



Carina Fiedler, BSc

**I want a Demotion:
A Systematic Evaluation of Cache
Side-Channel Attacks**

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Advisors

Fabian Rauscher

Daniel Gruss

Institute of Applied Information Processing and Communications

Graz, December 2024

AFFIDAVIT

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

Date, Signature

Acknowledgements

I would like to thank my supervisors Fabian Rauscher and Daniel Gruss for providing valuable feedback and guidance.

I am thankful for the financial support from both the German Ministry of Education and Research as well as the IAIK Information Security Scholarship allowing me to pursue my studies.

Thanks to my colleagues and friends for interesting discussions about the topic, the nature of science and the hidden mysteries of Latex.

Special mention goes to my dog Cookie for upholding office morale and ensuring steady supply of serotonin.



Abstract

The increasing computing demands of modern systems have prompted hardware optimizations that inadvertently make bug-free programs vulnerable. Cache timing side channels have gained more traction in light of speculative execution attacks in the recent decade. Techniques like Flush+Reload and Prime+Probe enable adversaries to spy on a victim’s behavior or stealthily transmit data across security boundaries. The recent introduction of the `cldemote` instruction into the x86 architecture provides the opportunity to implement faster SMT attacks.

In this thesis, we present three novel attacks based on the `cldemote` instruction: the SMT attacks Demote+Reload and Demote+Demote are a generic drop-in replacement for Flush+Reload type attacks modeled after Flush+Reload and Flush+Flush. The cross-core attack DemoteContention works on CPUs with non-inclusive LLC and does not require shared memory or physical addresses. We systematically evaluate our novel attacks alongside the existing techniques Flush+Reload, Flush+Flush, Evict+Reload and Prime+Probe. While previous papers have reported individual metrics such as covert channel capacities for their presented attacks, these measurements are highly system-dependent. We perform standardized measurements for all tested attacks on a Xeon Silver 4410T CPU and a Xeon Silver 4514Y CPU, which support the novel `cldemote` instruction. We measure nine metrics: topological constraint, spatial precision, hit-miss margin, attack time, temporal precision, blind spot, covert channel capacity, noise resilience and detectability. Our novel attacks rank at the top in most metrics, with our optimized Demote+Reload covert channel surpassing all channel capacities of existing SMT attack techniques by at least 69 %, achieving a true capacity of 15.5 - 17.0 Mbit/s. Demote+Demote and Flush+Flush outperform the other tested attacks in several metrics: For both attacks, we measure a blind spot length of 0 CPU cycles. Demote+Demote demonstrates the fastest attack time without a victim access at 138 - 186 cycles, while we measure the fastest attack time with a victim access for Flush+Flush at 198 - 265 cycles, followed by Demote+Demote at 217 - 289 cycles. DemoteContention is the only attack that does not induce a detectable increase in performance counters.

Keywords: Demote+Reload · Cache Attacks · Timing Side Channel · Systemization

Kurzfassung

Die steigenden Rechenanforderungen moderner Systeme haben zu Hardware-Optimierungen geführt, die fehlerfreie Programme angreifbar machen. Cache-Timing-Seitenkanäle haben in der letzten Dekade angesichts von spekulativen Attacks zunehmend an Bedeutung gewonnen. Techniken wie Flush+Reload und Prime+Probe ermöglichen es Angreifern, das Verhalten eines Opfers auszuspähen oder heimlich Daten über Sicherheitsgrenzen hinweg zu übertragen. Die kürzlich erfolgte Einführung der `cldemote`-Instruktion in die x86-Architektur bietet die Möglichkeit, schnellere SMT-Angriffe zu implementieren.

In dieser Arbeit stellen wir drei neuartige Angriffe vor, die auf der `cldemote`-Instruktion basieren: Die SMT-Angriffe Demote+Reload und Demote+Demote sind ein generischer Drop-in-Ersatz für Flush+Reload Angriffe, modelliert in Anlehnung an Flush+Reload und Flush+Flush. Der kernübergreifende Angriff DemoteContention funktioniert auf CPUs mit nicht inklusivem LLC und benötigt keinen gemeinsamen Speicher oder physikalische Adressen. Wir bewerten unsere neuartigen Angriffe systematisch neben den bestehenden Techniken Flush+Reload, Flush+Flush, Evict+Reload und Prime+Probe. In früheren Arbeiten wurden zwar einzelne Metriken wie z. B. Verdeckter-Kanal-Kapazitäten für die vorgestellten Angriffe angegeben, diese Messungen sind jedoch stark systemabhängig. Wir führen standardisierte Messungen für alle getesteten Angriffe auf einer Xeon Silver 4410T CPU und einer Xeon Silver 4514Y CPU durch, die die neue `cldemote`-Instruktion unterstützen. Wir messen neun Metriken: topologische Einschränkung, räumliche Präzision, Hit-Miss-Marge, Angriffszeit, zeitliche Präzision, blinder Fleck, Verdeckter-Kanal-Kapazität, Störungssignal-Resistenz und Detektierbarkeit. Unsere neuen Angriffe liegen in den meisten Metriken an der Spitze, wobei unser optimierter Demote+Reload-basierter verdeckter Kanal alle Kanalkapazitäten bestehender SMT-Angriffstechniken um mindestens 69% übertrifft und eine echte Kapazität von 15,5 bis 17,0 Mbit/s erreicht. Demote+Demote und Flush+Flush übertreffen die anderen getesteten Angriffe in mehreren Metriken: Für beide Angriffe messen wir eine Blindspot-Länge von 0 CPU-Zyklen. Demote+Demote weist die schnellste Angriffszeit ohne Victim Access mit 138 - 186 Zyklen auf, während wir die schnellste Angriffszeit mit Victim Access für Flush+Flush mit 198 - 265 Zyklen messen, gefolgt von Demote+Demote mit 217 - 289 Zyklen. DemoteContention ist der einzige Angriff, der keinen erkennbaren Anstieg der Performance Counter zur Folge hat.

Schlagwörter: Demote+Reload · Cache-Attacken · Timing-Seitenkanal · Systemisierung

Contents

1	Introduction	1
2	Background	4
2.1	CPU Caches	4
2.2	Cache Timing Side Channels	7
2.3	Side-Channel Mitigations	16
3	Demote-based Attacks	20
3.1	Demote+Reload	20
3.2	Demote+Demote	22
3.3	DemoteContention	23
4	Systemization of Cache Side-Channel Attack Techniques	26
4.1	Topological and Spatial Constraints	26
4.2	Temporal Metrics	28
4.3	Covert-Channel Metrics	48
4.4	Detectability	57
5	Discussion	61
6	Conclusion	63
	Bibliography	64

Chapter 1

Introduction

In the era of digitization, there is a plethora of applications that require fast and reliable computation. Computation-intensive processes such as artificial intelligence are employed in real-time systems. To optimize performance, efficient software algorithms are a necessary but insufficient component on their own. Hardware optimizations enabling caching, parallelization and speculative execution also play a central role in increasing code execution speed. Caching exploits the principles of spatial and temporal locality to keep data, that is likely needed in the future, readily available [58]. In certain situations, it is necessary to write these locally modified copies back to memory, e.g. to propagate changes to failure-protected memory [34] or to other threads running on a different CPU core. User-level programs can do this via the unprivileged x86 instruction `clflush`, which invalidates a cache line at every level of the cache hierarchy, including caches of other cores. Recently, Intel introduced a new instruction with a similar functionality. On typical processor designs, the `cldemote` instruction moves a cache line from same-core L1 and L2 caches to the LLC. Compared to `clflush`, `cldemote` enables data propagation to threads running on different cores through the shared L3 cache while avoiding the performance penalty associated with slower subsequent RAM accesses [29].

However, novel performance improvements of modern processors often conflict with security interests. The ability to observe the microarchitectural state from software opens up a wide range of attacks on software that is verified and secure at the architectural level [85, 42]. One method of observing the microarchitectural state is via timing attacks. Instruction and data caches facilitate faster access to recently used memory addresses, which can lead to information leakage about a program execution via the generated cache state. Specifically, a memory access can prompt loading a cache line into the cache. An attacker can infer the presence of a cache line via several techniques based on timing memory accesses [85, 51, 22] or measuring the execution time of certain x86 instructions, e.g. `clflush` [21]. The leakage rate of Flush+Reload is constrained by the time-consuming RAM access of the attacker.

In this thesis, we present three novel side-channel attacks based on the `cldemote` instruction: the SMT attacks Demote+Reload and Demote+Demote as well as the cross-core attack DemoteContention. Our systematic evaluation runs novel and existing attacks in a standardized setup on two CPUs supporting the instruction, a Sapphire Rapids

Xeon Silver 4410T and an Emerald Rapids Xeon Silver 4514Y. We evaluate nine metrics: topological constraint, spatial precision, hit-miss margin, attack time, temporal precision, blind spot, covert channel capacity, noise resilience and detectability.

Our experiments show that Flush+Reload is a reliable technique with the largest hit-miss margin (124 - 232 cycles), but its time-consuming attacker RAM access (225 - 292 cycles) leads to the highest attack time (295 - 713 cycles), the longest detection delay (76 - 133 ns), the largest absolute blind spot (344 - 552 cycles) and a low covert channel capacity (1.9 - 2.4 Mbit/s). Compared to Flush+Reload, the faster attacker L3 access (76 - 97 cycles) of our novel Demote+Reload attack enables a 30 % faster attack time without a victim access, a 60 % smaller absolute blind spot and a 2.7 - 3 times higher SMT covert channel capacity. We demonstrate the fastest optimized SMT covert channel with Demote+Reload (15.5 - 17.0 Mbit/s). Since the attack loops of Flush+Flush and Demote+Demote do not induce any memory accesses, they have the fastest attack times (138 - 289 cycles), high temporal precision (34 - 45 ns detection delay with 15 - 19 ns SD), do not miss victim accesses and consequently, have high-performing covert channels (8.3 - 10.3 Mbit/s). Hence, we demonstrate that our novel attacks significantly improve existing cache timing attacks.

Furthermore, our evaluation reveals that Evict+Reload has a very small hit-miss margin (4 - 10 cycles), shows the largest standard deviation of the detection delay (39 - 41 ns) and is the only attack in which the attack times with and without a victim access are comparable (333 - 399 cycles). We show that Evict+Reload and Prime+Probe demonstrate small blind spot lengths (53 - 84 cycles). Since DemoteContention relies on temporary contention, it has the largest blind spot relative to attack time (90 %). Due to the minuscule difference between the hit-miss distributions, the channel is very susceptible to noise and has the lowest capacity of 0.2 Mbit/s. We determined that Flush+Reload and Flush+Flush are the only tested techniques that work both in SMT threads and cross-core without significant adaptations. We note that Prime+Probe and DemoteContention do not require shared memory but have lower spatial granularity than a cache line. Our evaluation reveals deeper insights into novel and existing cache attacks to aid the understanding of the strengths and limitations of individual techniques.

We make the following main contributions:

- We introduce the novel SMT attacks Demote+Reload and Demote+Demote as generic drop-in replacements for Flush+Reload.
- With DemoteContention, we present a cross-core attack on LLC. In contrast to previous techniques, DemoteContention does not require shared memory nor reverse-engineering of unknown hardware structures.
- We systemize Flush+Reload, Flush+Flush, Evict+Reload, Prime+Probe and our novel attacks according to nine metrics: topological constraint, spatial precision, hit-miss margin, attack time, temporal precision, blind spot, covert channel capacity, noise resilience and detectability.

Chapter 1 Introduction

- We demonstrate the fastest published CPU covert channel with our optimized Demote+Reload attack.

Outline. Chapter 2 provides background knowledge on CPU cache architectures, cache timing side channels and mitigations. Chapter 3 proposes our novel same-core cache timing side-channel techniques Demote+Reload and Demote+Demote and cross-core DemoteContention. Chapter 4 details an extensive systemization of selected existing side channel techniques and the new attacks. Chapter 5 discusses the implications of our results. Chapter 6 concludes.

Chapter 2

Background

2.1 CPU Caches

In this section, we present fundamental knowledge concerning the functionality of caches. We explain the common concepts of the cache hierarchy, central cache parameters and cache addressing.

2.1.1 Cache Architecture

The CPU performs operations on data stored in memory. CPU processing power increased drastically, but there are physical limits on memory access speed. While most instructions can be executed within a few CPU cycles [1], main memory accesses can consume several hundred CPU cycles. Hence, memory accesses constitute a bottleneck for the number of instructions that the CPU can execute within a given time. To solve this issue, caches have been introduced. A small amount of data is stored in smaller and faster memory closer to the CPU. Modern chips have a hierarchy of several caches with increasing storage capacities and decreasing access speeds. Figure 2.1 depicts the cache hierarchy of an Xeon Silver 4410T processor. At the L1 level, there are small and fast dedicated instruction and data caches per physical core, while the per-core L2 cache unifies instructions and data. In contrast, the L3 cache is shared among all physical cores. The data is split up into Last-Level Cache (LLC) slices that are interconnected as well as attached to one core each [29].

Data is dynamically loaded into the cache on the first access and stored for future accesses. This is sensible due to the principle of temporal locality: accessed data is likely to be needed again soon, for example, in a code loop that repeatedly updates the value of a variable. In addition, the principle of spacial locality is utilized: data close to the accessed data may also be accessed next, for example, during the iteration through an array (data access) or loading and executing source code (instruction access). Hence, the hardware does not load a single memory address but rather blocks of data called cache lines. The size of a cache line varies among processors with cache lines being typically 64 bytes wide on Intel processors [32]. Besides the data, a cache line contains metadata,

Chapter 2 Background

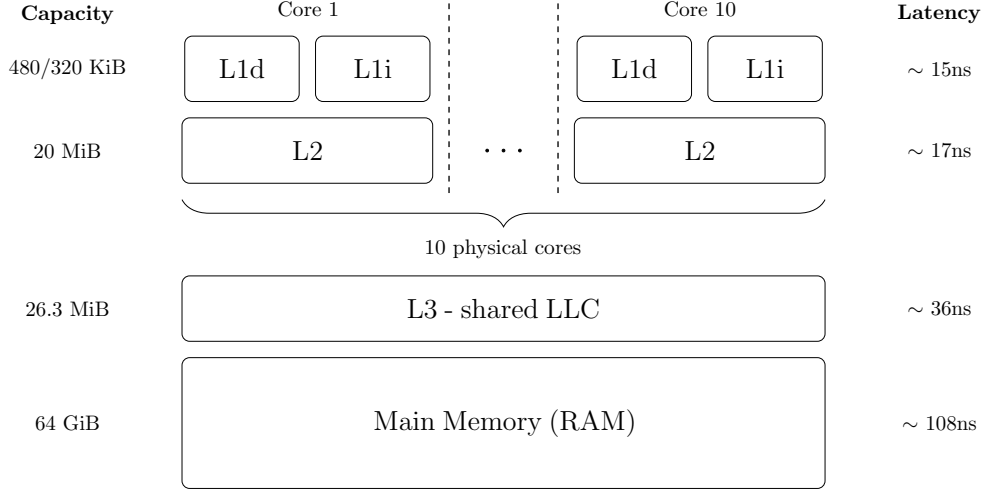


Figure 2.1: Cache Hierarchy of a modern processor (Xeon Silver 4410T). The last-level L3 cache is shared among all cores. L1 and L2 caches are individually per core. Cache capacity and access latency increase with the distance from the CPU.

such as a valid bit indicating that the data is present in the cache. A cache set can store a fixed number of cache lines, which is denoted as the associativity of the cache. How data is moved around in the cache hierarchy depends on a variety of different policies that are partially documented or reverse-engineered:

- **Eviction strategy:** Since the storage capacity of the caches is limited, bringing a new cache line into the cache can cause the eviction of a cache line resident in the same cache set. Different eviction policies determine which cache line is evicted. The Least Recently Used (LRU) strategy evicts the cache line accessed the longest time ago, following the principle of temporal locality. Since it is non-trivial to efficiently track all access times across different programs and cores, Intel processors employ pseudo-LRU algorithms [33, 1] to approximate LRU behavior.
- **Coherency:** When data is updated in a faster cache, the changes can be immediately written back to the slower caches (write-through). While this enforces consistency between caches and main memory, the write-back strategy provides better performance [32]: Data is marked dirty when modified since the last load and is written back to slower caches or main memory on eviction.
- **Inclusivity:** An inclusive cache contains all data residing in smaller caches closer to the CPU. This can reduce the access time of shared data that is not cached in the private caches of a different core. However, it limits the overall capacity of the cache hierarchy since an eviction of data from the slower inclusive cache also leads to eviction from the faster cache. With the recent Xeon Scalable processor family,

Intel switched to a non-inclusive LLC to utilize a larger L2 cache more efficiently [35].

2.1.2 Cache Addressing

To perform cache addressing, a memory address is split into three parts: a t -bit tag, an i -bit cache index and an o -bit offset. Cache data is organized in 2^i cache sets containing one or several cache lines. The cache index is used to select the corresponding cache set. Since the main memory is much larger than the cache by design, 2^t cache lines will be mapped to the same cache entry. To distinguish the different cache lines, their tag is stored alongside the data. Combined, the cache index and tag uniquely identify a cache line. The size and associativity of the cache influence the size of the tag and cache index. The offset is used to address individual bytes within a cache line. Hence, the size of a cache line determines the offset size [58].

In a directly mapped cache, each cache set can only store a single cache line. On a memory access, the i -th cache set is accessed and the tag is compared with the stored tag. On a match, a cache hit has occurred and the data is accessed at the desired offset. On a mismatch, a cache miss is issued and the data will be fetched from a cache further from the CPU or main memory. The data will eventually be loaded into the cache, evicting the cache line currently present at the cache index. If several addresses are mapped to the same cache index and accessed alternately, this setup leads to a high overhead since the addresses are evicting each other from the cache, leading to a cache miss on each access [58].

Hence, modern Intel processors employ set associative caches [29, 1]. An n -way cache stores n cache lines in each cache set. Figure 2.2 visualizes a 2-way set associative cache. On a memory access, the cache set is selected by the cache index. The tag is compared with each of the stored tags in the cache set. If any of them match and correspond to a valid entry, a cache hit occurs; otherwise, a cache miss occurs.

The size of the physical main memory is limited. To allow several processes to access the entire memory range without interfering with each other, virtual memory provides an idealized abstraction of main memory. Each process has a range of virtual addresses that are mapped to physical addresses, which are swapped between main memory and disk as needed. The memory address used to index the cache and calculate the tag can be a virtual or physical address. Typically, the low-latency L1 cache is virtually indexed and physically tagged (VIPT), while the L2 and L3 caches are physically indexed and physically tagged (PIPT). Virtual indexing is fast since it does not require the virtual address to be translated to its corresponding physical address first. Physical indexing and tagging ensure that shared memory is only present once in the cache. This is particularly useful for coherency in last-level caches that are shared among physical cores [58].

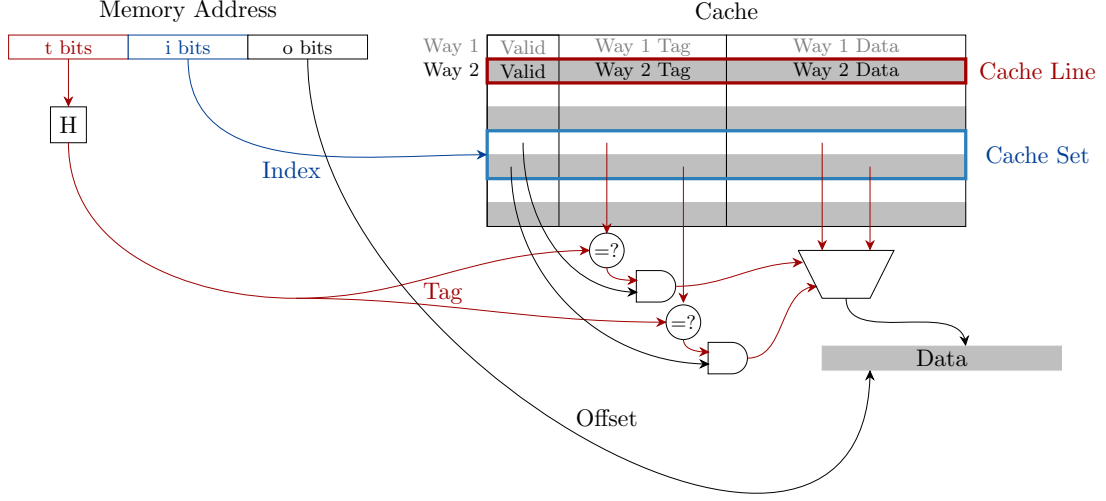


Figure 2.2: Memory access in a 2-way set associative cache. The index selects the cache set. If any of the tags in this cache set matches and the entry is valid, the corresponding data is accessed at the appropriate offset.

