-eviction in the victim, because in randomized skewed caches, *reliable* self-eviction only occurs with substantial amounts of memory accesses as part of the secret-dependent operations or active victim participation.

SassCache uses a function that maps physical addresses to cache sets by generating indices $\texttt{idx}_i$, where i counts the ways. (Figure 7.1). We assume the function is known to the attacker but uses a random secret key and a security domain (SDID) to separate security contexts. The key is inaccessible to the attacker, whereas the address is fully controlled. Attacker and victim are separate tenants on a multi-tenant system, where each has their own SDID. Consequently, they are located in different security contexts, and the attacker only has control over few contexts (c.f. Section 7.5). While the attacker may be able to read the SDID, it cannot set it; only privileged software (e.g., the hypervisor) is allowed to set it. The generated indices $\texttt{idx}_i$ are not observable directly but only via cache contention. Physical attacks on the function, its intermediate values, or secret parameters and bugs in the privileged software are out of scope.

3.2 Required Attributes

Functionally, a secure cache should be mostly transparent to software but maintain performance that is comparable to standard caches. On the security side, we extend the security attributes of ScatterCache [26] as follows to address more recent attacks [3] and attacks commonly considered out-of-scope (e.g., the cache occupancy channel [5, 16, 26]):

- Software-defined security domains (based on properties like virtual machine (VM), tenant, user, or process ID) must not share cache lines unless cross-domain coherency is explicitly required, e.g., writable shared memory.

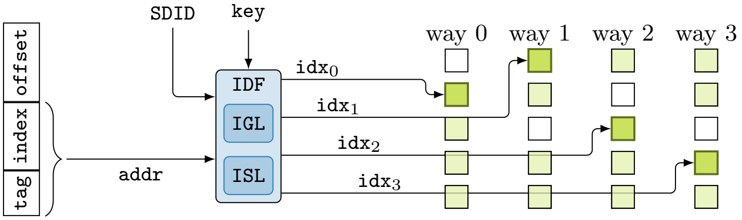


Figure 7.1: Our two-layer Index Derivation Function (IDF): The first Index Generation Layer (IGL) is a cryptographically randomized mapping of addresses to indices, like in ScatterCache; the second Index Spacing Layer (ISL) reduces the set of reachable cache lines through another cryptographically randomized mapping (cf. Section 5.1).

- It must be hard to find congruent addresses in the cache, i.e., it should be hard for adversaries to infer a connection between accessed physical addresses and cache set index.
- Partially accessible addresses should become hidden with a high probability to reduce their observability.
- It must be impossible to evict, measure or control all cache lines from another security domain.

4 The SassCache Architecture

We present SassCache, a novel probabilistically-skewed secure cache that achieves the desired security properties (cf. Section 3.2) and maintains a backward-compatible interface.

Purnal et al. [3] show that randomized secure caches can still be attacked both with old attacks and new attack variations, albeit at a lower attack performance. The underlying problem is that the full cache is still accessible to the attacker. On the other hand, approaches based on cache partitioning such as cache coloring [38, 60] and Intel's Cache Allocation Technology (CAT) offer strict security by splitting the cache into fixed allocations but lack flexibility and scalability. To overcome the drawbacks of previous randomized and partitioned caches, we combine the two principles in Sass Cache. Hence, to mitigate even statistical and occupancy attacks, the idea for SassCache is to cryptographically limit the total number of cache lines accessible to an attacker, in addition to the permutation performed by ScatterCache. We refer to the number of accessible cache lines (=share of the total cache) as *coverage*.

SassCache follows a two-layered approach for its IDF (Figure 7.1): First, the Index Generation Layer (IGL) is a permutation of cache sets as in ScatterCache. This breaks the link between cache set and physical address across different security domains and makes it very hard to profile the cache for an attack [3]. If this layer were a known, non-skewing mapping (such as a bit mask on the physical address), the number of unique sets would be limited. A small number of fixed sets not only makes profiling trivial, it also makes self-eviction deterministic and more likely. This is because some addresses would always share the same set, and therefore compete for the same unobservable ways (Section 7.1). Additionally, with potentially millions of profiling attempts per successful attack (Section 7) and minutes per attempt [3], the IGL ensures high costs for attackers. In short, the IGL provides important support for the security of the second layer and defense-in-depth properties.

Second, the Index Spacing Layer (ISL) restricts accessible cache lines similar to partitioning-based approaches. However, instead of statically slicing the cache into fixed allocations, SassCache selects the accessible cache lines pseudorandomly based on the cryptographic function we propose in Section 5. Therefore, overlaps between security domains become probabilistic. Some cache lines are inaccessible to other security domains, which makes eviction of these cache lines by an attacker impossible. The second layer's parametrizable construction determines the share of the cache available per security domain. As outlined in Section 3, SassCache is focused on a server setting with multiple security domains, identified by a Security Domain Identifier (SDID). We target this environment in particular because the security domains are well defined, and concurrent use is typical. Each security domain is assigned to one tenant, with the hypervisor running in its own security domain, i.e., all virtual machines of one tenant run in one security domain. However, SassCache's design would also allow the definition of other security domains and use cases, such as VMs, users, groups of processes (e.g., for container software), single processes, or even address ranges (e.g., in-process isolation mechanisms). Our generic approach leaves the choice for security domains to the most privileged software (e.g., the hypervisor). Whatever the use case, one domain should never be able to generate more domains under its control to avoid collusion (c.f. Section 7.5). As multiple security domains, i.e., tenants, will use a CPU concurrently, each domain evicts fewer lines from other domains, increasing fairness. Additionally, multiple users already limit each other's cache share, which further alleviates the reduction in cache size per domain.

We propose SassCache as an inclusive or non-inclusive last-level cache (LLC). We opt for a set-associative base design with W ways, i.e., W cache arrays, exactly like existing caches deployed in current CPUs. Each cache array with a size S is indexed individually by one of the W indices. Because the sets are dynamic, we use a random replacement policy. This approach results in S^W possible cache sets after the first layer, similar to ScatterCache [26]. The second layer ISL restricts the possible indices in each way. While this reduces the number of sets per domain, it brings a novel security property: certain cache lines cannot be evicted by an attacker. Our cryptographic design is the basis that makes it improbable (cf. Section 7) for an attacker to evict a target cache line or measure cache occupancy.

5 Cryptographic Design

For SassCache, we need a function to generate cache-set indices from addresses to skew the cache, i.e., the Index Derivation Function (IDF). Additionally, the IDF must limit the number of accessible cache lines per security domain, i.e., the coverage. Hence, in this section, we introduce the two-layered cryptographic primitive at the core of SassCache. We design a low-latency IDF that maps the address to W indices $\texttt{idx}_i$, where the mapping is controlled by the `SDID` and the `key` (Figure 7.1). The IDF consists of two layers: an Index Generation Layer (IGL) that maps the address to W independent intermediate identifiers $\texttt{id}_i$ and an Index Spacing Layer (ISL) that maps each identifier to the final index $\texttt{idx}_i$ in the index space. This index $\texttt{idx}_i$ is selected uniformly from a subset of the index space defined by the `SDID`, `key`, and way i. The ISL is designed such that the subset is expected to cover a defined share of the full index space that we refer to as *coverage*.

5.1 Design of the Index Derivation Function

We design both layers using keyed permutations, i.e., block ciphers or tweakable block ciphers. For the first IGL layer, we profit from permutations with larger block sizes, whereas the permutations for the second ISL layer are smaller and faster. We refer to these as `BC` and `TinyBC`, respectively.

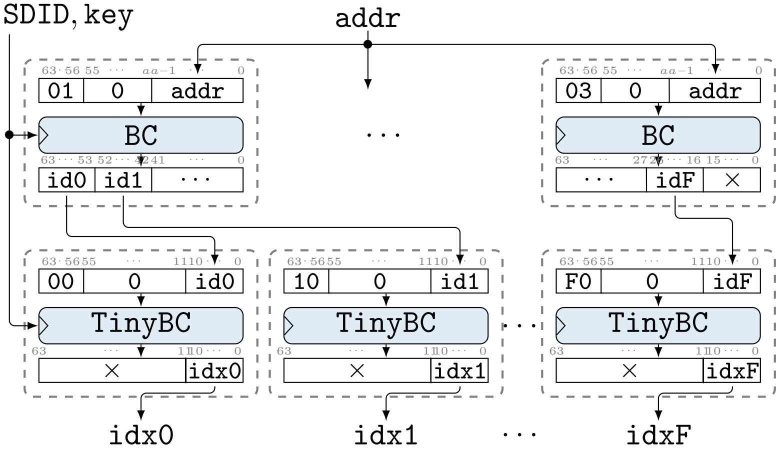


Figure 7.2: The address-to-index mapping function IDF with 63 % coverage, for $W = 16$ ways and 11-bit indices ($\ell = n = 11$). The top layer is the IGL, the bottom ISL.

Assuming the IDF maps a-bit addresses (e.g., $a = 48$) to n-bit indices $\texttt{idx}_i$ (e.g., $n = 11$), we use intermediate identifiers $\texttt{id}_i$ of $\ell = n + t$ bits (t controls the coverage).

Index Generation Layer. For the identifiers $\texttt{id}_i$, the IGL uses BC in a counter-based streaming mode with SDID, key as key, and the address addr as nonce to produce $W \cdot \ell$ bits of keystream. Figure 7.2 (top half) illustrates this construction for $\ell = n = 11$, i.e., $t = 0$. For example, if BC is a 64-bit block cipher, the IGL performs $\lceil (W \cdot \ell)/64 \rceil = 3$ parallel calls to BC with inputs $c \parallel 0 \parallel \texttt{addr}$, where $c \in \{\texttt{0x01}, \texttt{0x02}, \texttt{0x03}\}$ is the 8-bit counter. We start counting at 01 for domain separation with the second layer to support the choice BC = TinyBC. The outputs of BC are cut into ℓ-bit chunks $\texttt{id}_i$. These identifiers are expected to be uniformly and (practically[2]) independently distributed in $\{0, 1\}^\ell$. They are unpredictable, not controllable, and not directly observable for an attacker; learning them still does not allow recovering information about the key. If two addresses are mapped to the same identifier $\texttt{id}_i$, they are mapped to the same index $\texttt{idx}_i$; the reverse is not true.

Index Spacing Layer. To generate the final indices $\texttt{idx}_i$ from $\texttt{id}_i$, the ISL uses W parallel TinyBC invocations. Prepending the counter i before $\texttt{id}_i$ in the input and truncating the output of TinyBC effectively gives us

[2] There is a tiny bias due to the bijectivity of BC, which is not detectable in $< 2^{32}$ calls with constant SDID/key due to the birthday paradox.

a family of ℓ-bit to n-bit pseudorandom functions, keyed with $\texttt{SDID}, \texttt{key}$. For `TinyBC`, a cipher smaller than `BC` also suffices, with a block size of at least $\max(n+\varepsilon, \ell+\log_2 W)$ bits for some small integer ε.

Coverage. When considering such a random (non-injective) function f with domain $\{0,1\}^\ell$ and co-domain $\{0,1\}^n$, we can derive the expected coverage of the co-domain, i.e., $\mathbb{E}[\#\{f(x) \mid x \in \{0,1\}^\ell\}/2^n]$, as follows. Randomly choosing f means randomly choosing each value $f(x)$ uniformly and independently. Then, the expected coverage is the same as the probability for any specific y to appear among the values $f(x)$ for any $x \in \{0,1\}^\ell$. In other words, the coverage equals the probability of drawing a single golden ball from an urn containing 2^n balls at least once when drawing $2^\ell = 2^{n+t}$ times with replacement. Thus, the expected coverage C is

$$C = 1 - \left(1 - \tfrac{1}{2^n}\right)^{2^{n+t}} = 1 - \Big(\underbrace{\left(1 - \tfrac{1}{2^n}\right)^{2^n}}_{\to \mathrm{e}^{-1} \text{ for } 2^n \to \infty} \Big)^{2^t} \approx 1 - \mathrm{e}^{-2^t} .$$

We list the resulting coverage C for $t \in \{-3, -2, \ldots, 2\}$ in Figure 7.3. These values depend primarily on t and are essentially identical for all relevant values of n, e.g., $n = 11$.

If `TinyBC` is only slightly larger than $\max(n, \ell+\log_2 W)$ bits, this truncated construction can be modeled more precisely by taking into account the bijectivity of the block cipher. With $b > \max(n, \ell + \log_2 W)$, the block size of `TinyBC` (i.e., 2^{b-n} inputs produce the same truncated n-bit output $\texttt{idx}_i$), the expected coverage C_b is the ratio of permutations mapping $i \parallel 0 \parallel \texttt{id}_i$ to $\texttt{idx}_i$ for any $\texttt{id}_i$ among all b-bit permutations:

$$\begin{aligned} C_b &= 1 - \frac{\binom{2^b - 2^{b-n}}{2^\ell} \cdot 2^\ell! \cdot (2^b - 2^\ell)!}{2^b!} = 1 - \frac{\prod_{i=0}^{2^{b-n}-1} (2^b - 2^\ell - i)}{\prod_{i=0}^{2^{b-n}-1} (2^b - i)} \\ &= 1 - \prod_{i=0}^{2^{b-n}-1} \left(1 - \frac{2^\ell}{2^b - i}\right) \approx 1 - \left(1 - \tfrac{1}{2^{b-n-t}}\right)^{2^{b-n}} . \end{aligned}$$

For example, for $n = 11$ and a $b = 16$-bit block cipher `TinyBC`, the resulting expected coverage C_b differs by up to 0.9 % from the result C for large block ciphers. For clarity, we take the expected value of C as a given for the security analysis, which is appropriate as its variance is very low.

5.2 Instantiation with `QARMA` and `QARTA`

We want to instantiate this design with efficient cryptographic functions `BC` and `TinyBC`. Several low-latency block ciphers [10, 53] and tweakable block ciphers (TBCs) [31, 36] have been published, though some provide insufficient security [28], which share several design ideas with `PRINCE` [53]. We propose an instantiation using (parts of) `QARMA` [31], a TBC used for ARM pointer authentication.

Conservative instantiation. A conservative instantiation is to use the 64-bit variant of `QARMA`, $\texttt{QARMA}_7\texttt{-64}$, for both `BC` and `TinyBC`. This TBC encrypts 64-bit plaintext blocks with a 128-bit key K and 64-bit tweak T, fitting with the dimensions given in Figure 7.2. The 192-bit combined tweakey (K, T) is available for key material from the SDID and key, fitting, e.g., a 128-bit `key` as K and 64-bit `SDID` as T, or the XOR of two 128-bit values as K with $T = 0$. The same key can be used for `BC` and `TinyBC`. The expected coverage C for this construction can be derived with the model for large block ciphers.

Low-latency instantiation. To avoid the latency of two calls to the full TBC, we propose an optimized variant with a latency comparable to one $\texttt{QARMA}_7\texttt{-64}$ call: We instantiate `BC` with the round-reduced $\texttt{QARMA}_5\texttt{-64}$ (with 12 instead of 16 rounds) and use operations from the remaining 4 rounds to run 4 ultra-light 16-bit $\texttt{QARTA}_4\texttt{-16}$ ciphers in parallel. The total circuit size of the IDF with fully unrolled `BC` and `TinyBC` instances corresponds roughly to 4 $\texttt{QARMA}_7\texttt{-64}$ instances. The expected coverage C_b of this instantiation can be derived with the model for small block ciphers.

$\texttt{QARTA}_4\texttt{-16}$ operates on one 16-bit column of a `QARMA` state and key using the `QARMA` 4-bit S-box layer S (`SubCells`), mixing layer M (`MixColumns`), and round tweakey addition (`AddRoundTweakey`), without applying an equivalent of the permutation layer τ (`ShuffleCells`). Four parallel instances of one $\texttt{QARTA}_4\texttt{-16}$ round correspond to one round of `QARMA` without the τ operation (cf. Figure 7.4). The key (w^0, k^0) and tweak T are again derived from the `SDID` and `key`. In this instantiation, the key material for `TinyBC` should be independent of that of `BC`. Figure 7.5 shows which parts of the key material influence which index computations. Notice that the tweak schedule implies that the tweak material influences several columns, and acts differently on each tweak column. The `QARMA` round constants $\mathfrak{c}_i$ are also different for each column.

$\texttt{QARTA}_4\texttt{-16}$ is not a generically secure tweakable block cipher due to its small size and low-latency design. It is tailored for the proposed application, where an attacker has little control and never learns the cipher inputs except for a few index and padding bits. Since each column depends on at least 8 cells of the tweak T (Figure 7.5), the effective tweakey size

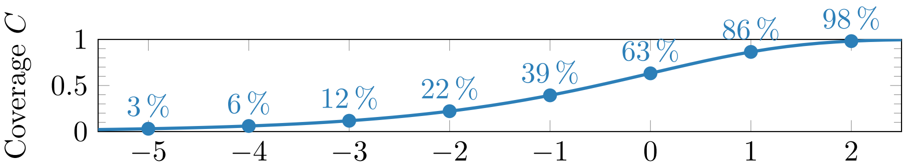


Figure 7.3: Expected coverage $C \approx 1 - e^{-2^t}$ for $t \in \{-5, \ldots, 2\}$.

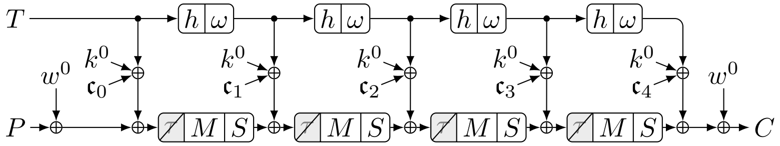


Figure 7.4: Four parallel invocations of $\texttt{QARTA}_4\texttt{-16}$ (with different, but partially related keys).

for $\texttt{QARTA}_4\texttt{-16}$ is at least 64 bits. Regarding its cryptanalytic properties, $\texttt{QARTA}_4\texttt{-16}$ is expected to reach its full algebraic degree after 3 rounds since its S-box has an algebraic degree of 3 and $3^3 > 15$. The MixColumns matrix has a branch number of $\mathcal{B} = 4$; there are several truncated differential and linear patterns with a total of 8 active S-boxes that are compatible with the input format, e.g., the iterative pattern $(0, 0, *, *)$, where $*$ denotes active cells. Since the maximum differential probability and absolute linear bias of the S-box are 2^{-2}, the maximum achievable probability for differential characteristics is 2^{-16}. Even with potential clustering effects, this is hard to exploit for largely unknown cipher inputs. Still, an attacker that observes a large number of cipher outputs might succeed in recovering a few key bits by exploiting the few known cipher input bits and the partially overlapping key material between indices. However, since the key material is independent of the key used in BC, there is little information to derive from this potential knowledge beyond the image set of TinyBC, which is easily obtained through direct observation rather than cryptanalysis. The main criterion for security is, however, the statistical behavior, which we analyze next.

Coverage evaluation. Figure 7.6 shows the observed distribution of the coverage for both instantiations of TinyBC for 100 random keys with 16 counter values i each. Both $\texttt{QARMA}_7\texttt{-64}$ and $\texttt{QARTA}_4\texttt{-16}$ behave as expected, with average coverages C and C_b ($b = 16$), respectively. For example, for $t = -1$, the coverage for $\texttt{QARMA}_7\texttt{-64}$ ranges from 37 % to 41 %, with an

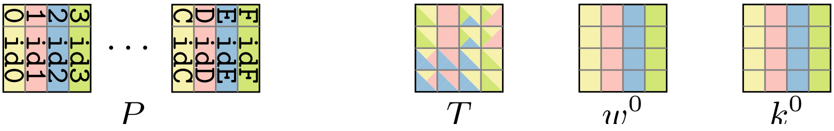


Figure 7.5: Mapping of `QARMA-64` and `QARTA-16` states.

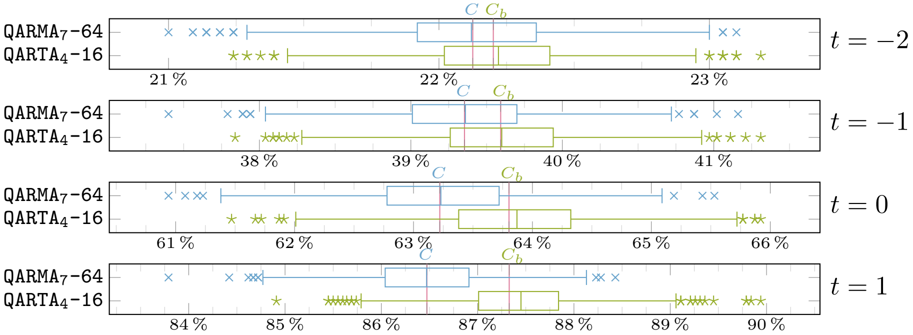


Figure 7.6: Experimental coverage for $t \in \{-2, \dots, 1\}$ for 100 random keys and 16 ways i, with `QARMA` or `QARTA`$_4$`-16`, and expected values C and C_b, respectively.

average of $C \approx 39\,\%$. The coverage for `QARTA`$_4$`-16` ranges from $37\,\%$ to $41\,\%$, with an average close to $C_b \approx 39\,\%$.

6 Implementation of SassCache

SassCache is designed as a last-level cache (LLC) for a server environment with multiple co-located security domains. The use in an LLC enables hiding the latency of the cryptographic functions during lower-level cache lookups. Especially in servers (and desktops), we believe the added energy consumption of the lightweight cryptography will be negligible compared to regular cache and memory lookups.

6.1 Hardware Modification

While recent advances in the RISC-V community lead to the first RISC-V CPUs with experimental L2 caches [17, 27], they are far from state-of-the-art high-performance L3 LLCs used in server-grade CPUs and thus not yet suited for estimations on the hardware modification costs that

Intel, AMD or ARM would see. Hence, we estimate SassCache's hardware costs by determining the hardware costs for the building blocks in terms of chip-area and latency.

Area. The cryptographic primitives for the IDF make up the main cost in hardware. We propose $\texttt{QARTA}_4\texttt{-16}$, a custom low latency instantiation for 16 ways (cf. Section 5.2) corresponding to roughly four $\texttt{QARMA}_7\texttt{-64}$ instances with 34.4kGE each [31]. Therefore, in total, the IDF requires less than 140kGE. Previous work [26] estimated that the open-source BROOM core's LLC takes up about 5.5MGE. Hence, our design should result in less than 3 % of that.

The additional area required to skew the cache is specific to the overall design. In general, Djordjalian [69] noted that additional decoders and wiring are required for each way. For a 2-way skewed cache, Spjuth et al. [66] saw a 17 % increase in energy consumption. However, Sardashti et al. [48] show that cache skewing only has 1.5 % to 15.3 % overhead. Furthermore, most Intel cache architectures already feature cache slices; thus, they partition the cache into smaller caches with multiple addressing circuits already.

Latency. SassCache's latency depends on the IDF's latency for generating the set indices. We use `QARMA`, which is already used for latency-critical applications like pointer authentication [35] or memory tagging [12]. Notably, `QARMA` achieves a latency as low as 2.20 ns when fully unrolled at a 7 nm process [31]. For our low-latency instantiation (cf. Section 5.2), the combined latency of $\texttt{QARMA}_5\texttt{-64}$ and $\texttt{QARTA}_4\texttt{-16}$, used for IGL and ISL, is comparable to that of $\texttt{QARMA}_7\texttt{-64}$ at about 3.25 ns. In comparison, this is still lower than L2 cache access on current CPUs (e.g., Intel Xeon 8280 at 5.18 ns [18], AMD Epyc 7742 [18] at 3.86 ns, and ARM Ampere Altra Q80-33 at 4.11 ns [11]). Since SassCache is designed for shared LLCs, most if not all of the latency is hidden in lower-level cache lookups, and our IDF becomes viable for practical implementation. In line with other skewed cache designs, the skewing itself does not introduce additional latency [5, 29].

6.2 SassCache Interactions

One goal of SassCache is to make its adoption as frictionless as possible by providing strong security benefits even without software support.

Backward Compatibility. To make SassCache backward compatible, we suggest to initially deactivate the spacing layer of the IDF. While booting, privileged software (e.g., the hypervisor) aware of SassCache and security domains activates the second layer via a configuration register. Thus, legacy software still benefits from a cache similar in functionality to ScatterCache without security domains, which already protects against some cache attacks [26].

SassCache Software Interface. The software must supply the correct SDID to SassCache. Depending on the architecture and what constitutes a security domain (we focus on the cloud use case), this can be achieved in numerous ways. On x86-64, ARM, or RISC-V systems, we can use unused bits in the PTEs to supply the SDID either directly or indirectly via an additional lookup table. Using the SDID directly only allows for a limited number of security domains, which we therefore do not recommend. Intel x86-64 has 14 bits [20], ARM64 4 bits [50], and RISC-V 10 bits [23] reserved for future use in the PTEs that can be used for this.