2.2 Cache Timing Side Channels

Side channels exploit signals that were originally not intended to transmit any information but rather appear as a side effect of another operation. Cryptographic implementations are the classical target of side channels [40]. In a key-recovery attack, the attacker observes the side effects of one or several encryptions and draws conclusions about the used key material from the recorded traces. Early timing side channels rely on key-dependent program execution time [43, 5]. These attacks are specific to one program and can be prevented by enforcing constant execution time. Microarchitectural side channels provide a different attack vector even in programs with constant execution time. In this thesis, we focus on generic cache timing side channels that can be measured in software.

By construction, there is a considerable timing difference between a memory access of cached versus uncached data. The access time can also be distinguished between different levels within the cache hierarchy. From an engineering perspective, this is a desired performance speed-up. From a security perspective, these side effects are particularly interesting if they are observable by another party (a potential attacker). The leakage of a vulnerable program is not in its execution time but rather in the cache state that a victim program creates. The attacker performs timing measurements to spy on the cache state.

```

1 res = 1
2 for i = 1 to key length:
3   res = square(r) mod n
4   if key_i == 1:
5     res = multiply(r, plaintext) mod n

```

Figure 2.3: Square and Multiply Algorithm for modular exponentiation. Sequence of `square` and `multiply` function calls depends on key bits.

2.2.1 High-level Scenario

We distinguish two scenarios. The first is a spy-victim scenario. A benign victim program is running normally and performing sensitive memory operations. An attacker can observe which cache lines or cache sets the victim has accessed. The following sections will outline several techniques to monitor victim accesses. With all techniques, the attacker actively runs a repeated attack loop to spy on the victim. Note that the attacker cannot observe the accessed data directly. However, knowing which instructions a victim has loaded and, hence, likely executed can be sufficient to leak sensitive information. A classical example is a naive square and multiply implementation of the modular exponentiation [2] in RSA (see Figure 2.3). Depending on the value of the i -th key bit, the intermediate value is either only squared or both squared and multiplied. If an attacker can observe the sequence of executions of the square and multiply function calls, they can reconstruct the key bits.

In the second scenario, two processes collaborate to secretly transmit data between each other. This communication channel is called a covert channel [47]. We consider side channels that communicate via memory operations. The communicating parties can run on two logical cores that share the same physical core via simultaneous multithreading (SMT). In a more powerful variant, the two processes run on different physical cores on the same machine. Hereafter, we refer to these setups as SMT and cross-core, respectively. This scenario could be used by an attacker to stealthily exfiltrate data across boundaries with no official communication interface, for example, cross-VM.

2.2.2 Flush+Reload

Yarom et al. [85] presented Flush+Reload, a cross-core timing side-channel technique targeting the L3 cache based on the attack by Gullasch et al. [23]. Flush+Reload uses the x86 instruction `clflush`, which invalidates a cache line from every level of the cache hierarchy, including caches of other cores [31].

First, the attacker executes the `clflush` instruction on a shared memory address to prepare the cache state. Second, the attacker performs a timed memory access on the

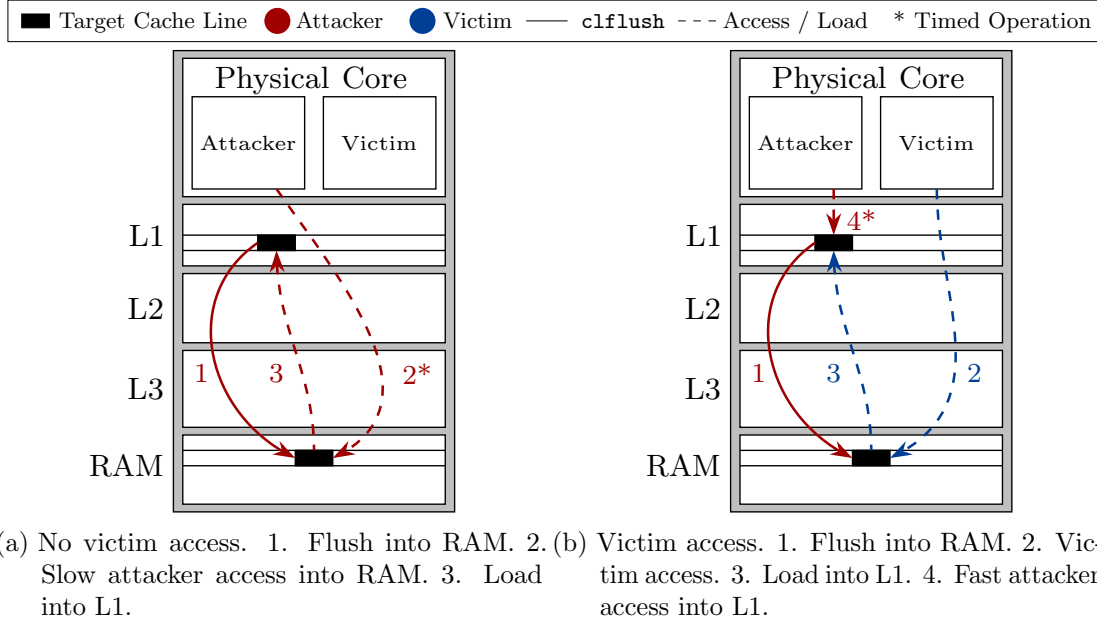


Figure 2.4: Working of Flush+Reload in the SMT case.

shared address. Flush+Reload exploits the timing differences between accessing cached versus uncached data to determine whether a victim access has occurred. If the access is fast, the data has been accessed before and is cached (victim access). If the access is slow, the data is not cached and must be fetched from the main memory (no victim access). However, the attacker's access can change the cache state - the accessed data is now cached. To reset the cache state before each measurement, the attacker repeatedly executes the two steps above.

Figure 2.4 visualizes the SMT case. Note that all actions on intermediate caches are omitted for simplicity, e.g. load into L2 during load into L1 cache. After a victim access, the attacker performs an L1 access. Figure 2.5 demonstrates the cross-core case. After a victim access, the attacker performs an L3 access.

2.2.3 Flush+Flush

Exploiting the same microarchitectural principles as Flush+Reload, Flush+Flush measures the execution time of the `clflush` instruction itself rather than measuring the time of a memory access [21]. The setup is identical to Flush+Reload with the exception that the attacker only performs a single `clflush` instruction in the attack loop to monitor victim accesses. If the execution of the `clflush` instruction is slow, the data is cached and needs to be flushed from the cache hierarchy (victim access). If the execution is fast, the data was not cached and no additional work needs to be performed by the `clflush`

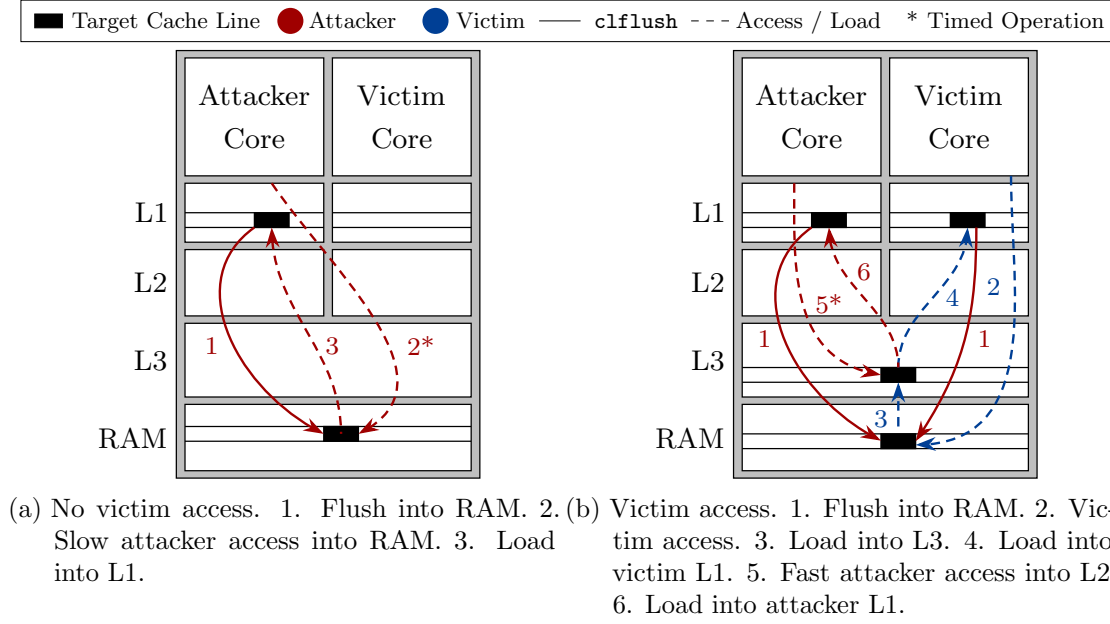


Figure 2.5: Working of Flush+Reload in the cross-core case.

instruction (no victim access). Figure 2.6 visualizes the SMT case. After a victim access, the attacker performs an L1 flush removing the data from its core’s L1 and L2 caches and the L3 cache. Figure 2.7 demonstrates the cross-core case. After a victim access, the attacker performs a cross-core L1 flush, removing the data from the victim’s core’s L1 and L2 caches and the shared L3 cache.

2.2.4 Evict+Reload

Evict+Reload follows the same working principle as Flush+Reload. However, Evict+Reload does not rely on an instruction to explicitly remove an address from the cache. Instead, addresses in the same cache set are accessed to evict the target address [22]. The set of cache lines that are brought into the cache to evict the target address is called an eviction set. For the SMT case visualized in Figure 2.8 it is sufficient to evict data from the L1 cache by filling up the L1 cache set. Since the L1 cache is typically virtually indexed, the addresses within the same L1 cache set can be trivially calculated by adding multiples of the virtual page size to the target address. After a potential victim access the access time of the target address is measured.

More precisely, the order of operations is as follows. First, the attacker accesses all addresses in the eviction set. The eviction set is now fully present in the L1 cache, while the target address resides in the L2 cache. Then, the attacker measures a memory access on the target address. A slow access indicates that the data is present in the L2 cache

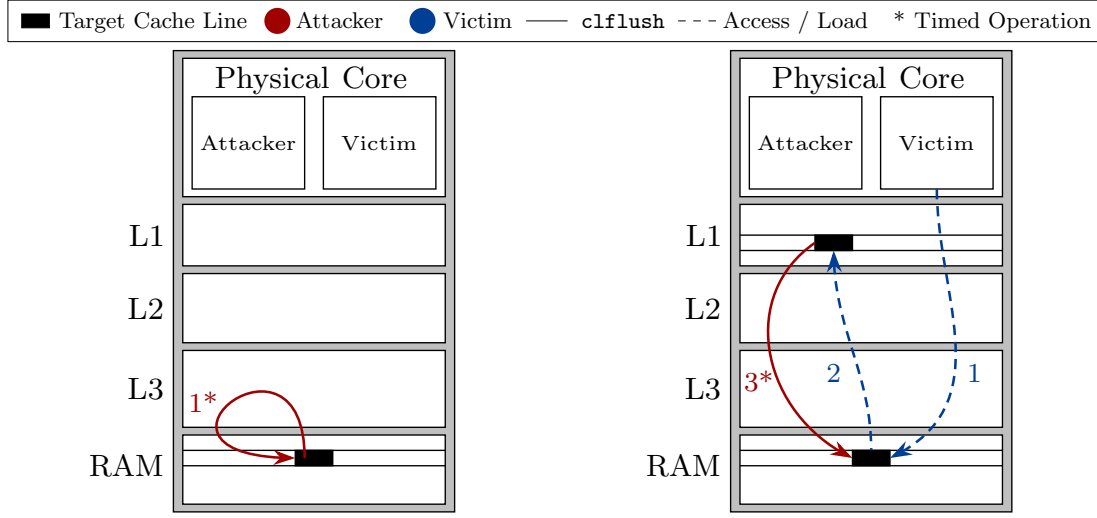


Figure 2.6: Working of Flush+Flush in the SMT case.

(no victim access). This case is depicted in Figure 2.8a. Upon access, the address is loaded into the L1 cache, evicting an address from the eviction set from the L1 cache. A fast access indicates that the data is present in the L1 cache (victim access). This case is depicted in Figure 2.8b. In contrast to the no victim access case, here the victim access already caused the load of the target address into the L1 cache before the attacker access.

The cross-core scenario is more complicated since the L1 and L2 caches are not shared. To target the L3 cache, the attacker needs to determine addresses that map to the same L3 slice and cache set (covered in Section 2.2.6). Moreover, the attack requires the L3 cache to be inclusive. The attacker can then evict the target address from the L3 cache, which leads to evictions in the victim core's L1 and L2 caches. Subsequently, a fast attacker access indicates that the data is present in the L3 cache (victim access). A slow access indicates that the data is not cached (no victim access). These scenarios are visualized in Figure 2.9.

2.2.5 Prime+Probe

Like Evict+Reload, Prime+Probe also utilizes an eviction set to evict the target address. However, instead of measuring the access time of the target address, the attacker times accesses on the eviction set cache lines [56, 51]. They can distinguish two cases. A fast combined time of all accesses on the eviction set indicates that no victim access has occurred in the meantime. Figure 2.10a depicts this case. Without a victim access, most

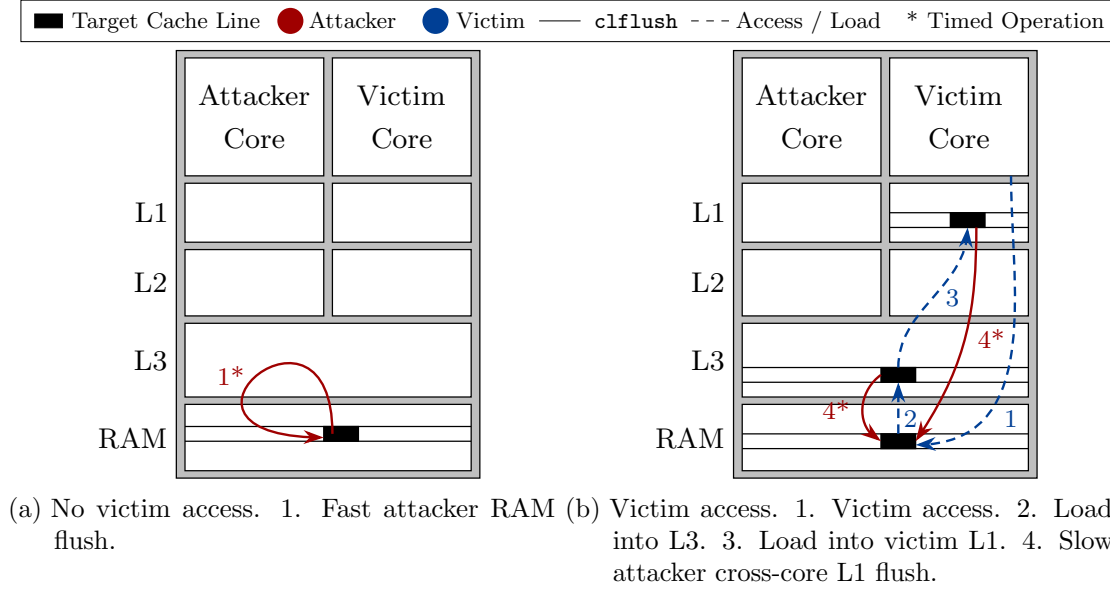


Figure 2.7: Working of Flush+Flush in the cross-core case.

or all of the eviction set cache lines are present in the L1 cache, resulting in a fast access time. A slow combined time of all accesses on the eviction set indicates the occurrence of a victim access. This case is depicted in Figure 2.10b. At the beginning, the whole eviction set is resident in the L1 cache. Upon a victim access, the target address is loaded into the L1 cache, evicting an eviction set cache line. When the attacker performs the measurement and accesses the eviction set cache lines, the evicted cache line is accessed in the L2 cache and loaded into the L1 cache. This can induce further evictions of other eviction addresses, as discussed below. Hence, the attacker's measurements contain one or more L2 accesses leading to a longer execution time than in the no-victim access case.

The timing difference between the two cases depends on how many eviction set cache lines are evicted from the L1 cache and need to be reloaded from the L2 cache in the next iteration. This depends on the timing of the victim access relative to the accesses on the eviction set, as well as the eviction strategy of the cache. Consider an LRU eviction policy and a victim access that occurs just after the previous iteration of accesses on the eviction set has finished. The victim access will cause a load of the target address into the L1 cache, evicting the least-recently used cache line of the eviction set. Since all eviction set cache lines are accessed in the same order in each iteration, this will be the first address in the sequence. When the attacker accesses and loads the first eviction set cache line from the L2 cache, it will, in turn, evict the least-recently used eviction set cache line, which is now the second address in the sequence. In this manner, all eviction set cache lines are evicted from the L1 cache and need to be loaded from the L2 cache, resulting in a considerably longer access time. However, if the victim access occurs during

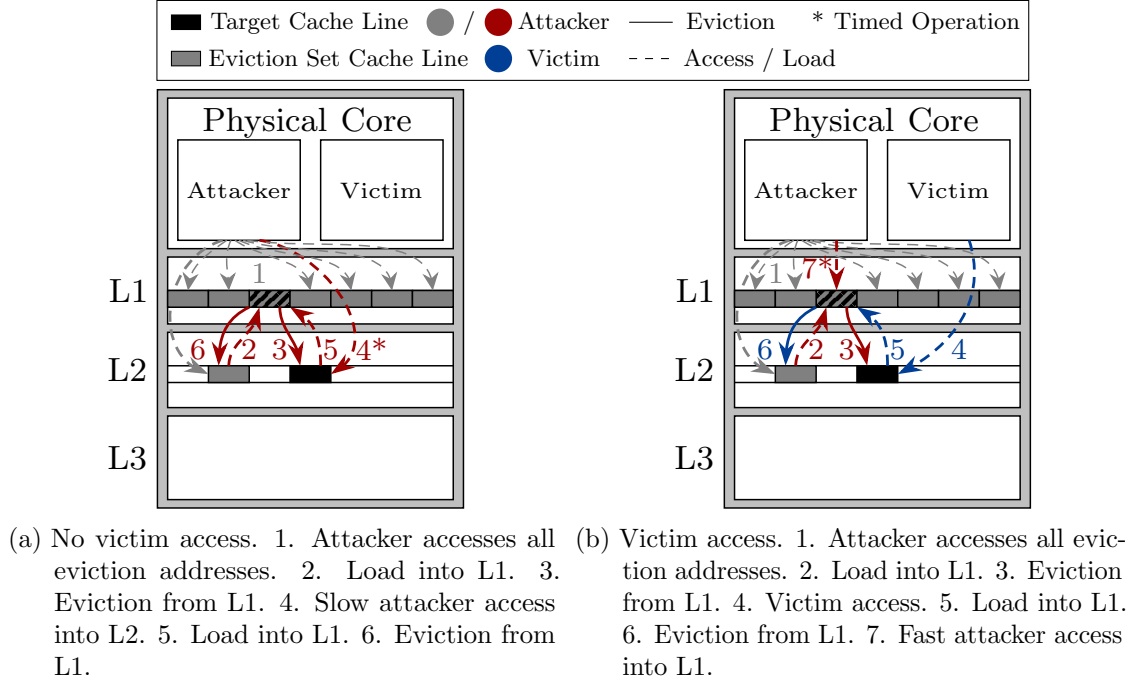


Figure 2.8: Working of L1 Evict+Reload.

the eviction set access sequence all concluded accesses of the current iteration have been L1 accesses. The LRU address is the one, which will be accessed next. The eviction of the LRU cache line causes the same domino effect of eviction set accesses evicting other cache lines of the eviction set as above. This affects the remainder of the eviction set cache lines that have not been accessed yet in the current iteration. Depending on the timing of the eviction address, we can have any number of L2 accesses up to the eviction set size in the measurement.