However, by using the PTE bits as a lookup, very large SDIDs are possible, such that there will never be domain collisions. To do so, an additional SDID list is implemented in hardware. The list is only writeable by privileged software (e.g., hypervisor or OS) and has a number of slots corresponding to the available bits in the PTE. This allows privileged software to load specific sets of SDIDs for a security domain during context switches. During a memory access the Memory Management Unit (MMU) then uses the bits from the PTE to look-up the SDID in parallel and forward it to the cache alongside the memory request.

This indirection allows for privileged software to easily change a large number of SDIDs on demand. Moreover, if the number of SDID indices loaded in parallel is insufficient for an application, one specific index can be used to trigger an exception, albeit with a performance hit. The exception then allows privileged software to examine the situation and switch certain SDIDs, before returning to the application. The size of the list limits the number of security domains available *in parallel*. The overall number is only limited by the size of the SDID and the input-size of the IDF.

Alternatively, Intel's Page Attribute Table (PAT) and ARM's Memory Attribute Indirection Register (MAIR) offer similar functionality for this purpose, i.e., define memory types and specify caching behavior. The available bits in the PTE index list can be used to implement the SDIDs.

Implementation Considerations. Addressing the SDID indirectly allows for effective tenant-based separation with only a single bit and two domain registers holding the full SDID. This enables using two security domains simultaneously, e.g., privileged software (e.g., the hypervisor) and application (e.g., the tenant's VM). Thus, it enables coarse separation of execution paths that require context switches, during which the configuration registers are maintained, and certain types of shared memory. During context switches, the privileged software swaps the SDID for the application. To preserve separation in the cache, different SDIDs are used for shared memory. Hence, read-only shared memory (e.g., libraries) is shared across security domains with different SDIDs without further modifications. However, shared memory used in different security domains is loaded to different locations in the cache for each domain.

For writable shared memory, a change in the SDID also changes which cache lines are part of the cache set. This can lead to cache coherency issues for writeable shared memory. Therefore, for this memory type, the privileged software must assure that the SDID is the same across all security domains that can access the shared memory. With the single-bit approach, the SDID for privileged software can be reused for shared memory. Similarly, to copy data between privilege levels (e.g., hypervisor and VM), the correct SDID must be loaded. Multiple VMs of one tenant are in the same security domain to prevent collusion using multiple SDIDs.

6.3 Key Management and Rekeying

Both the IGL and ISL require a secret key to make cache indices unpredictable. Thus, an attacker cannot map addresses to indices or vice-versa, even if SDID, IGL, and ISL are publicly known. However, this makes it essential to prevent software from extracting the secret key used by Sass Cache.

SassCache uses a boot-time hardware-generated key, like CEASER-S [29] and ScatterCache [26]. The key is stored in a hidden CPU register and is inaccessible to software. This prevents an attacker from predicting the resulting cache set from a physical address and vice-versa. Only the SDID may be set by the software to provide different security contexts. This keeps the required software and hardware changes low and modular, decreasing the effort to implement SassCache.

CEASER-S and ScatterCache require rekeying [26, 29]. The frequency of rekeying is a security parameter to adapt to improved attacks and security margins [3]. Key changes work well on write-through caches as no data inconsistencies can occur. However, rekeying costs performance as data is invalidated, causing cache misses. Thus, changing the key corresponds to a cache flush in terms of performance. Implementing rekeying for write-back caches is more costly: The key must not change before all dirty cache lines were written to memory.

While SassCache could support rekeying for both write-through and write-back caches, we do *not* recommend it. SassCache's main security gain over previous designs derives from the Index Spacing Layer (ISL). With rekeying, each epoch will have a certain chance that a given address is fully reachable by an attacker. If the user decides that this chance is too high, and the additional security provided by the Index Generation Layer (IGL) is too low, the rekeying period then necessarily depends only on the IGL, which renders the ISL superfluous. This is because under the assumption that a victim address is attackable, the rekeying period will now have to be determined only by how long it takes to succesfully attack the IGL, i.e., ScatterCache.

Instead, we suggest choosing the coverage parameter t such that the remaining risk is acceptable.

7 Security Evaluation

In this section, we evaluate the security of SassCache against state-of-the-art attacks. The security is based on two steps:

1. For $t{=}-1$, $W{=}16$ SassCache prevents full eviction of the target cache line in 99.999 97 % of cases (cf. Section 7.1). Thus, on average, the attacker can try to construct an eviction set for 1 in 3 000 000 cache lines, or 64 B in 185 MB of memory. On CPUs with 20 ways per cache set, this increases 1 in 125 000 000 cache lines, or 64 B in 7.6 GB of memory.
2. For attackable cache lines, security reduces to that of ScatterCache, with attack times from prior work [3].

Like previous designs [5, 26], SassCache precludes attack techniques based on shared read-only memory (e.g., Flush+Reload) by placing them separately in the cache per security domain. Thus, we evaluate SassCache

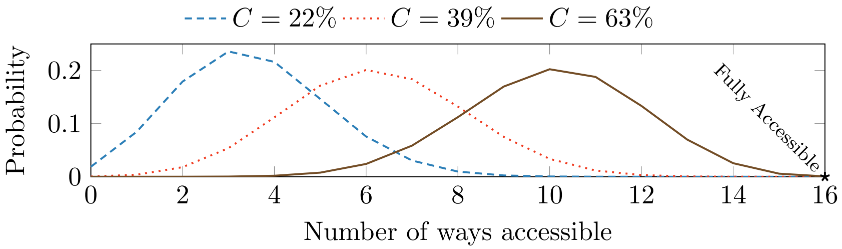


Figure 7.7: Accessibility distribution of cache lines ($W = 16$) for different cache coverages C. *Fully accessible* lines are observable to an attacker; all others become unobservable.

against the remaining attack classes, i.e., contention- and occupancy-based attacks.

7.1 Properties of SassCache

Partially Accessible Lines. In the worst case, a victim line falls within the attacker's coverage in each of the W ways. The probability $P_{\mathrm{W}} = C^W$ is low for practical configurations (e.g., $P_{\mathrm{W}} = 0.06\,\%$ in a 16-way Sass Cache with $C = 63\,\%$ or $P_{\mathrm{W}} = 0.000\,033\,\%$ with $C = 39\,\%$). For such *fully accessible* lines, the security of SassCache is equivalent to that of Scatter Cache with the same security domain size (i.e., W ways, $C \cdot S$ sets). All other lines are *partially accessible*, i.e., they can hide in the victim region, where they are not observable nor controllable by the attacker. Figure 7.7 highlights the abundance of partially accessible lines in the distribution. The exponential dependence of P_{W} on the number of indices W motivates SassCache to build on ScatterCache instead of other skewed designs.

For victim programs where M lines encode the same information (e.g., AES T-Tables, $M = 16$), the probability that at least one of them is fully accessible is $P_{\mathrm{M}} = 1 - (1 - C^W)^M$. Figure 7.8 illustrates P_{M} for a 16-way cache.

Repeated Observations. Cache attacks typically require many observations of sensitive lines, both for contention-based attacks [3, 13, 25, 43] and for cache occupancy attacks [24, 45]. In SassCache, accesses to victim lines are only *observable* to the attacker when they evict an attacker's line. The probability for a partially accessible victim line to be observable N times before being hidden is $\frac{P[N]}{(1-C^W)}$, where

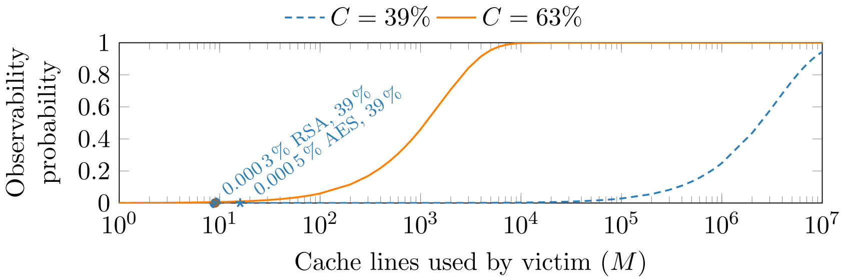


Figure 7.8: Observability probability ($W = 16$, $C = 39\,\%$ & $C = 63\,\%$), increases with cache lines. A covert channel requires thousands of cache lines. The probability of a successful occupancy-based attack is extremely low. Example points are openSSL AES T-Tables and mbedTLS RSA-4096.

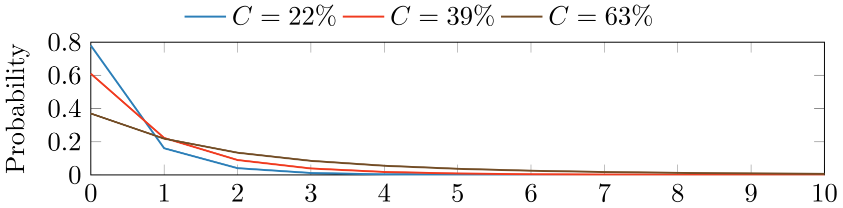


Figure 7.9: Probability to observe partially accessible victim line N times before it is hidden ($W = 16$).

$P[N] = \sum_{i=0}^{W-1}[(\frac{i}{W})^N \cdot \frac{W-i}{W} \cdot \binom{i}{W} \cdot C^i \cdot (1-C)^{(W-i)}]$ (cf. Figure 7.9). The expected number of observable victim accesses until the line is hidden can be computed as $E[N] = \sum_{n=0}^{\infty} n \cdot P[N = n]$. In a 16-way SassCache with $C = 39\%$, this is already after $E[N] \approx 0.72$ accesses on average, for $C = 63\%$ after $E[N] \approx 2$ accesses. This reflects the number of potentially observable victim accesses to the cache line, i.e., assuming that the attacker occupies its full share of the cache ($C \cdot S \cdot W$ lines).

Summarized, **the vast majority of lines are not fully accessible by the attacker, and partially accessible lines rapidly become unobservable.** Section 7.2 examines the security of SassCache against contention-based attacks, evaluating state-of-the-art profiling and exploitation success rates. Section 7.3 evaluates SassCache for cache occupancy leakage, an attack vector largely unmitigated by prior designs.

7.2 State-of-the-Art Attack Evaluation

We implement SassCache in CacheFX [1], which models attack strategies on real systems with attacker and victim code, which the framework invokes accordingly. Attacker and victim issue requests to cache implementations that respond with hits and misses. The framework analyzes attack success statistically and offers several knobs to test caches configured with different parameters, e.g., number of sets, ways, and replacement policy. With the framework we test the applicability of state-of-the-art profiling (Single Holdout Method [29], Group Elimination Method [22], and Prime+Prune+Probe [3]) on SassCache, to find partially congruent addresses, i.e., lines congruent with the target in one or more ways. We configure SassCache with 16 ways, and $t = -1$ (i.e., $C = 39\%$) and test each profiling algorithm 500 times. We sample a random victim address, let the algorithm find an eviction set, and test the quality of the found set by (a) determining the share of truly conflicting addresses (*True Positive Rate* (TPR)) in the eviction set, and (b) computing the success rate (SR) of evicting the victim address. We compute the minimum, maximum, mean, and median for TPR and SR over all runs to statistically analyze profiling on Sass Cache.

We observe that all techniques fail with overwhelming probability, as expected (cf. Section 7.1). Our analysis of the algorithms' progress shows that after a few victim evictions, the victim address is not observable to the attacker anymore and hidden in the cache, as expected (cf. Figure 7.9). Hence, both the minimum and median of TPR and SR are 0. Maximum TPR and SR vary depending on the algorithm, the eviction set size, and the concrete cache parameters, but occur infrequently when the victim address is fully attacker-reachable (and SassCache falls back to Scatter Cache). For instance, the most efficient technique Prime+Prune+Probe achieves a maximum TPR of 1 and a SR according to Figure 5 in [26]. Mean TPR and SR are skewed according to the maximum and the probability of fully accessible lines.

Note that our empirical analysis considers a single target victim line. Figure 7.8 covers the probability for profiling to succeed on at least one out of M redundant lines, which is low for real-world attack targets. Advanced profiling methods [3] enlarge eviction sets (of sufficient size) without relying on further victim accesses. This does not affect SassCache, as the address is mostly not reachable in the first place. Furthermore, once a line is hidden, the attacker cannot proceed without self-eviction by the victim (cf. Section 3).

7.3 Cache Occupancy Leakage

The cache occupancy channel is arguably the most primitive cache side-channel. When the number of cache lines a victim uses depends on a secret bit, an attacker can recover the bit by simply measuring cache utilization. Traditional shared caches as well as secure randomized caches [5, 22, 26, 29] cannot completely close the cache occupancy channel. This can easily be seen when looking at a fully-associative cache: While there is no cache-set information to gain, the occupancy of a different number of cache lines can still be observed. SassCache improves over previous secure randomized caches, as only part of the cache occupancy of the victim is visible to the attacker. As the cache occupancy channel covers a significant amount of irrelevant cache lines, all cache occupancy attacks so far [24, 45] require a large number of repetitions. Unless the victim's cache line is fully coverable by the attacker, it will quickly be hidden in an attacker-unreachable part of the cache (cf. Section 7.1). Furthermore, self-eviction of a specific cache line is generally unreliable and unlikely (cf. Section 3). If a victim occupies large amounts of memory, self-eviction can occur, but with adverse effects as it reduces the overall number of cache lines occupied by the victim as compared to the non-self-eviction case, reducing exposure to the attacker. This is particularly relevant for the covert channel case where the attacker occupies a large fraction of the cache. To obtain worst-case numbers, we assume that in this case the attacker is completely lucky and there is no self-eviction that conceals some of the fully coverable visible cache lines. Consequently, the formula from Section 7.1 also applies to the cache occupancy channel: The probability that a cache occupancy channel encoding '0' and '1' into M cache lines works successfully is the probability that at least one of these cache lines is fully coverable (cf. P_M in Section 7.1).

With the default configuration of 39 % coverage, for very low values, the success probability for the occupancy channel is close to 0 % (cf. Figure 7.8). If the difference between a secret bit '0' and a secret bit '1' is reflected in the access to more than two million cache lines, the probability that the attacker can observe the occupancy channel is 50 %. For an observability probability of 95 %, more than 10 million cache lines must encode a '1'. This number of cache lines is far beyond normal cache attack targets: OpenSSL AES T-Table encryption encodes key-bit information in 16 cache lines resulting in an observability probability below 0.000 5 %. mbedTLS RSA-4096 signature computation encodes equivalent key-bit information in up to 9 cache lines [13], resulting in an observability probability below 0.000 3 %. Hence, SassCache also closes the cache occupancy channel in many attack scenarios.

7.4 Asymmetric Domain Sizes

A convenient feature of SassCache is that its parameters can be configured for each system. By adjusting the coverage parameter t of the Index Derivation Function (IDF) (cf. Section 5), we can fine-tune the security and performance tradeoff. We will see in Section 8.4 that this tradeoff is not necessarily bad for systems that are shared by many users. Domain sizes can cover 12 % to 98 % of the available cache. While smaller coverage increases the security, it reduces the application's performance due to the reduced cache size.

Maybe counterintuitively, restricting security-critical domains more does not increase their security. Since attackers only need to find congruent addresses to the victim's in their own address space, the only domain size that matters is the attacker's. As it is generally unkown which parties on a system will be attackers, the largest domain size should be chosen based on the security requirement for all domains. Lower priority applications can use reduced domain sizes.

7.5 Multi-Domain Attacks and the Choice of t

Ristenpart et al. [58] showed that achieving co-location in the cloud is possible. Inci et al. [39] also showed that, while rare, co-location of more than 1 VM on the same system is possible. Fang et al. [2] also show container scheduling (e.g., Kubernetes) is vulnerable to co-location.

If attackers can occupy multiple security domains on the same system, they may collude to mount a stronger attack on a victim VM. When we combine the coverage of multiple domains, the expected total coverage changes with the attacker-controlled domains n_d according to $C_t = 1 - (1 - C)^{n_d}$ (cf. Section 5.1). Doubling the amount of domains n_d effectively reduces the security equal to an increase of the parameter t by 1. For example, two colluding attackers with $C = 39\,\%$ could, at best, mount an attack as if the coverage C were 63 %. In practice, cloud providers need to select an appropriate base coverage for security (e.g., $t = -1$), and then adjust down with the above formula based on their expectation of co-location probability.

7.6 Trusted Execution Environments

The threat model of Intel SGX [37] and AMD SEV-SNP [7] explicitly allows malicious privileged software and hypervisors. Without further modification for these TEEs, this goes beyond our threat model because it contradicts our goal of backwards compatibility. If the ISL is disabled, SassCache falls back to the security of a secure randomized cache.

8 Performance Evaluation

In this section, we analyze SassCache's performance in a default configuration with 39 % coverage (unless stated otherwise). We use gem5 to run MiBench [71], lmbench [73], scimark2 [67], and the GAP benchmark [41], in line with previous works [22, 26, 29] to show the skewed cache characteristics of SassCache. With our custom simulator we run extensive workloads (e.g., SPEC CPU 2017) efficiently and evaluate the cache's behavior based on real-world recorded memory access traces, e.g., for a multi-tenant cloud scenario. We show that SassCache performs similar to traditional set-associative caches of the same size, and can increase in relative performance as multiple tenants compete for the cache. All caches are evaluated without rekeying, as this would be highly implementation dependant.

8.1 gem5 Test Setup

To evaluate SassCache in the gem5 full system simulator, we run gem5 as a 32-bit ARM DerivO3 3 GHz CPU. We configure a two-level cache system. The L1 is made up of 2 32 kB, 8-way caches for instructions and data. As this is the same for all configurations, the differences in performance stem from the respective L2 implementations only. All L2 caches are configured with 1 MB and 16 ways. The L1 and L2 cache line size is 64 B. This is similar to typical server and desktop configurations today with

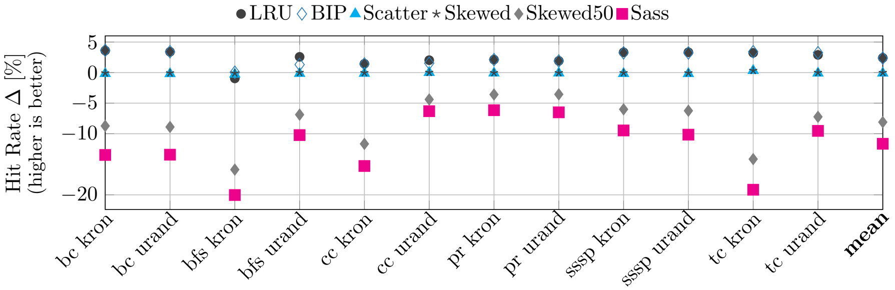


Figure 7.10: gem5 cache hit rate in the GAP benchmark. Comparison against `Rand` as a baseline.

slices functioning as independent caches (i.e., 1 MB per cache slice and 64 B cache line size), since when all cores on a system are in use, the average per-core share of the LLC amounts to the size of 1 slice. We test 6 cache designs: (1) Bimodal Insertion Policy (`BIP`), (2) Least Recently Used (`LRU`), and (3) random replacement (`Rand`), for regular caches; (4) ScatterCache (hash-based), (5) skewed associative caches, and (6) Sass Cache, for skewed caches. For SassCache, we evaluate the default security level with the coverage parameter $t = -1$, i.e., a single-tenant coverage $C \approx 39\,\%$. We also include Skewed50, a skewed cache at 50 % capacity, as this is the nearest size to 39 % SassCache for a standard configuration without changing W.

We deploy the same software setup as prior work [26], with poky Linux (19.0.2) in Yocto 2.5 (“sumo”) and Linux 4.14.67 patched for compatibility with gem5. For the evaluation, we use the cache statistics from the gem5 simulator. We configure the `QUARTA` instances for our IDF with the exact distribution and latency properties described in Section 6.

8.2 gem5 Results

In our evaluation in gem5, the benchmark is the only workload on the system, i.e., the only active tenant (one security domain). Therefore, cache hit rates and performance of SassCache suffer from the smaller cache size of each domain, as only ~39 % of the cache are used by the single tenant. We evaluate concurrent security domains in Section 8.3. We measure only the L2 hit rate, as the L1 does not change.

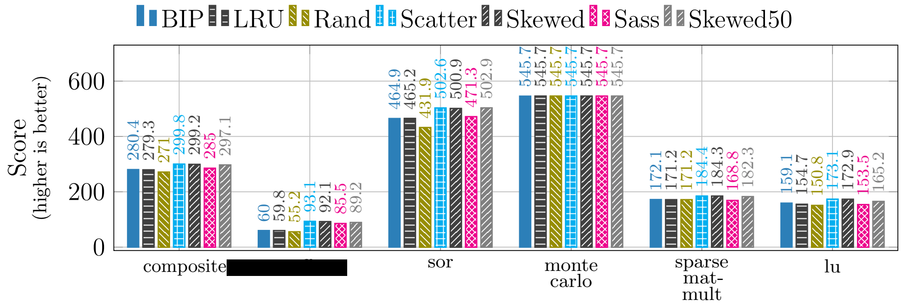


Figure 7.11: Result score for the scimark2 benchmarks for the gem5 simulator with different L2 replacement policies.

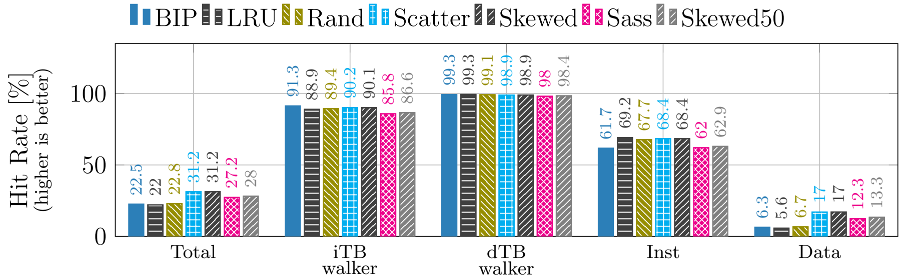


Figure 7.12: Cache hit rate by origin of cache requests for the scimark2 benchmark in gem5 for different caches.

The GAP benchmark consists of 6 workloads (bc, bfs, cc, pr, sssp, tc) with 2 input graphs, `kron` (`-g27 -k16`) and `urand` (`-u27 -k16`). `Rand` is the baseline for the hit rate (cf. Figure 7.10). As expected, SassCache's smaller effective cache size lowers the hit rate on average by 11.6 p.p. (6.1 p.p. to 20 p.p.). On average, SassCache's hit rate is 11.5 p.p. lower than ScatterCache. For GAP, (the best performing) `LRU` has an average 14.1 p.p. higher hit rate than SassCache. Skewed50 approximates Sass Cache's effective size and closely follows its hit rate, which supports that the delta stems mostly from the smaller effective size.

To evaluate the impact of the smaller effective cache size, we use scimark2 [67] in the `-large 1` configuration. The cache hit rate (Figure 7.12) shows that SassCache performs similar to other skewed caches for the total hit rate and data accesses. All skewed caches show a notably higher hit rate for the `fft` and `composite` benchmark than the non-skewed `BIP`, `LRU`, and `Rand` policies. Consequently, for scimark2, skewed caches like SassCache outperform the non-skewed caches due to these benchmarks

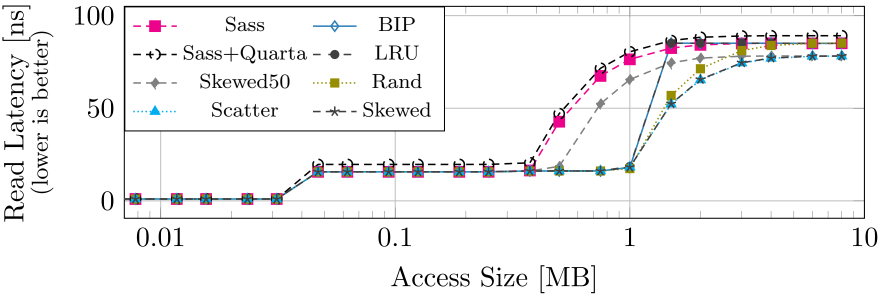


Figure 7.13: Memory read latency with `lat_mem_rd` for different cache configurations with 64 B strides.

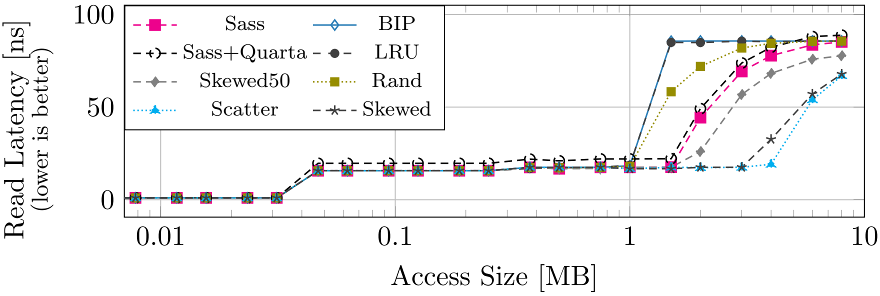


Figure 7.14: Memory read latency with `lat_mem_rd` for different cache configurations with 256 B strides.

(cf. Figure 7.11) [26]. As expected, SassCache's hit rate in this benchmark lies slightly below the Skewed50 configuration, due to SassCache's smaller effective size of ∼39 %. Here, ScatterCache and the skewed cache have the highest total hit rate, with SassCache being 4 p.p. lower, due to the data (−4.7 p.p.) and instruction (−6.3 p.p.) hit rates. The total hit rate of `LRU` is 5.2 p.p. lower than SassCache. Although `LRU`'s instruction hit rate outperforms SassCache by 7.1 p.p., its data hit rate is 6.7 p.p. lower. The reason for this is a high code locality but a weaker data locality in scimark2.

Caches with random replacement policy, e.g., our skewed caches, show a smoother roll-off after the L2 cache [26]. We verified this for SassCache using the lmbench `lat_mem_rd` benchmark [73] for 8 MB size and 64 B (i.e., cache line size) strides, cf. Figure 7.13. SassCache closely follows Skewed50, due to its smaller effective size.

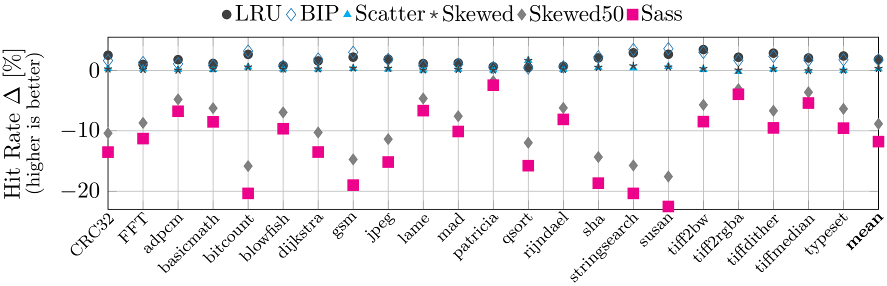


Figure 7.15: MiBench cache hit rate in gem5 with a small dataset. Percentage points over Rand.

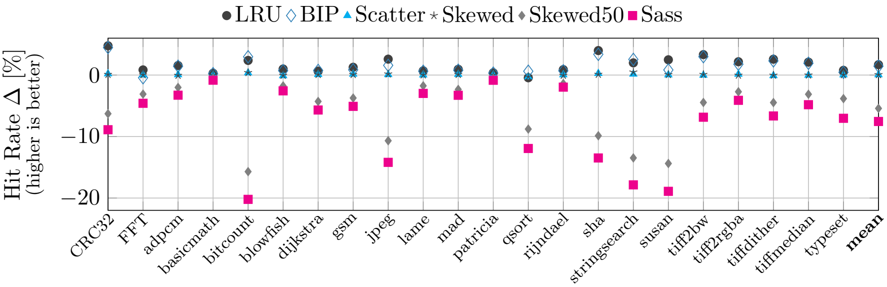


Figure 7.16: MiBench cache hit rate in gem5 with a large dataset. Percentage points over Rand.

With a 256 B stride size the step between the L2 and higher memory levels is shifted to a larger access size, cf. Figure 7.14. This shift results from skewed caches breaking the alignment of addresses and the cache set indices. In traditional set-associative caches, lat_mem_rd's 256 B stride size performs sparse but aligned memory accesses that use every fourth cache index. With skewed caches the indices are random and, hence, there are fewer cache conflicts. Thus, for this type of access pattern, skewed caches generally improve the hit rate and lead to lower read latencies for larger ranges of memory [26]. However, due to SassCache's smaller effective cache size, this effect is less pronounced than for, e.g., ScatterCache. Still, the apparent size shifts from 1 MB to about 1.5 MB ($\approx 4 * 0.39$).

We also run the MiBench benchmark (small and large setting) on gem5 with the Rand cache as a baseline. The average hit rate in MiBench (small dataset) is 11.8 p.p. (2.4 p.p. to 22.5 p.p.) lower than Rand, cf. Figure 7.15. For MiBench with the large dataset, cf. Figure 7.16, this average improves to be 7.5 p.p. (0.8 p.p. to 20.2 p.p.) lower than Rand. SassCache has on

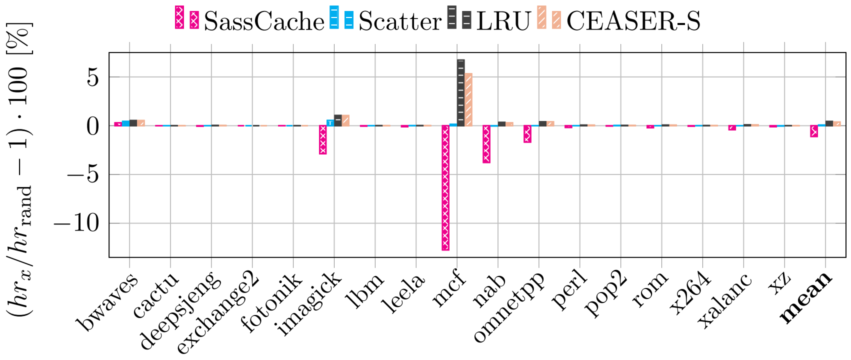


Figure 7.17: Change in LLC hit rate over Rand for SPECspeed 2017. Std.dev. < 0.03% for all tests over 10 runs.

average a 12.1 p.p. and 7.6 p.p. lower hit rate for MiBench small and large, respectively, than ScatterCache. SassCache's hit rate is on average 13.6 p.p.(MiBench small) and 9.2 p.p.(MiBench large) lower than for LRU and BIP caches, which have the highest hit rates. In both cases, the Skewed50 consistently has a higher hit rate than SassCache but strongly correlates with it, confirming skewed cache characteristics for SassCache and lower hit rates in the single-tenant scenario with the lower effective cache size.

Over all GEM5 benchmarks, SassCache has a 9.8 p.p. lower hit rate than Rand and 11.7 p.p. lower than LRU in the single-tenant gem5 evaluation. For comparison, at C=63 %, the average hitrates go down by 4.6 p.p. and 6.4 p.p. compared to Rand and LRU. The scimark2 benchmark shows that in some workloads, skewed caches outperform non-skewed caches, and Sass Cache benefits from this as well. The variation in hit rate is expected due to each benchmark having different access patterns with different locality properties. Thus, each workload benefits differently from a particular cache architecture which are tailored towards certain locality properties, e.g., via their replacement policy.

QUARTA Latency. Though we expect to hide latency in lower level accesses (see Section 6.1), we also evaluate a 12c (3.25 ns@3 GHz) overhead in our L2 (see Figures 7.13 and 7.14). This is a costly 30 % increase at our LLC base latency of 40c. Compared to SassCache without overhead, hitrates remain virtually the same (<±0.2 p.p. avg. difference), while scimark scores drop 4.1 % on average.

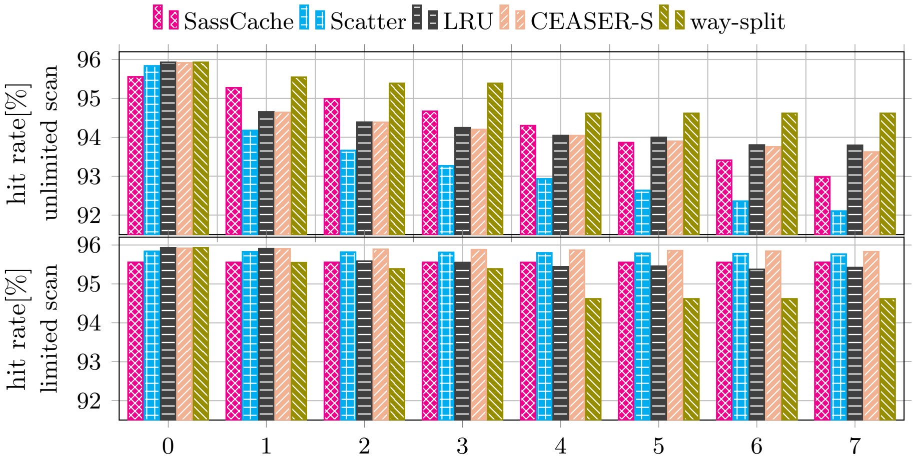


Figure 7.18: Average simulated hit rates of SPECspeed 2017 with 0-7 parallel domain workloads. Size of way-split partions: 0: 100%, 1: 50%, 2-3: 25%, 4-7: 12.5%. Top: adversarial workload with 64 B stride, unlimited size. Bottom: 1 024 B stride repeating 512 addresses.

8.3 Custom Cache Simulator Setup

We build a cache simulator based on the model by Purnal et al. [3]. We evaluate SassCache against ScatterCache, CEASER-S, standard LRU, Rand, and a way-split cache (an approximation of Intel CAT [42]), in SPEC CPU 2017.

To run benchmarks in our simulator, we collect real memory access traces (including instructions) with the Intel PIN tool [54]. While this is much faster than gem5 emulation, we still need to limit the size of our traces. Like prior works [22, 26, 62], we collect a representative sample trace over 250 million instructions for each benchmark.

Our simulator implements 2 cache levels. The L1 consists of 2 set-associative, 32 kB, 8-way data and instruction caches with tree-PLRU replacement and is the same for all LLCs. The LLC has a size of 1 MB and 16 ways, similar to modern Intel slice configurations. SassCache is configured for 39% coverage, for CEASER-S we use 2 partitions and LRU replacement. Our way-split cache supports evenly splitting the cache into separate domains along the ways.

8.4 Custom Cache Simulator Results

We run our recorded traces through the cache simulator to measure LLC hit rates in SPECspeed 2017. Since average hit rates vary between benchmarks, we compare the ratio of hit rates for 4 cache implementations to the `Rand` cache in Figure 7.17. Owing to its reduced size per security domain, SassCache performs worse than other caches in most tests, which is more pronounced in benchmarks with larger working sets or those optimized for `LRU` replacement. As in Section 8.2, we see that SassCache is at a disadvantage in single-threaded performance evaluations. When compared to the overall best, `LRU`, we see average hit rate drops of 1.75 %, 0.46 %, 0.08 %, and 0.52 % for SassCache, ScatterCache, CEASER-S, and `Rand`, respectively.

We also examine SassCache when several competing workloads run concurrently, e.g., in a multi-tenant cloud. Here we simulate only the LLC, as the L1 is not shared among cores. We run SPEC traces as before, but interleave them with up to 7 adversarial workloads. We test 2 different types of adversarial workloads: a linear scan over an infinite range with 64 B stride (e.g., streaming a lot of data), and a scan over a limited set of 512 addresses with a stride of 1 024 B, both shown in Figure 7.18. We configure our way-split cache such that it always has at least as many separate domains as workloads, i.e., 1/2/4/8, which results in the visible performance plateaus. In this particular test, we run only the memory accesses of the benchmark traces, leaving out the instruction accesses. This accentuates the different behaviors of the caches for these parallel workloads. As Figure 7.18 shows, caches like SassCache and ScatterCache suffer under workloads that use many cache lines without re-referencing them because of their random replacement, which can cause evictions even on recently used lines. For SassCache, this effect is at first counteracted by its isolation property. With few parallel workloads, many cache lines are exclusive to each domain. `LRU`-based caches fare better in general, as often-referenced cache lines can consistently survive streaming. The way-split cache leads under this workload, because the complete domain separation provides excellent thrashing protection as long as the workloads fit within the reduced cache size. The second workload is more adversarial to `LRU`-based caches, as not all sets are filled optimally. Here, SassCache can provide higher hit rates than a way-split cache while offering almost the same security.

These parallel tests reveal an important property of SassCache. While single-threaded tests show decreased performance because of the reduced size and random replacement, relative performance of SassCache *increases* with higher parallelism. This is because parallel workloads proportionally reduce the average share of cache a thread can use. Critically, this reduction does *not compound* multiplicatively with the coverage C of SassCache, so a coverage of 39 % will already lose importance with 4 cores.

Finally, we compare the hit rates for different C of SassCache for the 2-level setup. The average SPEC hit rate for SassCache (cf. Figure 7.17) is 88.36 % at 39 % coverage. For coverages of 12 %, 22 %, 63 %, 86 %, and 98 %, this changes to 86.18 %, 87.29 %, 88.89 %, 89.17 %, and 89.28 %. This almost reaches ScatterCache's performance for 98 % coverage and clearly drops towards 12 % coverage as expected.

9 Conclusion

In this paper, we proposed SassCache, a novel secure cache design based on a low-latency cryptographic function tailored to this use case. Sass Cache eliminates the attacker's capability of building an eviction set in 99.999 97 % of the cases. We found that the *hiding* property allows for a higher share of the cache than static partitioning, while providing virtually the same security. Furthermore, we showed that SassCache also mitigates the cache occupancy channel, e.g., a cache occupancy attack on OpenSSL AES T-Tables or mbedTLS RSA-4096 can succeed in less than 0.000 5 % of cases. Our performance evaluation revealed that Sass Cache only has an overhead of 1.75 % on average on the last-level cache hit rate in the SPEC2017 and an average decrease of 11.7 p.p. in hit rate for MiBench, GAP, and Scimark benchmark compared to a set-associative `LRU` cache. Hence, we conclude that SassCache is a promising design for use in appropriate, security-critical contexts.

Acknowledgments

This research is supported in part by the European Research Council (ERC `#101020005`), the Flemish Government (FWO project `TRAPS`) and Cyber-Security Research Flanders (`#VR20192203`). Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO). Additional funding was provided by a generous gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. CacheFX: A Framework for Evaluating Cache Security. In: arXiv:2201.11377 (2022) (pp. 138, 159).

[2] Chongzhou Fang, Han Wang, Najmeh Nazari, Behnam Omidi, Avesta Sasan, Khaled N Khasawneh, Setareh Rafatirad, and Houman Homayoun. Repttack: Exploiting Cloud Schedulers to Guide Co-Location Attacks. In: arXiv:2110.00846 (2021) (p. 161).

[3] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In: S&P. 2021 (pp. 137, 140–146, 156, 157, 159, 168).

[4] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In: CCS. 2021 (p. 140).

[5] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security Symposium. 2021 (pp. 137, 141, 143, 144, 153, 156, 160).

[6] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it. In: S&P. 2021 (pp. 140, 143).

[7] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. 2020. URL: `https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf` (visited on 06/2020) (p. 162).

[8] Rahul Bodduna, Vinod Ganesan, Patanjali Slpsk, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. In: IEEE Computer Architecture Letters 19.1 (2020), pp. 9–12 (p. 137).

[9] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In: MICRO. 2020 (pp. 137, 140–143).

[10] Dušan Božilov, Maria Eichlseder, Miroslav Kneževic, Baptiste Lambin, Gregor Leander, Thorben Moos, Ventzislav Nikov, Shahram Rasoolzadeh, Yosuke Todo, and Friedrich Wiemer. PRINCEv2 – More Security for (Almost) No Overhead. In: SAC. 2020 (p. 149).

[11] Andrei Frumusanu. The Ampere Altra Review: 2x 80 Cores Arm Server Performance Monster. 2020. URL: `https://www.anandtech.com/show/16315/the-ampere-altra-review/` (p. 153).

[12] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory. In: arXiv:2012.06761 (2020) (p. 153).

[13] 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 (pp. 157, 161).

[14] Brian C. Schwedock and Nathan Beckmann. Jumanji: The Case for Dynamic NUCA in the Datacenter. In: MICRO. 2020 (p. 142).

[15] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In: arXiv:2008.01957 (2020) (p. 141).

[16] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In: NDSS. 2020 (pp. 137, 141, 143, 144).

[17] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In: Fourth Workshop on Computer Architecture Research with RISC-V. 2020 (p. 152).

[18] Johan De Gelas. AMD Rome Second Generation EPYC Review: 2x 64-core Benchmarked. 2019. URL: `https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/` (p. 153).

[19] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In: USENIX Security Symposium. 2019 (pp. 142, 143).

[20] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (p. 154).

[21] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in Scatter-Cache. In: arXiv:1908.03383 (2019) (p. 143).

[22] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In: ISCA. 2019 (pp. 137, 141–143, 159, 160, 162, 168).

[23] RISC-V Foundation. The RISC-V Instruction Set Manual, Vol. II: Privileged Architecture, Version 20190608-Priv-MSU-Ratified. Ed. by Andrew Waterman and Krste Asanović. 2019 (p. 154).

[24] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through The Cache Occupancy Channel. In: USENIX Security Symposium. 2019 (pp. 143, 157, 160).

[25] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In: S&P. 2019 (pp. 140, 142, 157).

[26] 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 (pp. 137, 140, 141, 143, 144, 147, 153–156, 159, 160, 162, 163, 165, 166, 168).

[27] Pi-Feng Chiu, Christopher Celio, Krste Asanović, David Patterson, and Borivoje Nikolić. An out-of-order RISC-V processor with resilient low-voltage operation in 28nm CMOS. In: IEEE Symposium on VLSI Circuits. 2018 (p. 152).

[28] Maria Eichlseder and Daniel Kales. Clustering Related-Tweak Characteristics: Application to MANTIS-6. In: IACR Transactions on Symmetric Cryptology 2018.2 (2018), pp. 111–132 (p. 150).

[29] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: MICRO. 2018 (pp. 137, 141, 142, 153, 155, 156, 159, 160, 162).

[30] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In: DAC. 2018 (pp. 137, 142).

[31] Roberto Avanzi. The QARMA Block Cipher Family: Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. In: IACR Transactions on Symmetric Cryptology 2017.1 (2017), pp. 4–44 (pp. 137, 150, 153).

[32] 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 Symposium. 2017 (p. 142).

[33] Zecheng He and Ruby B Lee. How secure is your cache against side-channel attacks? In: MICRO. 2017 (p. 142).

[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 (p. 137).

[35] ARM Connected blog. Armv8-A architecture: 2016 additions. 2016. URL: https://www.community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions (p. 153).

[36] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In: CRYPTO. 2016 (p. 150).

[37] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (p. 162).

[38] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In: USENIX Security Symposium. 2016 (pp. 141, 145).

[39] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-location detection on the cloud. In: COSADE. 2016 (p. 161).

[40] 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. 137).

[41] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP Benchmark Suite. In: arXiv:1508.03619 (2015) (p. 162).

[42] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Cache. 2015. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf` (p. 168).

[43] 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. 137, 140, 157).

[44] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015 (p. 139).

[45] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (pp. 157, 160).

[46] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015 (p. 137).

[47] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In: MICRO. 2014 (p. 137).

[48] Somayeh Sardashti, André Seznec, and David A Wood. Skewed compressed caches. In: MICRO. 2014 (p. 153).

[49] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (p. 140).

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

[51] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In: PACT. 2013 (p. 142).

[52] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 137).

[53] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. In: ASIACRYPT. 2012 (pp. 149, 150).

[54] Intel. Pin - A Dynamic Binary Instrumentation Tool. 2012. URL: `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool` (p. 168).

[55] 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) 8.4 (2011) (p. 142).

[56] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In: ISCA. 2011 (p. 142).

[57] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: HPCA. 2009 (p. 137).

[58] 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. 137, 161).

[59] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security Symposium. 2009 (p. 141).

[60] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In: EuroSys. 2009 (p. 145).

[61] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In: MICRO. 2008 (pp. 137, 141).

[62] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In: ACM SIGARCH Computer Architecture News 35.2 (2007), p. 381 (p. 168).

[63] 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 (pp. 137, 141).

[64] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 137, 140).

[65] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (p. 140).

[66] Mathias Spjuth, Martin Karlsson, and Erik Hagersten. Skewed caches from a low-power perspective. In: Conf. Computing Frontiers. 2005 (p. 153).

[67] Roldan Pozo and Bruce R. Miller. Scimark 2.0. 2004. URL: `https://math.nist.gov/scimark2/` (pp. 162, 164).

[68] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In: CHES. 2003 (p. 140).

[69] Andrés Djordjalian. Minimally-skewed-associative caches. In: Symposium on Computer Architecture and High Performance Computing. Proceedings. 2002 (p. 153).

[70] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. In: Cryptology ePrint Archive, Report 2002/169 (2002) (p. 140).

[71] Matthew R. Guthaus, Jeff Ringenberg, Dan Ernst, Todd Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In: WWC. 2001 (p. 162).

[72] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 140).

[73] Larry McVoy and Carl Staelin. Lmbench: Portable Tools for Performance Analysis. In: USENIX ATC. 1996 (pp. 162, 165).

[74] André Seznec. A case for two-way skewed-associative caches. In: ACM Computer Architecture News (1993) (p. 141).

[75] Alan Jay Smith. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. In: IEEE Trans. Software Eng. 4.2 (1978) (p. 140).

8

Fast and Efficient Secure L1 Caches for SMT

Publication Data

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

Contributions

Main author.

Fast and Efficient Secure L1 Caches for SMT

Lukas Giner, Roland Czerny, Simon Lammer, Aaron Giner,
Paul Gollob, Jonas Juffinger, and Daniel Gruss

Graz University of Technology

Abstract

Secure randomized caches use the latency budgets of last-level caches to isolate data by security domain. In contrast, L1 caches are very latency- and size-constrained (by cache ways and page size), hindering both the adoption of secure randomized designs and increases in size without losing backward compatibility due to page size changes.

We propose a new secure and larger L1 cache design for SMT cores: SMT Cache. SMTCache uses separate, identical L1 caches (slices) to isolate security domains. The overall cache size scales with the number of SMT threads, with individual slices mirroring current designs without changing the page size. SMTCache consumes less power than larger sets and does not increase hit latency. We show that SMTCache is a principled mitigation against L1 cache attacks and fundamentally precludes vulnerabilities like L1TF. Further, we measure that SMTCache improves L1 cache performance compared to current designs and even remains competitive with larger caches. For instance, on a system with SMT-2, SMTCache provides equivalent hit ratios across the SPEC CPU2017 suite to a state-of-the-art L1 cache of comparable size while improving system security and significantly reducing energy costs.

1 Introduction

Caches hide the high access times of main memory by storing recently used data within the CPU. With low latency, limited space, and sharing across security contexts, they are an attractive target for attacks. Attacks range from side channels [10, 33, 34, 35, 36] to severe vulnerabilities like Meltdown [13] and its variants [11, 19, 23, 24, 31, 32]. All of these attacks rely on the cache being a shared resource without security domain isolation.

While recent secure cache proposals address this problem for the large last-level caches [3, 5, 9, 17, 22, 25], low latency is crucial for L1 caches. Hence, we cannot simply apply last-level secure cache designs to the L1 cache. Furthermore, partitioning the L1 cache is costly as it is already very size-constrained. Due to the virtual indexing, the L1 size is determined by the number of ways times the page size, which for commodity laptop, desktop, and server CPUs has been 4 KiB for over a decade. This limits an 8-way L1 cache to a maximum size of $4\,\text{KiB} \cdot 8 = 32\,\text{KiB}$. Thus, there are currently only two options to increase the L1 cache size, without even taking security considerations into account: First, like some recent Intel server CPUs, the number of ways per set is increased (e.g., from 8 to 12), at the cost of a super-linear increase in energy consumption [50, 56]. Second, like recent Apple CPUs, the page size could be increased (e.g., to 16 KiB [12]). However, this is only possible given Apple's firm control of both hardware and software on their machines, reducing the need for backward compatibility. Still, with this change, Apple increased the L1 cache size to 128 KiB. While this shows that *die area near the execution core is available*, it further emphasizes the page size as a limiting factor to efficiently scale the L1 and its lack of security that becomes increasingly interesting for attacks.

This leads us to investigate the following research questions:
How can we prevent L1 cache attacks in a principled way? Is it possible to increase L1 cache size and security without substantial efficiency loss or software-breaking changes? What is the energy cost of scaling the L1 cache?

In this paper, we propose SMTCache, a secure L1 data cache (L1D) design that offers advantages in L1 cache size, security, and energy efficiency on CPUs with simultaneous multithreading (SMT). SMTCache stays within the existing ISA specifications as well as power and latency budgets of commodity off-the-shelf CPUs. We achieve this by creating independent L1D *slices* (like L3 slices) accessed by a memory address and a security domain. Every domain has its own L1 slice, ensuring principled data separation.

In our default configuration, SMTCache does not require operating system support and switches domains based on existing mechanisms, *i.e.*, user mode and kernel mode. This provides out-of-the-box data separation between processes and the operating system. From the point of view of processes, they have their own private L1D cache without interference or data leakage from one SMT thread or process to another. Often, the number of processes active on a core is higher than the number of L1D slices (we evaluate up to 9 slices per core). If a process is scheduled to run on a core where it does not have a slice assigned, the least recently used (LRU) slice is flushed to higher cache levels, and the new process gets this slice exclusively until it is eventually flushed for a different process.

Each slice can be the same size as current L1D caches, as sets are addressed by their domain in addition to the virtual address, thereby sidestepping the page-size limitation while scaling the cache. The maximum active number of slices is limited to the number of SMT threads. For SMT-2, this doubles the effective available L1D cache space for simultaneously running processes. When the operating system schedules only a small number of processes on a core, this markedly increases performance as processes are not competing over cache space. With more slices than SMT threads, inactive slices store currently unused data for different processes that are not running while only drawing static power.