As for Evict+Reload, we require an inclusive L3 cache for the original cross-core attack [51] depicted in Figure 2.11. However, Yan et al. [84] demonstrated a cross-core Prime+Probe attack on the cache directory of a non-inclusive L3 cache. They reverse-engineered a set-associative inclusive directory structure in an LLC slice that contains entries for L2 and L3 cache lines. By filling up all ways of the directory structure, a cache line can be evicted from the victim’s private L2 and L1 caches [84].

2.2.6 Locating Addresses within the Same Cache Set

Evict+Reload and Prime+Probe require addresses that map to the same cache set. For the virtually indexed L1 cache, such addresses can typically be determined by adding multiples of the page size. For L3 cache sets, modern processors use complex cache indexing. The shared last-level cache is split into per-core slices, which are connected

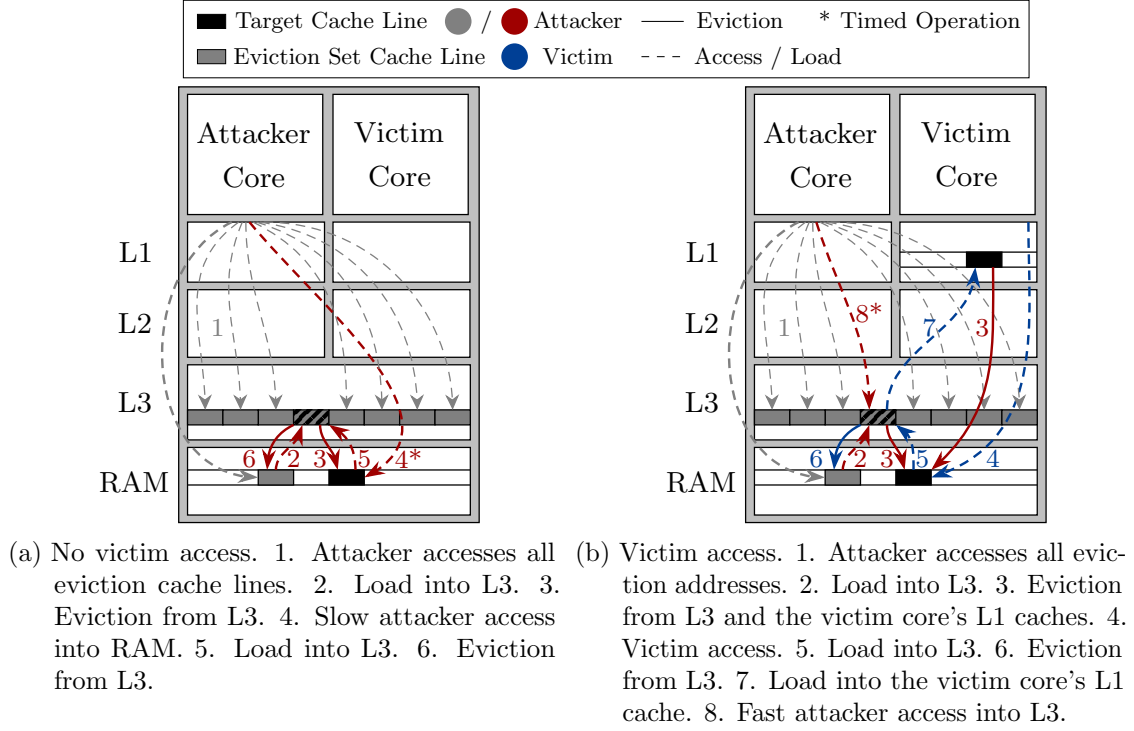


Figure 2.9: Working of cross-core Evict+Reload on an inclusive L3 cache.

to the L2 caches via a ring bus on Intel processors [29]. To locate two addresses in the same L3 cache set, we also need to determine which addresses are mapped to the same L3 cache slice. The mapping is done via an undocumented hash function.

Hund et al. [27] manually reverse-engineered the hash function of Sandy Bridge with four cores. However, Liu et al. [51] demonstrated that this hash function is not used on Sandy Bridge processors with a number of cores that is not a power of 2. Irazoqui et al. [37] proposed an automated tool to reverse engineer the slice hash function via Prime+Probe and recovered the slice selection algorithm of 6 platforms. Maurice et al. [53] presented an automated tool based on performance counters. For Xeon and Core processors with 2^n cores, the reverse-engineered hash function consists of linear xor functions of address bits. However, recovering a compact function via their method in the general case is NP-hard [53]. Gerlach et al. [16] introduced the first generic approach to recover non-linear slice selection algorithms.

2.2.7 Variants and Use Cases

Cache timing side channels were traditionally used to target cryptographic implementations. Most prominently, OpenSSL AES T-table implementations are vulnerable to

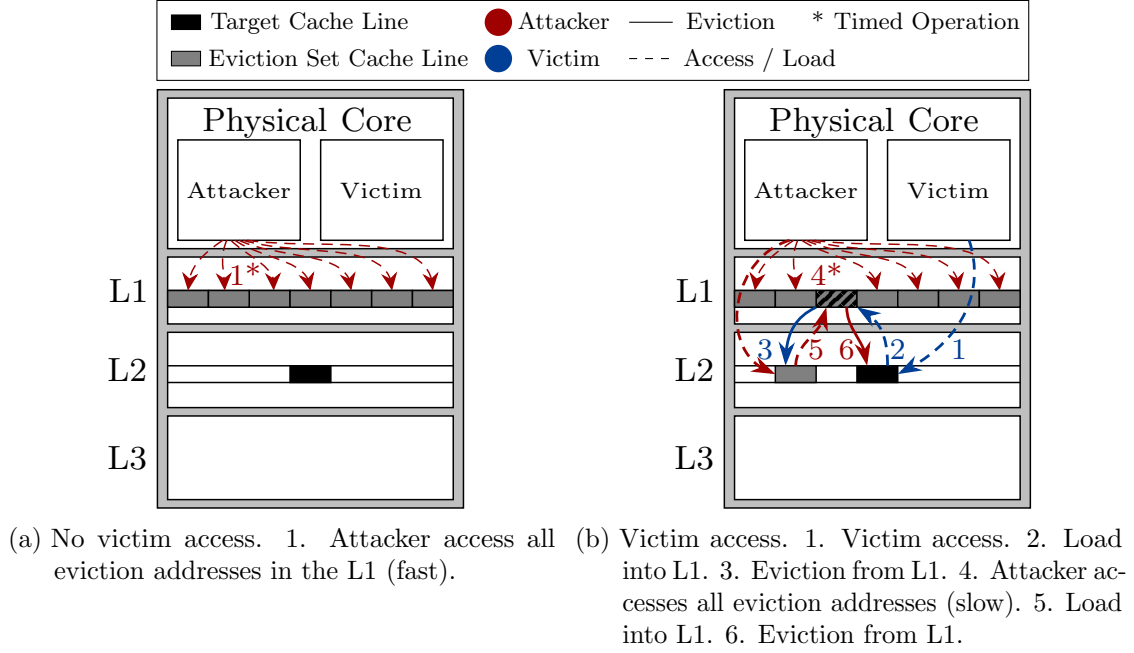


Figure 2.10: Working of L1 Prime+Probe.

key recovery attacks via cache timing side channels [56, 23, 38, 3]. With OpenSSL 3.0.2 reverting back to a non-constant-time implementation as the default for performance reasons [12], these attacks are not purely academic.

However, the ability to monitor cache state also enables attacks on a wider range of targets, including user behavior. Gruss et al. [22] presented Cache Template Attacks, which automatically detect cache side channels in a program to spy on keystrokes, window switches or encryption. Individual key presses result in data-dependent cache hits in the GDK library. Hence, an attacker can distinguish different (groups of) key presses by monitoring cache hits on profiled addresses [22]. Inter-keystroke timing attacks [74, 86] can also be used to recover user input. Cache timing side channels can be used to monitor addresses that are accessed on a keypress. In this manner, the attacker records a trace of inter-keystroke timings with high temporal precision [22]. Moreover, Oren et al. [55] demonstrated remote monitoring of further user behavior, such as network and mouse activity, via a Prime+Probe implementation in Javascript.

Furthermore, cache timing side channels can be utilized as building blocks in more complex attacks. For example, attackers can use cache timing side channels to bypass defense mechanisms such as KASLR [27, 20] or to leak temporary microarchitectural state in Spectre attacks [42]. However, cache side channels can also be used on the defensive side: Zhang et al. [88] utilized Prime+Probe to detect the co-location of other VMs, and Schwarz et al. [66] employed Flush+Reload to detect double-fetch bugs in the kernel.

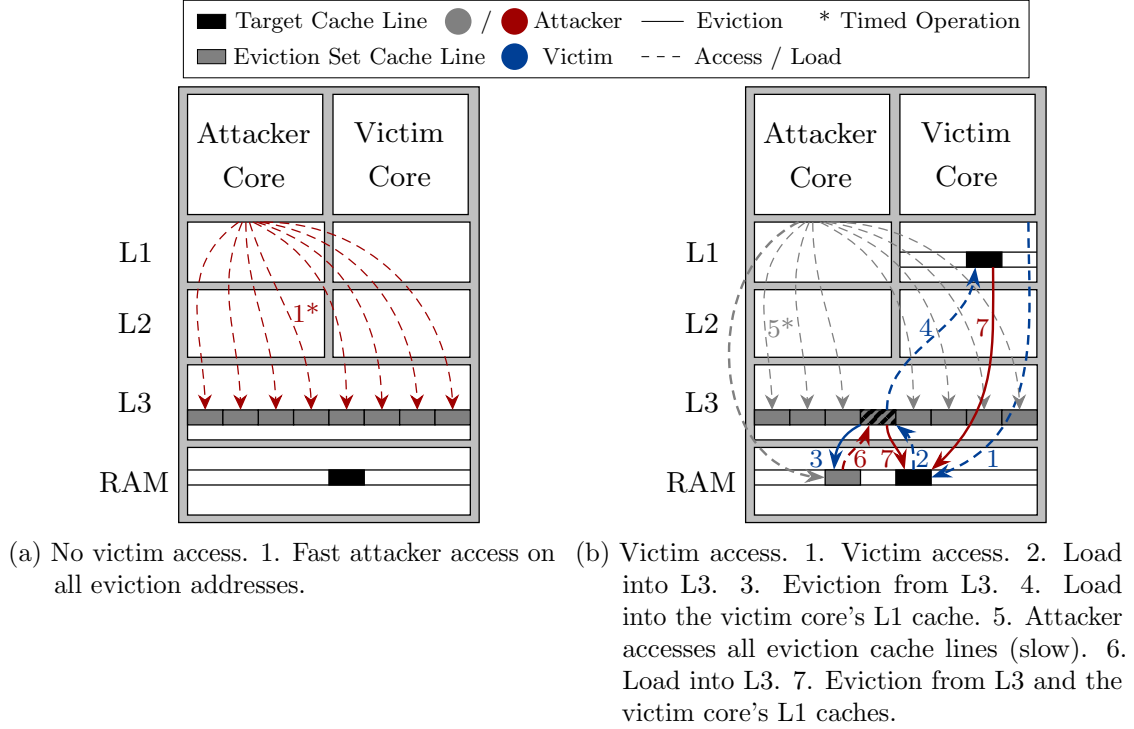


Figure 2.11: Working of cross-core Prime+Probe on an inclusive L3 cache.

The cache timing side-channels described in the previous sections were initially demonstrated on Intel x86 CPUs operating on the same physical core or cross-core on the same physical machine. Since then, numerous works have shown that cache-side channels are powerful attacks across security boundaries. Adaptations of the presented side channels have been demonstrated on non-rooted mobile devices [48], attacking secure enclaves from operating system [78] and from co-located enclaves [67], as well as targeting remote machines [55, 70, 46].

2.3 Side-Channel Mitigations

Cache timing side channels are powerful attacks against process isolation across virtualization boundaries and physical cores and even on remote machines. In this section, we give an overview of proposed mitigations at the hardware-, system- and application level. While hardware mitigations can fix the issues at the source, it takes considerably more effort and time to push hardware changes into consumer devices. System-level mitigations are easier to push out via operating system updates, but high overheads hinder adaption. Application-level mitigations on an individual program basis are error-prone and unlikely to offer complete protection.

2.3.1 Mitigating Core Requirements

The presented cache timing side channels rely on four components. This subsection discusses the feasibility of mitigating cache side channels in general by restricting access to these fundamental requirements.

High precision timers. Cache timing side channels exploit timing differences in the range of nanoseconds. Hence, the attacker needs to be able to monitor execution time with fine temporal granularity. To restrict access to high-precision timers, considerations include the complete removal of the `rdtsc` instruction [23], masking of the least significant bits [56], and adding a random offset [39]. Besides their disruptiveness to time-reliant programs, these mitigations are also insufficient since an attacker can implement a high precision software clock [60]. Without a hardware-supplied timer, numerous works utilize counting threads [83, 48, 69, 67]. In the absence of a timer, performance counters can also be used to monitor the cache state [77].

Martin et al. [52] proposed microarchitectural modifications to decrease the granularity of the `rdtsc` instruction and detect and mitigate software timing primitives. Vattikonda et al. [79] demonstrated the feasibility of fuzzing [26] the `rdtsc` instruction on Xen-virtualized x86 machines without disrupting legitimate processes. Kohlbrenner et al. [44] implemented time fuzzing in web browsers to mitigate remote side-channel attacks.

Execution latency. Cache timing side channels require one or several instructions whose variable execution time allows the attacker to distinguish between two states. Implementation-specific timing side channels on a program’s execution time have been combatted by constant-time requirements of cryptographic implementations. This approach cannot be generally applied to memory operations. Memory access timing differences are essential for performance. Requiring all memory accesses to take the time of a RAM access would defeat the purpose of the cache architecture. We can achieve de-facto constant-time access by locking sensitive data in the cache, as discussed, concerning the requirement of a shared resource below. To prevent sensitive cache state leakage, software developers must ensure secret-independent code and data access patterns [28]. Hence, the mitigations are implemented in software on a per-program basis. The call to make the `clflush` instruction constant time [21] has not been followed since it would only mitigate Flush+Flush.

Cache state restoration. The presented cache side channels rely on a mechanism to prepare the cache state for a measurement. Flush+Reload and Flush+Flush require the availability of the `clflush` instruction. Again, removing or restricting the instruction [85, 21] would be an insufficient mitigation for cache timing attacks in general since Evict+Reload and Prime+Probe only need the ability to perform memory accesses.

Shared Resource. Cache side channels operate on a shared resource. The specifics differ among the presented attack techniques. While Flush+Reload and Flush+Flush require shared memory to access the same cache line, Evict+Reload and Prime+Probe only require colocation on the same CPU to access the same cache set. To mitigate the

former class of attacks, Yarom et al. [85] proposed to remove shared memory. In specific cases, shared memory has indeed been limited at the kernel level. Memory deduplication refers to the merging of memory locations that contain the same content. Side-channel mitigations included disabling memory deduplication at runtime by default and restricting deduplication to the same security domain [10, 72]. However, it is difficult to distinguish malicious resource sharing from legitimate one and shared libraries continue to provide an attack vector [71].

2.3.2 Secure Caches

As a mitigation against Evict+Reload and Prime+Probe in the cloud, several works explore cacheline coloring implemented at the hypervisor level [63, 73, 41, 18, 9]. For security-sensitive operations, memory pages are mapped to isolated cache lines [73]. The isolation is achieved by assigning cache lines to different cache partitions corresponding to a security domain [18]. Cache lines within different cache partitions cannot evict each other. Kim et al. [41] presented a protection mechanism that locks a limited number of sensitive cache lines in the L3 cache, preventing eviction. In contrast to page coloring, this approach also isolates the cache lines from the root partition [41]. Undocumented microarchitectural optimizations reduce the effectiveness of cache partitioning and increase the difficulty of implementation [9]. Another kernel-level approach consists of repeated cache cleansing during context switches [89, 18]. Liu et al. [49] implemented a system-level cache partitioning mechanism based on Intel’s Cache Allocation Technology (CAT).

A possible mitigation at the hardware level is a partitioned cache [57]. In a PLcache, private partitions are created dynamically by locking individual cache lines in a cache [81]. In addition to this separation approach, Wang et al. [81] also proposed a random permutation cache (RPcache). On a cache miss, a random cache set is selected to store the newly loaded cache line. Hence, if an attacker’s cache line is evicted, the attacker only learns that the victim has accessed some address [81]. NewCache [80] follows a similar permutation approach, enables cache partitioning and cache line locking, and demonstrates increased performance compared to previous solutions.

The previous proposals differ significantly from conventional set associative cache architectures. An alternative approach retains the concept of cache sets but uses a keyed hash function to deterministically select a cache set based on a memory address [61, 62] and optionally on the process id [76]. ScatterCache builds on this idea, providing more entropy to remove the requirement for frequent re-keying [82]. Several works present fully associative randomized caches [65, 14]. ClepsydraCache [75] combines time-based evictions with cache index randomization. The randomized SassCache [17] utilizes a custom cryptographic function that spaces out the mapping of cache lines, resulting in statistical cache partitioning.

While the above designs mitigate the eviction set construction, there is also a proposal to mitigate the address reuse-based Flush+Reload and Flush+Flush attacks. Liu et al. [50]

presented a random fill cache architecture that does not load and store the demanded cache line in the cache on a cache miss.

2.3.3 Individual Mitigations and Detection

While it is difficult to mitigate cache side channels in their entirety, there are specific mitigations for individual use cases. Keydrown [68] mitigates keystroke timing attacks by injecting fake keystrokes in the kernel. Several works proposed mitigations against KASLR breaks and circumvented protection repeatedly [20, 15, 19, 6, 45].

Besides prevention mechanisms, there are also generic detection techniques. Flush+Reload generates many cache misses, which can be detected via hardware performance counters at runtime [25, 7, 87, 59]. Flush+Flush is much stealthier. Since it does not generate any cache misses itself, Gruss et al. [21] claimed that Flush+Flush was not detectable via performance counters. However, Flush+Flush does increase the number of victim accesses that result in cache misses. NIGHTs-WATCH [54] and FortuneTeller [24] leverage machine learning algorithms to detect an attack via recorded performance counters. Cho et al. [8] presented a real-time detection system.

Since cache side channels are difficult to mitigate entirely, the responsibility lies on the software developer to write side-channel resistant programs. Multiple works have focused on identifying cache side channels in binaries automatically via static analysis [13, 36] or symbolic execution [4]. Cache Template Attacks [22] allow developers to detect cache side channels in their programs for specific events such as keystrokes or encryptions automatically. Finally, also cache side channels themselves can be utilized to detect ongoing cache attacks [88, 22].

In conclusion, while the surveyed cache timing side-channel techniques are generic, effective mitigations either induce high overheads, involve a lengthy process to push out the changes to the consumer or are highly dependent on the use case and prone to be incomplete. As a result, the presented attacks are still possible on modern processors running up-to-date operating system versions.

Chapter 3

Demote-based Attacks

In Tremont and Sapphire Rapids processors, Intel introduced the unprivileged `cldemote` instruction [30]. This instruction demotes a cache line from a cache closer to the processor to a more distant cache, usually the shared last-level L3 cache. The `cldemote` instruction can be utilized to explicitly remove currently unneeded data to free up cache space. Furthermore, `cldemote` enables faster inter-core sharing of data. In particular, demoting written-to (dirty) data to the shared last-level cache can facilitate faster future accesses by other cores [31]. Similar to the `clflush` instruction, the use of the `cldemote` instruction exhibits timing behavior that can be exploited. In this section, we present the SMT attacks Demote+Reload and Demote+Demote as well as the cross-core attack Demote-Contention. We test all attacks on the two processors available to us that support the `cldemote` instruction: a Sapphire Rapids Xeon Silver 4410T CPU (SR) running Ubuntu 22.04 LTS, Linux 6.2.0 and an Emerald Rapids Xeon Silver 4514Y CPU (ER) running Ubuntu 24.04 LTS, Linux 6.8.0.

3.1 Demote+Reload

With the new `cldemote` instruction, we can build a Flush+Reload type generic attack for SMT. The attacker and the victim run on different logical cores on a shared physical core. The attacker can spy on the victim’s behavior by measuring the cache state. We demonstrate in Section 4.2.1 that an L3 access takes considerably more time than an L1 access (see Figure 4.1a). Hence, the attacker can distinguish if a cache line is resident in the L1 cache or only in the L3 cache by measuring memory access time. The attacker and victim need to share memory so that the attacker can access the memory and observe the load time.

Figure 3.1 demonstrates the workings of the attack in the standard attacker-victim scenario. The attacker can ensure that the data resides in the cache hierarchy at the beginning by performing an initial access on the address. The attacker code runs in a loop and works as follows: First, the attacker prepares the cache state by executing the `cldemote` instruction. The shared data consequently resides only in the L3 cache. Second, the victim potentially accesses the shared address, which, third, issues a load

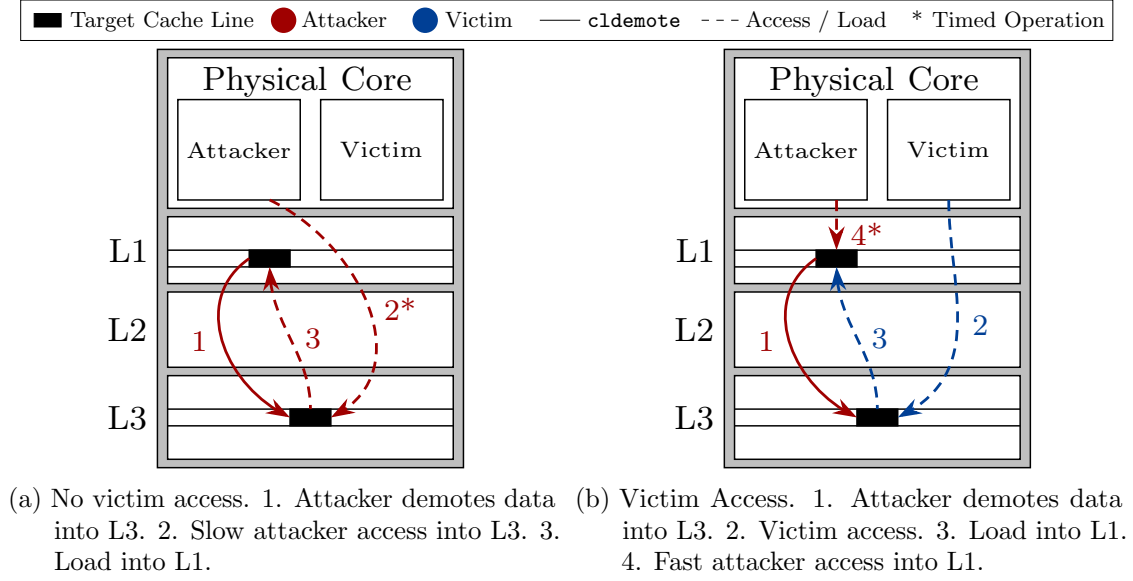


Figure 3.1: Working of the novel Demote+Reload attack in the SMT scenario.

into the L2 and L1 caches. Fourth, by measuring the access time of the data, the attacker can detect whether a victim access has occurred. If the access is fast, the attacker can conclude that the cache line was present in the L1 cache and, therefore, a victim access must have occurred. If the access is slow, the attacker concludes that the cache line was only present in the L3 cache and hence no victim access has occurred.

The main benefit of utilizing the `cldemote` instead of the `clflush` instruction lies in the reduced overhead in the no-victim-access case. The attacker only needs to access the L3 cache instead of the significantly slower main memory if no victim access has occurred. This leads to a shorter attack time in the no victim access case (see Figure 4.3a). For the covert-channel scenario, this implies that Demote+Reload can use a shorter transmission window per bit than Flush+Reload, therefore achieving a higher channel capacity than Flush+Reload (see Table 4.6).

Demote+Reload is not portable to the cross-core scenario as the L1 and L2 caches are not shared among CPU cores. Therefore, a potential victim access only loads the cache line from the shared L3 cache into the victim core's private L1 and L2 caches. Since the data does not reside in the attacker's L1 cache, a subsequent attacker access loads the cache line from the L3 cache. Hence, the attacker performs an L3 access regardless of whether a victim access has occurred. Therefore, the effect of a victim access is not observable by measuring access time on the attacker core.

3.2 Demote+Demote

Analogously to Flush+Flush, we can also construct an attack based on the execution time of the `cldemote` instruction. The attacker and victim run on different logical cores on a shared physical core. The attacker can spy on the victim’s behavior by monitoring the cache state. We demonstrate in Section 4.2.1 that the execution of the `cldemote` instruction on a cache line present in the L1 cache is considerably slower than on a cache line only present in the L3 cache (see Figure 4.2a). If the data resides in the L1 cache, the `cldemote` instruction needs to perform work by moving the data to the L3 cache. This leads to an increase in execution time compared to the case in which the instruction only needs to verify that the data is not present in the L1 and L2 caches. Hence, the attacker can distinguish if a cache line is resident in the L1 cache or only in the L3 cache by measuring the execution time of the `cldemote` instruction. The attacker and victim need to share memory so that the attacker can access the memory and observe the load time.

Figure 3.2 visualizes the workings of SMT Demote+Demote. The attacker can ensure that the data resides in the L3 cache at the beginning by performing an initial access and `cldemote` on the address. The attacker code runs in a loop and functions as follows: First, the victim potentially performs a memory access, which, second, issues a load into the L1 cache. Third, the attacker measures the execution time of the `cldemote` instruction. This simultaneously restores the cache state. If the execution is slow, the attacker concludes that the cache line was present in the L1 cache and, therefore, a victim access must have occurred. If the execution is fast, the attacker can conclude that the cache line was only present in the L3 cache and hence no victim access has occurred.

In contrast to the Flush+Reload vs. Demote+Reload comparison, the attack time of Demote+Demote does not benefit from the speed-up of an L3 access compared to a RAM access since the attacker does not perform any memory accesses. Demote+Demote still achieves a slightly faster attack time than Flush+Flush in the no-victim-access case (see Figure 4.3b) since `cldemote` on an L3 cache line is faster than `clflush` on data in RAM (see Figure 4.2a). In the covert-channel scenario, we benefit from the faster sender access into the L3 cache instead of a RAM access. Hence, Demote+Demote can use a shorter transmission window per bit than Flush+Flush, achieving a higher single-bit channel capacity than Flush+Flush (see Table 4.6).

In contrast to `clflush`, the `cldemote` instruction only demotes data from its own core’s private L1 and L2 caches and does not affect cross-core private caches. Hence, we don’t observe a difference in the execution time of the `cldemote` instruction on the attacker core when the cache line is present in the victim core’s private caches. Furthermore, we cannot use the `cldemote` instruction to remove the cache line from the victim core’s private caches. Hence, Demote+Demote is also not directly portable to the cross-core scenario.

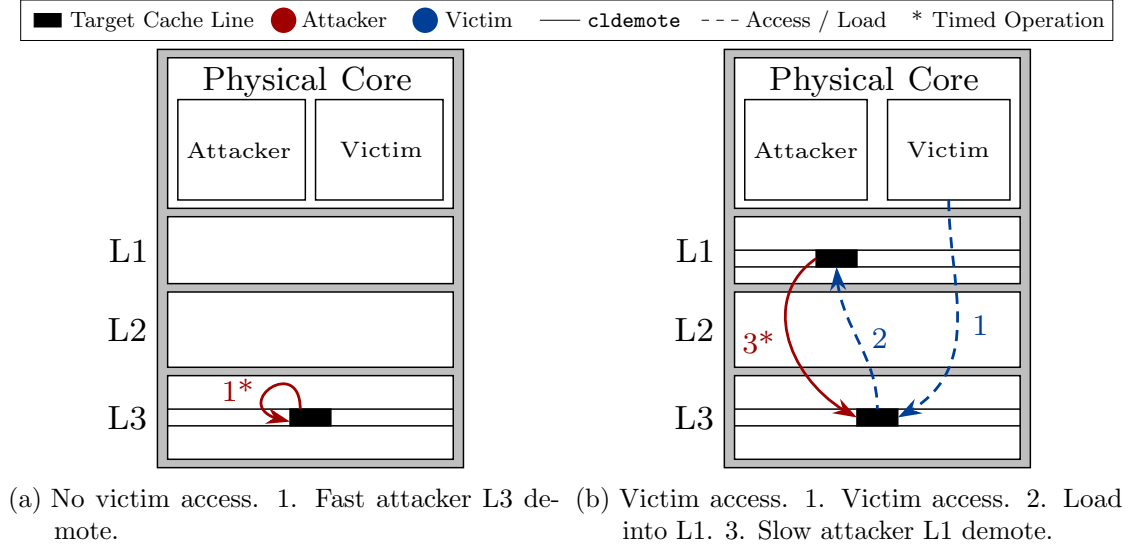


Figure 3.2: Working of the novel Demote+Demote attack.

3.3 DemoteContention

In this section, we construct a cross-core attack based on the `cldemote` instruction. For Demote+Reload and Demote+Demote, the only victim behavior that we assume is a memory access on a shared memory address. For our cross-core attack, we assume that the victim additionally performs a `cldemote` instruction. Again, if the victim executes the `cldemote` instruction, this does not demote the cache line from the attacker core's private caches. However, we can construct a different attack based on the execution time of the `cldemote` instruction without moving the data between the L1 and L3 cache of the attacker core.

Performing a victim access and `cldemote` directly before measuring the execution time of a second `cldemote` on the same L3 cache set results in an increase in the execution time of the `cldemote` instruction (see Figure 4.2b). We assume that this is likely due to contention in the L3 cache set or cache directory. We observe this effect even if the cache line is not present in the attacker core's private caches. Hence, despite no work to be done in terms of demoting the cache line, the `cldemote` instruction accesses the corresponding L3 cache set. This may be due to some performance benefit if data were to be written back. An access to the L3 cache set may be necessary, for example, to perform some coherency checks. It appears that the instruction is accessing the L3 cache set in parallel to the check whether the cache line is present in lower-level caches. These checks on the cache set do not appear to be parallelizable. If two `cldemote` requests on the same L3 cache set arrive within a short period, there is a slight delay in the request scheduled behind the first request.

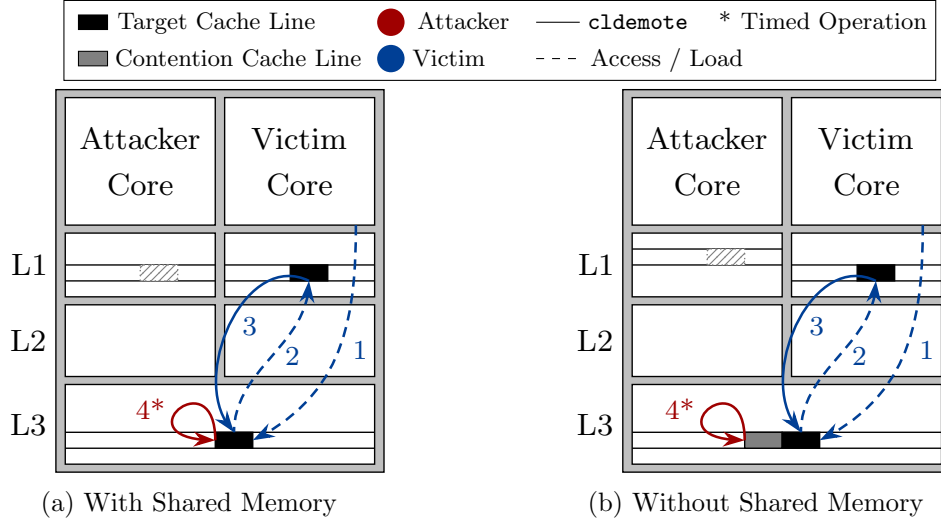


Figure 3.3: Working of the novel cross-core DemoteContention attack. 1. Victim access (optional). 2. Load into victim L1 (optional). 3. Victim demote into L3 (optional). 4. Timed attacker demote on same address (a) or address in same L3 cache set (b).

Hence, we can construct an attack in which the attacker repeatedly executes the `cldemote` instruction on the same address and measures each execution time. If there is a spike, the attacker can infer that the victim has performed a memory operation that reached the same L3 cache set. Figure 3.3 demonstrates the working of DemoteContention with and without shared memory. First, the victim optionally performs a memory access which, second, issues a load into the L1 cache. Third, the victim optionally demotes the cache line back into the L3 cache. Fourth, the attacker measures the execution time of the `cldemote` instruction. If the execution time is slow, the attacker can conclude that the victim has recently performed an access and the `cldemote` instruction on a cache line in the same L3 cache set. If the execution time is fast, it may be that the victim has not recently performed these operations or that the contention is not visible to the attacker. Either the victim operation has just concluded before or is scheduled after the measured attacker `cldemote`.

For Demote+Reload and Demote+Demote, the attacker actively moves cache lines around. A victim access performs a change to the cache state that persists and remains detectable until the next attack loop iteration. In contrast, for DemoteContention, the cache line always remains only in the shared L3 from the attacker's point of view. After a victim access and `cldemote`, the cache state is indifferent. The timing side effect is only observable within a specific time window while the victim operations are performed. Hence, victim and attacker operations need to be executed within a short amount of time from each other to observe a difference in the execution time of the `cldemote` instruction.

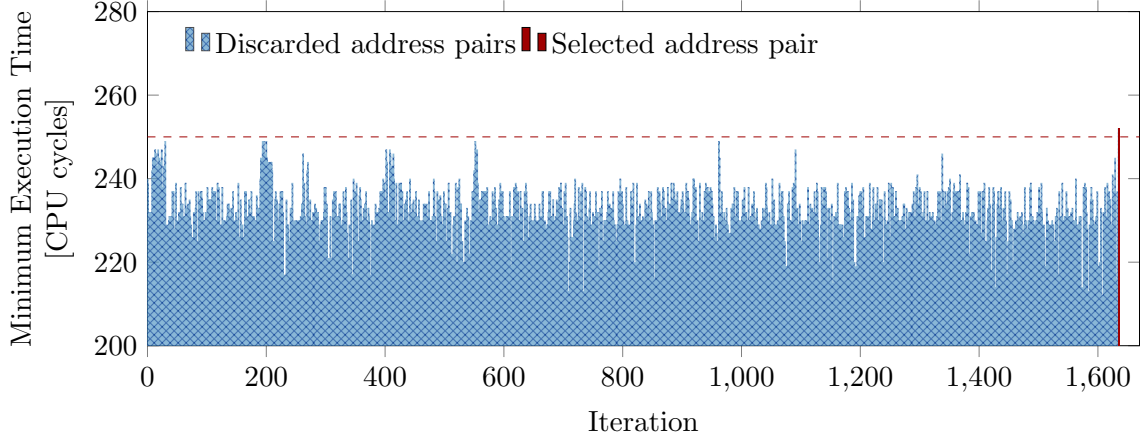


Figure 3.4: Selecting an address pair for DemoteContention. Pairs with execution time below the threshold are discarded. In each iteration, an address pair is randomly selected. The graph shows the minimum execution time among all measurements for each address pair.

Two addresses seem to affect each other’s `cldemote` timings precisely if they reside in the same L3 cache set. Since we do not have knowledge of the slice selection algorithm used on our tested CPUs, we locate suitable addresses for the covert channel via the same DemoteContention method we use for transmitting data. We randomly select two addresses and measure the execution time of subsequent memory accesses and `cldemote` executions on the two addresses. We perform 1024 such measurements for an address pair, making up one set of measurements. If the average of the measured execution time is below a certain threshold, we discard the pair and randomly select a new one. Otherwise, we perform up to 1024 such sets of measurements. If none of the sets has a below-threshold average, we have located an address pair suitable for our side channel. Figure 3.4 visualizes the minimum execution time among all supersets of time measurements per address pair.

Chapter 4

Systemization of Cache Side-Channel Attack Techniques

In Section 2.2, we described a variety of existing cache side-channel attacks from the literature. While the corresponding papers published some metrics, such as covert channel capacity, explicit results are system-dependent. Therefore, the published numbers from different papers are not directly comparable, and a general framework for evaluating attacks needs to be developed. In this chapter, we perform an extensive systemization of the existing attacks as well as the novel Demote+Reload, Demote+Demote and DemoteContention attacks that we presented in Chapter 3. We measure and analyze nine metrics: topological constraint, spatial precision, hit-miss margin, attack time, temporal precision, blind spot length, covert channel capacity, noise resilience and detectability. For standardization, we perform all experiments on the two processors available to us that support the `cldemote` instruction: a Sapphire Rapids Xeon Silver 4410T CPU (SR) running Ubuntu 22.04 LTS, Linux 6.2.0 and an Emerald Rapids Xeon Silver 4514Y CPU (ER) running Ubuntu 24.04 LTS, Linux 6.8.0. We focus our analysis on the original Flush+Reload, Flush+Flush, L1 Evict+Reload, L1 Prime+Probe techniques and our new attacks Demote+Reload, Demote+Demote and DemoteContention.

4.1 Topological and Spatial Constraints

In this section, we discuss whether the tested attack techniques require the attacker and victim to run on the same physical core or also work across physical cores. Cross-core attacks are more generic and powerful since we do not need to make assumptions about the cores on which the attacker and victim are being scheduled. We also note the spatial granularity with which an attacker can monitor a victim’s memory operations. Fine-grained cache line precision allows an attacker to learn precisely which cache line the victim has accessed. Cache set precision refers to the attacker’s knowledge that the victim has accessed one of the cache lines mapped to a specific cache set without the ability to differentiate between the individual cache lines within that set. Finally, we state whether shared memory between attacker and victim is required. Attacks that do not rely on shared memory are more potent, as they can even monitor memory accesses

Table 4.1: Topological and spatial constraints of the tested attacks.

Attack	Topological Constraint	Spatial Precision	SHM required
Flush+Reload	CPU	cache line	✓
Demote+Reload	Core	cache line	✓
L1 Evict+Reload	Core	cache line	✓
Flush+Flush	CPU	cache line	✓
Demote+Demote	Core	cache line	✓
DemoteContention	CPU	L3 cache set	✗
L1 Prime+Probe	Core	L1 cache set	✗

of a non-readable, statically compiled binary in environments where page deduplication is disabled. An overview of the discussed attacks is given in Table 4.1.

Flush+Reload, Demote+Reload and L1 Evict+Reload have the spatial precision of a cache line as it is loaded and flushed/demoted/evicted as a unit. The attacker learns which cache line the victim has accessed but not the specific offset within. Shared memory is required for the attacker to access the same cache line as the victim. Flush+Reload is easily adaptable to the cross-core scenario. The attacker flushes the cache line from all caches into main memory. A victim access on a second core will bring the cache line into the shared L3 as well as the victim’s private L1 and L2 caches. Since the data is not present in the attacker’s L1 or L2 cache, a subsequent attacker access performs an L3 access. Hence, compared to the SMT case where the attacker differentiates between an L1 and a RAM access, the difference is now between an L3 and a RAM access. Figure 4.1a demonstrates that an L3 access is considerably faster than a RAM access. To port the attack to the cross-core scenario, we may simply need to increase the threshold distinguishing between the victim access versus no-victim access cases to account for an L3 access instead of an L1 access in the victim access case.

In contrast, Demote+Reload cannot be adapted for the cross-core case analogously. The attacker demotes the cache line to the L3 cache. A victim access will bring the cache line into the L1 and L2 caches of the victim core. Since the data is not present in the attacker’s L1 or L2 cache, a subsequent attacker access performs an L3 access. Hence, the attacker performs an L3 access both with and without a victim access.

SMT Flush+Flush, Demote+Demote and cross-core Flush+Flush all operate with cache line precision requiring a shared cache line. Similarly to Flush+Reload, Flush+Flush can be easily adapted to the cross-core case. The attacker flushes the cache line from all caches into the main memory. A victim access on a second core will bring the cache line into the shared L3 as well as the L1 and L2 caches of the victim core. A subsequent attacker `clflush` removes the cache line from all caches in the coherency domain, i.e. including the victim core caches. Hence, the execution of the instruction takes considerably longer than without a victim access, in which case no data needs to be moved. We may want to slightly adapt the threshold distinguishing between victim access and no-victim access

compared to the SMT case, but the timings of L1 flush and cross-core L1 flush are pretty similar (see Figure 4.2a).