We evaluate the performance of SMTCache in CacheSim [4] and on traces recorded on a native Linux server running different workloads, as well as the functionality via micro-benchmarks with Linux on gem5 [14, 51]. Our evaluation shows that performance scales very well with SMT and often exceeds the performance of an equivalently large standard cache due to the inherent thrashing protection. For SMT-2, and especially SMT-4 (Section 6.1), SMTCache increases the available L1 cache per thread while guaranteeing fairness and security. We find that a number of slices higher than SMT ways + 1 only minimally improves performance, as processes rarely return to an empty L1D cache at that point.

Contributions. In summary, our main contributions are:

- We propose a novel secure L1 cache design, SMTCache, providing strict isolation between security domains on the hardware level.
- SMTCache builds on the synergetic introduction of security and performance enhancements to expand cache sizes without breaking backward compatibility.
- We provide a security argument for SMTCache, showing that it mitigates a range of state-of-the-art attacks in a principled way.
- We evaluate the performance of SMTCache in many different configurations and demonstrate that it offers competitive hit ratios even when compared to larger monolithic L1 caches like Apple's M1.

Outline. Section 2 presents background and Section 3 the design. Section 4 discusses energy and area costs and Section 5 security. Section 6 evaluates the performance. Section 7 presents related and future work. Section 8 concludes.

2 Background

In this section, we discuss caches, limiting factors for their size, traditional and secure L1 designs as well as their attack surfaces.

Caches. CPU caches are buffers close to the CPU, orders of magnitude smaller than main memory. They hide high memory access latencies for recently used data. In modern set-associative caches, addresses are statically mapped to one of many sets of cache lines and occupy any of the ways within that set.

Traditional caches are organized in a 3-level hierarchy, with the cache closest to the CPU (L1) being the smallest and fastest, with acritical impact on CPU performance. Unlike higher-level caches, most L1 caches use the virtual address to index the cache set to reduce latency by already looking for a cache line while translating the address. To not map a physical address to multiple sets, the index is taken only from bits shared with the virtual address, *i.e.*, the page offset, typically 12 bit. This limits the L1 cache size by the page size and number of ways, e.g., 4 KiB·8 ways = 32 KiB. Hence, traditional L1 cache size can only be increased with the page size, the number of ways, or by dropping the virtually indexed design. Intel increased the L1D cache size of the “Core” CPUs from 16 KiB to 32 KiB and 48 KiB with the number of ways, from 4 to 8 and 12. In the Apple M1, the 16 KiB page size allows for a 128 KiB L1D cache and a 192 KiB L1 instruction cache (L1I).

The drawback of increasing the associativity is a rising energy cost per access due to two factors: Firstly, the number of tags that need to be searched to determine a cache hit increases proportionally to the number of ways. Implementations may also load the data of all lines in a set at the time of the tag comparison [47], further adding to the increased energy demand. Secondly, super-linear components to the power draw grow with the size of a cache set [50, 56].

Cache Coherence. When multiple cores access the same memory location on a CPU with private, per-core caches, writes on one core need to become visible to other cores as soon as possible. There are various coherence protocols that ensure this memory consistency. In the simplest case, caches need to know if a local copy of a cache line is modified, shared, or invalid (MSI). There are two common methods to implement coherence protocols, *snooping* and *cache directories* [15]. Snooping protocols work by broadcasting each memory request to all caches but are only viable for a low number of caches. Directories can solve this problem by centralizing the protocol's state information at a point of coherence. In inclusive cache hierarchies, the last-level cache (LLC) stores all cache lines found in lower levels and can, therefore, also act as the directory.

Cache Attacks. As caches are shared and introduce timing differences, they have been a popular target for side-channel research. In a cache attack, an attacker observes different access times to their data to infer a victim's behavior. With knowledge about cache architecture, refined cache attacks are possible.

Attacks on Cache Metadata. The simplest form of these attacks are time-driven attacks, such as Bernstein's attack [60] or **Evict+Time** [59]. The latter, for example, evicts an AES T-Table entry by filling the cache set with attacker memory. By timing the victim's execution the attacker can infer if this entry was used. A more noise-resilient evolution is **Prime+ Probe**, where the attacker first *primes* a set by filling it, and then *probes* it by timing accesses. If the victim used a line in this set in between, the attacker measures a longer access and can observe the victim's accesses at the granularity of cache sets. Prime+Probe requires a set of addresses that map to the same cache set, called an *eviction set*. The **Flush+ Reload** [48] attack enables cache line accuracy if attacker and victim share memory, by not relying on set conflicts but measuring the target line directly. The attacker uses the `clflush` instruction to evict the targeted address precisely, and later measures it again to see if the victim has brought it into the cache. Achieving shared memory with a victim is more challenging than co-location, and `clflush` might be unavailable. **Evict+ Reload** [40, 43] removes the need for `clflush` by replacing it with set eviction like Prime+Probe.

Attacks with Caches. Meltdown, Spectre, etc. [11, 29, 31] use caches for their covert-channel to recover data encoded during speculative execution. This is possible because the state of the caches is not reversed when a speculatively execution is aborted. Meltdown variants leaking from the L1 Cache (or the Line Fill Buffer) exploit caches that does not check permissions when data is served.

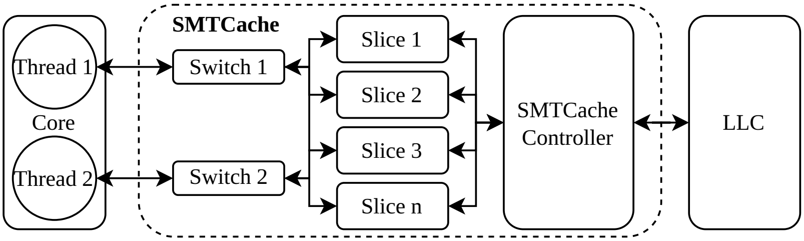


Figure 8.1: SMTCache abstract design for n slices. At most 2 slices are active at the same time, one per SMT thread. The SMTCache controller ensures coherence between slices and that SMTCache appears like a normal cache to higher cache levels.

Secure Cache Designs and Related Work. With some of the attacks known for decades, many secure cache designs have been proposed, generally based on two methods: randomization or partitioning. The former tries to obscure access patterns by making them seemingly random, while the later tries to make accesses unobservable. Many designs require complex functions whose latency is too large for the L1 and only target the LLC [3, 5, 6, 7, 8, 9, 17, 22, 25, 30, 41] assuming the other caches are secure. In Section 7 we detail these secure caches and highlight how SMTCache is orthogonal to many of them and discuss how SMTCache can complement them for improved security and performance.

3 The SMTCache Architecture

At the heart of SMTCache are a number of n identical *slices*; complete L1 data caches with standard parameters, e.g., 8 ways, 32 KiB size as shown in Figure 8.1. At each context switch (security domain switch), one slice is assigned to the process. Until the next context switch, requests from the SMT thread are statically routed to this slice by a switch. From the perspective of the core, cache hits on this slice behave identical to a standard L1 design cache hit. The communication with the higher-level caches, however, runs through the SMTCache controller (Section 3.3), presenting SMTCache as a standard L1 cache. This, of course, adds extra latency. While our design could also be used for instruction caches, we focus on L1 data caches to limit the scope.

3.1 Domains

An important aspect of isolation-based designs is how security domains are derived. We propose a basic in-hardware implementation augmented with optional software control. The default configuration changes the slice assignment when the process (PCID/CR3) or the protection ring change. When the protection ring is 3, the CR3 register represents the domain ID, when it is less than 3, it is considered the kernel domain, regardless of the CR3 value. All kernel threads therefore share one slice, while userspace processes are isolated. This ensures security boundaries in line with standard OS process isolation. This is the backward-compatible mode of the design that works regardless of OS version.

With OS support, this could be enhanced to be more or less precise via MSRs. A process might, e.g., want to isolate its threads to maximize its L1D cache size, while another might want to share one slice among an entire process group. As this is highly workload dependent, we do not evaluate OS support in this work.

Hypervisors and SGX. With a hypervisor, we can simply consider ring -1 the only mandatory domain in the default configuration. Thus, the hypervisor and guest can never share an L1. Intel SGX [2] has the unique situation that the hardware is generally under the control of the untrusted OS, yet SGX must be secure. We can accommodate this by always treating each enclave as a unique domain, irrespective of any configuration the OS might have chosen.

3.2 Slice Swapping

When a new domain not currently associated with a slice is assigned to a core, one of the slices is chosen to be evicted. On eviction, the hardware flushes all modified (dirty) cache lines to the higher cache levels, unmodifed (clean) cache lines can simply be dropped. When the number of slices equals the number of SMT ways $n = n_{SMT}$, the slices can be statically assigned to logical cores, and the current slice will be reused. For $n = n_{SMT} + 1$, the additional slice is always used for the kernel.

When $n > (n_{SMT} + 1)$, SMTCache chooses the slice to be evicted with a modified LRU algorithm. Domains scheduled often are therefore likely to keep their data in an inactive slice while they are descheduled, ready to resume work when they are scheduled again (see Section 6.2).

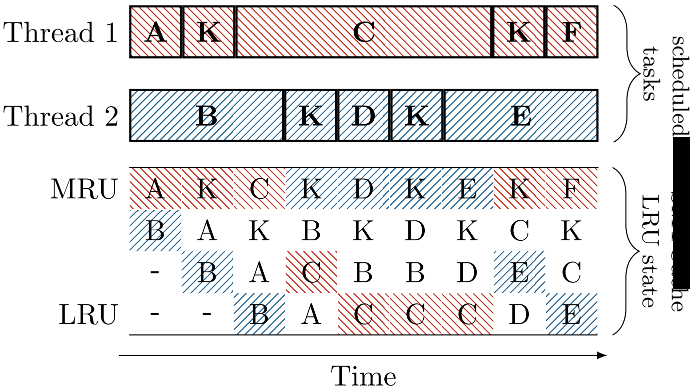


Figure 8.2: Domain swapping with modified LRU for SMTCache with 4 slices. Context switches cannot replace active caches and bring the last active slice to the second-most recently used position. As the context switches are performed by the kernel (**K**) its slice is always most or second-most recently used automatically and can never be evicted.

Figure 8.2 shows an example of our modified LRU for 4 slices and SMT-2. A thread is moved to the MRU position the moment it is newly scheduled on the core. Because the process scheduling is always performed by the kernel (**K**) it can always only be at the most or second-most recently used position. This ensure that the slice of the kernel is “reserved” and never evicted, guaranteeing fast kernel entries when there are more slices than SMT ways. We modify standard LRU such that active threads can never have their slice taken from them, regardless of their LRU position. Additionally, swapped-out threads are placed in the second-most recent position. This prevents long-running threads from immediately being the new eviction candidate (see switch from B to D or C to F in Figure 8.2). For a configuration of 5 slices, this means that 4 slices will be available for user space domains. If the kernel is scheduled on both SMT threads at the same time, they share one cache slice similar to normal CPUs where both SMT threads share the L1 cache.

3.3 SMTCache Controller and Coherence

Multi-core processing with several simultaneously executing threads and shared, writeable memory requires caches to implement a coherence protocol that ensures all threads work with consistent copies of modified data.

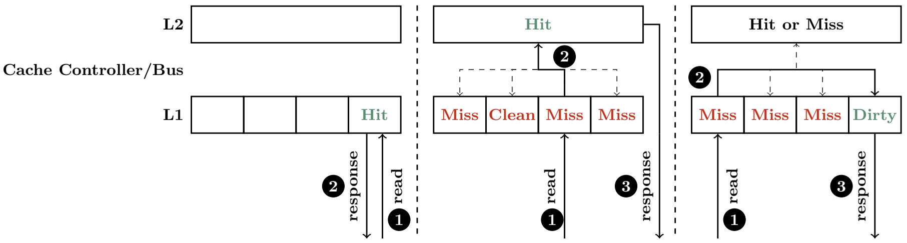


Figure 8.3: A read request can be satisfied by the same slice (left), by the L2 (middle), and by a sibling slice (right). Hits on clean lines in sibling core are served from upper levels to prevent side-channel leakage.

Information about changes to a location in one cache is propagated to other caches as soon as possible. SMTCache includes an extra coherency controller that facilitates security-aware snooping for the L1 slices (see Figure 8.1) to support shared memory, stay coherent between threads and processes, and curb high lookup latency for writeable shared memory. It handles misses from currently active L1 caches and requests from higher cache levels.

The snooping protocol works like the standard coherence between L1I and L1D caches. It avoids moving the attack surface from the L1 cache slice one layer higher to a directory [26], as there are no evictions from underprovisioning. Contrary to caches on different cores, the slices are also in much closer physical proximity, which reduces the cost of snooping. To reduce the energy costs of querying all slices for data, we propose a dual-mode line lookup for each cache line's tag and state data. When answering a request from the local core, only the currently active slice's set is searched, and tag, state, and data can be loaded in parallel. In response to a sibling-slice or remote miss, tag and state information from all slices is requested in parallel, without loading cache line data simultaneously.

Requests from the core first go to the assigned slice, then, for a miss, are forwarded to other slices and L2 cache at the same time (Figure 8.3). The cache controller can also aggregate cache line states w.r.t. upper levels, it can distinguish between a total miss in SMTCache and a hit on a clean or dirty line in a sibling slice. When a miss occurs in the controller, the request is served from the L2 cache. Likewise, when the data is found but is clean, the request is still served from the L2 cache to prevent Flush+ Reload (see Section 5). When a sibling slice contains the requested data and it is dirty, it can be served directly from there with limited security concerns. Since the position of the line is already known from the lookup request, the corresponding set does not need to be searched again, saving time and energy.

The slices together with the controller also keep track of copies and only forward modified data when the last copy is evicted or a coherence message from the upper level requires it. This avoids generating unnecessary traffic up the hierarchy when a line is evicted from one slice but still present in others. From a top-down view, SMTCache presents as a standard cache controller within the larger coherency protocol while maintaining its own internal state. Upon a request from a remote core, the controller can locate the address in the slices and adjust the cache lines accordingly, *i.e.*, changing ownership, responding with data, or flushing lines. Again, finding an address via the first broadcast already includes the location in the slice's set, so an extra lookup is unnecessary.

4 Energy and Area Estimation

Estimating energy and area overheads for commercial large-scale CPUs is difficult as CPU vendors do not open-source competitive state-of-the-art designs. Therefore, we follow the methodology of prior work [6, 9] and use McPAT [54] with CACTI [56] to estimate energy and area overheads, close to the actual hardware costs for commercial large-scale CPUs [54]. Like Townley et al. [6], we use the most recent Intel Xeon that McPAT supports. For the cache, we configure CACTI [56] directly, providing more fine-grained configuration and detailed information. The slices of SMT Cache behave like separate caches that each contribute to the static power consumption of the CPU. We interpolate unsupported non-power-of-two values.

Area. The main area overhead of SMTCache is storage area, closely resembling that of L1 caches in recent Apple CPUs. An increase from 32 KiB to a 128 KiB cache (like Apple's) comes with a proportional area growth of factor 4. The bus area increase is entirely negligible compared to the storage. SMTCache has a about 1 % area overhead from a basic 8-way cache due to additional complexity and tag bits added. However, SMTCache scales much better than a naive extension of current cache designs with a higher number of ways.

Table 8.1: Area and power overheads estimated with McPAT [54] and CACTI [56].

Number of Ways	8-way	12-way	16-way	2 slices (8-way)	24-way	3 slices (8-way)	32-way	4 slices (8-way)	40-way	5 slices (8-way)
Total L1 Cache Size	32 KiB	48 KiB	64 KiB	64 KiB	96 KiB	96 KiB	128 KiB	128 KiB	160 KiB	160 KiB
Number of SMT Cores†	1	1	2	2	2	2	4	4	4	4
Bus Area [mm^2]	0.37	0.37	0.38	0.38	0.39	0.40	0.41	0.42	0.53	0.54
Bus Peak Dynamic [W]	2.04	2.08	2.11	2.13	2.15	2.13	2.27	2.14	2.41	2.45
Bus Subthreshold Leakage [W]	0.05	0.05	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
Bus Runtime Dynamic [W]	2.04	2.08	2.11	2.13	2.15	2.13	2.27	2.14	2.41	2.45
L1 Dynamic read energy [nJ]	2.53	6.87	11.22	2.53	29.16	2.53	47.09	2.53	81.55	2.53
L1 Dynamic write energy [nJ]	2.58	7.01	11.45	2.58	29.73	2.58	48.01	2.58	82.72	2.58
L1 Standby Leakage [mW]	42.50	64.41	86.33	85.83	132.47	127.49	178.61	169.98	276.95	212.48
L1 Area [mm^2]	3.77	9.26	14.75	7.63	36.52	11.32	58.30	15.10	100.54	18.87
L1 Max. Total Leak. (2 loads+stores/cycle) [W]	10.26	27.84	45.42	10.30	117.90	10.34	190.38	10.38	328.83	10.43
L1 Max. Total Leak. (4 loads+stores/cycle) [W]	-	-	90.77	20.52	235.70	20.56	380.63	20.60	657.47	20.64
L1 Max. Total Leak. (8 loads+stores/cycle) [W]	-	-	-	-	-	-	761.04	41.02	1314.59	41.07

† For a fair comparison, we adjusted the number of SMT cores to reflect the L1 cache sizes: 1 SMT core below 64 KiB, 2 SMT cores for the 64 KiB to 96 KiB range, and 4 SMT cores above. We simulate the results for 3 different configurations for the load and store ports from 2 to 8 loads and store per cycle. The maximum total leakage significantly changes with the number of SMT cores and the number of loads and stores per cycle. As the slices of SMTCache act as independent caches, they scale almost linearly in the maximum total leakage.

Energy. The dynamic read and write energy for a single operation (2.53 nJ and 2.58 nJ respectively) stays at the level of the initial cache design (see Table 8.1). While standby leakage increases significantly it is negligible compared to overall power consumption. For the maximum total leakage, we use the metrics of a current CPU, *i.e.*, a throughput of 0.5 cache reads and writes per cycle. On a CPU with a 4 cycle cache latency, two load and two store ports, and 4 GHz clock, the upper bound for the throughput is 2 billion cache reads and cache writes each per second, which we also empirically tested on an Intel i7-8565U CPU.

For a fair comparison across all designs, we compute the maximum total leakage for 1 SMT core for all caches with less than 64 KiB, 2 SMT cores for all caches from 64 KiB to 96 KiB, and 4 SMT cores for 128 KiB or more. As SMTCache slices act as entirely separate L1 caches, their energy consumption only increases linearly with the number of slices. The two slice variant of SMTCache has twice as much maximum total leakage, as both caches can be fully utilized by the two SMT threads. However, even at this point the maximum total leakage is lower than the 12-way L1 cache without SMT and significantly lower than the 16-way L1 cache with two SMT threads. This trend continues for the 96 KiB to 160 KiB caches. The energy costs for the 40-way L1 cache are particularly prohibitive, whereas SMTCache with SMT-4 support stays below the maximum total leakage of the 16-way L1 cache.

5 Security

SMTCache provides strong isolation guarantees for the L1 cache. Therefore, we discuss how different cache contention and cache utilization channels are mitigated by our design. However, equally importantly, we show how SMTCache is a defense-in-depth against data leakage attacks.

Data Leakage (Defense in Depth). The strict separation of L1 slices ensures that the L1 cache can no longer be a source for leakage of data at rest, such as Meltdown [11, 13] and L1TF [11, 31, 32]. As requests from one domain are never directly routed to the slice of a different domain, the active L1 slice can never respond with data outside its domain. The request to other domains is only issued with the request to the L2, which happens after the permission check on Meltdown-affected hardware. Though orthogonal to SMTCache, a similar separation (or static partitioning) of the line fill buffer (LFB) could be implemented to additionally prevent leaking data in use, as seen in several microarchitectural data sampling (MDS) attacks [19, 23, 24]. Though these vulnerabilities have been mitigated in current CPU generations, designs with clear isolation boundaries provide defense in depth against possible future leakage from similar sources. We conclude that had these processors already followed a design like SMT Cache, Meltdown [11, 13] and L1TF [11, 31, 32] would have had very little security impact.

Kernel Domain. As mentioned in Section 3.1, the kernel shares a single domain. This is in line with standard process isolation but leaves open the possibility of (transient) confused deputy attacks. We weigh this against the significant overhead of providing each process with a separate kernel slice. We consider this an acceptable tradeoff, primarily because confused deputy attacks in the case of SMTCache require both a disclosure gadget in the victim's kernel code and a leakage gadget in the attacker's. Additionally, this attack surface is known, and gadgets have been systematically reduced in recent years.

OpenSSL AES. The AES T-Table implementation in OpenSSL is often considered as a benchmark for side channels. The typically page-aligned block of T-Tables (`Te` and `Td`) is accessed during the encryption, e.g., in the first round with a byte-wise xor of plaintext and key. With SMT Cache, the initial `prefetch256` call loads the tables into the L1 cache, *i.e.*, they are placed in separate slices of SMTCache. Consequently, we cannot observe any contention.

mbedTLS RSA. Another side-channel attack commonly used as a benchmark is the mbedTLS RSA implementation. mbedTLS uses a windowed square-and-multiply implementation. However, prior attacks [37, 45] exploited that a window size of 1 results in a simple square-and-multiply where the buffer containing the exponent is used in different ways, allowing to observe different contention patterns With SMTCache, the buffer is first loaded into the L1 cache, *i.e.*, again in separate slices of SMTCache where we cannot observe any contention.

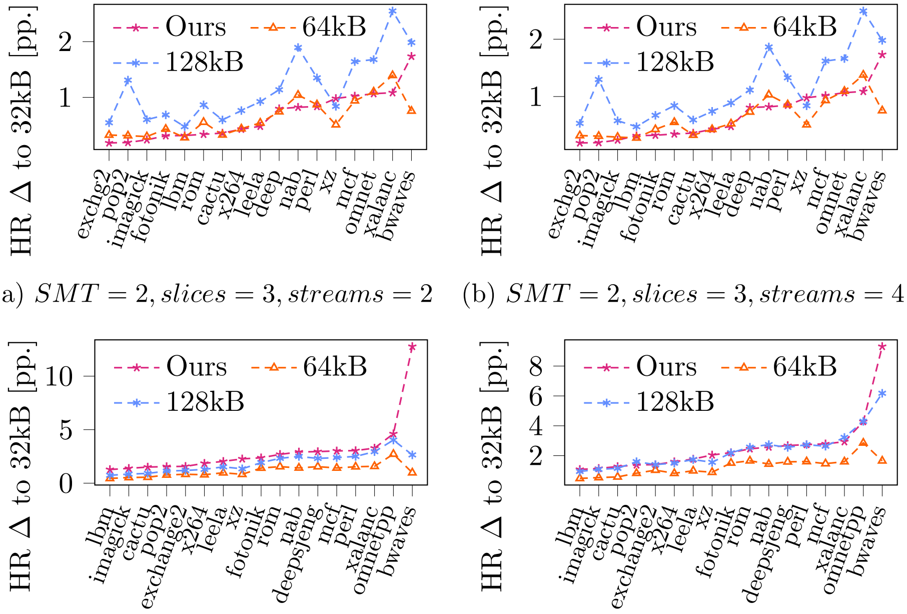
Generic Side Channels. In general, Prime+Probe builds on the foundational assumption that an attacker can find the set that the victim process' targeted address is cached in and interact with it. Specifically, the *Prime* step fills the entire set, thereby evicting the victim cache line. The *Probe* step then measures how many of the attacker's own addresses are still cached after the victim has executed some code. If an address has been replaced, the attacker infers that, with some likelihood, an address from the victim was loaded. SMTCache cuts this primitive off at the root, as two different security domains cannot interact with each other's cache line allocation anymore. As the sets are separated in both the slice and the L1 directory, the victim's set contents are unaffected by Prime+Probe or other attacks that manipulate the replacement algorithm.

Flush+Reload and Flush+Flush rely on shared memory between victim and attacker. However, with SMTCache, sibling slices do not respond to requests for unmodified data (see Figure 8.3 middle). Thus, neither Flush+Reload nor Flush+Flush on unmodified data are possible on SMTCache.

Cache side channels on writeable shared memory are still possible. However, this is a special case that was not handled by prior work on secure last-level caches either, as writeable shared memory already requires trust between victim and attacker for these shared memory regions. Hence, we also conclude that given the lack of a plausible threat model it is no case that SMTCache should cover.

6 Performance Evaluation

As gem5 lacks SMT support, we cannot use it to test SMTCache performance, as its benefits only materialize with SMT. Instead, we evaluate performance in CacheSim [4] and on an Intel CPU, both with SMT, using the SPEC benchmark. We evaluate real-world single-threaded and SMT



(a) $SMT = 2, slices = 3, streams = 2$ (b) $SMT = 2, slices = 3, streams = 4$

(c) $SMT = 4, slices = 5, streams = 4$ (d) $SMT = 4, slices = 5, streams = 6$

Figure 8.4: Average simulator hit ratios over SPEC-speed 2017 benchmark combinations of different L1D cache configurations compared to a standard 32 KiB cache. Each datapoint represents the average hitrate of that benchmark measured in all combinations with other benchmarks. Base hit ratios are around 90-99%. Benchmarks sorted by ascending SMTCache hit ratio.

switching behaviour on Linux server workloads over several hours, resulting in data for SMTCache performance estimates for different numbers of slices.