In contrast, Demote+Demote cannot be ported to the cross-core scenario analogously. The attacker demotes the cache line to the L3 cache. A victim access will bring the cache line into the L1 and L2 caches of the victim core. Contrary to the `clflush` instruction, `cldemote` only removes data from the L1 and L2 caches of the core the instruction is executed on. Since the data is not present in the L1 or L2 cache of the attacker’s core, a subsequent `cldemote` does not need to move data. Hence, a single cross-core victim access does not have a significant impact on the execution time of a subsequent `cldemote` instruction. We cannot spy on a single cross-core victim access via this technique. However, we can construct a covert channel with the DemoteContention adaptation presented in Section 3.3 via contention on the L3 cache set. This leads to the spatial precision of an L3 cache set for DemoteContention. The attacker and victim do not need to share memory since the attacker only needs to be able to access other cache lines within the same L3 cache set.

For the SMT channels, L1 Prime+Probe is the only exception with regard to spatial precision since we only measure if the probe addresses are still in the L1 cache. Hence, any access to the same L1 cache set is indistinguishable; in other words, this technique has L1 cache set granularity. Again, the attacker and victim do not need to share memory since the attacker only needs to be able to access other cache lines within the same L1 cache set. Textbook Evict+Reload and Prime+Probe are not directly portable to the cross-core scenario since the L3 cache is not inclusive. An eviction from a non-inclusive L3 cache does not automatically cause an eviction from the victim’s private L1 and L2 caches. Hence, the target cache line generally remains in the victim’s private cache hierarchy independently of the occurrence of a potential victim access. Therefore, the cache state can not be used as a side channel to detect a victim access. Cross-core attacks on the Extended Directory via these techniques are possible [84], but we do not consider these in this thesis.

Takeaway: Among the considered attacks, only DemoteContention and L1 Prime+Probe do not require shared memory and operate with less spatial precision than a cache line. Flush+Reload, Flush+Flush and DemoteContention work cross-core, while Demote+Reload and Demote+Demote require the attacker and victim to run on the same physical core. Evict+Reload and Prime+Probe do not function in the cross-core scenario if the shared last-level cache is not inclusive.

4.2 Temporal Metrics

In this section, we measure several metrics by timing different parts of the attacks. These characteristics quantify how reliably and precisely an attacker can monitor victim behavior. They are also relevant for the maximum possible throughput of a covert

channel, which we discuss in Section 4.3. Hit-miss margins (Section 4.2.1) indicate noise resilience. The attack time (Section 4.2.2) measures the duration of a single attack loop iteration. The precision (Section 4.2.3) measures the delay between victim access and attacker detection. The blind spot length (Section 4.2.4) indicates the size of the time window in which an attacker misses potential victim accesses. Since varying frequency scaling during the experiments could influence the duration that an operation takes, we fixed the CPU frequency for all measurements in this section to obtain more comparable results for each attack.

4.2.1 Hit-Miss Margins

The hit-miss margin measures the strength of a side-channel signal, quantifying the difference between two distinguished states and serving as an indicator of noise resilience. A large hit-miss margin makes misclassifications less likely, resulting in a reliable side channel with fewer transmission errors.

We consider timing side channels that exploit memory access or instruction execution time differences to infer the system state. In most cases, we distinguish whether the data is cached in the L1 cache or is resident in a lower-level cache or main memory. However, the execution time does not only vary due to the location of the data which we want to infer. It is also influenced by several other factors, such as scheduling and interrupts. Hence, we repeat all measurements $4 \cdot 10^6$ times to capture the distribution of the execution time. The standard error of the calculated averages is ≤ 0.01 cycles for all tested attacks.

We prepare the required cache state by performing a memory access on the address to bring it into the L1 cache, evicting it to L2 by accessing other addresses in the same L1 cache set, demoting it to L3, or flushing it to main memory. For one set of experiments, we simulate a victim access (for `DemoteContention` multiple victim accesses and `cldemote` executions). For the no-victim access case, we omit this step. Then, we execute the corresponding attack code and measure its execution time with the `rdtsc` instruction.

We then have two distributions of measured execution time, one set of measurements after a victim access has occurred and one without an intermediate victim access. We select a threshold between the two distributions. A system state is then classified as a victim access or no-victim access case according to whether the execution time is above or below the threshold. The accuracy of that classification depends on the overlap of the two distributions, in other words, their respective means and variation. We define the hit-miss margin of a cache side-channel technique as the gap between the 95th percentile of the faster execution case and the 5th percentile of the slower execution case. This definition allows us to limit the impact of outliers and quantify how much typical execution times of the victim access and no-victim access cases differ.

The results are summarized in Table 4.2 and visualized in Figure 4.1 and Figure 4.2. The first three attacks all follow the same principle. We first move the data to the RAM

Table 4.2: Averages of corresponding hit and miss types for the tested attacks and hit-miss margins on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

Attack		Hit Type Victim Access	Hit Average		Miss Type No-Victim Access	Miss Average		Hit-Miss Margin	
CPU		SR/ER	SR	ER	SR/ER	SR	ER	SR	ER
SMT	Flush+Reload	L1 access	42.0 C	36.0 C	RAM access	292.4 C	225.6 C	232 C	166 C
	Demote+Reload	L1 access	42.0 C	36.0 C	L3 access	97.3 C	76.1 C	48 C	34 C
	L1 Evict+Reload	L1 access	39.2 C	34.0 C	L2 access	46.6 C	40.5 C	10 C	4 C
	Flush+Flush	L1 flush	205.1 C	144.2 C	RAM flush	122.7 C	89.7 C	72 C	48 C
	Demote+Demote	L1 demote	214.3 C	147.9 C	L3 demote	109.5 C	78.9 C	86 C	60 C
	L1 Prime+Probe	L2 accesses	270.1 C	243.6 C	L1 accesses	199.0 C	171.7 C	78 C	68 C
CC	Flush+Reload	L3 access	97.3 C	76.1 C	RAM access	292.4 C	225.6 C	170 C	124 C
	Flush+Flush	L1 cc flush	221.8 C	144.4 C	RAM flush	122.7 C	89.7 C	86 C	44 C
	DemoteContention	contention	121.4 C	101.4 C	no contention	119.8 C	100.3 C	-	-

(Flush+Reload), L3 cache (Demote+Reload) or L2 cache (Evict+Reload), simulate a potential victim access and measure the access time afterward. First, with Flush+Reload, we consider the timing difference between an L1 access (victim access) at 42.0 cycles (SR) and 36.0 cycles (ER) and a RAM access (no victim access) at 292.4 cycles (SR) and 225.6 cycles (ER). The hit-miss margins of 232 cycles (SR) and 166 (ER) are the largest of the three attacks since the data resides at the closest and farthest locations from the core, respectively. Second, in a Demote+Reload attack, we differentiate between an L1 access (victim access) at 42.0 cycles (SR) and 36.0 cycles (ER) and an L3 access (no victim access) at 97.3 cycles (SR) and 76.1 cycles (ER). Since the data resides closer together, the hit-miss margins are considerably smaller, with 48 cycles (SR) and 34 cycles (ER). Third, for L1 Evict+Reload the difference is measured between an L1 access (victim access) at 39.2 cycles (SR) and 34.0 cycles (ER) and an L2 access (no victim access) at 46.6 cycles (SR) and 40.5 cycles (ER). The hit-miss margins are very small at 10 cycles (SR) and 4 cycles (ER). Hence, L1 Evict+Reload is very susceptible to noise and we expect a considerably higher number of misclassifications than for the other attack techniques.

The second group of attacks times the execution of the instruction or memory accesses used to evict the target address. Again, we move the data to the RAM (Flush+Flush), L3 cache (Demote+Demote) or L2 cache (Prime+Probe), but now we measure the time these operations take after a potential victim access. For Flush+Flush, we differentiate between the execution time of an L1 `clflush` (victim access) at 205.1 cycles (SR) and 144.2 cycles (ER) and the execution time of a RAM `clflush` (no victim access) at 122.7 cycles (SR) and 89.7 cycles (ER). The hit-miss margins amount to 72 cycles (SR) and 48 cycles (ER). The timing difference can be explained by the amount of work that needs to be performed by the instruction. If the data resides in the L1 cache, `clflush` performs a write-back to memory. Otherwise, if the data resides in main memory, the instruction only needs to check that the data is not stored in the cache hierarchy.

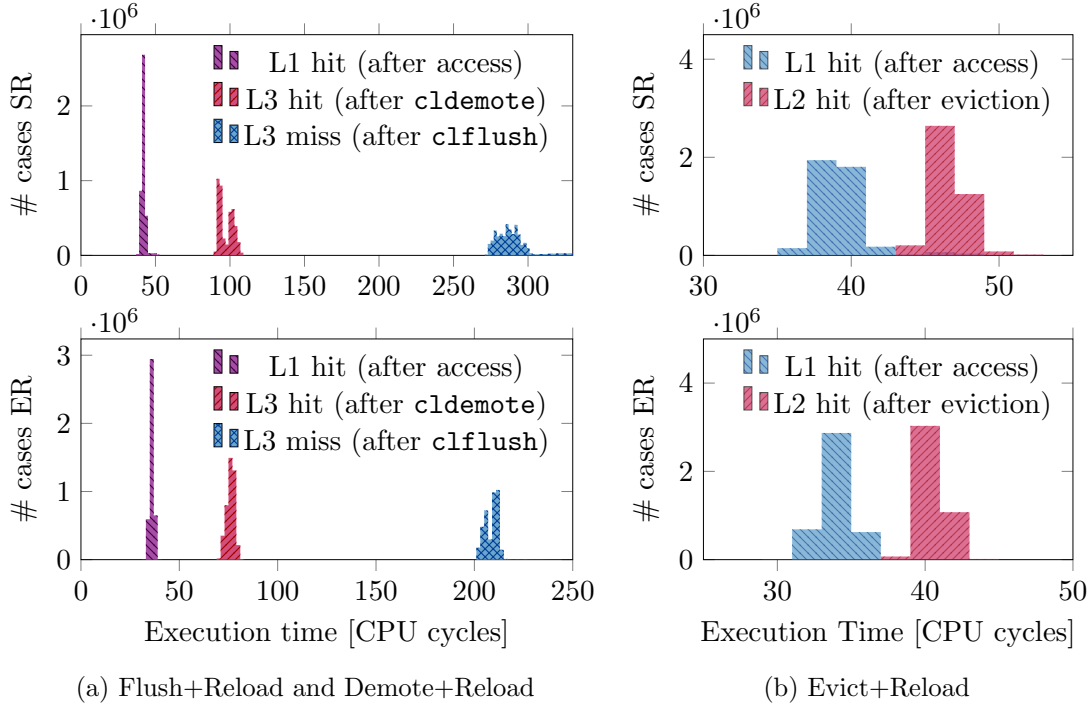


Figure 4.1: Target Address Access Timing Histograms on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs. Figure 4.1a shows clearly separated distributions between an L1 hit and an L3 hit (Demote+Reload) or an L3 miss (Flush+Reload), respectively. Figure 4.1b shows the adjacency of the distributions for an L1 and an L2 hit (Evict+Reload).

Similarly, for Demote+Demote, if the data resides in the L1 cache (victim access), the `cldemote` instruction takes 214.3 cycles (SR) and 147.9 cycles (ER) versus only 109.5 cycles (SR) and 78.9 cycles (ER) on an L3 cache hit (no victim access). With a value of 86 cycles (SR) and 60 cycles (ER), the hit-miss margins of Demote+Demote are larger than of Flush+Flush. Again, the `cldemote` instruction performs more work during the write-back from the L1 and L2 caches than if the data does not reside in the L1 and L2 caches.

For L1 Prime+Probe the measurements look different. The eviction from L1 (victim access) takes 270.1 cycles (SR) and 243.6 cycles (ER), while the eviction for an already evicted address (no victim access) lasts 199.0 cycles (SR) and 171.7 cycles (ER). If a victim access has taken place, cache lines used for probing will have been evicted to the L2 cache. Hence, the execution time is longer after a victim access. With 78 cycles (SR) and 68 cycles (ER), respectively, the hit-miss margins are considerably larger than for Evict+Reload. The hit-miss margin of Evict+Reload measures the timing difference between a single L1 and L2 access. The larger margins for Prime+Probe suggest that

the cache eviction policy induces the eviction of many eviction set cache lines due to a single victim access.

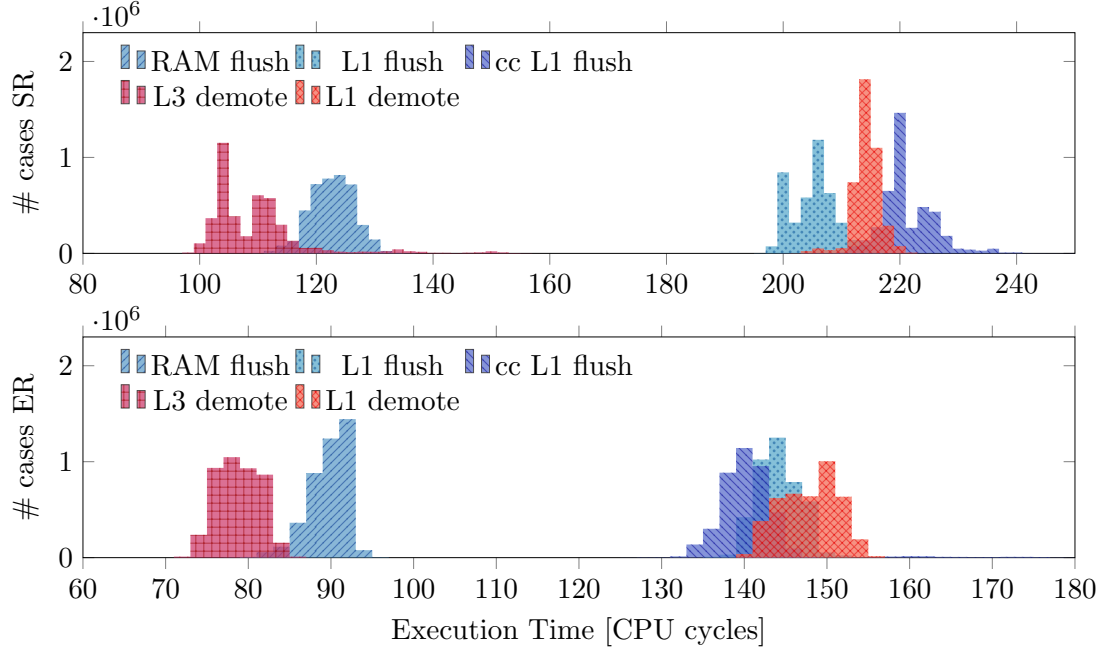
Flush+Reload has considerably larger hit-miss margins than Demote+Reload, which has larger margins than Evict+Reload. This is due to the direct relation between the distance between the locations of the cache line in the two edge cases and the timing difference in their access time. In contrast, Flush+Flush, Demote+Demote and Prime+Probe have comparable hit-miss margins. In general, the measurements on the two CPUs agree on the ranking of the attacks concerning hit-miss margins. However, on our Sapphire Rapids CPU, Demote+Demote has a slightly larger hit-miss margin than Prime+Probe, while we observe the opposite on our Emerald Rapids CPU. This could be due to different implementations of the `cldemote` instruction on the two CPUs.

In the cross-core Flush+Reload attack, we differentiate an L3 hit (data in L1 of victim core) at 97.3 cycles (SR) and 76.1 cycles (ER) from a RAM access (no victim access) at 292.4 cycles (SR) and 225.6 cycles (ER). The hit-miss margins at 170 cycles (SR) and 124 cycles (ER) are lower than in SMT Flush+Reload but still approximately twice as large as the hit-miss margins of any other attack techniques. For cross-core Flush+Flush, we compare a cross-core L1 flush at 221.8 cycles (SR) and 144.4 cycles (ER) with a RAM flush (no victim access) at 122.7 cycles (SR) and 89.7 cycles (ER). The hit-miss margins amount to 86 cycles (SR) and 44 cycles (ER), respectively. On our Sapphire Rapids CPU, in contrast to Flush+Reload, here, the cross-core case has a larger hit-miss margin than SMT Flush+Flush since the data needs to be flushed from the victim core’s L1 and L2 caches. The data indicates that this consumes about 16.7 additional cycles compared to an L1 flush on the attacker core on our Sapphire Rapids CPU. However, on our Emerald Rapids CPU, an L1 flush on the attacker core is slightly slower than a cross-core L1 flush, leading to a slightly lower hit-miss margin in the cross-core case compared to SMT Flush+Flush. While the slower cross-core `clflush` observed on the Sapphire Rapids CPU could be explained by the delay caused by the communication with the cross-core caches, a possible explanation of the faster cross-core `clflush` on the Emerald Rapids CPU is that the executing core does not need to remove the cache line from its own private caches.

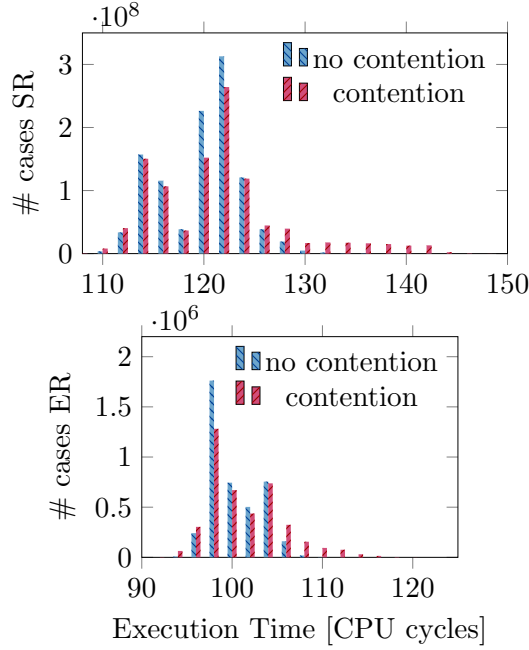
Finally, for cross-core DemoteContention, we compare the contention case at 121.44 cycles (SR) and 101.4 cycles (ER) with the no-contention case at 119.8 cycles (SR) and 100.3 cycles (ER). The distributions overlap to a large extent. As shown in Figure 4.2b, the majority of both cases on our Sapphire Rapids CPU fall into the range of 110 to 130 CPU cycles, with the no contention case even demonstrating a higher number of occurrences at 120 and 122 cycles, which lies in the upper half of the range. Hence, there is no sensible value to assign to the hit-miss margin. However, the contention case demonstrates a considerably large number of outliers above 130 cycles (SR) that do not occur for the no contention case. Similarly, there are outliers of the contention case above 112 cycles on our Emerald Rapids CPU. Therefore, we can correctly classify executions above 130 cycles (SR) and 112 cycles (ER) as contention with high certainty, while we will have a high error rate for most execution times under 130 cycles (SR) and 112 cycles

(ER), respectively. This means that a side channel established via this technique will be very noisy, but it is still possible to transmit data.

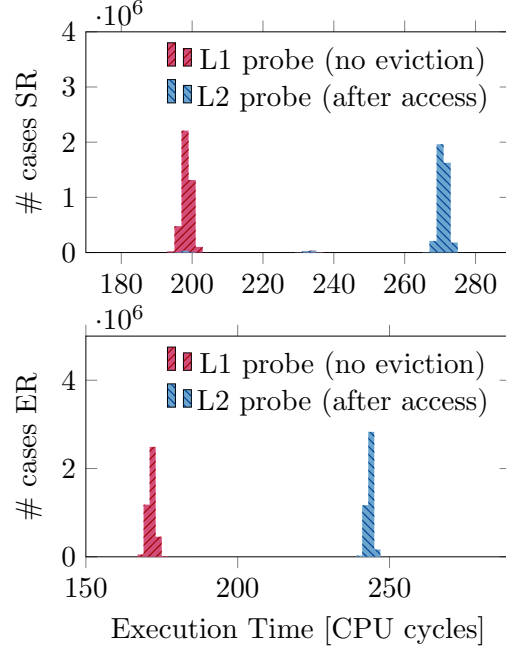
Takeaway: Flush+Reload clearly shows the largest hit-miss margins in both the SMT and cross-core scenario due to the large timing differences between an L1 access and a RAM access. Flush+Reload, Demote+Reload and L1 Evict+Reload show a correlation between hit-miss margin and memory access time in the no-victim-access case with Demote+Reload demonstrating a lower hit-miss margin than Flush+Reload and Evict+Reload in turn having a lower hit-miss margin than Demote+Reload. Flush+Flush, Demote+Demote and L1 Prime+Probe have similar hit-miss margins and clearly separated distributions. The distributions for DemoteContention overlap the most, leading to no sensible definition of a hit-miss margin, followed by Evict+Reload, which shows the lowest measurable hit-miss margin.



(a) Flush+Flush and Demote+Demote



(b) Cross-core DemoteContention



(c) Prime+Probe

Figure 4.2: Instruction and Eviction Timing Histograms on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs. Figure 4.2a shows the hit-miss margins for Flush+Flush and SMT Demote+Demote. The distributions for cross-core DemoteContention in Figure 4.2b overlap to a large extent. Figure 4.2c shows a much clearer separation for Prime+Probe compared to Evict+Reload in Figure 4.1b.

4.2.2 Attack Time

The attack time measures the duration of a single attack loop. The loop body consists of the respective measurement and a potential store operation if the measured value indicates a hit. A low attack time is desirable to maximize the frequency at which an attacker can monitor victim accesses. For most techniques, the attack time varies with the occurrence of an intermitted victim access. In a spy-victim scenario, the no-victim access case needs to have a short attack time to enable a high monitoring frequency, while a longer attack time after a victim access is less critical. Conversely, in a covert channel, both cases must be accommodated within each transmission window, requiring both attack times to be minimized.

We perform two sets of measurements for each attack type, once with a victim access and once without. In each set, we perform 10^6 measurements and calculate the average. The results are summarized in Table 4.3 and the distributions are visualized in Figure 4.3. The standard error of the average is ≤ 0.1 cycles for all tested attacks.

In Flush+Reload, the attack loop takes 376.4 cycles (SR) and 294.7 cycles (ER) after a victim access and 614.1 cycles (SR) and 462.8 cycles (ER) without a victim access. Without a victim access, the attacker needs to perform a RAM access instead of an L1 access to retrieve the data. The difference of 237.7 cycles (SR) and 168.1 cycles (ER) aligns with the measured hit-miss margin of 232 cycles (SR) and 166 cycles (ER) in Section 4.2.1. In Demote+Reload, the attack loop takes 385.9 cycles (SR) and 299.5 cycles (ER) after a victim access and 424.3 cycles (SR) and 320.6 cycles (ER) without a victim access. Similar to Flush+Reload, the no-victim-access case requires an L3 access instead of an L1 access. The difference between the two cases amounts to 38.4 cycles (SR) and 21.1 cycles (ER), which is roughly comparable to the hit-miss margin of 48 cycles (SR) and 34 cycles (ER).

In L1 Evict+Reload, the attack loop takes 398.6 cycles (SR) and 333.3 cycles (ER) after a victim access and 390.0 cycles (SR) and 339.8 cycles (ER) without a victim access. While the hit-miss margin is very small for L1 Evict+Reload there is an additional factor at play here in contrast to the two previous attacks. Flush+Reload and Demote+Reload measure the access on a single address, so the attack time difference amounts approximately to the hit-miss margin. When we measure the attack time for L1 Evict+Reload, however, this does not only include the measured access time of the target address but also the access times of the addresses used for eviction. Since the attacker performs accesses on the target address and the entire eviction set, we expect all accesses to be L2 hits in the no-victim-access case. If a victim access occurs, the subsequent attacker access on the target address results in an L1 hit. The eviction set accesses result in L2 hits. Therefore, we expect the victim-access and no-victim-access cases to differ by the speed-up of one L1 instead of an L2 hit. Our measurements show that this difference is insignificant and both cases have similar runtimes. While the no-victim-access case is indeed slightly faster than the victim-access case on our Emerald Rapids CPU, we observe the opposite effect on our Sapphire Rapids CPU.

Table 4.3: Attack Time: Average attack time in CPU cycles for all tested attacks. We performed the measurements on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

Attack		Victim Access		No Victim Access	
CPU		SR	ER	SR	ER
SMT	Flush+Reload	376.4 C	294.7 C	614.1 C	462.8 C
	Demote+Reload	385.9 C	299.5 C	424.3 C	320.6 C
	L1 Evict+Reload	398.6 C	333.3 C	390.0 C	339.8 C
	Flush+Flush	264.9 C	197.7 C	192.0 C	155.2 C
	Demote+Demote	289.2 C	216.5 C	185.8 C	137.6 C
	L1 Prime+Probe	363.3 C	316.5 C	281.2 C	245.0 C
CC	Flush+Reload	712.5 C	434.0 C	614.1 C	462.8 C
	Flush+Flush	278.6 C	232.2 C	192.0 C	155.2 C
	DemoteContention	?		185.8 C	137.6 C

Comparing the victim-access-case attack times of Flush+Reload, Demote+Reload, and Evict+Reload, we note that the timings for the victim-access case increase with the time required to remove the target address from the relevant caches. The attack loop performs an L1 access and consequently flushes/demotes/evicts the data. The average duration of an L1 flush is the fastest, followed by L1 demote and L1 eviction (see Section 4.2.1). Hence, the victim access attack times of Flush+Reload is the smallest, followed by Demote+Reload and Evict+Reload.

In the victim access case, Flush+Reload has the shortest attack time, followed by Demote+Reload and Evict+Reload has the longest attack time. In the no-victim access case, we observe contrasting behavior. The differences between victim and no victim access depend on the additional work required to be performed in the no-victim-access case: instead of an L1 access in the victim-access case, Flush+Reload performs a RAM access, Demote+Reload performs an L3 access and Evict+Reload performs an L2 access in the no-victim-access case. On our Sapphire Rapids CPU, this leads to Flush+Reload having the longest attack time in the no-victim access case, followed by Demote+Reload and Evict+Reload. On our Emerald Rapids CPU, the victim-access case attack time of Evict+Reload is 33.8 cycles larger than Demote+Reload. Hence, the additional 21.1 cycles of the L3 access performed in the no-victim access case of Demote+Reload is insufficient to result in a larger no-victim-access attack time than Evict+Reload. Therefore, on our Emerald Rapids CPU, Flush+Reload also has the largest no-victim-access attack time, but Demote+Reload has a lower attack time than Evict+Reload in the victim-access and no-victim-access cases.

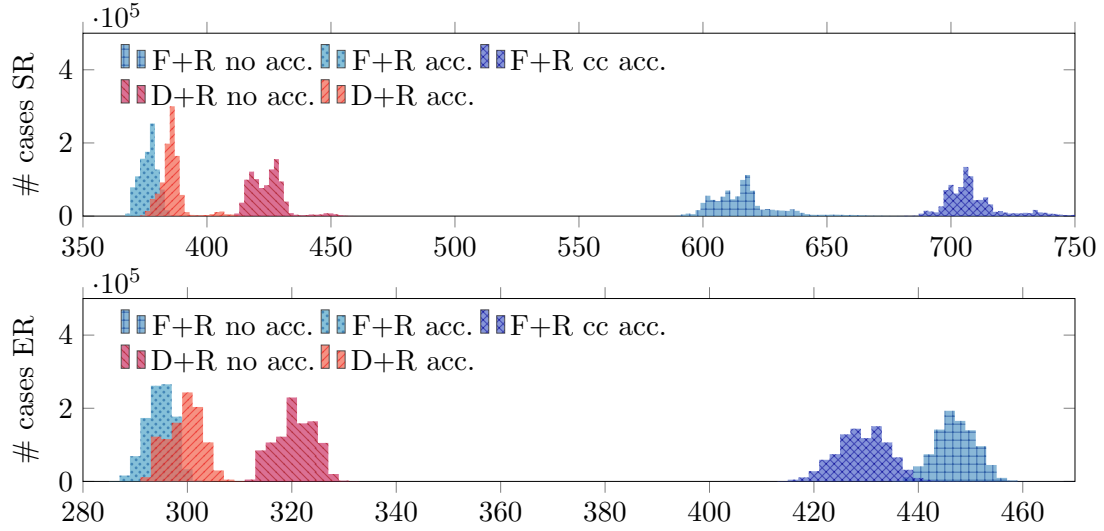
In Flush+Flush, the attack loop takes 264.9 cycles (SR) and 197.7 cycles (ER) after a victim access and 192.0 cycles (SR) and 155.2 cycles (ER) without a victim access. The

differences of 72.9 cycles (SR) and 42.5 cycles (ER) align with the measured hit-miss margins of 72 cycles (SR) and 48 cycles (ER). In Demote+Demote, the attack loop takes 289.2 cycles (SR) and 216.5 cycles (ER) after a victim access and 185.8 cycles (SR) and 137.6 cycles (ER) without a victim access. The differences of 103.4 cycles (SR) and 78.9 cycles (ER) overapproximate the hit-miss margins of 86 cycles (SR) and 60 cycles (ER). In L1 Prime+Probe, the attack loop takes 363.3 cycles (SR) and 316.5 cycles (ER) after a victim access and 281.2 cycles (SR) and 245.0 cycles (ER) without a victim access. The differences of 82.1 cycles (SR) and 71.5 cycles (ER) compare well with the hit-miss margin of 78 cycles (SR) and 68 cycles (ER).

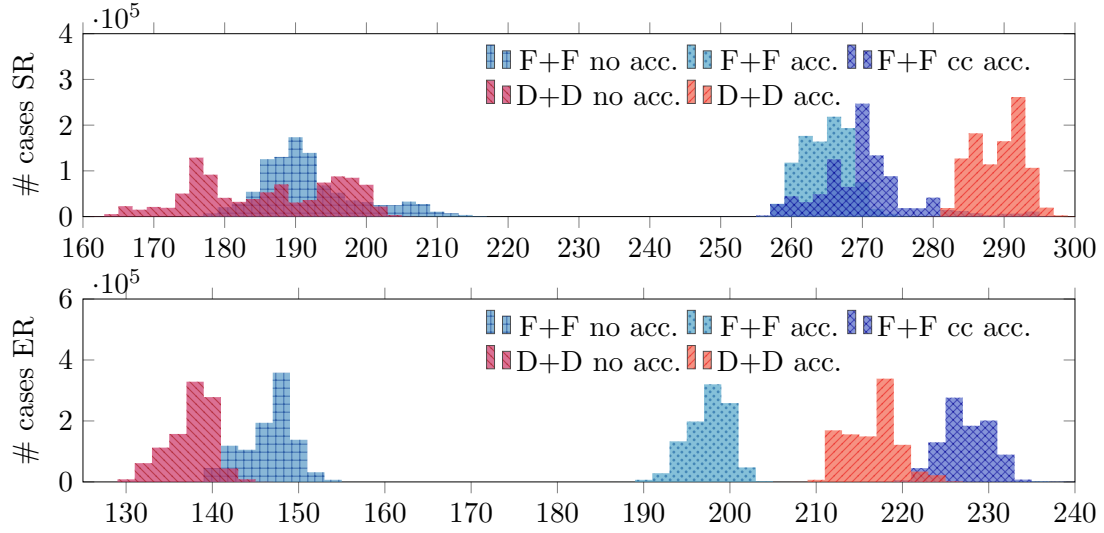
Comparing the victim access case attack times of Flush+Flush, Demote+Demote and Prime+Probe, the timings for the victim access case are again increasing with the time required to remove the target address from the relevant caches. The attack loop performs an L1 flush/demote/eviction. An L1 eviction has the longest average duration, followed by L1 demote and L1 flush (see Section 4.2.1). Hence, Prime+Probe has the the largest victim-access attack time, followed by Demote+Demote and Flush+Flush. In the no-victim-access case, Demote+Demote has the lowest attack time, followed by Flush+Flush and L1 Prime+Probe. Again, this is consistent with the fact that an L3 demote is the fastest, followed by a RAM flush and L1 eviction (see Section 4.2.1).

Comparing Flush+Reload with Flush+Flush, note that both victim access and no victim access attack times are shorter for Flush+Flush. The attack loop in Flush+Flush only performs a `clflush` while Flush+Reload performs an access and a `clflush`. For the victim access case, the Flush+Flush attack loop is 111.5 cycles (SR) and 97.0 cycles (ER) faster than Flush+Reload. In the no-victim-access case, we benefit even more from the omitted memory access. Not only do we perform a time-consuming RAM access in Flush+Reload, but in addition, we need to perform an L1 flush. In Flush+Flush, we only need to perform a significantly faster RAM flush. For the no-victim-access case, the Flush+Flush attack loop is 422.1 cycles (SR) and 307.6 cycles (ER) faster than Flush+Reload.

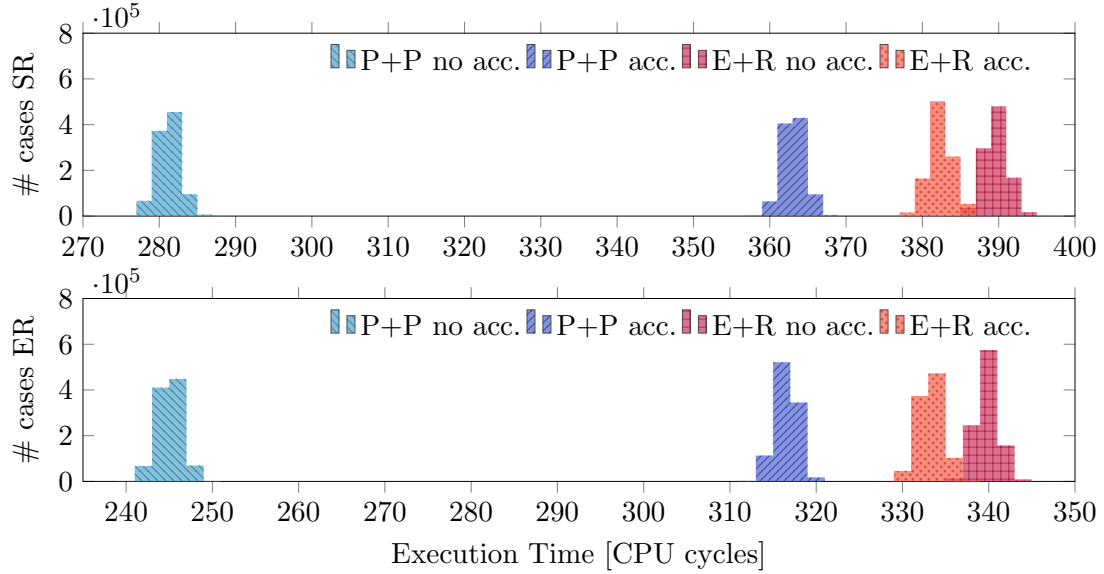
Similarly, Demote+Demote is 96.7 cycles (SR) and 83.0 cycles (ER) faster than Demote+Reload in the victim access case due to the omitted memory access. In the no-victim-access case, the Demote+Demote attack loop is 238.5 cycles (SR) and 183.0 cycles (ER) faster than Demote+Reload since we omit the L1 access and perform a significantly faster L3 demote instead of an L1 demote. The Prime+Probe attack loop is 35.3 cycles (SR) and 16.8 cycles (ER) faster than Evict+Reload in the victim access case due to the omitted L1 access. In the no-victim-access case, the Prime+Probe attack loop is 108.8 cycles (SR) and 94.8 cycles (ER) faster than Evict+Reload. By omitting the L1 access on the target address, we are also not evicting any probing addresses. Therefore, the Prime+Probe attack loop only performs fast L1 accesses in the no-victim-access case.



(a) Flush+Reload and Demote+Reload



(b) Flush+Flush and Demote+Demote



(c) Prime+Probe and Evict+Reload

Figure 4.3: Attack Time: Execution time in cycles of a single attack iteration for all tested attacks on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

An interesting observation is that the impact of the additional memory access in the victim access case varies among the attacks. The victim access is started before the attacker access. If the victim access has not concluded at the time of the attacker access, the memory accesses are likely merged. For Evict+Reload, the victim access performs a fast L2 access and the timing of the subsequent attacker’s memory access equates to less than an L1 access time. However, for Demote+Reload and Flush+Reload, the increased memory access times indicate that the victim access has not concluded yet at the time of the attacker access. Even though the attacker access follows a victim access, its timing does not correspond to an L1 access. In our experimental setup, we only test attack loops that run immediately after a victim access. With an added random delay, we expect the attack times of Flush+Reload and Demote+Reload in the victim access case to be slightly lower on average but still higher than Flush+Flush and Demote+Demote due to the additional memory access.

In cross-core Flush+Reload, the attack loop takes 712.5 cycles (SR) and 434.0 cycles (ER) after a victim access, constituting the highest attack time among all techniques. In contrast to the SMT variant, cross-core Flush+Reload needs to perform a slower L3 access instead of an L1 access after a victim access (see Figure 4.1a). Additionally, the cross-core L1 flush takes more time than an L1 flush on our Sapphire Rapids CPU (see Figure 4.2a), resulting in a larger attack time in the victim-access case compared to the no-victim access case at 614.1 cycles (SR). In contrast, on our Emerald Rapids CPU, the cross-core L1 flush is less time-consuming (see Figure 4.2a), leading to a shorter attack time in the victim-access case compared to the no-victim access case at 462.8 cycles. In cross-core Flush+Flush, the attack loop takes 278.6 cycles (SR) and 232.2 cycles (ER) after a victim access and 192.0 cycles (SR) and 155.2 cycles (ER) without a victim access. Since the cross-core L1 flush is faster than an L1 flush on our Emerald Rapids CPU, the longer attack time in the victim access case compared to the SMT variant lacks an explanation. In cross-core DemoteContention, the attack loop takes 185.8 cycles (SR) and 137.6 cycles (ER) without a victim access. The victim case is difficult to evaluate due to high noise and missed transmissions.

Takeaway: Flush+Flush, Demote+Demote and DemoteContention have the lowest attack times since they do not induce any cache misses and only perform a single instruction in their attack loop. Our novel Demote+Demote and DemoteContention attacks have the shortest attack times in the no-victim-access case. SMT Flush+Flush has the lowest attack time in the victim access case. Cross-core Flush+Reload has the highest victim-access attack times, followed by L1 Evict+Reload. Cross-core Flush+Reload and SMT Flush+Reload have the highest no-victim-access attack times, followed by Demote+Reload and Evict+Reload.

4.2.3 Precision

The temporal precision quantifies how accurately an attacker can estimate the exact time of a victim access. A small delay until detection enables an attacker to monitor

several victim accesses in a short time frame. A small standard deviation of the detection delay allows for accurate estimates of the duration between victim accesses as required in various timing-based attacks, e.g. inter-keystroke timing attacks.

When a victim performs a memory operation, it takes time for the effects of the operation to become visible to the attacker. When the victim accesses data residing in the main memory in a Flush+Reload attack, the attacker can only detect the access once the loading operation has brought the data into the cache hierarchy. Depending on the current position within the attacker loop, it will take additional time from when the operation becomes detectable (e.g. present in the cache hierarchy) to the time it is actually detected by the attacker (e.g. by performing a timed memory access).

In this subsection, we measure the time from the start of the victim operation until the attacker can detect the effects of the operation. However, it is a non-trivial question of when to finish the measurement. We want to exclude the time it takes the attacker to execute the attack loop (covered in Section 4.2.2). Hence, we end the time measurement at the beginning of the attacker’s loop iteration that detects the access. Note that this choice can result in a negative time measurement if an access is performed shortly after an attacker loop iteration has started and detected within the same loop iteration.

A measurement is performed as follows: First, we take the starting timestamp s via the `rdtsc` instruction before the victim access is performed. Second, the victim issues an access on the target address. For DemoteContention, the victim also performs a `cldemote` on the address. Third, we take a timestamp e_i at the beginning of each attacker’s loop iteration. Fourth, the attacker executes the attack loop. They measure the access/instruction time corresponding to the attack type. Fifth, if the measured value is above the chosen threshold, indicating a victim access, the temporal precision of the detection is calculated as $e_i - s$ and stored.