6.1 CacheSim Hit Ratio Simulation

We use CacheSim [4] to evaluate hit ratios in SMTCache in different SMT configurations and two levels of cache. Like prior work [3, 25, 52], we use a representative sample of 250 million instructions from SPECspeed CPU 2017 benchmarks. We use a standard 8 way, 32 KiB L1 instruction cache, combined with the different L1 data caches we evaluate.

SMT workloads are simulated by interleaving memory accesses of the currently active workloads. We test all 153 pairwise combinations (with repetition) of 17 SPEC workloads. To fill up to 8 SMT threads, we use multiples of the pairs to create up to 8 workload streams and avoid an explosion of simulation time. We shift the recorded addresses of streams such that no two workloads share memory addresses. To simulate context switches by the operating system, threads change their workload in regular intervals of 3 000 000 accesses, which roughly equals 500 Hz on a 3 GHz machine, assuming 2 memory accesses per cycle. Between each switch, the implementation of SMTCache briefly loads a fictitious kernel domain. In addition to context switches, we also add the option to simulate a number of syscalls in every context switch interval, e.g., 5 syscalls for every context switch. A syscall here is simulated simply by loading the kernel domain and switching back to the last workload.

We examine the hit ratio of these combinations in Figures 8.4 to 8.5. In Figure 8.4, we plot the averages for each benchmark combination. We simulate configurations where the number of workloads is equal or higher than the number of slices. This shows an ideal and non-ideal case for SMTCache. SMTCache performs about on par with a standard cache of equivalent size to the maximum active number of slices, *i.e.*, the number of SMT ways. We only see a significant deviation for the benchmark combinations that include *bwaves* (and, to a minor extent, *xz*), as this workload seems to use a particularly large working set. The example of *bwaves* also demonstrates the thrashing resistance of SMT Cache, as thrashing can only spill over to the second thread via evictions caused by inclusivity in higher caches. The combination of 2 *bwaves* workloads (Figure 8.4a) produces a 1.73 pp higher hit ratio on SMTCache than a standard cache on SMT-2 with 3 slices, compared to 1.98 pp for the 128 KiB standard cache with twice the concurrently available cache memory. This becomes even more pronounced for SMT-4 (Figure 8.4c) with hit ratio increases of 12.78 pp vs 2.64 pp for SMTCache (5 slices) vs. a 128 KiB cache.

Figure 8.5 reinforces the result that thrashing resistance becomes increasingly more pronounced with more logical cores. In this graph, the size of SMTCache increases with the number of SMT ways. While SMTCache starts with hit ratios very similar to the standard cache with the corresponding size, we can see that the hit ratio of standard caches quickly drop as the 8 workloads start to interfere more and more, while SMT Cache remains somewhat static.

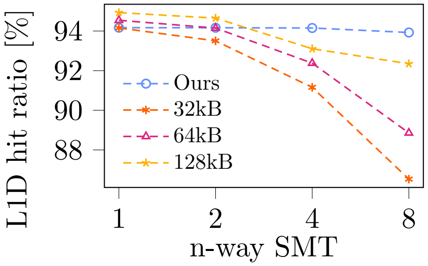


Figure 8.5: Mean simulator hit ratio over SPEC combinations for different number of slices with standard designs for reference. 1,2,4,8-way SMT. 8 workloads. $n = n_{SMT} + 1$ for SMT Cache.

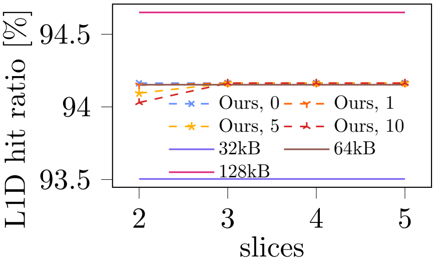


Figure 8.6: Mean simulator hit ratio over SPEC combinations for different number of slices with different numbers of syscalls per context switch. Standard designs for reference. SMT-2, 6 workloads.

As Figure 8.6 shows, the increase in hit ratio for each extra slice beyond $SMT + 1$ is fairly small, compared to the benefit from increasing the effective cache size. This coincides with our observations in other benchmarks (cf. Section 6.2) that returning to an empty cache is not a significant cost when the uninterrupted runtime is significantly larger than the time it takes to refill the cache. The overhead of the full-flush mitigation is mostly small, but some applications see a significant loss in performance [28]. In our tests, when we add a number of simulated syscalls per context switch similar to what we find in Section 6.2, we see that the gap from 2 to 3 slices grows slightly. Specifically, this occurs when the number of slices is not higher than the number of SMT ways, as then each syscall results in a full cache eviction. For example, in the depicted configuration with 2 slices, 6 workloads and SMT-2, we see the average hit ratio drop from 94.16 pp to 93.50 pp when we increase number of syscalls per context switch from 0 to 10. Therefore, the number of slices in our proposed default configuration of SMTCache is the number of SMT ways + 1. This ensures that applications always return to a full cache from a syscall.

6.2 Server Context-Switch Evaluation

We analyze real-world switching behavior with SMT-2 by running several server workloads in different configurations on a native Linux system and simulate the impact of our design. We use the applications proposed

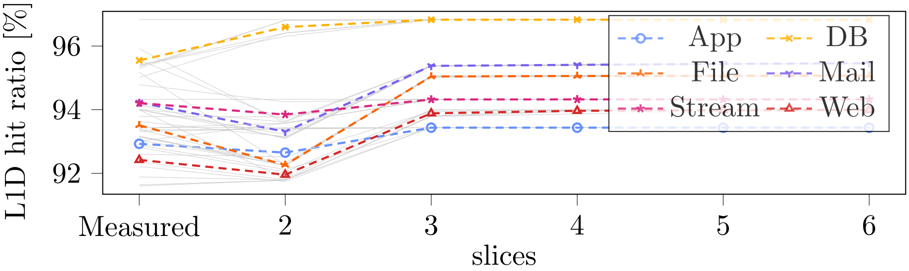


Figure 8.7: Measured hit ratios from conventional cache architecture compared to expected hit ratios for different numbers of slices in SMTCache. SMT-2.

by prior work ([38, 53]) to evaluate the performance of SMTCache in a realistic cloud scenario. These benchmarks include the following server applications combined in pairs of two to form our server workloads: Apache Tomcat (application server), MySQL (DB server), Postfix (mail server), Samba (file server), FFserver (streaming), and Apache (http server). We run all experiments on an Intel i7-6700K CPU with 4 cores and SMT. We isolate one physical core to eliminate interference from unrelated tasks and execute the workloads on the two SMT cores.

We modify a Linux v5.13 Kernel to record all context switches and syscalls with tracepoints in the `context_switch` and `do_syscall_64` functions. In addition to information about the current and next process, we also record L1D performance counters of hit ratios for all applications. The context switching and syscall information now lets us simulate LRU replacement for varying numbers of slices for each application in different workload combinations. The EER indicates how often a process receives a cleared cache upon being scheduled. A higher number of slices results in a lower EER. With the performance counters, we create two L1 cache hit ratio baselines for each server application on an isolated core. The first baseline is standard switching without flushing, yielding the highest possible hit ratio for each application (HR_{high}). For the second baseline, we evict the cache in every context switch and syscall, producing each application's lowest possible hit ratio (HR_{low}). We assume that the hit ratio decreases linearly from an EER of 0% (HR_{high}) to an EER of 100% (HR_{low}), that a process with a dedicated cache performs roughly the same as a process running on an isolated core, and that kernel threads only interfere minimally on isolated cores. Based on these assumptions, a process receiving a cleared cache each time it is scheduled (EER 100%) has the hit ratio of the process

Table 8.2: Comparison of measured syscall and scheduling metrics.

	App	DB	File	Mail	Stream	Web
Syscalls per scheduled period	6.86	4.84	10.56	10.98	46.34	4.12
Avg. scheduled period (ms)	0.033	0.088	0.066	0.298	9.843	0.028
Avg. time to context switch or syscall (ms)	0.004	0.015	0.006	0.025	0.208	0.005

operating on a dedicated core, where the cache is flushed at every context switch and syscall (HR_{low}) and vice versa (EER 0%, HR_{high}).

For the evaluation, we record the application's L1 hit ratios, context switches, and syscalls in all workload combinations. We use the information about context switches and syscalls to compute the EER for each application and varying numbers of slices. We then use the EER values to interpolate between the hit ratio baselines. This yields the expected hit ratios of each application in a SMTCache architecture with different numbers of slices.

Figure 8.7 shows the expected hit ratios for all workload combinations with different numbers of slices. The leftmost values represent the measured hit ratios for each application in all workload combinations in a conventional cache architecture. The other values are the expected hit ratios for the respective number of slices in a SMTCache architecture. The grey lines represent specific workload combinations. The colored lines show the average hit ratio for each application. Our evaluation shows an expected performance improvement for SMT workloads when there are *SMT ways + 1* slices compared to the measured value in a conventional cache architecture. We observe a decrease in performance when using as many slices as *SMT ways* in most workload combinations. The cause for this expected performance decrease roots in syscalls. Since syscalls constantly refresh the kernel to be the most recently used cache domain, only one slice is left for parallel tasks. Table 8.2 shows that, on average, between 4 and 46 syscalls occur during a scheduled period, depending on the application.

Figure 8.8 depicts each server application's average expected eviction ratios in a SMTCache architecture for different numbers of slices. We see a high eviction ratio when using as many slices as *SMT ways*. For some workloads, the computed eviction ratio is almost 100 % when using 2 slices on our machine with 2 *SMT ways*. Moreover, we see that using more than *SMT ways + 1* slices brings almost no performance improvement, given

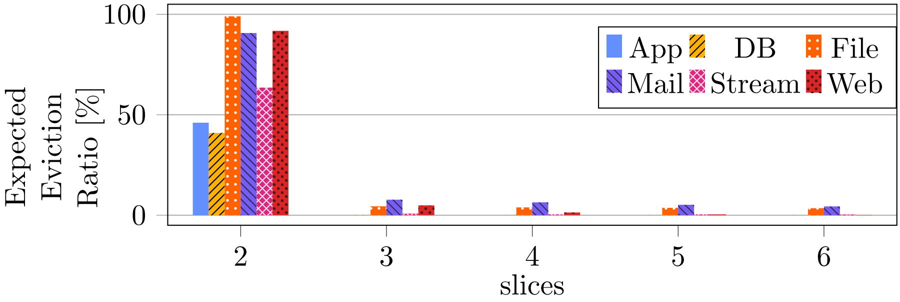


Figure 8.8: Expected eviction ratios computed from context switch and syscall information for different numbers of slices. SMT-2.

that the eviction ratio is already almost as low as 0 % for 3 slices for all applications.

The average time for a full L1D cache flush in our applications is 3 836 cycles (2 332 cycles to 10 272 cycles, median 2 401 cycles). We flush the L1D cache upon every syscall and context switch to record this value. We observe a higher duration of 10 272 cycles for the stream testcase compared to the other server applications. The average L1D flush duration correlates with the average time between syscalls and context switches for each application. The stream testcase runs uninterrupted for 0.208ms between syscalls and context switches on average (Table 8.2), allowing a longer time for data to be written to the L1D cache. As all dirty cache lines are flushed to higher cache levels, the L1D cache flush duration increases with the number of writes.

To confirm these L1 flush delays, we also micro-benchmark the full-cache-flush duration in gem5. The results show 350 writebacks on average taking an average of 1700 cycles, comparable to our real-world results.

7 Related and Future Work

Many secure cache designs have been proposed to curb these attacks. We can divide these designs into two groups: designs based on randomization and on partitioning. The former tries to obscure access patterns by making them seemingly random to an attacker, while the latter tries to make accesses unobservable. Many designs require complex functions whose latency is too large for implementation in an L1 cache or simply target the

LLC because they assume the underlying caches are secured in a different way. These designs therefore only target the LLC [3, 5, 6, 7, 8, 9, 17, 22, 25, 30, 41].

While prior partition-based designs may be applicable to the L1, they have so far come at a reduced cache utilization or available cache size. Only way-based partitioning even has the option to increase cache size, though as examined in Section 4, may come with increased energy needs. In this sense, SMTCache achieves an orthogonal goal of offering security and an increase in the overall L1 size, which is complementary to the partition-based designs. We anticipate that for a fully secure system memory subsystem, SMTCache will be combined with one or more of the secure cache approaches for L2 and L3 caches.

Wang et al. [58] presented *PLCache* and *RPCache*. *PLCache* has the ability to lock critical cache lines dynamically in the cache. While less wasteful than static partitioning, the programmer has to mark secrets. Instead, *Random Permutation Cache* tries to prevent observable interference between cache lines of different processes by randomizing their locations with a permutation table. Both *PLCache* and *RPCache* have low-overhead implementations, though Kong et al. [57] point out security-related shortcomings of both. Further approaches have been proposed that offer fine-grained specification of cache partitions, on a cache-line and cache bank granularity respectively [49, 55]. Still, all these designs are size-limited, where SMTCache offers an orthogonal approach to increase the overall L1 size.

Some works explored way-based partitioning [27, 52] similar to Intel CAT [39, 44] with additional security by disabling cross-domain cache hits and moderate performance costs. We believe that compared to our work, these way-split designs could not benefit from power savings in the way SMTCache does because of the dynamic nature of the designs. *Hybcache* [20] proposes selective cache partitioning that incurs only a low overhead and only for protected code. It does so by combining random replacement with a small but fully-associative sub-set of the cache for a trusted execution environment. *Jumanji* [16] partitions the L3 cache dynamically by splitting it into software-defined shares. Still, partitioning reduces the effective cache size, which is unsuitable for the size-limited L1 cache. *Newcache* [42] is a pseudo-fully-associative cache with random replacement, that maps address and domain ID of a load to a possible random location in the cache, at moderate performance, area, and energy costs.

TEE-SHirT [1] is a design with partitioned L3 caches and private L2 caches, and non-partitioned private L1 caches. To secure the L1 cache, they simply flush the cache on context switches, which is not overly expensive, given that refills from L2 and L3 are possible. Ge et al. [21] estimated the overhead for L1 flushing to be as low as 1 % on the L4 kernel. However, benchmarks on the Linux kernel showed a significantly higher cost of 10 % [28] on commodity CPUs. SMTCache complements *TEE-SHirT* from a security perspective while offering better performance than L1 flushing. Similarly, for *MI6* [18], SMTCache offers a better alternative to simple L1 flushing.

Future Work. Our experiments have shown that while scaling the number of slices with the number of SMT threads provides a performance boost very similar to an equivalent increase in cache size, going beyond has quickly diminishing returns. The impact of context switches and syscalls, however, shows that an extra domain for the kernel is useful. An open question for future work is, therefore, if a separate but smaller slice dedicated to the operating system would be a good tradeoff between performance and chip area.

To maintain energy consumption on par with current designs, we assumed the same bandwidth between the core and SMTCache as in standard caches. SMTCache supports twice that bandwidth for SMT-2. Future work could investigate dynamic scaling of the amount of issued loads and stores by the core to optimally fit power budgets and provide increased performance.

8 Conclusion

We proposed SMTCache, a secure L1D cache increasing cache size and thrashing resistance while being energy efficient. SMTCache achieves strong domain isolation, as security critical memory accesses from one domain are never served from another. With CacheSim and a simulation based on traces from native Linux benchmarks, we also showed that increasing the cache size with multiple slices provides not only the performance boost from simply increasing the cache size, but also from preventing interference between workloads. Lastly, our CACTI power simulation revealed that SMTCache design is significantly more energy-efficient than a traditional design of comparable size. We conclude that the SMTCache design shows promising results in terms of security, performance, and energy efficiency.

Acknowledgments

This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85 and FWF project NeRAM 10.55776/I6054). Additional funding was provided by generous gifts from Intel, Red Hat and Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

9 Appendix

9.1 Implementation of SMTCache in gem5

To demonstrate the functionality of SMTCache we implemented it in gem5. At the moment, the gem5 simulator does not support simultaneous multithreading in full system mode. Therefore, we cannot use it to estimate full system-level performance overheads for SMTCache, as it scales with simultaneous multithreading. We still modelled the additional latencies caused by our design realistically, allowing for micro-benchmarks of specific operations. Our implementation aims to functionally represent the features of the design described in Section 3, while working within the limitations of the gem5 codebase.

9.1.1 Implementation Overview

The gem5 framework simulates a freely configurable set of CPU cores, caches, crossbars (XBar), peripheral devices, etc. connected through ports on with each other. To avoid a complete overhaul of the memory subsystem, our implementation works within the system as much as possible, only swapping the default cache configuration with our SMTCache implementation.

Because all the SMTCache L1 slices behave like independent caches, we can build SMTCache on top of the existing L1 cache implementation. More specifically, we add functionality to perform a full cache flush (Section 9.1.3). In a typical CPU, the gem5 CPU core is directly connected to a L1 data cache. For SMTCache, we instead add multiple L1 caches and connect all of them to the CPU core through a custom SMTCache-XBar that implements the switch, as shown in Figure 8.1. Additionally, this XBar also simulates the SMTCache coherence behavior. The design of the XBar is described in detail in Section 9.1.2. The L2 cache in our system is shared between cores and the point of coherence. Usually gem5 connects all L1 caches of all cores to the shared L2 cache through the L2XBar. For SMTCache we do exactly the same, with all L1 data slices of all cores connected to the L2XBar. Finally, we customize the move-into-control-register instruction implementation (`MOV_C_R`) to inform our custom SMT Cache-XBar about a `CR3` change.

9.1.2 SMTCache-XBar Coherence Controller

The CPU core communicates with its SMTCache-XBar by writing to a special address, whenever the `CR3` register is written. This communication is necessary to allow the SMTCache-XBar to respond to a switch in the active domain. The SMTCache-XBar implements the LRU slice eviction and causes a full flush of all lines in the slice about to be assigned to a new domain.

Finally, the SMTCache-XBar also simulates the snooping coherence behavior. In a real implementation, every memory access would go to the active L1 slice, which may then forward the request to the controller if it is a miss. The controller then forwards the request to a slice that contains the cache line if there is one, or the L2 cache. In our gem5 implementation, the SMTCache-XBar directly checks all connected L1 slices and forwards the request to the correct one, if appropriate (*i.e.*, the line is found in the current slice or is modified in a different slice). By adding the correct latencies differentiating a cache hit vs a miss in the active L1 slice, our implementation can simulate the correct overhead. For the tag-matching in the slices, we budget one extra cycle. With this, we implement the behavior of the SMTCache coherence controller without requiring a separate component.

9.1.3 Flushing

Whenever a process without an associated slice is scheduled, the least recently used cache slice must be flushed and write back dirty data into higher cache levels or the main memory. Because we only have to write back dirty data, the flush latency is dependent on the number of dirty cache lines. Intel Skylake and later CPUs have a bandwidth of 1 cache line per cycle between the L1 and L2 [46]. This gives a lower bound of 512 cycles for a full flush if every single line is dirty. The flushing can take longer if, e.g., the L2 has to write data into the main memory to make space for the flushed data from the L1 slice. We implement our cache flushing to simulate this behavior and latency.

In gem5, caches can tell the CPU that they are blocked for various reasons. We use this mechanism to block the cache while flushing, as this can take many clock cycles. The CPU waits for the flushing to be finished, treating it as a fully serializing operation. This is important to avoid speculative loads or stores to the wrong slice during this step.

References

[1] Kerem Arikan, Abraham Farrell, Williams Zhang Cen, Jack McMahon, Barry Williams, Yu David Liu, Nael Abu-Ghazaleh, and Dmitry Ponomarev. TEE-SHirT: Scalable Leakage-Free Cache Hierarchies for TEEs. In: NDSS. 2024 (p. 200).

[2] 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. 186).

[3] 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 (pp. 181, 185, 193, 199).

[4] Giner, Lukas. CacheSim Cache Simulator. 2023. URL: `https://github.com/isec-tugraz/CacheSim` (pp. 182, 192, 193).

[5] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Chunked-cache: On-demand and scalable cache isolation for security architectures. In: NDSS. 2022 (pp. 181, 185, 199).

[6] Daniel Townley, Kerem Arikan, Yu David Liu, Dmitry Ponomarev, and Oğuz Ergin. Composable Cachelets: Protecting Enclaves from Cache {Side-Channel} Attacks. In: USENIX Security. 2022, pp. 2839–2856 (pp. 185, 189, 199).

[7] Thomas Unterluggauer, Austin Harris, Scott Constable, Fangfei Liu, and Carlos Rozas. Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks. In: SEED. 2022 (pp. 185, 199).

[8] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin Qureshi. Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. In: SEED. 2021 (pp. 185, 199).

[9] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security. 2021 (pp. 181, 185, 189, 199).

[10] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In: S&P. 2021 (p. 180).

[11] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In: FC. 2021 (pp. 180, 184, 191).

[12] Andrei Frumusanu. Apple Announces The Apple Silicon M1: Ditching x86 - What to Expect, Based on A14. Nov. 2020. URL: `https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive` (p. 181).

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading Kernel Memory from User Space. In: Commununications of the ACM 63.6 (May 2020) (pp. 180, 191).

[14] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, et al. The gem5 Simulator: Version 20.0+. 2020 (p. 182).

[15] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. Springer Nature, 2020 (p. 184).

[16] Brian C. Schwedock and Nathan Beckmann. Jumanji: The Case for Dynamic NUCA in the Datacenter. In: MICRO. 2020 (p. 199).

[17] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In: NDSS. 2020 (pp. 181, 185, 199).

[18] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In: MICRO. 2019 (p. 200).

[19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (pp. 180, 191).

[20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In: USENIX Security. 2019 (p. 199).

[21] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In: EuroSys. 2019 (p. 200).

[22] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In: ISCA. 2019 (pp. 181, 185, 199).

[23] 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. 180, 191).

[24] 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. 180, 191).

[25] 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 (pp. 181, 185, 193, 199).

[26] 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. 188).

[27] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: MICRO. 2018 (p. 199).

[28] Michael Larabel. An Early Look At The L1 Terminal Fault ”L1TF” Performance Impact On Virtual Machines. 2018. URL: `https://www.phoronix.com/review/l1tf-early-look` (pp. 195, 200).

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security. 2018 (p. 184).

[30] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: MICRO. 2018 (pp. 185, 199).

[31] 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. 180, 184, 191).

[32] 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/` (pp. 180, 191).

[33] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (p. 180).

[34] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In: EuroSec. 2017 (p. 180).

[35] Zhen Hang Jiang and Yunsi Fei. A novel cache bank timing attack. In: ICCAD. 2017 (p. 180).

[36] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (p. 180).

[37] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 192).

[38] Hao Wu, Fangfei Liu, and Ruby B. Lee. Cloud Server Benchmark Suite for Evaluating New Hardware Architectures. In: IEEE CAL 16.1 (2017), pp. 14–17 (p. 196).

[39] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In: HPCA. 2016 (p. 199).

[40] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security. 2016 (p. 184).

[41] 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 (pp. 185, 199).

[42] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. In: IEEE Micro 36.5 (2016), pp. 8–16 (p. 199).

[43] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security. 2015 (p. 184).

[44] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Cache. 2015. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf (p. 199).

[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. 192).

[46] Julius Mandelblat. Technology Insight: Intel's Next Generation Microarchitecture Code Name Skylake. 2015. URL: https://en.wikichip.org/w/images/8/8f/Technology_Insight_Intel%E2%80%99s_Next_Generation_Microarchitecture_Code_Name_Skylake.pdf (p. 203).

[47] Baker Mohammad. Embedded Memory Design for Multi-Core and Systems on Chip. Vol. 116. Analog Circuits and Signal Processing. Springer, 2014 (p. 183).

[48] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security. 2014 (p. 184).

[49] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In: PACT. 2013 (p. 199).

[50] Mutaz Al-Tarawneh. An Investigation of the Impact of Instruction Cache (I-Cache) Organization on Power-Performance Trade-Offs in the Design of Scalar Processors. In: European Journal of Scientific Research 115 (Nov. 2013), pp. 7–26 (pp. 181, 183).

[51] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. In: ACM SIGARCH Computer Architecture News (2011) (p. 182).

[52] 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 TACO 8.4 (2011) (pp. 193, 199).

[53] Dawei Huang, Deshi Ye, Qinming He, Jianhai Chen, and Kejiang Ye. Virt-LM: a benchmark for live migration of virtual machine. In: ACM/SPEC ICPE. 2011 (p. 196).

[54] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In: ICCAD. 2011 (pp. 189, 190).

[55] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In: ISCA. 2011 (p. 199).

[56] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. In: HP Laboratories 27 (2009), p. 28 (pp. 181, 183, 189, 190).

[57] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: CSAW (2008), p. 25 (p. 199).

[58] 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. 199).

[59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 184).

[60] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (p. 184).

9

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX

Publication Data

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

Contributions

Main author.

Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX

Lukas Giner[1], Andreas Kogler[1], Claudio Canella[1], Michael Schwarz[2], and Daniel Gruss[1]

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

Abstract

Load Value Injection (LVI) uses Meltdown-type data flows in Spectre-like confused-deputy attacks. LVI has been demonstrated in practical attacks on Intel SGX enclaves, and consequently, mitigations were deployed that incur tremendous overheads of factor 2 to 19. However, as we discover, on fixed hardware LVI-NULL leakage is still present. Hence, to mitigate LVI-NULL in SGX enclaves on LVI-fixed CPUs, the expensive mitigations would still be necessary.

In this paper, we propose a lightweight mitigation focused on LVI-NULL in SGX, LVI-NULLify. We systematically analyze and categorize LVI-NULL variants. Our analysis reveals that previously proposed mitigations targeting LVI-NULL are not effective. Our novel mitigation addresses this problem by repurposing segmentation, a fast legacy hardware mechanism that x86 already uses for every memory operation. LVI-NULLify consists of a modified SGX-SDK and a compiler extension which put the enclave in control of LVI-NULL-exploitable memory locations. We evaluate LVI-NULLify on the LVI-fixed Comet Lake CPU and observe a performance overhead below 10% for the worst case, which is substantially lower than previous defenses with a prohibitive overhead of 1220% in the worst case. We conclude that LVI-NULLify is a practical solution to protect SGX enclaves against LVI-NULL today.

1 Introduction

Transient-execution attacks, *i.e.*, Meltdown [42], Spectre [24], or ZombieLoad [27], are powerful microarchitectural attacks for leaking sensitive data. These attacks are commonly classified into Spectre-type and Meltdown-type attacks [20]. Spectre-type attacks [24, 36, 40, 41, 43] exploit that the transient instructions following a wrongly predicted branch are not committed but still leave traces in the microarchitectural state.

With Load Value Injection (LVI), Van Bulck et al. [15] presented a new type of transient-execution attacks related to Meltdown-type attacks. Meltdown-type attacks trigger a faulting load in the attacker domain to transiently consume its value, circumventing permission checks. LVI causes the fault in the victim domain, making the victim transiently consume a value from the attacker, *i.e.*, LVI transiently injects data into a victim.

Recent processors mitigate Meltdown-type attacks in silicon [6], e.g., Comet Lake processors have no known Meltdown-type vulnerability. As hardware defenses for Meltdown-type attacks in general also mitigate the corresponding LVI attacks, attackers cannot inject arbitrary data into the victim domain. However, on several microarchitectures, the hardware defense, instead of returning a value from the victim domain, only zeroes out the value [4, 15]. While this prevents data leakage, it can be used as a side channel to detect whether an address is valid, e.g., to break KASLR [4]. Even worse, this remains exploitable in an LVI attack variant, namely LVI-NULL [15]. With LVI-NULL, the attacker can still inject '0' values into the victim domain.

The LVI paper showed the dangers of LVI-NULL in an AES-NI attack, but the proposed defenses for LVI-NULL have not been thoroughly evaluated [15]. While the software workarounds for LVI also prevent LVI-NULL, they are costly, and Intel suggests that developers "should determine the level of software hardening that their environment requires, based on risk analysis and an evaluation of the performance impacts of mitigation" [8]. Potentially, every load instruction can suffer from a fault, requiring memory fences for such instructions [15]. This also includes replacing certain instructions, e.g., the return instruction, with sequences of other instructions [8, 15]. The worst-case overhead for these mitigations on real-world workloads is between factor 2 and 19 [11, 15]. This raises the question whether these prohibitively expensive defenses are still required on processors with hardware mitigations against LVI just to defend against LVI-NULL.

In this paper, we propose a lightweight mitigation tailored to LVI-NULL in SGX. Our mitigation, LVI-NULLify, is built on a systematic analysis of LVI-NULL variants, yielding new insights on the attack building blocks of each variant. In particular, we identify that four out of six variants rely on pointer redirection to the null page. Based on our analysis and experimental validation, we discover that LVI-NULL mitigations proposed by Van Bulck et al. [15] are not effective.

The idea of LVI-NULLify is to offset all memory accesses performed within the SGX enclave relative to the start of the enclave's memory region. To implement this property, LVI-NULLify repurposes segmentation. Segmentation is a fast legacy hardware mechanism on x86 that is used during address translation for every memory operation. The first part of LVI-NULLify is a compiler extension, which generates only segment-relative data loads. Consequently, any '0' injection only loads data from the start of the enclave's memory region, which is under full control of the SGX enclave. The second part of LVI-NULLify is a modified SGX-SDK that maintains interoperability with the untrusted userspace program.

The security of LVI-NULLify relies on special preparation of the enclave's memory region, mitigating transient injection of arbitrary values. LVI-NULLify marks the first pages in the enclave's memory region as non-executable. Transiently executing non-executable memory leads to an immediate stall, preventing any attack. LVI-NULLify also marks these pages as non-readable. We empirically validated that this immediately stalls the load and dependent instructions.

In our evaluation, we show that LVI-NULLify is extremely lightweight, with runtime overheads below 10% in the worst case. This is substantially faster than previous defenses against LVI with a prohibitive overhead of 1220% in the worst case in our tests. The memory overhead of LVI-NULLify is around 21.5% on the code size due to the generation of instruction sequences that explicitly use segmentation. We illustrate that our mitigation is a practical solution to protect SGX enclaves on hardware vulnerable to LVI-NULL but not LVI.

To summarize, we make the following contributions:

1. We systematically analyze and categorize LVI-NULL variants, revealing common attack requirements, and insufficiencies of previously proposed defenses.
2. We propose, LVI-NULLify, a novel lightweight defense against LVI-NULL in SGX, repurposing segmentation in a peculiar fashion.[1]

[1] We open-source LVI-NULLify on github : `https://github.com/IAIK/LVI-NULLify/`.

3. We evaluate the security and performance of LVI-NULLify. We demonstrate that SGX enclaves on the LVI-fixed Comet Lake CPU are only secure with our defense. We observe a performance overhead below 10%.

Outline. Section 2 provides background. Section 3 details our threat model. Section 4 systematically analyzes LVI-NULL variants. Section 5 presents the design and implementation of LVI-NULLify. Section 6 evaluates its security and performance. Section 7 discusses limitations. Section 8 concludes.

2 Background

2.1 Transient-Execution Attacks

Transient-execution attacks [20] are a new class of attacks that exploit so-called transient instructions, *i.e.*, instructions that are executed but never retired, to leak sensitive data. Kocher et al. [24] introduced the first sub-class with Spectre, while Lipp et al. [42] introduced the second with Meltdown. While Spectre attacks exploit control- or data-flow predictions made by the hardware, Meltdown exploits the deferred permission check when accessing memory from a different security domain. This deferred permission check allows the out-of-order execution to encode the normally inaccessible data in the cache from where the attacker then extracts it. Subsequent work showed additional variants in both sub-classes [12, 16, 19, 20, 27, 28, 36, 40, 43, 46]. Additional work has summarized the state-of-the-art of both transient-execution attacks [2, 17, 20] and defenses [3, 20].

2.2 Load Value Injection

Load Value Injection (LVI) turns Meltdown around by exploiting faults in the victim [15]. Thus, instead of leaking values, LVI injects values into the transient execution of the faulting victim. For LVI, the attacker prepares a microarchitectural buffer, e.g., the store buffer or L1, by filling it with the values that should be injected into the victim. Then, the victim has to suffer a fault or a microcode assist when fetching data from memory to transiently use the values injected by the attacker. The execution of subsequent instructions with the injected value is then exploited to either encode secrets in the microarchitecture or hijack the control or data flow. Similar to Spectre, LVI requires the gadget to be in the victim and has to additionally induce a fault or assist in the victim.

For unmitigated processors, the state-of-the-art solution for LVI is to insert `lfence` instructions after memory loads [8]. These fences ensure that faulting loads retire before the next instruction, effectively stopping all variants of LVI. However, this type of software mitigation comes with a performance penalty between factor 2 and 19 [11, 15].

2.2.1 LVI-NULL

Starting with the Cascade Lake microarchitecture, Intel processors include in-silicon mitigations against Meltdown, Foreshadow, and MDS attacks, including LVI [6]. These mitigations prevent non-zero value injections through all currently known buffers. However, this mitigation only prevents the attacker from injecting attacker-controlled data. Instead of stalling, faulting loads still transiently forward '0' to dependent instructions [4, 15]. Hence, by inducing a fault in the victim domain, an attacker injects the constant value '0' into the transient execution of the victim. This variant of LVI is called LVI-NULL. Even injecting '0' can be exploited to great effect, e.g., to transiently inject round keys consisting entirely of '0' into AES-NI computations [15].

The Comet Lake series represents Intel's latest SGX-enabled generation available for both mobile and desktop workstation models that is affected by LVI-NULL [6]. Ice Lake processors based on the Sunny Cove architecture appear to be unaffected by LVI-NULL [6].

2.3 Intel SGX

To provide processor-level isolation and attestation for secure enclaves, Intel developed Software Guard Extensions (SGX) [60]. By design, SGX assumes that only the processor is trustworthy. Hence, an attacker can have full control of the operating system while still being within the threat model.

When a secure enclave is run, it is placed in the virtual address space of an untrusted user-space process. While the operating system is untrusted, it is still responsible for maintaining the virtual-to-physical address mappings.

Naturally, this would make the enclave vulnerable to address remapping attacks [60]. To prevent these, SGX maintains its own shadow entry in the Enclave Page Cache Map (EPCM) containing the expected virtual address and the permission bits (R-W-X) for each valid enclave page. In case an illegal virtual-to-physical mapping is encountered, an EPCM page fault is raised.

Although side-channel attacks are not in scope of the SGX threat model, previous work showed that powerful side-channel attacks can be mounted against SGX. A root attacker can still mount low-noise side-channel attacks through the cache [49, 53, 54], page-table accesses [58, 59, 66], interrupt timing [48], or branch predictors [5, 33, 52]. SGX is also vulnerable to transient-execution attacks [21, 26, 27, 47] and Intel has released microcode updates to protect against them [22, 38].

2.4 Virtual Memory and Segmentation

In modern systems, virtual address spaces are used as an abstraction and to isolate processes. Hence, they are natively supported by the hardware. Each process works in its own, largely non-overlapping, virtual address space and cannot unintentionally interfere with the memory of another process. The used virtual addresses need to be translated to the corresponding physical addresses using a multi-level page translation table. The location of the table for the current process is indicated by a dedicated register and is switched by the operating system upon a context switch.

Another concept besides paging is segmentation. The idea is to have a set of segments for different uses, e.g., code, data, stack. While older processors used segmentation to enable the use of more physical memory, newer ones mainly use it as a protection mechanism.

Segments are configured via segment descriptors that are located in memory and are then used in conjunction with paging. Each segment descriptor has a base address and a limit. During the address translation, the CPU adds the base address to the segmented virtual address, yielding a non-segmented virtual address. Some instructions use segments implicitly (e.g., `push` and `pop` with the stack segment), and code fetches are implicitly performed via the code segment. Data segments can be used explicitly with memory referencing instructions.

On modern systems, paging has completely replaced segmentation for virtual address translation. Consequently, processor manufacturers removed this feature in the 64-bit long mode (IA-32e) for all segments but `fs` and `gs`. All but these two segments are now required to have a base of 0 and the maximum possible size. The segments `fs` and `gs` still support base and limit as they are broadly used to implement thread local storage for user threads and core local storage in operating systems. Hence, to use the base and limit feature of segmentation on 64-bit systems, user-level software has to use instructions that use `fs` or `gs`, and the operating system has to set up `fs` or `gs` with a base and a limit.

2.5 Object Relocations

Relocations are an essential part of the ELF file format [64, 68]. If a symbol is referenced inside an object file, the linker or the dynamic loader has to resolve the symbol's address and replace all the occurrences of this reference with the real symbol address. The relocation type specifies how this address should be calculated and which symbol is referenced.

SGX enclaves behave similarly to dynamic libraries and can be loaded on arbitrary addresses inside the main program's virtual address space. Therefore, enclaves and dynamic libraries need a mechanism to adjust addresses inside the image to point to the desired position in the address space. The most common way to achieve this is by using relative addressing, where all the absolute addresses inside the library are calculated over the instruction pointer. This type of relocation can be resolved during linking of the dynamic library.

In contrast to relative addressing, dynamic libraries also support absolute addressing where the dynamic loader resolves the addresses after the base address where the image is loaded is known. Here, the loader replaces placeholders inside the dynamic library with the real symbol address.

3 Threat Model

Hardware. For our mitigation, we assume a current or future Intel processor with SGX that mitigates LVI in hardware but does not prevent LVI-NULL, such as, e.g., the Comet Lake microarchitecture. We assume that there are no Meltdown-type transient-execution attacks [26, 27, 42, 47] that directly leak data from enclaves. Moreover, hardware vulnerabilities such as Rowhammer [34, 51, 67] or undervolting [9, 13, 25] are out of scope. We also assume that Spectre-type attacks [20, 21, 24] are either mitigated in hardware, firmware, or software. Additionally, we assume hyperthreading to be disabled. The Intel SGX Attestation Service indicates whether hyperthreading is enabled, so the verifying party can enforce its status.

Software. We assume a privileged attacker that is explicitly within the scope of the Intel SGX threat model. For the enclave, we assume that it is not vulnerable to traditional side-channel attacks, such as cache attacks [49, 53, 54] or controlled-channel attacks [66]. We assume that an attacker can start the enclave as often as required and thus rely on precise execution control, such as single- or zero-stepping [58]. Bugs in the enclave, e.g., synchronization problems [45, 63], or missing validations on the ABI or API level [29], are out of scope.

We consider only 64-bit enclaves, since enclaves can be (cf. Section 2.4) attacked via 32-bit segmentation [35], but not via 64-bit segmentation due to differences in the behavior.

Takeaway: Our mitigation targets 64-bit SGX enclaves on CPUs vulnerable to LVI-NULL, but not vulnerable to LVI.

4 Detailed Investigation of LVI-NULL

In this section, we first investigate the prevalence and impact of different LVI-NULL scenarios, and their applicability to SGX. We then examine the overhead and efficacy of current and proposed mitigations.

4.1 LVI-NULL Categorization

We distinguish control-flow and data-flow attacks (cf. Figure 9.1).

Control-flow Attacks. In control-flow attacks, the instruction pointer is transiently redirected in a way that serves the attacker. Again, we distinguish two cases: direct code redirection (❶) *to* the null page, or indirect redirection to arbitrary locations (❷ and ❸) *via* the null page. Direct redirection (❶) is achieved by faulting the load that reads the

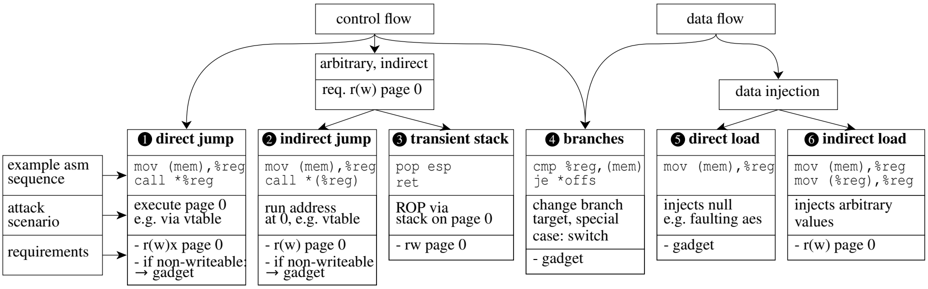


Figure 9.1: Categorization of LVI-NULL into control-flow and data-flow attacks. Subcategories list the different attack vectors and example assembly sequences for each (in AT&T syntax), the attack scenario, and their requirements.

call target, thus injecting '0' and redirecting code execution to the null page. In contrast, arbitrary redirection allows code execution anywhere in memory if the null page is attacker-controlled, e.g., for Intel SGX enclaves (cf. Section 2.3). It applies to indirect jumps (❷), which load their targets from memory via at least one indirection. Faulting the second to last load causes the jump target to be loaded from an offset in the null page, which allows arbitrary redirection. A special case of indirect redirection are sequences like `pop rsp; ret`, which load the stack pointer from memory and then return. This allows an attacker to set up a transient stack (❸) on the null page by faulting the stack pointer load, and perform a well understood ROP [70, 72] attack from there.

Data-flow Attacks. Data-flow attacks inject data into the victim's execution. We distinguish between direct (❺) and indirect (❻) loads, which allow the injection of either '0' or arbitrary values. Van Bulck et al. [15] showed that injecting '0' into the hardware AES-NI key schedule leaks the full key. A special case are binary branches and switch statements (❹), which can be compiled as jump tables. Here, data-flow manipulation changes the control flow, but only to the available branches.

Limitations. While LVI-NULL attacks are possible to execute from user space, several significant limitations apply. First, user-space attackers cannot manipulate page tables directly. This prevents these attackers from arbitrarily causing assists or faults on targeted loads. Secondly, most operating systems do not allow mapping of the null page by default. Both Linux and Windows require privileged access to map it, limiting the user-space attack surface to two cases (❹ and ❺). The exploitability of direct load ‘0’ injection (❺) depends on the targeted algorithm, and is thus best mitigated by developers themselves. Manipulating regular branches with ‘0’ injections (❹) is similar to Spectre variants and can be mitigated the same way.

4.2 Control-flow Injection

C/C++ compilers, such as GCC and Clang, commonly emit code patterns containing jump instructions whose target depends on an address or value loaded from memory. In our analysis, we found 3 categories of such jumps that are potentially susceptible to LVI-NULL.

Case 1: Virtual Function Calls in C++ (❶ and ❷) When objects in C++ call a virtual function, it is not known at compile time which function is being called. To solve this, each object has its own table (vtable), which contains the location of its virtual functions. Because the location of a dynamically allocated object itself (and thus its vtable) is also not known at compile time, calling a virtual function requires at least 2 loads. This creates 2 possible points of injection. First, the attacker may inject ‘0’ when the target is read from the vtable. This load may be generated by an indirect call instruction or a `mov` before a direct call, and can transiently redirect execution to the null page (❶). Secondly, the attacker can inject ‘0’ one load earlier, *i.e.*, when the address of the vtable is read. This causes the null page to act as the vtable, allowing transient redirection of execution to any location (❷). As the offset in the vtable is known at compile time, it is compiled to an immediate value that cannot be manipulated by LVI-NULL.

Exploitable in SGX enclaves? Very likely.
Indirect function calls (❷) occur frequently and are almost always immediately exploitable, as they allow redirection to any suitable gadget.

Case 2: Global Offset Table (❶) Another potentially interesting case is the global offset table (GOT), which enables programs to use functions in dynamically linked libraries. Unlike vtables, the GOT is always at a known location, and so only the call target is loaded. After the initial dynamic relocation of the symbols in the GOT, this only creates the potential to transiently redirect execution to address 0x0.

Exploitable in SGX enclaves? No.
While direct function calls (❶) are frequent, they are not exploitable in SGX.

Case 3: Switch Statements (❹) For certain switch cases, compilers generate a jump table, first loading the variable in question, and then looking up the corresponding jump target. This only applies to switch variables loaded from a single memory location, and not derived calculations. In position-independent code (PIC), this creates 2 attack points: injecting '0' into the variable itself, or injecting '0' when the jump target is calculated. The former transiently leads the switch into the '0' case, executing code there as if the variable were '0'. The latter causes the program to jump into the data section instead, as both GCC and Clang load offsets relative to the jump table. These offsets are likely not valid code, and furthermore, as Canella et al. [20] showed, the executable bit is respected in transient execution, so this injection is not exploitable here. When compiled as non-relocatable (no-pic), execution can again be redirected to the first case. Additionally, it can now be redirected to address 0x0, as the jump table contains absolute addresses, which can be zeroed on load (❶).

Exploitable in SGX enclaves? Unlikely.
Similiar to Spectre-PHT [20], exploitability is highly dependent on the specific case, but case-0-injections can be prevented reliably (see Section 5).

These are the three cases of commonly used code we found to enable control-flow injection. We expect there are more cases in other code patterns, compilers, or languages. However, for SGX, LVI-NULLify copes with all types of control-flow injection, as all control-flow injections rely on the null page. Table 9.1 explores the prevalence of such gadgets in standard SGX code.

4.3 Data Injection

Data injection gadgets are simply direct or indirect loads from memory, and as such, they are ubiquitous in all programs. Van Bulck et al. [15] have shown that in some cases, even '0' injections can be exploited to great effect. However, in cases where data injection does not lead to changes in control flow, it depends entirely on the algorithm at hand whether it can be exploited. As a direct '0' injection (❶) cannot be mitigated by software changes short of adding a load-serializing instruction after all potentially problematic loads, we do not consider this case. Instead, we leave it to the authors of software to guard their critical computations, such as cryptography, with this possibility in mind. However, in Section 5, we propose a way to prevent arbitrary data injection via indirect loads (❻).

Exploitable in SGX enclaves? Likely.
The danger of transient data injection depends on the targeted algorithm, but arbitrary value injection provides high flexibility for exploitation.

4.4 Applying LVI-NULL Variants in SGX

Of the attack vectors presented in Section 4.1 (cf. Figure 9.1), 4 out of 6 require at least read access to the null page. Variants ❹ and ❻ have no particular requirements and apply in any case. Most modern operating systems do not map the null page by default and typically require root privileges to do so [32]. Since the purpose of SGX enclaves is to protect against malicious or compromised operating systems, their threat model currently allows attackers to use the null page as they see fit.

From within an enclave, all memory of the user-space process is available for reading and writing according to its page-table entries, as it would be to the process itself. This implies that variants ❷, ❸, and ❻ apply fully if the null page is writable or with limitations, if it is not. Variant ❶, however, requires the null page to be executable. Van Bulck et al. [15] experimentally found that code outside of enclave memory is not executable from within an enclave, even during transient execution. This was later confirmed by Intel [8], and we have reproduced this result as well. It follows that the only way to execute instructions at address 0x0 is to load the enclave itself starting at the null page. Since the Intel SDK does not build enclaves with execute permissions on this page [8], we consider variant ❶ *not exploitable in SGX* enclaves.

To evaluate the prevalence of assembly sequences that allow LVI-NULL types ❷ and ❸, we search several prebuilt- and SDK-generated binaries for a limited selection of exploitable assembly patterns. As Table 9.1 shows, indirect calls (❷) are plentiful in these binaries, though they are currently mitigated by `lfence` instructions. We find that there are even some gadgets for variant ❸. An especially interesting observation is that

Gadget/File	QE	LE	PCE	PVE	trts	tsdc	tcxx	tcmalloc

`mov (mem), %reg`
`mov (%reg), reg`
`call *reg`
	216	0	123	216	0	0	0	4

`mov (mem), %reg`
`call *(%reg)`
	0	45	0	0	0	0	19	3

`mov (mem), %rsp`	0	2	0	0	1	0	1	0

`mov (mem), %reg`
`mov %reg, %rsp`
`ret`
	1	1	1	1	1	0	0	0

`mov (mem), %reg1`
`mov %reg1, %reg2`
`mov %reg2, %rsp`
`ret`
	0	0	0	0	0	0	0	0

`pop %reg`
`mov %reg, %rsp`
`ret`
	0	0	0	0	0	0	0	0

Table 9.1: Number of control-flow gadgets found in Intel's prebuilt (quoting, launch, provisioning) enclaves and SDK libraries. Search was limited to instruction sequences with fewer than 10 separating instructions.

the original transient stack gadget, as described by Van Bulck et al. [15], is still present in unmitigated form in the prebuilt launch enclave for Linux provided by Intel as of SDK release 2.11.

An analysis of how some code patterns generate vulnerable instruction sequences is shown in Section 4.2.

Takeaway: ❷, ❸, ❹, ❺, and ❻ are all feasible in SGX.
❶ is feasible, but mitigated by default.

4.5 Current and Proposed Mitigations

In this section, we discuss the two main types of mitigations against LVI and LVI-NULL for SGX.

4.5.1 Memory Fences

The officially suggested mitigation against LVI is to stop transient execution before it can be exploited. Similar to the mitigations for Spectre [39], Intel also suggests to use memory fences for aborting transient execution [8]. As it is infeasible to add memory fences manually, these memory fences are supposed to be emitted by the compiler. With the publication of LVI [15], Intel has provided 2 levels of mitigation [7, 8], and Google engineer Zola Bridges another [1, 10]:

Control-Flow Mitigation. This mitigation replaces `ret`, `call`, and `jmp` instructions by fenced alternatives. It protects transient control-flow redirection at the cost of effectively disabling all control-flow predictors. However, it does not generally protect against value injection and only prevents these special cases. Compilation options: `-mlvi-cfi`

SESES. "Speculative Execution Side Effect Suppression" aims to prevent more than just LVI by adding an `lfence` instruction before every instruction that operates on memory. This approach fully mitigates LVI, LVI-NULL, and other transient execution attacks. Compilation options: `-mseses`

Optimized Cut. In addition to CFI, this mitigation for LVI (which we call "optimized cut") tries to separate loads from potential transmit gadgets by analyzing the control-flow graph of applications. Hence, the compiler can insert far fewer `lfence` instructions than SESES while still providing the same security guarantees w.r.t. LVI. Compilation options:

```
-mlvi-hardening -mllvm -x86-lvi-load-opt-plugin=
OptimizeCut.so -x86-experimental-lvi-inline-asm-hardening
```

While these three levels of mitigation differ in the amount of `lfence` instructions (cf. Table 9.2), they all incur heavy performance penalties in the range of factor 2 to 19 [11, 15].