Concerning the general setup, the attacker performs a continuous loop of the attack code and the time measurement store as described above. Keeping the attack loop as short as possible is crucial to get a fine time measurement resolution. Meanwhile, the victim thread performs memory accesses at random times sampled from a uniform distribution in each iteration. We ensure that the attacker has time to detect the access before the next victim access is performed by including a fixed minimum waiting period in addition to a random offset. The random time intervals ensure that the attacker will be in different stages of its loop iteration when the access happens, simulating a real-world scenario in which an attacker is spying on a victim access. If victim accesses are missed due to external memory access noise or a blind spot (see Section 4.2.4), we discard the measurement.

We perform 10^6 measurements for each attack technique and calculate the average and the standard deviation (which is a relevant metric for inter-keystroke timing attacks [21]). The standard error of the average is ≤ 0.1 cycles for all tested attacks. The results are summarized in Table 4.4 and visualized in Figure 4.4.

Table 4.4: Temporal Precision: Average and standard deviation of time until access is detected in cycles and nanoseconds on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

Attack		Average Time until Detection				Standard Deviation			
CPU		SR		ER		SR		ER	
SMT	Flush+Reload	74.2 ns	200.3 C	75.5 ns	150.9 C	20.1 ns	54.2 C	19.9 ns	39.8 C
	Demote+Reload	65.9 ns	178.0 C	73.6 ns	147.1 C	27.1 ns	73.3 C	24.3 ns	48.5 C
	L1 Evict+Reload	52.0 ns	140.4 C	55.6 ns	111.1 C	38.7 ns	104.8 C	41.4 ns	82.9 C
	Flush+Flush	36.4 ns	98.2 C	41.3 ns	82.5 C	19.1 ns	51.5 C	18.2 ns	36.4 C
	Demote+Demote	34.0 ns	91.7 C	40.3 ns	80.6 C	17.4 ns	47.1 C	16.5 ns	33.0 C
	L1 Prime+Probe	35.2 ns	95.0 C	35.7 ns	71.3 C	33.2 ns	89.8 C	33.2 ns	66.3 C
CC	Flush+Reload	132.6 ns	358.1 C	95.6 ns	191.1 C	24.7 ns	66.8 C	14.9 ns	29.7 C
	Flush+Flush	45.4 ns	122.7 C	44.4 ns	88.8 C	12.5 ns	33.7 C	14.6 ns	29.1 C
	DemoteContention	48.7 ns	131.5 C	84.1 ns	168.1 C	16.1 ns	43.5 C	24.3 ns	48.6 C

In a Flush+Reload attack, it takes 200.3 cycles (74.2 ns, SR) and 150.9 cycles (75.5 ns, ER) until the access is detected with a standard deviation of 54.2 cycles (20.1 ns, SR) and 39.8 cycles (19.9 ns, ER). Recalling the average RAM access time of 292.4 cycles (SR) and 225.6 cycles (ER) (see Table 4.2), this is a reasonable result. The attacker can detect the access once the data is loaded at least into the L3 cache, in other words, possibly before the victim’s load operation is finished. Depending on the chosen threshold, the access can be detected even earlier if the victim and attacker accesses are merged. In a Demote+Reload attack, it takes 178.0 cycles (65.9 ns, SR) and 147.1 cycles (73.6 ns, ER) until the access is detected with a standard deviation of 73.3 cycles (27.1 ns, SR) and 48.5 cycles (24.3 ns, ER). In an L1 Evict+Reload attack, it takes 140.4 cycles (52 ns, SR) and 111.1 cycles (55.6 ns, ER) until the access is detected with a standard deviation of 104.8 cycles (38.7 ns, SR) and 82.9 cycles (41.4 ns, ER).

As expected, the timings for Demote+Reload and L1 Evict+Reload are lower than for Flush+Reload, but there are two observations to note. First, recall the average L3 and L2 access timings of 97.3 cycles (SR)/75.1 cycles (ER) and 46.6 cycles (SR)/40.5 cycles (ER), respectively (from Section 4.2.1). Contrary to Flush+Reload, for Demote+Reload and Evict+Reload, the average number of cycles until detection is higher than the victim’s access time. Second, while the average time until detection is going down, the closer to the CPU the data resides, the standard deviation is significantly higher.

A possible explanation of these phenomena is the longer execution time of L1 demote and L1 eviction compared to an L1 flush (see Section 4.2.1). The memory access time provides a lower bound for the temporal precision. However, the actual detection time and variance are highly dependent on the range of possible positions of the attacker within the attacker loop that lead to a detection. Since we are disregarding missed accesses in this section, the standard deviation depends on the size of the interval within the attack loop in which an access can be detected in the current or next iteration. Assuming

a uniform distribution of the attacker’s position within the attack loop at the time of access, the size of this interval correlates with the standard deviation.

Victim accesses that occur after the attacker access has been issued and before the `clflush` or `cldemote` execution do not have a persisting impact on the cache state and, therefore, cannot be detected by the attacker. Hence, the size of the detection interval depends on the fraction of the `clflush` or `cldemote` execution during which a victim access leads to the data being present in the L1 cache after instruction and access have concluded. The measurements suggest that the longer execution time of `cldemote` also includes a larger time interval during which a victim access is persistent, leading to a larger standard deviation for the temporal precision of Demote+Reload compared to Flush+Reload.

While it is not clear under which circumstances a victim access is persistent during a `clflush` or `cldemote` execution, for Evict+Reload, we can be more precise. In an LRU cache, the victim address will be present in the L1 cache in the next iteration of the attack loop if it occurs at any point in the attack loop after the first attacker eviction access has occurred. Instead of the victim address, the first accessed eviction address will be evicted by the last eviction access of the iteration. Hence, for most of the eviction time window, a victim access has a persistent cache state impact until the next iteration. This is a possible cause for the high standard deviation of the temporal precision of Evict+Reload.

For the instruction and eviction time-based attacks, the detection times lie closer together. In a Flush+Flush attack it takes 98.2 cycles (36.4 ns, SR) and 82.5 cycles (41.3 ns, ER) until the access is detected with a standard deviation of 51.5 cycles (19.1 ns, SR) and 36.4 cycles (18.2 ns, ER). In a Demote+Demote attack it takes 91.7 cycles (34 ns, SR) and 80.6 cycles (40.3 ns, ER) until the access is detected with a standard deviation of 47.1 cycles (17.4 ns, SR) and 33.0 cycles (16.5 ns, ER). In an L1 Prime+Probe attack it takes 95.0 cycles (35.2 ns, SR) and 71.3 cycles (35.7 ns, ER) until the access is detected with a standard deviation of 89.8 cycles (33.2 ns, SR) and 66.3 cycles (33.2 ns, ER).

In contrast to Flush+Reload, Demote+Reload and Evict+Reload, we do not perform attacker memory accesses. Hence, we cannot miss victim accesses due to prior attacker accesses. A victim access that occurs during a flush/demote/eviction either concludes first (for Evict+Reload at least before the last eviction access) and is therefore detected in the current iteration or the flush/demote/eviction finishes first and the access will be detected in the next iteration. Hence, the average detection time of Flush+Reload, Demote+Reload, and Prime+Probe is drastically reduced compared to Flush+Reload, Demote+Reload and Evict+Reload due to the increased number of accesses detected in the current iteration.

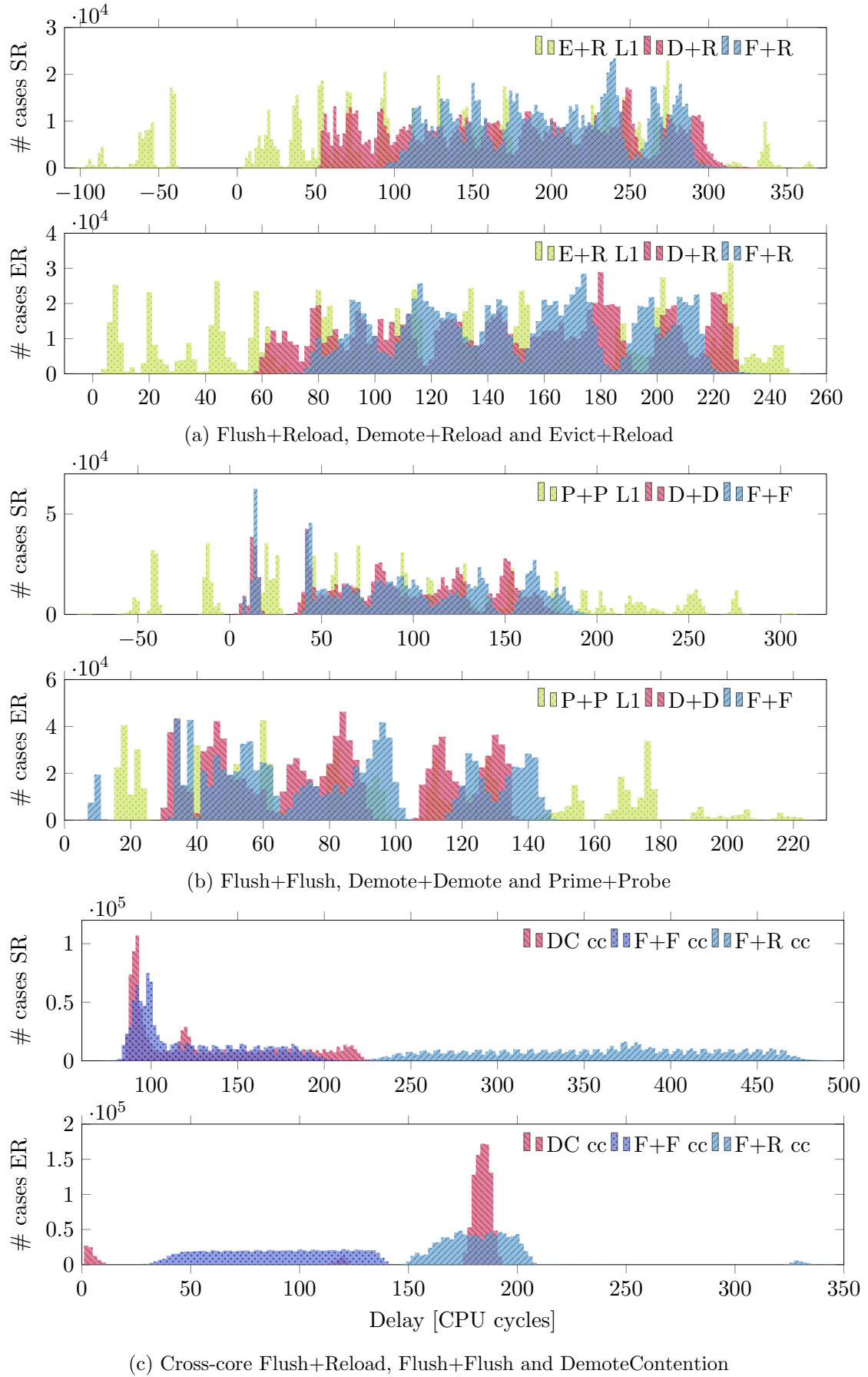


Figure 4.4: Delay between victim access and beginning of attack loop iteration that detects the access (for DemoteContention access and `cldemote`) for all tested attacks on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

Looking beyond averages and standard deviation, the temporal precision distributions present some interesting behavior that we cannot explain. All distributions show periodic spikes. On our Sapphire Rapids CPU, Flush+Flush and Demote+Demote show a large spike around 14 cycles separated from the largest part of the distribution. While Flush+Flush also displays a smaller spike on our Emerald Rapids CPU, Demote+Demote does not exhibit the same behavior here.

In a cross-core Flush+Reload attack it takes 358.1 cycles (132.6 ns, SR) and 191.1 cycles (95.6 ns, ER) until the access is detected with a standard deviation of 66.8 cycles (24.7 ns, SR) and 29.7 cycles (14.9 ns, ER). On our Sapphire Rapids CPU, the distribution cross-core Flush+Reload shows a large variation between approximately 225 and 480 cycles. In contrast, on our Emerald Rapids CPU, the distribution of cross-core Flush+Reload is much more concentrated between approximately 150 and 210 cycles and has a smaller outlier peak around 328 cycles. In a cross-core Flush+Flush attack it takes 122.7 cycles (45.4 ns, SR) and 88.8 cycles (44.4 ns, ER) until the access is detected with a standard deviation of 33.7 cycles (12.5 ns, SR) and 29.1 cycles (14.6 ns, ER). On our Sapphire Rapids CPU, the distribution of cross-core Flush+Flush shows a large spike around 100 cycles and fades out until around 200 cycles. In contrast, on our Emerald Rapids CPU, the distribution of cross-core Flush+Flush demonstrates no major spikes and covers a similar range to SMT Flush+Flush. We are uncertain what causes the spike at the lower end of the cross-core Flush+Flush distribution and the larger detection delay compared to the SMT scenario on our Sapphire Rapids CPU.

In a cross-core DemoteContention attack, it takes 131.5 cycles (48.7 ns, SR) and 168.1 cycles (84.1 ns, ER) until the access and `cldemote` execution are detected with a standard deviation of 43.5 cycles (16.1 ns, SR) and 48.6 cycles (24.3 ns, ER). In the other tested attacks, victim accesses cause data to be loaded into the L1 cache, where it persists until the next attack loop iteration. This leads to high variation in the temporal precision since the access can be detected after an arbitrarily long time as long as the data is not evicted in the meantime. In contrast, the contention caused by DemoteContention is only temporarily detectable. Hence, the temporal precision distribution is concentrated in a large spike around 93 cycles (SR) and 184 cycles (ER) and has a very low standard deviation on our Sapphire Rapids CPU. On our Emerald Rapids CPU, DemoteContention has a considerable amount of outliers at around 4 cycles and 120 cycles, increasing the standard deviation. Comparing the distributions of an L3 demote with the temporal precision of DemoteContention, we note that on our Sapphire Rapids CPU, the contention is mostly observable shortly before the instruction retires. In contrast, on our Emerald Rapids CPU, the main temporal precision peak lies outside of the L3 demote duration.

Takeaway: Demote+Demote, Flush+Flush and Prime+Probe show the smallest delay between victim access and attacker detection since they likely detect more accesses within the same attack loop iteration. Cross-core Flush+Flush shows the smallest standard deviation, followed by DemoteContention (on SR) / cross-core Flush+Reload (on ER) and Demote+Demote. Cross-core Flush+Reload has the longest delay until detection due

to the required time-expensive RAM access. L1 Evict+Reload shows the largest standard deviation since detected accesses can occur during the majority of the eviction.

4.2.4 Blind Spot

In subsection Section 4.2.3, we measured the time it takes until a victim operation becomes detected by the attacker. We ignored the cases where the attacker failed to detect the operation entirely. In this subsection, we measure how many victim operations are missed for each attack technique. In most cache attack techniques, the attacker performs a detection and restores the cache state. Therefore, the victim access is no longer detectable after a full attack iteration. If the victim operation is performed after the detection but before the cache state restoration, the operation will remain undetected. This period during which the victim operation is missed is called the ‘Blind Spot’. A large blind spot length means the attacker will likely miss many victim accesses. This is especially critical in the spy-victim scenario in which the attacker and victim are not synchronized.

To measure the blind spot length, we have a similar experimental setup as in Section 4.2.3. The attacker code is executed in a continuous loop and the victim thread performs a memory access at random times. Hence, the attacker’s program counter will be at different stages of its attack loop when the victim access occurs. If the access occurs during the blind spot, the attacker will miss the access. After each victim access, we check if the attacker detects the access within 800 cycles. Otherwise, we consider the access to be missed. The relative blind spot is the fraction of missed accesses over all accesses. Assuming a uniform distribution of the accesses among the different stages of the attack loop, we can infer the length of the blind spot in cycles as the product of the relative blind spot length and the attack time. We perform an experiment with 200 accesses for each attack type. The standard error of the average is $\leq 1\%$ for all tested attacks. The results are summarized in Table 4.5.

For Flush+Reload, 75.1 % (SR) and 74.3 % (ER) of accesses are not detected, which equates to a blind spot of 461.2 cycles (SR) and 343.8 cycles (ER). Demote+Reload performs significantly better with 42.8 % (SR) and 42.6 % (ER) of accesses going undetected, equating to a blind spot of 181.6 cycles (SR) and 136.6 cycles (ER). L1 Evict+Reload further reduces the fraction of undetected accesses to 21.4 % (SR) and 15.5 % (ER) and a blind spot of 83.5 cycles (SR) and 52.7 cycles (ER). The correlation between blind spot and required memory access time on a miss can be explained by the fact that victim accesses, issued after the attacker’s memory access has started, are missed. The second memory access does not reduce the timing of the current attacker’s memory access and the subsequent cache state restoration removes the traces of the victim access.

In stark contrast to all previous attacks, we did not detect any missed victim accesses for both Flush+Flush and Demote+Demote, leading to a blind spot of 0 cycles on both tested CPUs. The detection measurement and cache state restoration are performed

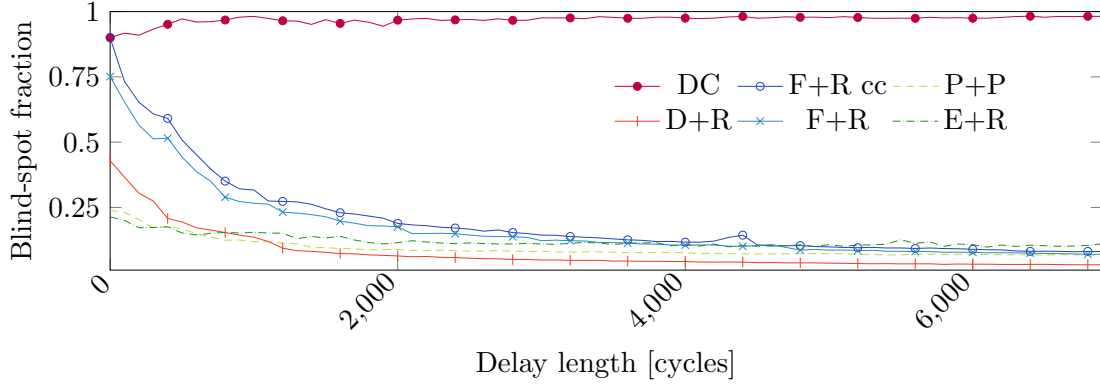
Table 4.5: Blind Spot: Average relative blind spot in percentage of detected accesses and estimated absolute blind spot in cycles for all tested attacks on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

Attack		Relative Blind Spot		Absolute Blind Spot	
CPU		SR	ER	SR	ER
SMT	Flush+Reload	75.1 %	74.3 %	461.2 C	343.8 C
	Demote+Reload	42.8 %	42.6 %	181.6 C	136.6 C
	L1 Evict+Reload	21.4 %	15.5 %	83.5 C	52.7 C
	Flush+Flush	0 %	0 %	0 C	0 C
	Demote+Demote	0 %	0 %	0 C	0 C
	L1 Prime+Probe	23.9 %	24.8 %	67.2 C	60.8 C
CC	Flush+Reload	89.9 %	86.3 %	552.1 C	399.4 C
	Flush+Flush	0 %	0 %	0 C	0 C
	DemoteContention	90.1 %	89.5 %	167.4 C	123.1 C

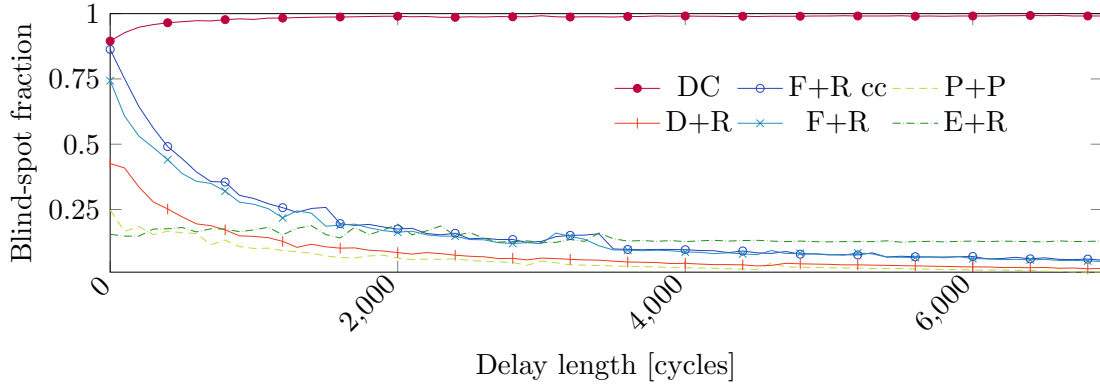
within a single instruction in both attack techniques. If the victim access is performed before this instruction, the access is detected in the current attack loop iteration. If the victim access is performed after this instruction, the access will be detected in the next attack loop iteration. Flush+Reload, Demote+Reload and Evict+Reload require additional instructions to restore the cache state. Hence, victim accesses, executed after the detection instruction but before the cache state is restored, are not detected.