4.5.2 Page Table Protections

Van Bulck et al. [15] also proposed specific mitigations for LVI-NULL. To prevent execution of the null page (❶), they suggest marking the first page in an enclave as non-executable or placing an infinite loop at the base of the enclave image.

As described in Section 4.4, marking a page non-executable indeed prevents execution in the transient domain. Experiments on our i5-10210U show that this holds even if the OS marks a page as executable after loading the enclave. Read, write, and execute permissions are also stored with the expected virtual address in the protected Enclave Page Cache Map

(EPCM) entries. Our experiments suggest that in transient execution, the CPU considers the permission bits of both the page table and the EPCM and applies whichever is less permissive. As this prevents ❶, an infinite loop or similar is not necessary.

To stop transient null pointer dereferences (❷, ❸, and ❻), Van Bulck et al. [15] suggest marking the null page as uncacheable. This has also been proposed as a possible mitigation for Spectre attacks [14], as uncachable memory cannot be read during transient execution. Any transient access to uncachable memory simply stalls [14]. While this would indeed prevent loads from this page, the OS can simply change these flags at any time, as they are not protected by SGX. We verified that in contrast to the read, write, and execute permissions, the memory type is not enforced by SGX and can be manipulated to mount an attack. Additionally, injection via the shared line-fill buffer is possible on some architectures [12, 14, 26, 27] if hyperthreading is enabled.

Takeaway: Current mitigations are too costly or are insufficient. All LVI-NULL variants are preventable by LVI mitigations, but incur substantial performance degradation. Other proposed mitigations are only partially effective.

5 LVI-NULLify

Previous mitigations have been designed primarily for LVI, not LVI-NULL. Hence, they mitigate attacks that are already mitigated more efficiently in current and future Intel processors, e.g., the recent Comet Lake microarchitecture. Following the analysis of Section 4.5, we can see that these previous mitigations either have a substantial performance overhead or are limited to only certain variants of LVI-NULL. This motivates the need for a defense that is more tailored to LVI-NULL. In this section, we present LVI-NULLify, our mitigation for LVI-NULL affected hardware that achieves a better balance between performance cost and remaining attack surface than previous LVI mitigations. The worst-case overhead on our LVI mitigated Comet Lake is only $\approx 9\%$, our older LVI-vulnerable Coffee Lake-R reaches a maximum of 36% overhead.

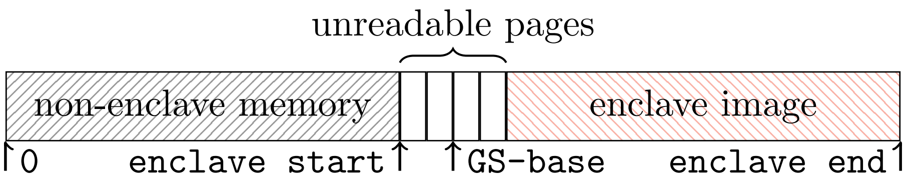


Figure 9.2: Memory layout of enclaves protected with LVI-NULLify.

5.1 LVI-NULLify Design

LVI-NULLify aims to prevent all LVI-NULL variants in our threat model (see Section 3). This includes variants ❶, ❷, ❸, and ❻. Though we categorize switch expressions as a subclass of branches (❹) in Section 4.1, we briefly describe a mitigation in Section 5.1.4 that is also applicable outside of SGX. Since all other variants involve the null page, the central feature of LVI-NULLify must be to control either its contents or accessibility. Therefore, we devise a way to effectively *move the LVI-NULL target page into the enclave*, even if address 0x0 is not in the enclave's linear address space.

SGX does not currently offer any control over pages outside of the enclave memory range. Hence, loading an enclave anywhere but page null means giving up control of the null page. As multiple enclaves can and often need to be loaded simultaneously, only loading an enclave if it is mapped at address 0x0 is not a practical option. Our solution is to *offset every memory load in the enclave* such that any pointers that are loaded from memory are added to an immutable constant. For this constant, we use the virtual base address of the enclave image. The resulting memory layout is shown in Figure 9.2. Even if an address load faults and transiently returns '0', the resulting load address is still within the enclave. This puts the control over the pages targeted by LVI-NULL into the hands of the enclave, regardless of where an attacker maps it.

5.1.1 Using Segmentation

To offset loads on commodity Intel CPUs, we rely on segmentation. In 64-bit mode, segments typically have to start at address 0x0. However, the GS and FS segment registers are an exception and can have non-zero base addresses. Conveniently, the `EENTER` and `ERESUME` instructions automatically set these base addresses to the enclave base address plus a developer-controlled, positive offset. As the GS and FS registers are set on each entry, and the offsets are stored in enclave memory in the thread control structures (TCS), these values are inaccessible to the OS. They are also part of the enclave's attestation, preventing manipulation at load time. The Intel SGX-SDK sets both registers to the same value, creating an unnecessary redundancy. We can thus repurpose one of the two segment registers, in our case GS, for LVI-NULLify.

Since address calculation with segment bases is an integral part of x86 hardware, there is no noticeable slowdown for GS-relative loads, as we experimentally verify in Section 6. We set the GS base to the beginning of the enclave, such that data loads in our enclave are now relative to the beginning of the enclave. This means that we essentially build a non-relocatable object (no-pic) that gains its position independence by adding GS base to all addresses.

Applying this mitigation to generic SGX enclaves requires the modification of 3 components: the compiler, the Intel SGX-SDK, and the Intel Platform Software (PSW). The compiler has to emit GS-relative loads for all memory-load instructions. We discuss how this is implemented in LLVM in Section 5.2. Section 5.3 then details the custom relocations that are necessary after compilation. Changing all load instructions also implies that all addresses inside the enclave are invalid outside and vice versa. Hence, this necessitates an automatic pointer conversion from the trusted runtime system (tRTS) to the untrusted runtime system (uRTS). Additionally, the tRTS itself needs to be built with our compiler modification. We detail the necessary modifications to the SDK in Section 5.3.2. As the GS offset is in part calculated by the PSW, we need to enable it to distinguish between enclaves mitigated by LVI-NULLify and unmitigated enclaves. The changes to the PSW are also described in Section 5.3.2.

5.1.2 First Enclave Pages

With GS-relative addressing, most transiently faulting indirect loads read from the first page of the enclave. For this reason, it is critical that reading from it does not return values that can be used in an attack. To prevent variant ❶ of LVI-NULL, which requires code execution, simply making this page non-executable is sufficient. Additionally, marking the first page non-readable in the EPCM entry prevents all variants that load from this page (❷, ❸, and ❻). Transient accesses to such pages simply stall, as we have experimentally confirmed on several CPUs, including our i5-10210U Comet Lake. This stops all further dependent accesses. However, the first page in an enclave may contain data that the dynamic loader inside the enclave needs to access. We, therefore, shift the enclave image and prepend empty pages that are neither readable nor executable. The amount of such pages that are needed depends on the enclave program; loads with offsets, where the base address can be zeroed, may transiently load more than 1 page from the GS base address or even below it. Negative offsets could, therefore, lead to loads that escape the enclave. Fortunately, the size of all immediate offsets is known at compile time. We can determine a safe amount of empty pages to map before and after the GS base after compilation. An example with 2 pages on each side of the GS base address is shown in Figure 9.2.

The residual attack surface to this approach are dynamic arrays. When neither position nor size are known at compile time, loads can take the form *base + offset*, where both base and offset are loaded from memory. If an attacker faults the base, the transient load target is potentially anywhere from the enclave beginning up to the size of the dynamically allocated structure.

5.1.3 Alternative Approaches

There are other, less peculiar mechanisms to prevent faulting loads from reaching the null page. We want to examine two candidates here, as their shortcomings are not immediately apparent.

First, we could simply add the base address to all loads by converting every load to use complex addressing. For this, we cannot load the base address from memory, as this load could again be zeroed. A solution would be to always keep the base address in a CPU register, but this reduces the number of registers available to the compiler. This can incur significant performance impacts as less efficient code can be generated. Alternatively, we could use memory fences for these loads, which would again result in large performance overheads. Additionally, any load that already uses complex-addressing needs an additional offset computation beforehand.

The second approach would be to employ immediate addresses by using the dynamic loader to relocate all symbols. An undesirable side effect of this method is the need to set the text section of the enclave to writable and executable. As enclave pages cannot currently change access permissions at runtime, this would weaken the enclave's protection against traditional exploits that might otherwise not be exploitable. As x64 does not generally support 64-bit displacement in complex addressing [62], this is also not a possibility. Only `AL`, `AX`, `EAX`, and `RAX` may be the source or target of immediate 64bit-addressed memory loads, which introduces the need for more intermediary registers [23].

5.1.4 Mitigating Switch Statements

Regular branches (❹) can be mitigated with Spectre-PHT mitigations, e.g., speculative load hardening. Switch statements present a special case of branches. On processors with hardware mitigations for branch target injection, e.g., single thread indirect branch predictors (STIBP) [37], such as the Comet Lake series, switch statements are protected from cross-hyperthread manipulation through the branch target buffer. However, they are still vulnerable to LVI-NULL when the compiler generates a jump table (see Section 4.2, case 3). We can mitigate the case of single-variable conditions (conditions that only depend on one in-memory variable) by targeted insertion of `lfence` instructions. This is described in more detail in Section 5.2. It is the only variant we mitigate that can also be exploited outside of SGX, and this mitigation can also be applied alone.

5.2 Compiler Changes

We developed an open-source implementation of the compiler-part of LVI-NULLify that is based on the LLVM compiler framework [71]. It handles the insertion of fences in switch statements (cf. Section 4.2) and ensures that every load is relative to the GS segment (cf. Section 5). Our modification consists of two new passes, one module pass (*i.e.*, a transformation pass) that works on the LLVM intermediate representation (IR) and one machine function pass in the x86 backend. The mitigation for switches is purely done in the pass on the IR, while the GS-relative addressing is implemented in the backend pass. For our modification, we added 1 024 lines of code to the LLVM code base (24 in 10 existing files, 1 000 in 3 new files).

Switches. When compiling an application with optimizations enabled, LLVM already tries to optimize the performance of switches [65, 69]. These optimizations are implemented as a transformation pass that iterates over all functions in the translation unit. If a switch is encountered, the pass tries to apply these optimizations. Unfortunately, this transformation pass is only executed when the application is compiled with at least optimization level 1. Hence, we cannot use it and need to implement a new pass.

We extend LLVM with a new module pass (*-flvi-null*) that iterates over all functions in the translation unit and searches for a switch within the basic blocks that comprise the function. If such a switch is found, the mitigation is applied. If the switch already contains a '0' case, the compiler simply modifies the case such that the first instruction within the basic block is a fence instruction. Otherwise, LVI-NULL falls back to the default case of the switch. Either one can lead to exploitable behavior. To prevent this, the compiler inserts a new '0' case that contains a fence instruction and then performs an unconditional branch to the default case. This new '0' case is only ever executed if the default case is supposed to handle zero values, or if an attack takes place. This significantly reduces the performance impact.

By considering these two cases, the compiler part of LVI-NULLify can mitigate LVI-NULL targeting switches. This mitigation is not specific to SGX and does not require segmentation, hence this can also be used in non-enclave applications. While this mitigation on its own is not sufficient to fully prevent LVI-NULL, as it only mitigates a subset of variant ❹, it is a necessary building block for LVI-NULLify.

GS-Relative Addressing of Loads. As discussed in Section 5, every load, explicit or implicit, has to be relative to the GS segment. Thus, in case of an LVI-NULL attack, the control flow is re-directed to a location that is controlled by the SGX enclave. To achieve this, we add a new machine function pass in the x86 backend of LLVM.

In the machine function pass, we iterate over each instruction of a given machine function, and replace explicit loads, implicit loads from pushing to and popping from the stack, as well as calls, jumps and returns, as all of these loads are also exploitable by LVI-NULL (❶, ❷, ❸, and ❻). Hence, the transformation pass replaces each such instruction with an equivalent sequence of instructions that use GS-relative addressing where necessary. Figure 9.3 shows some cases that we consider for implicit loads and how our modified compiler mitigates them.

As jump and call instructions cannot use the GS segment, and the CS segment cannot be changed (see Section 2.4), we manually convert the

```
push %rbp

callq 400480 <func>




pop  %rbp

retq


call *16(%r12,%r13,8)
```

(a) unmodified

```
sub  $0x8,%rsp
mov  %rbp,%gs:(%rsp)
lea  $return_address(%rip),%r11
sub  $0x8,%rsp
mov  %r11,%gs:(%rsp)
jmpq 400480 <func>
mov  %gs:(%rsp),%rbp
add  $0x8,%rsp
mov  %gs:(%rsp),%rcx
add  $0x8,%rsp
jmpq *%rcx
lea  __ImageBase(%rip),%r11
add  %gs:16(%r12,%r13,8),%r11
sub  $0x8,%rsp
mov  %r11,%gs:(%rsp)
lea  $return_address(%rip),%r11
xchg %r11,%gs:(%rsp)
jmp  *%r11
```

(b) modified

Figure 9.3: Figure 9.3a shows the unmodified assembly instructions containing implicit loads while Figure 9.3b shows the instruction sequence with which they get replaced by our modified compiler.

relative call address of, e.g., an indirect call, to an absolute address by adding the image base. To protect this pointer conversion from LVI-NULL, we use a rip-relative `lea` instruction to calculate the image base instead of loading the address from memory. Hence, we ensure that all pointers inside the enclave are relative to GS regardless of their data representation.

As our pass is run as the last pass before the actual code is emitted, no additional load can appear that is not GS-relatively addressed. With these compiler modifications, and in combination with the further LVI-NULLify components, we successfully mitigate LVI-NULL, as we show in Section 6.

5.3 Relocation and SGX-SDK Changes

In addition to the compiler changes (cf. Section 5.2), we require additional changes in the relocations of the object files (cf. Section 2.5) and changes in the SGX-SDK and SGX-PSW to realize the GS-relative addressing. We discuss these changes in this section.

5.3.1 Relocation Types

The generated instructions from our compiler pass do not use the instruction pointer for relative addressing, and therefore, the compiler emits absolute relocations. Since the enclave image is signed by the author of the enclave and the signature is verified during the enclave initialization, absolute address relocations cannot be resolved before verifying the signature [60]. Additionally, enclave memory cannot be modified from outside, so each enclave contains an ELF loader to resolve any relocations during the initialization phase. A disadvantage of this is that the page flags cannot be changed after the enclave is instantiated. Therefore, pages containing absolute relocations must remain writable at runtime, even if they contain text sections. Pages that are writable *and* executable are a traditional security concern. While the enclave signer issues an error message if it finds such text absolute relocations, this error can be suppressed. Therefore, keeping the original page flags without making pages writable is a design goal of LVI-NULLify.

Since we do not need absolute address relocations when using the GS segment to specify the image base, we fully replace these absolute relocations. We build an additional tool to change the relocation types directly in the object files after compilation. LVI-NULLify does not enforce additional requirements on the build environment by reusing the existing relocation types instead of implementing a new one for GS-relative addressing.

The compiler extension emits `R_X86_64_x` absolute relocations as part of the code generation. We then iterate over the generated ELF object file and exchange these absolute relocations with the `R_X86_64_COPY` relocation type. The copy relocation type fills in the relocation destination with the symbol's offset from the image base, exactly what is needed for the GS-relative addressing. The copy relocation's addend is set to the difference between the GS-base and the real enclave base to implement the additional pages in front of the enclave (see Section 5.1.2 and Figure 9.2), *i.e.*, shifting the address of the relocated symbol.

5.3.2 SGX-SDK and SGX-PSW

For LVI-NULLify to work, some changes to the SGX-SDK and the SGX-PSW are required. The changes detailed here are made mostly to the enclave-loading mechanism and for automated pointer conversion between enclave and host application.

SGX-PSW. The SGX-PSW is used globally for loading all enclaves. For LVI-NULLify, the PSW has to set the GS segment in the thread-control-structure template (see Section 5.1.1). This template is used for creating threads inside the enclave, and is used for the attestation process. Hence, the same changes are also required in the SGX signer. We ensure backward compatibility with enclaves that are not protected by LVI-NULLify by indicating the use of LVI-NULLify in the enclave signature structure. Hence, the PSW only modifies the GS segment if LVI-NULLify was used for building the enclave.

SGX-SDK. Most changes in the SDK affect the ECALL and OCALL interface. With LVI-NULLify, the enclave and the host application basically operate in different virtual address spaces. Hence, the ECALL and OCALL interface have to apply pointer conversion. On enclave entry, the GS segment is automatically set by the `ENCLU` instruction. We modify the `enclave_entry` function to convert the stack pointer, the base pointer, and the pointer to the structure used to pass additional data into the enclave. Some minor changes also adapt the elf loader inside the enclave for GS-relative addressing, as some of the supported relocation types refer to the absolute enclave base.

In the SGX-SDK, the *edger8r* application is responsible for parsing the enclave interface definition file and generating the trusted and untrusted part of the enclave interface. To ensure the enclave can use pointers passed to an ECALL implementation without manual modification of the code, we modified the *edger8r* application for the code generation of the trusted enclave API. As this parser already has all the information about functions and their parameter types, it can automatically generate code for converting pointers from absolute pointer addresses to GS-relative addresses. Thus, all the default cases of passing pointers into an ECALL or OCALL are handled automatically.

The enclave definition language also allows the definition of data structures for ECALLs and OCALLs. These structures can be automatically copied into the enclave memory, but nested structures or structures containing additional data over pointers must be copied by hand from the enclave developer [61]. Hence, we also leave pointer conversion for such data types to the enclave developer.

6 Evaluation

6.1 Security Evaluation

For the security evaluation, we first perform a theoretical analysis of all variants in the context of our mitigation. Additionally, we also evaluate our own proofs of concept demonstrating LVI-NULL (see Section 9.3). All experiments are run on an Intel Core i5-10210U Comet Lake that is vulnerable to LVI-NULL but not to LVI. In all of our experiments, LVI-NULLify successfully prevents all targeted variants of LVI-NULL.

Variant ❶, direct jumps. If the target of a jump instruction, such as `call` or `jmp`, is loaded from memory, it can be zeroed using LVI-NULL. As a result, transient execution continues at address 0x0, which is either outside the enclave, and therefore not executable in the context of SGX, or on the first page of the enclave, which is also not executable as ensured by the SDK. This behavior stays the same with LVI-NULLify, and is thus *not exploitable.*

Variant ❷, indirect jumps. When the first of the two loads in an indirect jump is zeroed, the jump target is read from address 0x0, plus potentially an offset used in the indirect-jump instruction. This is, e.g., the case for an entry in a vtable (cf. Section 4.2). Without LVI-NULLify, this address points to the virtual address 0x0+offset. This address can be outside of the enclave and thus under attacker control.

With LVI-NULLify, the load is performed with GS base, which ensures that the address is inside the enclave. As a number of pages (that depend on the largest such offset in the enclave) directly after GS base are non-readable, the address load stalls, and no jump occurs. Since function offsets can generally be determined at compile time, the required number of buffer pages can be reported by the compiler. An example would be finding the maximum number of entries in a vtable. This does not consider programs that use 'manually' constructed jump tables. While in rare cases, a dynamic offset could be large enough to reach beyond the allocated buffer pages, e.g., when manually constructing dynamic jump tables, most cases are prevented, and the remaining rely on very specific circumstances. Hence, *we consider this variant mitigated.*

Variant ❸, transient stack. As shown by Van Bulck et al. [15], function epilogues that load the value of the stack pointer from memory and return can be exploited to transiently use the null page as the stack by zeroing the load. This attack allows for arbitrary code redirection using transient return-oriented programming. LVI-NULLify replaces the return instruction (`ret`) with a GS-relative load and jump (cf. Figure 9.3). As a result, mounting an LVI-NULL attack moves the transient stack to a non-readable page within the enclave. *This prevents the transient stack attack*, as long as the enclave developer does not actively try to circumvent that, e.g., by loading the stack pointer with an indirect load instruction using a very large offset.

Variant 2b, branches. For regular branches, LVI-NULL behaves very similar to Spectre-PHT and are thus *out of scope* for LVI-NULLify. Developers can mitigate them with speculative load hardening if they choose. As discussed in Section 4.1, switch constructs represent a special case of branches, as they can be implemented as a jump table. Without mitigations, execution can be redirected to case '0', which may also be the default case. LVI-NULLify places an `lfence` instruction in the affected case, thereby mitigating it. As the deciding variable may depend on more than one memory load, *we consider this variant only partially mitigated.*

Variant ❺, direct load. All data loads are still susceptible to direct '0' injection with our mitigation. Thus, an attacker can use LVI-NULL for data-only attacks, e.g., as shown for AES-NI [15]. Since exploitability highly depends on the victim algorithm, mitigation is left to the enclave developer. Cryptographic libraries need to consider different side channels in their implementation already. Variant ❺ becomes one more issue on this list. We therefore consider it *out of scope* for LVI-NULLify.

Variant ❻, indirect load. When values are loaded indirectly, *i.e.*, by loading the target address from memory, arbitrary values can be injected when the first load is zeroed. Similar to ❷, values are loaded from address 0x0 with a possible offset. With LVI-NULLify, the now GS-relative load ensures that this load is inside the enclave's address space. If the offset falls within the non-readable pages at the beginning of the enclave, arbitrary value loading is prevented. Again, offsets are dependent on the program, and most can be statically determined at compile time, which allows adjusting the number of buffer pages accordingly in the compiler. Dynamic arrays of unknown size may still produce transient loads that reach into the enclave memory itself. In these cases, non-zero data injection may still occur. Because most cases are prevented, and the remaining rely on very specific circumstances, *we consider this variant mostly mitigated.*

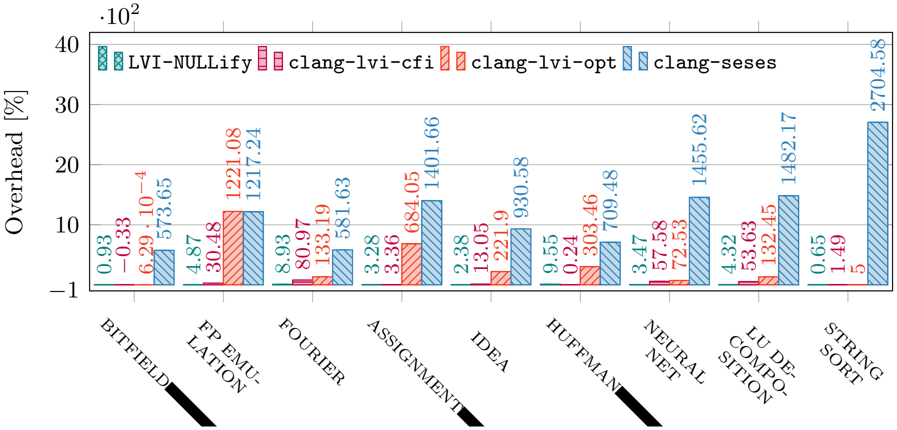


Figure 9.4: Mean runtime overhead in sgx-nbench [57] on our i5-10210U@1.6GHz of LVI-NULLify vs. Intel's control-flow and optimized-cut mitigations as well as SESES. N=50, standard deviations vs unmitigated mean plotted, but too small to be visible.

All told, our analysis suggests that LVI-NULLify prevents the majority of LVI-NULL variants and cases at a significantly lower performance impact than Intel's optimized-cut solution, not to mention SESES. Additionally to this reasoning, we also evaluated our claims with proof-of-concept implementations of the attack variants. Where our proofs of concept were successful without LVI-NULLify, enabling it prevents leakage in all cases.

We have no indication whether the discussed remaining vulnerabilities occur in real-world code. However, C and C++ grant developers vast freedom to implement features in non-standard ways (e.g. manual jump tables for ❷ that our compiler extension is unaware of) which we would not catch and, thus, not mitigate. Therefore, our mitigation bridges the gap between the very expensive optimized-cut mitigation and the less secure control-flow mitigation. When enclaves are not subject to one of the described caveats, our mitigation provides the same level of security as the optimized-cut mitigation at much lower performance cost.

6.2 Performance Evaluation

For the performance evaluation, we first investigate the number of emitted instructions, *i.e.*, the number of `lfences` and GS-relative loads, of LVI-NULLify and Intel's control-flow and optimized-cut mitigations as well as SESES(cf. Section 4.5 and Section 9.2). Our expection is that the number of `lfence` instructions has a direct and significant impact on the performance while GS-relative loads provide better performance. We substantiate this by benchmarking SGX applications with all of the above mentioned mitigations.

In line with previous work [18, 30, 31, 50, 55, 56], we evaluate the performance of our mitigation based on SGX benchmarks written in C/C++, the *nbench* adaptation for SGX [50, 57] and *SGXBENCH*[44]. As our mitigation is highly specialized for the SGX environment, we can only benchmark SGX enclaves, preventing us from measuring the compiler-introduced overhead for regular benchmarks, such as the SPEC benchmarking suite.

In our setup, we build enclaves with clang 11 and optimization level *O3*. As the optimized-cut mitigation by default does not mitigate the enclave entry assembly, it was compiled with the experimental mitigation for assembly to provide a better comparison with our mitigation in SGXBENCH, which measures enclave-entry performance. We evaluate on an Intel Core i5-10210U (1.6GHz, Comet Lake) and an Intel Core i9-9900K (3.5GHz, Coffee Lake-R). While the i9-9900K is also affected by LVI, it serves as a reference for workstation performance, compared to the mobile Comet Lake. Moreover, this CPU has also been used by Phoronix [11] to benchmark the overhead of Intel's LVI mitigations. We provide these results in Section 9.1. All experiments were run on isolated cores with fixed frequencies to reduce the variance of the measured values.

6.2.1 Analysis of Emitted Instructions

Table 9.2 shows the result for our evaluation of emitted instructions for the two benchmarks as well as three libraries that are essential components of SGX. The SESES mitigation issues the largest amount of `lfence`s, *i.e.*, more than 29 700 for libsgx_tstdc.a, which is to be expected as it simply fences every memory read and write that it encounters. Intel's optimized-cut mitigation improves upon this by removing more than 23 000 `lfence`s. The control-flow mitigation further reduces this number, down to 1 400 `lfence`s, but at the cost of reduced security as it does not mitigate all loads. None of these three mitigations issue a significant number of GS-relative loads, *i.e.*, 6 at most. Contrary to the other mitigations, LVI-NULLify issues the lowest amount of `lfence` instructions but the highest amount of GS-relative loads. This change in behavior significantly improves the performance, as our subsequent performance evaluation of the benchmarks shows.

Naturally, due to LVI-NULLify replacing certain instructions with a longer sequence of secure instructions (cf. Figure 9.3), we expect the binaries that LVI-NULLify generates to be larger than for the Intel mitigations. As Table 9.2 shows, this is indeed true: in the worst case, we see an increase of 21.5% over the unmitigated baseline.

6.2.2 nbench

The relative performance overhead shown in Figure 9.4 clearly demonstrates that the strong LVI-NULL mitigation provided by LVI-NULLify comes in at or even below the cost of Intel's control-flow-mitigation, which only covers variants ❷ and ❸. Table 9.3 contains the benchmark's raw results in iterations per second. We also see that some of the tests, like `String Sort` and `Bitfield`, operate almost entirely on registers, s.t. the overheads do not represent the differences of the mitigations very well. Memory heavier benchmarks like `FP Emulation`, on the other hand, clearly demonstrate the advantage of our mitigation vs. Intel's optimized-cut mitigation. Here we achieve an overhead reduction of 1216 percentage points. As this overhead is more in line with the original results by Van Bulck et al. [15] and Phoronix [11], we consider this to better represent the difference between the mitigations. We also note some benchmarks where LVI-NULLify performs better than the unmitigated reference. We consider this an artifact of cache alignment or similar effects specific to this benchmark and not representative of our mitigation.

6.2.3 SGXBENCH

When compiling the SGXBENCH[44] suite, we found that some loads in the benchmarks are not fenced by Intel's optimized-cut mitigation. For benchmarks that copy memory, this makes the comparison to our mitigation rather uninteresting, as tests show very similar performance. The results for a selection of benchmarks are listed in Section 9.1. Two of the benchmarks still provide a useful comparison, *einit/edestroy* and *empty ocall*. They show that at $\approx 0.17\%$ and $\approx 3.3\%$ lower performance, respectively, our mitigation does not introduce any significant slowdown for this basic enclave functionality.

Software	LVI-NULLify	control-flow	optimized cut	SESES
	LFENCE / GS / KB	LFENCE / GS / KB	LFENCE / GS / KB	LFENCE / GS / KB
nbench	37 / 11433 / 224(+19%)	319 / 6 / 192(+2%)	3289 / 6 / 200(+7%)	13780 / 6 / 233(+24%)
sgxbench	55 / 4323 / 113(+22%)	231 / 6 / 93(+0%)	1274 / 6 / 97(+4%)	5229 / 6 / 109(+18%)
libsgx_trts.a	4 / 1483 / 109(+10%)	105 / 6 / 102(+3%)	591 / 6 / 104(+5%)	1872 / 6 / 108(+9%)
libsgx_tstdc.a	0 / 23356 / 1322(+3%)	1400 / 0 / 1367(+7%)	6188 / 0 / 1383(+8%)	29754 / 0 / 1454(+13%)
libsgx_tcxx.a	1 / 14916 / 799(+10%)	812 / 0 / 722(-1%)	3353 / 0 / 730(+0%)	17818 / 0 / 775(+6%)

Table 9.2: We show the number of `lfence` and GS-relative instructions the different mitigation techniques insert and the overall file size in kB (and its change to baseline) for a selection of software, including benchmarks and SGX components.

Test/Mitigation	none (σ)	LVI-NULLify (σ)	control-flow (σ)	optimized cut (σ)	SESES (σ)
NUMERIC SORT	723.69 (0.181)	718.67 (0.093)	722.15 (0.061)	317.23 (0.018)	100.73 (0.008)
STRING SORT	70.46 (0.003)	70.01 (0.005)	69.43 (0.003)	67.11 (0.002)	2.51 (0.000)
BITFIELD	316 550 164 (165 589)	313 635 987 (102 467)	317 587 729 (228 815)	316 548 172 (411 084)	46 990 509 (1 326)
FP EMULATION	30.17 (0.002)	28.77 (0.009)	23.12 (0.002)	2.28 (0.000)	2.29 (0.000)
FOURIER	23 851.98 (7.853)	21 896.10 (12.773)	13 180.42 (3.819)	10 228.70 (2.137)	3 499.27 (0.086)
ASSIGNMENT	41.72 (0.013)	40.39 (0.004)	40.36 (0.003)	5.32 (0.000)	2.78 (0.000)
IDEA	7 257.17 (0.529)	7 088.14 (0.759)	6 419.30 (20.861)	2 254.47 (0.311)	704.18 (0.060)
HUFFMAN	2 335.15 (0.314)	2 131.65 (1.249)	2 329.53 (0.474)	578.78 (0.061)	288.48 (0.012)
NEURAL NET	66.20 (0.027)	63.98 (0.056)	42.01 (0.024)	38.37 (0.004)	4.26 (0.000)
LU DECOMP	1 467.54 (0.780)	1 406.82 (0.351)	955.22 (0.434)	631.33 (0.129)	92.75 (0.003)

Table 9.3: Average performance in sgx-nbench [57] on i5-10210U@1.6GHz of our GS mitigation vs. Intel's control-flow and optimized-cut mitigations as well as SESES. Clang 11 was used for all tests. Iterations/s, higher is better. N=50

7 Discussion and Limitations

Hardware and Microcode Changes. Ultimately, LVI and LVI-NULL have to be mitigated in silicon, as we can already see from CPUs that are not affected by any LVI variant. However, as it is infeasible to replace all affected CPUs, an intermediate solution compatible with affected CPUs is necessary. Van Bulck et al. [15] suggested the possibility of a microcode update that simply marks the null page uncachable. However, we identified several problems with this approach.

First, transient loads from an uncachable page can pick up values from the line-fill buffer [27, 42]. With hyperthreading enabled, clearing the line-fill buffer on enclave entry and exit is then also not sufficient.

Second, we experimentally verified that the operating system can change the memory type of enclave pages. Hence, a malicious operating system could change the memory type of the null page to cachable. Only if there is a method to lock entries in the TLB, SGX could ensure that the TLB entry for the null page stays in the TLB, preventing the operating system from changing the memory type.

Hence, we conclude that microcode mitigations are not as simple as assumed. The fact that there is no microcode update for any CPU to prevent LVI-NULL also indicates that microcode mitigations might not be possible.

Limitations. While LVI-NULLify conceptually prevents most variants of LVI-NULL, our technical implementation is currently limited by a few factors. Some of these limitations can be solved using additional engineering effort, while others can be solved directly by the enclave developer.

Most limitations are due to our proof-of-concept compiler transformation pass. The transformation pass currently uses a machine function pass to apply LVI-NULLify. However, as assembly is not handled by this machine function pass, we currently cannot directly patch inline assembly or assembly files automatically.

The remaining limitations are due to the pointer conversion between enclave and host application. While all the cases where the enclave developer adheres to best practice and the strict interface definitions are supported, there are corner cases that cannot be supported in an automated way, e.g., if the pointer is hidden behind an unknown type and reinterpreted by the developer.

8 Conclusion

In this paper, we presented a novel, lightweight defense against LVI-NULL in SGX. Based on a systematic analysis of LVI-NULL variants, we identified the attack requirements and discovered that previous mitigations targeting LVI-NULL are not effective. Our mitigation, LVI-NULLify, addresses this problem by repurposing segmentation to offset every load during enclave execution. LVI-NULLify consists of a modified SGX-SDK and a compiler extension that we open source. We evaluated LVI-NULLify on LVI-fixed CPUs and observed a performance overhead below 10% for the worst case, which is substantially lower than previous defenses. We conclude that LVI-NULLify is a practical solution to protect SGX enclaves on processors that remain susceptible to LVI-NULL.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Fangfei Liu, for their comments and suggestions. We also want to thank Aikata Aikata for her support with hardware procurement. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 681402). Additional funding was provided by generous gifts from Intel, Amazon and ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Zola Bridges. LLVM SESES pass for LVI. 2020. URL: `https://reviews.llvm.org/D75939` (p. 225).

[2] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. The Evolution of Transient-Execution Attacks. In: GLSVLSI. 2020 (p. 215).

[3] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In: GLSVLSI. 2020 (p. 215).

[4] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 213, 216).

[5] 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. 217).

[6] Intel. Affected Processors: Transient Execution Attacks. 2020. URL: `https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model` (pp. 213, 216).

[7] Intel. An Optimized Mitigation Approach for Load Value Injection. 2020. URL: `https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injection` (p. 225).

[8] Intel. Load Value Injection. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/load-value-injection.html (pp. 213, 216, 223, 225).

[9] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In: USENIX Security Symposium. 2020 (p. 219).

[10] Michael Larabel. Google Engineer Shows "SESES" For Mitigating LVI + Side-Channel Attacks. 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=LLVM-SESES-Mitigating-LVI-More (p. 225).

[11] Michael Larabel. The Brutal Performance Impact From Mitigating The LVI Vulnerability. 2020. URL: https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf (pp. 213, 216, 225, 238, 239).

[12] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In: USENIX Security Symposium. 2020 (pp. 215, 226).

[13] 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 (p. 219).

[14] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020 (p. 226).

[15] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (pp. 213–216, 220, 222–226, 235, 236, 239, 240).

[16] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In: NDSS. 2020 (p. 215).

[17] Wenjie Xiong and Jakub Szefer. Survey of Transient Execution Attacks. In: arXiv:2005.13435 (2020) (p. 215).

[18] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX. In: EuroSys. 2020 (p. 238).

[19] 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. 215).

[20] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. Extended classification tree and PoCs at https://transient.fail/. 2019 (pp. 213, 215, 219, 222).

[21] 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 (pp. 217, 219).

[22] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. 2019 (p. 217).

[23] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 2019 (p. 230).

[24] 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. 212, 213, 215, 219).

[25] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In: AsianHOST. 2019 (p. 219).

[26] 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. 217, 218, 226).

[27] 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. 212, 215, 217, 218, 226, 240).

[28] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 215).

[29] 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. 219).

[30] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In: RAID. 2019 (p. 238).

[31] 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. 238).

[32] Adam Chester. Exploiting Windows 10 Kernel Drivers - NULL Pointer Dereference. 2018 (p. 223).

[33] 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. 217).

[34] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (p. 219).

[35] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In: ESSoS. 2018 (p. 219).

[36] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 213, 215).

[37] Intel. Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088. 2018. URL: `https://software.intel.com/security-software-guidance/advisory-guidance/branch-target-injection` (p. 230).

[38] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. 2018 (p. 217).

[39] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (p. 224).

[40] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (pp. 213, 215).

[41] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (p. 213).

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 212, 215, 218, 240).

[43] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 213, 215).

[44] Raul Quinonez. SGXBENCH framework for benchmarking SGX enclaves. 2018. URL: `https://github.com/sgxbench/sgxbench` (pp. 238, 239).

[45] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS. 2018 (p. 219).

[46] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (p. 215).

[47] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 217, 218).

[48] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 217).

[49] 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. 217, 219).

[50] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer. 2017 (p. 238).

[51] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (p. 219).

[52] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. 2017 (p. 217).

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (pp. 217, 219).

[54] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 217, 219).

[55] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In: NDSS. 2017 (p. 238).

[56] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In: NDSS. 2017 (p. 238).

[57] utds3lab. Adaptation of nbench-byte-2.2.3 for Intel SGX. 2017. URL: `https://github.com/utds3lab/sgx-nbench` (pp. 237, 238, 240, 249).

[58] 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. 217, 219, 251).

[59] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security Symposium. 2017 (p. 217).

[60] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (pp. 216, 217, 233).

[61] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference. Rev 1.5. May 2016 (p. 234).

[62] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 2016 (p. 230).

[63] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In: ESORICS. 2016 (p. 219).

[64] David Drysdale. How programs get run: ELF binaries. 2015. URL: `https://lwn.net/Articles/631631/` (p. 218).

[65] Hans Wennborg. The recent switch lowering improvements. Oct. 2015. URL: `http://llvm.org/devmtg/2015-10/slides/Wennborg-SwitchLowering.pdf` (p. 230).

[66] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (pp. 217, 219).

[67] 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 (p. 219).

[68] Daniel Pierre Bovet. Special sections in Linux binaries. Jan. 2013. URL: `https://lwn.net/Articles/531148/` (p. 218).

[69] Anton Korobeynikov. Improving Switch Lowering for The LLVM Compiler System. In: SYRCoSE. May 2007 (p. 230).

[70] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS. 2007 (p. 220).

[71] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: IEEE / ACM International Symposium on Code Generation and Optimization – CGO. 2004 (p. 230).

[72] Nergal. The advanced return-into-lib(c) explits: PaX case study. 2001 (p. 220).

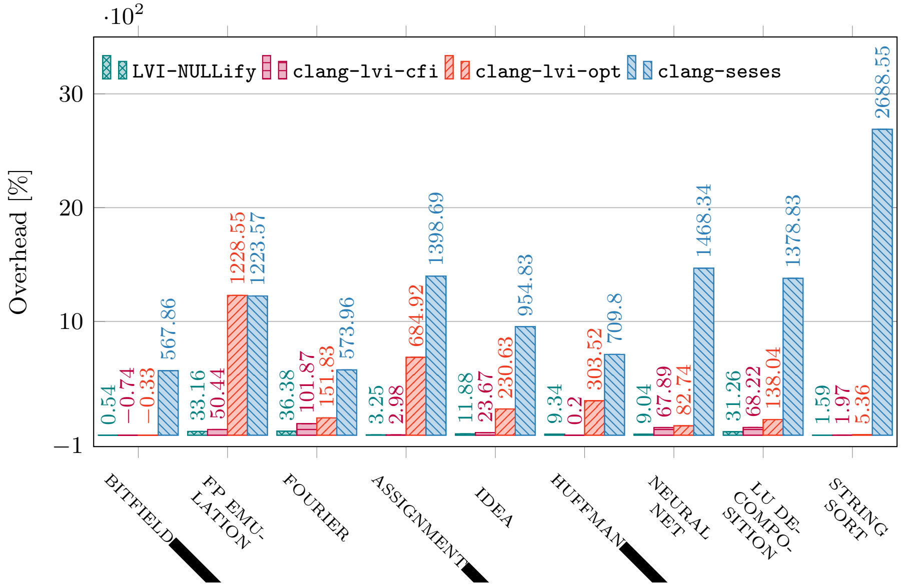


Figure 9.5: Mean performance overhead in sgx-nbench [57] on our i9-9900K@3.5GHz of LVI-NULLify vs. Intel's control-flow and optimized-cut mitigations as well as SESES. Clang 11 was used for all tests. N=50, standard deviations w.r.t. baseline mean are plotted, but too small to be visible.

9 Appendix

9.1 Benchmarking results

In addition to our LVI-NULL-only affected Comet Lake CPU, we also provide benchmark overheads for the older i9-9900K Coffee Lake CPU in Figure 9.5. We can see that while their are some differences, the relative

Test/Mitigation	**none** (σ)	**LVI-NULLify** (σ)	**control-flow** (σ)	**optimized cut** (σ)	**SESES** (σ)
empty function	37072.3 (2655.0)	37660.3 (1878.0)	37349.9 (1832.2)	39015.5 (1701.7)	42612.8 (3325.2)
empty ocall	14496.6 (1151.5)	14980.8 (1096.3)	14735.5 (1146.6)	16052.1 (1080.8)	15995.6 (995.6)
ocall in/out	15651.8 (835.7)	16433.0 (809.6)	16236.2 (702.1)	17510.5 (819.4)	23309.8 (1023.8)
encrypted read	14884.4 (756.5)	15228.5 (843.2)	14984.3 (758.9)	16429.6 (799.8)	21868.8 (966.6)
encrypted write	14720.8 (799.7)	15279.9 (827.8)	14989.7 (740.8)	16507.6 (779.2)	21866.2 (967.0)
einit/edestroy	141845200.9 (793624.2)	142087389.7 (849561.9)	142354110.9 (841980.8)	142506399.6 (798824.8)	142649039.2 (827876.1)

Table 9.4: Runtime of the SGXBENCH benchmarks on an i5-10210U@1.6GHz in cycles. Lower is better. N=1000000 for all except eint/edestroy where N=1000

performances between the mitigations is roughly the same on this desktop CPU as it is on the mobile i5-10210U.

Table 9.4 shows the execution times for various SGXBENCH benchmarks on our Comet Lake i5-10210U.

9.2 Sample Compilation Options for Mitigations

Control-flow Mitigation:

```
clang-lvi-cfi -mlvi-cfi -Iclang-lvi-cfi/sgxsdk/include
-Iclang-lvi-cfi/sgxsdk/include/tlibc -fpic -O3 -nostdinc
-fvisibility=hidden -fstack-protector -fpic -c Enclave.c
-o Enclave.o
```

SESES Mitigation:

```
clang-lvi-seses -mseses -Iclang-lvi-seses/sgxsdk/include
-Iclang-lvi-seses/sgxsdk/include/tlibc -fpic -O3 -nostdinc
-fvisibility=hidden -fstack-protector -fpic -c Enclave.c
-o Enclave.o
```

Optimized-Cut Mitigation:

```
clang-lvi-opt -mlvi-hardening
-mllvm -x86-lvi-load-opt-plugin=OptimizeCut.so
-mllvm -x86-experimental-lvi-inline-asm-hardening
-Iclang-lvi-opt/sgxsdk/include
-Iclang-lvi-opt/sgxsdk/include/tlibc -fpic -O3 -nostdinc
-fvisibility=hidden -fstack-protector -fpic
-c Enclave.c -o Enclave.o
```

9.3 LVI-NULL POC Implementation Details

In addition to LVI-NULLify, the relevant proofs of concept can also be found in our repository at `https://github.com/IAIK/LVI-NULLify/`.

For attacks on SGX, an attacker would typically use a framework like SGX-Step [58] to interfere with a victim enclave at more or less precise points. For our POCs however, we can use a more cooperative approach, which simplifies the code and imitates a very strong attacker. Right before vulnerable loads in our victim, we OCALL to the attacker who then removes the *accessed* bit from our target page. This reliably causes 0 to be injected into the next loads from this page, triggering our LVI-NULL attacks. We can then measure rates of leakage via a transmission gadget; in our case an access to a page outside the enclave.

When we compile with LVI-NULLify, we see that all leakage is completely prevented.

9.4 Artifact Appendix

9.4.1 Abstract

The public repository[2] contains all the code necessary to reproduce the data for all performance graphs/tables in the paper, as well as PoCs to demonstrate that the mitigation works. This includes patches and build instructions for LLVM11, the Intel SGX SDK and PSW, as well as the benchmarks. The artifact requires SGX to evaluate, and is easiest to run on Ubuntu 18.04 or 20.04.

9.4.2 Artifact check-list (meta-information)

- **Program:** Adapted versions of `nbench` and `sgxbench` are downloaded & installed via included scripts.
- **Compilation:** Requires a modified Clang 11, install & download script is included.
- **Transformations:** A tool to fix up relocations is included (relocator).

[2] `https://github.com/IAIK/LVI-NULLify/`

- **Run-time environment:** Needs a native Linux installation that supports SGX, Ubuntu 18.04 or 20.04 are strongly recommended. Build scripts need internet access at several points. Requires root for installation and evaluation. PoCs require the PTEditor kernel module.
- **Hardware:** Intel CPU with SGX support, needs to be vulnerable to LVI-Null for PoC tests (affected CPUs).

  The PoCs need a kernel module, which means either self-signing or disabling secure boot. This may require physical access to the machine.
- **Run-time state:** As this artifact includes performance benchmarks, a stable CPU frequency and isolated cores are recommended.
- **Execution:** For ideal testing, the system should have isolated cores, fixed frequency, and not much other activity.
- **Metrics:** Benchmarks report cycle count or iterations/s, PoCs report leakage percentage.
- **Output:** Benchmark outputs are .csv tables with performance, an included spreadsheet can convert to a graph similar to the paper.
- **Experiments:** Installation scripts are included and described here and in READMEs.
- **How much disk space required (approximately)?:** 4-5GB
- **How much time is needed to prepare workflow (approximately)?:** 2-3h
- **How much time is needed to complete experiments (approximately)?:** 3-6h, depends on hardware
- **Publicly available?:** `https://github.com/IAIK/LVI-NULLify/`
- **Code licenses (if publicly available)?:** zlib

9.4.3 Description

How to access Clone `https://github.com/IAIK/LVI-NULLify/tree/ae_final` and follow the README.md from there.

Hardware dependencies As this is a mitigation for Intel SGX, SGX support is a hard requirement. To fully evaluate the PoCs, and not just mitigation performance, the CPU also needs to be vulnerable to LVI. You can check if your CPU is vulnerable here: `https://software.intel.com/content/www/us/en/develop/topics/software-security-guidance/processors-affected-consolidated-product-cpu-model.html`

Software dependencies We strongly recommend Ubuntu 18.04 or 20.04 as these are officialy supported by Intel, and all our tools were tested on them.

Beyond standard compilation tools (ninja, cmake etc) our PoCs require the PTEditor kernel module [3]. Other requirements are listed in the README files at the appropriate points.

9.4.4 Installation

Follow the detailed README in the top-level directory to set up our modified clang compiler and relocator and install the SGX driver as well as our modified SGX SDK and PSW.

Once that is done, you can already test your installation with the PoCs by following the README file in the POC directory.

With a working PSW and driver, you can follow the README in the benchmarks directory to download and build the benchmarks.

9.4.5 Experiment workflow

After building the benchmarks, follow along in the README to start all or a subset of them. An important aspect to keeping benchmarks comparable is to fix the CPU's frequency to a sustainable level, and idealy run them on an isolated core.

PoCs can be run according to the README in the POC folder.

[3] `https://github.com/misc0110/PTEditor/`

9.4.6 Evaluation and expected results

The main results in our paper are contained in Figure 4/Table 3. These are the performance overheads of our LVI-NUll mitigation compared to other, similar mitigations. The second, more implicit result is the efficacy of LVI-NULLify.

For the benchmarks, the absolute performance overheads vary significantly between different machines and architectures (compare Figure 4 and Figure 5), but the relative differences should be roughly similar. That is: LVI-Nullify should be the fastest mitigation, or at least very close to Intel CFI, typically followed, with some distance, by Intel's optimized-cut mitigation.

For the PoCs, starting once without and once with mitigation should produce qualitatively similar results to the examples shown in the README. That means, for the 3 PoCs where LVI-Nullify is effective, leakage rate should drop to zero, or a level that is comparable to the noise-catching output ”other”. While absolute leakage rates before applying the mitigation may differ significantly from system to system, they should be clearly differentiable from ”other”.

The respective READMEs for benchmarks and PoCs detail how to reproduce these results.

9.4.7 Experiment customization

Attack PoCs need a cache miss threshold, which is automatically determined. If this doesn't work, it can be set manually in the corresponding *App.cpp* file. All PoCs include a conf.h file, in which the character that should be leaked can be changed if desired.

Both benchmark run-scripts contain a variable called ”isolated_core” that sets the core on which they should be run on. Set this to an isolated core, if available.

sgx-nbench contains a parameter to change the number of iterations in the file, see the benchmarking README.

9.4.8 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`

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.