Like Flush+Flush and Demote+Demote, Prime+Probe also does not perform a separate attacker access on the target address. However, the observed blind spot lengths of Prime+Probe are surprisingly high, with 23.9 % amounting to 67.2 cycles (SR) and 24.8 % corresponding to 60.8 cycles (ER). The measured blind spot is larger than of Evict+Reload. In a noise-free system, a victim access, performed before the last eviction address access, is detected in the current iteration. A victim access, performed after the last eviction address access, will be detected in the next iteration. One possible explanation for the blind spot of Prime+Probe is that the eviction set addresses are being prefetched. While we ensured that we chose eviction set addresses on different pages, further steps may be necessary to disturb the prefetcher on these recent Xeon CPUs. Our experimental setup does not allow us to determine where the blind spot lies within the attack loop. We only know the overall fraction of missed accesses. Hence, we cannot separate misses due to noise from misses due to cache restoration by the attacker.

In the cross-core application, the blind spot for Flush+Reload increases to 89.9 % (SR) and 86.3 % (ER) corresponding to 552.1 cycles (SR) and 399.4 cycles (ER). Cross-core Flush+Flush behaves the same way as in the SMT case, resulting in a blind spot of approximately 0 % and 0 cycles. DemoteContention operates based on contention and, therefore, does not share the zero blind spot of Demote+Demote. DemoteContention has



(a) Sapphire Rapids



(b) Emerald Rapids

Figure 4.5: Evolution of blind spot with increased delay in the attack loop.

a blind spot of 90.1 % (SR) and 89.5 % (ER), amounting to 167.4 cycles (SR) and 123.1 cycles (ER). The attack is highly dependent on the scheduling of the memory accesses and `cldemote` instructions in the memory controller. If the execution of the attacker `cldemote` instruction is delayed due to scheduled victim memory accesses and `cldemote` instructions, we can detect an increase in the attacker `cldemote` execution time. If the attacker `cldemote` instruction is issued after the victim operations have already finished or is scheduled before the victim operations, we cannot detect the victim operations. Hence, in contrast to Flush+Reload, Demote+Reload and Evict+Reload the blind spot length is not determined by the time to restore the cache state but by the time window during which contention is observable.

In a second step, we delay each iteration of the attacker's loop, thereby increasing the attack time while keeping the absolute blind spot in cycles constant for most attacks. Hence, for most attacks, adding more delay should cause the relative blind spot to converge to 0 %. The results are visualized in Figure 4.5. Flush+Flush and Demote+Demote are excluded from the diagram since their blind spot is 0 % irrespective of an

added delay. As expected, the blind spot approaches 0 % with increasing delay time for most attacks.

In contrast, the relative blind spot of DemoteContention approaches 100 %. As DemoteContention relies on contention, the effect of the victim operations is only observable within a short time frame. Adding delay in the attack loop increases the time between victim operations and detection code on average. Hence, the victim access becomes less observable and the blind spot approaches 100 % with increasing delay time. In other words, DemoteContention is the only technique among our tested attacks that is rendered unusable by an increased delay time in the attack loop. Prime+Probe retains the second-highest blind spot with a large delay length. Since its blind spot is likely due to noise instead of cache state restoration time, increased delay does not provide a significant benefit.

Takeaway: Flush+Flush and Demote+Demote clearly outperform all other attack techniques with a blind spot of approximately 0 % due to their single instruction measurement and cache state restoration. DemoteContention has the largest relative blind spot due to its dependence on subsequent memory controller scheduling. Cross-core Flush+Reload has the largest absolute blind spot due to the time-intensive RAM access and cross-core L1 flush. The absolute blind spot of Demote+Reload is less than half of Flush+Reload due to the reduced cache state restoration time.

4.3 Covert-Channel Metrics

In this section, we construct single- and multi-bit covert channels for each cache attack technique. We transmit randomly generated data in a standardized setup and measure the true transmission capacity. We then perform experiments to assess the noise resilience of each covert channel.

4.3.1 Setup

We construct a one-way covert channel that transmits data by performing operations on a shared memory address or several addresses in the same cache set. We want to measure the maximum covert channel capacity that two fully cooperating threads can achieve. Hence, we assume that the sender and receiver are synchronized at the beginning of the transmission. To maintain synchronization throughout the transmission period, we define a fixed-size transmission window during which the sender transmits a single bit or n bits for the multi-bit channel, respectively. In other words, the sender transmits the generated data one (n) bit(s) at a time in consecutive time windows without any feedback from the receiver. For transmitting multiple bits in parallel, we use addresses that are 4288 bytes apart so that they are resident on different virtual pages and do not

interfere with each other by being mapped to the same cache set. The sender performs the corresponding memory accesses on each address in sequence.

As measured in Section 4.2.3, there is some delay between the time of a sender operation and the time at which the receiver detects the operation. To account for this, we split the window size into two parts: a setup offset and a reading offset. At the beginning of the transmission, the receiver sleeps for the period of the setup offset, allowing the sender to perform the memory operation. The receiver then executes its detection code and stores a potential hit. After the reading offset has passed, the first window is complete and the sender sends the next bit(s). Figure 4.6 visualises the window slices.

Once the transmission is complete, we calculate the true capacity of the covert channel. The raw capacity C_{raw} measures the number of transmitted bits per second. It is solely dependent on the transmission window size. The bit error rate is defined as

$$p = \frac{\text{incorrectly transmitted bits}}{\text{transmitted bits}}.$$

We count the number of correctly transmitted bits via a bitwise comparison of the received and original data. The true capacity quantifies the number of correctly transmitted bits per second. Cover et al. [11] define the true capacity as

$$C_{\text{True}} = C_{\text{raw}}(1 + (1 - p) * \log_2(1 - p) + p * \log_2(p)).$$

We do not perform any error correction, as this is not influenced by the underlying technique used in the covert channel. Since it takes both raw capacity and error rate into account, the true capacity is a suitable measure to compare the capacities the different techniques can achieve. Any additional protocols layered on top will scale accordingly.

For Flush+Reload, Demote+Reload and Evict+Reload, we also implement a faster variation, which we hereafter refer to as optimized Flush+Reload, optimized Demote+Reload and optimized Evict+Reload. In the standard attacks, the receiver brings the cache line to a slower memory level, the sender performs a memory access and the receiver measures access time. Hence, the window size of each iteration allows for two memory accesses and one `clflush`/`cldemote`/`evict` operation. In the attacker-victim scenario, we assumed that the victim performs a sensitive memory address. In contrast, we assume that the sender and receiver fully cooperate in a covert channel. Hence, we are not limited by the actions that the sender can perform. We can remove the overhead of the additional memory access by letting the sender perform the `clflush`/`cldemote`/`evict` operation while the receiver only repeatedly measures access time.

4.3.2 Optimizing Offsets

To optimize transmission capacity, we choose the window size as small as possible. However, to prevent errors, we need to ensure that neither the setup offset is too small

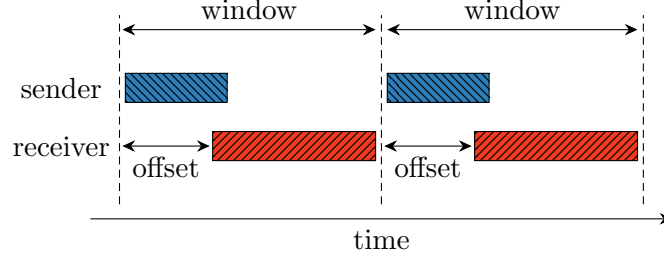


Figure 4.6: Window slices of the generic covert channel during which one or multiple bits are transmitted.

(and hence the receiver misses a sender operation) nor the reading offset is too small (and hence the receiver mistakenly reads the next bit instead of the current one). Currently, there is no fast and fully automated framework to determine the optimal setup and reading offsets. We perform a mix of automated fixed-sized increments and manual inspections to determine suitable values for the offsets.

Noise-sensitive attack techniques cannot be pushed under a certain error rate independently of the window size. Hence, we start the experiments with large values for both offsets to get a lower bound for the error rate. Theoretically, a large window size would ensure no errors due to timing asynchronies, minimizing errors but limiting the absolute throughput capacity. Next, we keep a large value for the setup offset but start with a minimal reading offset. We increase the reading offset in several steps and measure the true capacity with each setting. While the reading offset is still too small we will get random results with an error rate of approximately 50%. Once the window is large enough to accommodate the majority of reading attempts, the true capacity peaks quickly, revealing the optimal offset. If the true capacity drops with an increased reading offset and the maximum capacity is above the lower bound, that we have measured for a large reading offset, we have found the optimum. We then repeat the same for the setup offset. Once we have obtained the optimal offsets and capacity for the single-bit channel, we increase parallelism and repeat the optimization.

We optimized the offsets with the CPU frequency scheduler set to “Performance” for reliability. We tested that the obtained optimal capacities remain the same if the CPU scheduler is the default frequency governor. Before the first measurement, we perform a warm-up, during which the frequency is scaled up to provide the optimal settings for the covert channel.

4.3.3 Capacities and Error Rates

With the optimized offsets from the previous section we perform 100 measurements per attack technique. We transmit one megabit of randomly generated data and measure the error rate and resulting true capacity. The averaged single- and multi-bit channel

Table 4.6: Single and multi-bit covert channel error rates and true capacities for all tested attack techniques on our Sapphire Rapids (SR) and Emerald Rapids (ER) CPUs.

Attack		Single-bit Channel				n-bit Channel				
		Cap. [Mbit/s]		Error rate		Opt. n	Cap. [Mbit/s]		Error rate	
CPU		SR	ER	SR	ER	SR/ER	SR	ER	SR	ER
SMT	Flush+Reload	2.35	2.01	3.4 %	3.2 %	1	2.35	2.01	3.4 %	3.2 %
	Demote+Reload	6.38	6.09	0.7 %	0.7 %	1	6.38	6.09	0.7 %	0.7 %
	L1 Evict+Reload	2.51	3.32	6.8 %	1.4 %	1	2.51	3.32	6.8 %	1.4 %
	Opt. Flush+Reload	5.58	5.31	1.8 %	1.7 %	12	9.17	9.24	0.6 %	1.0 %
	Opt. Demote+Reload	11.93	11.10	0.2 %	0.6 %	12	15.48	17.03	2.0 %	1.4 %
	Flush+Flush	4.56	5.03	1.1 %	2.1 %	~ 800	8.55	9.03	5.6 %	2.6 %
	Demote+Demote	6.03	8.17	0.7 %	1.7 %	~ 10000	8.34	9.36	4.6 %	2.3 %
	L1 Prime+Probe	3.62	3.78	10.9 %	2.0 %	1	3.62	3.78	10.9 %	2.0 %
CC	Flush+Reload	1.94	2.15	2.2 %	3.6 %	1	1.94	2.15	2.2 %	3.6 %
	Opt. Flush+Reload	5.42	6.51	0.7 %	2.2 %	12	9.42	10.01	0.3 %	1.2 %
	Flush+Flush	4.43	5.74	2.7 %	2.0 %	~ 800	5.51	10.26	9.5 %	1.9 %
	DemoteContention	0.18	0.19	5.9 %	1.9 %	1	0.18	0.19	5.9 %	1.9 %

capacities for each attack are summarized in Table 4.6. While the single-bit capacity is a measure of the information leakage rate of the attack on a single vulnerable address, the multi-bit channel provides an upper limit on the transmission rate that fully cooperating threads can achieve. A high true channel capacity means that information can be transmitted quickly and reliably. The standard error is ≤ 0.2 Mbit/s for all tested attacks.

Single-bit SMT Flush+Reload has error rates of 3.4 % (SR) and 3.2 % (ER) and the lowest true capacities of all tested SMT attacks with 2.35 Mbit/s (SR) and 2.01 Mbit/s (ER). In stark contrast, single-bit SMT Demote+Reload only has an error rate of 0.7 % and a capacity of 6.38 Mbit/s (SR) and 6.09 Mbit/s (ER). Single-bit SMT L1 Evict+Reload has error rates of 6.8 % (SR) and 1.4 % (ER) with true capacities of 2.51 Mbit/s (SR) and 3.32 Mbit/s (ER). For all three attack techniques, transmitting several bits in parallel does not improve the capacity. Flush+Reload has the fastest covert channel, followed by Demote+Reload and Evict+Reload.

We expect higher capacities for the optimized versions of these attack techniques due to shorter required offsets. In fact, both optimized single-bit SMT attack techniques also show a lower error rate than their corresponding standard versions. Optimized single-bit SMT Flush+Reload has considerably reduced error rates of 1.8 % (SR) and 1.7 % (ER) with increased true capacities of 5.58 Mbit/s (SR) and 5.31 Mbit/s (ER). However, the bigger advantage is that we can utilize parallelization in this setting. Optimized 12-bit SMT Flush+Reload has error rates of merely 0.6 % (SR) and 1.0 % (ER) and higher true capacities than non-optimized Flush+Reload, Demote+Reload and Evict+Reload with 9.17 Mbit/s (SR) and 9.24 Mbit/s (ER). Optimized single-bit SMT Demote+Reload

further reduces the low error rates of the standard version to 0.2 % (SR) and 0.6 % (ER). These constitute the minimum error rates among all tested attacks. However, note that we did not minimize for error rates but maximized for true capacity. Hence, all reported error rates are not the achievable minimum for each attack but rather the error rate at maximum true capacity. With 11.93 Mbit/s (SR) and 11.10 Mbit/s (ER), the true capacities of optimized Demote+Reload are the maximum single-bit channel capacities among all tested techniques. Similarly to Flush+Reload, the optimal parallelism consists of 12 parallel bits. The optimized multi-bit SMT Demote+Reload channel has increased error rates of 2 % (SR) and 1.4 % (ER), but the best overall true capacities at 15.48 Mbit/s (SR) and 17.03 Mbit/s (ER).

Single-bit SMT Flush+Flush has error rates of 1.1 % (SR) and 2.1 % (ER) with true capacities of 4.56 Mbit/s (SR) and 5.03 Mbit/s (ER). A small amount of parallelization increases noise and reduces the true capacity. However, with a large amount of parallelization of 800 bits, the noise can be offset and higher capacities of 8.55 Mbit/s (SR) and 9.03 Mbit/s (ER) are achieved, almost double the single-bit capacities. The multi-bit error rates amount to 5.6 % (SR) and 2.6 % (ER). Single-bit SMT Demote+Demote also beats its `clflush`-based counterpart with error rates of 0.7 % (SR) and 1.7 % (ER) and true capacities of 6.03 Mbit/s (SR) and 8.17 Mbit/s (ER). Similarly to Flush+Flush, a high amount of parallelization improves the capacity further. At 10000 bits, we measure error rates of 4.6 % (SR) and 2.3 % (ER) with true capacities of 8.34 Mbit/s (SR) and 9.36 Mbit/s (ER), demonstrating similar capacity as multi-bit Flush+Flush. Single-bit SMT L1 Prime+Probe has high error rates of 10.9 % (SR) and 2.0 % (ER), but its true capacities of 3.62 Mbit/s (SR) and 3.78 Mbit/s (ER) still beat standard Flush+Reload and Evict+Reload. Parallelization does not improve the capacity.

In the cross-core case, the single-bit Flush+Reload true capacity drops to 1.94 Mbit/s (SR) with an error rate of 2.15 % (SR). However, on our Emerald Rapids CPU, the cross-core single-bit Flush+Reload capacity is slightly larger than with SMT at 2.15 Mbit/s with an error rate of 3.6 % (ER). As in the SMT case, parallelization brings no improvement. Cross-core single-bit optimized Flush+Reload has error rates of 0.7 % (SR) and 2.2 % (ER) and true capacities of 5.42 Mbit/s (SR) and 6.51 Mbit/s (ER). The 12-bit channel has error rates of 0.3 % (SR) and 1.2 % (ER) and true capacities of 9.42 Mbit/s (SR) and 10.01 Mbit/s (ER). Cross-core single-bit Flush+Flush has error rates of 2.7 % (SR) and 2.0 % (ER) and true capacities of 4.43 Mbit/s (SR) and 5.74 Mbit/s (ER) roughly comparable with the SMT case. Parallelisation with 800 bits improves the capacities to 5.51 Mbit/s (SR) and 10.26 Mbit/s (ER) with error rates of 9.5 % (SR) and 1.9 %. As the highest performing optimized Demote+Reload technique cannot be used in the cross-core scenario, the multi-bit optimized Flush+Reload (SR) and multi-bit Flush+Flush (ER) channels have the highest true capacities in the cross-core case. Finally, cross-core single-bit DemoteContention has error rates of 5.9 % (SR) and 1.9 % with true capacities of 0.18 Mbit/s (SR) and 0.19 Mbit/s (ER). Due to the L3 cache set granularity of the technique, parallelization does not improve the true capacity.

Takeaway: Our optimized Demote+Reload channel clearly outperforms the other attacks in the SMT scenario. In the cross-core scenario, optimized Flush+Reload and Flush+Flush have the highest true capacities. Only optimized Flush+Reload, optimized Demote+Reload, Flush+Flush and Demote+Demote benefit from parallelization. Demote+Demote outperforms Flush+Flush in the single-bit channel, while the optimum multi-bit channels have similar capacities.

4.3.4 Noise Resistance

Noise Resistance quantifies how well a covert channel performs under memory stress. If the channel retains a high transmission capacity under noise, the attack works well in realistic setups. The maximum capacities in the previous section have been measured under optimal conditions with minimal noise from other user programs. In a realistic scenario, other programs generate noise by performing memory accesses while our covert channel is running. This can lead to cache evictions of our target or eviction addresses or contention in the L3 directory, resulting in misclassification errors. To test the resilience of the different attack techniques to memory access noise, we measure the true channel capacity while adding stress-ng worker threads that access random memory locations.

For each number of running stress threads, we perform 100 (for DemoteContention 50 due to large window size) measurements transmitting one megabit of randomly generated data. The standard error is ≤ 0.2 Mbit/s for all tested attacks. The averaged results are visualized in Figure 4.7 and Figure 4.8. As expected, all channels suffer from noise. There is a general trend for the biggest percentual capacity loss per added thread for a small number of threads. A small amount of noise drastically reduces the channel's capacity compared to optimal conditions. After ten stress workers, introducing additional noise does not have a significant impact on the channel capacity.

Under maximum stress of 20 threads, optimized Demote+Reload performs the best both in terms of absolute true capacity as well as percentual loss relative to its capacity under optimal conditions. With true capacities of 5.33 Mbit/s (SR) and 2.84 Mbit/s (ER) under maximum stress optimized Demote+Reload retains 46.3 % (SR) and 27.7 % (ER) of its original capacity. optimized SMT Flush+Reload still retains 19.7 % (1.1 Mbit/s) (SR) and 24.7 % (1.34 Mbit/s) (ER) of its original capacity under maximum stress. Regarding the standard SMT versions, Demote+Reload sees a larger percental loss but still performs best in terms of absolute capacity with 14.8 % (0.96 Mbit/s) (SR) and 9.3 % (0.57 Mbit/s) (ER) while Flush+Reload retains only 4.7 % (0.11 Mbit/s) (SR) and 18.4 % (0.37 Mbit/s) (ER). On our Sapphire Rapids CPU, Evict+Reload is very susceptible to noise with only 1 % of its optimal capacity (0.03 Mbit/s) under maximum stress while it still retains 13.1 % (0.43 Mbit/s) on our Emerald Rapids CPU. SMT Demote+Demote retains 20.9 % (1.23 Mbit/s) (SR) and 15.9 % (1.29 Mbit/s) (ER), Flush+Flush 12.3 % (0.56 Mbit/s) (SR) and 16.0 % (0.81 Mbit/s) (ER) and Prime+Probe 3.6 % (0.13 Mbit/s) (SR) and 18.5 % (0.70 Mbit/s) (ER).

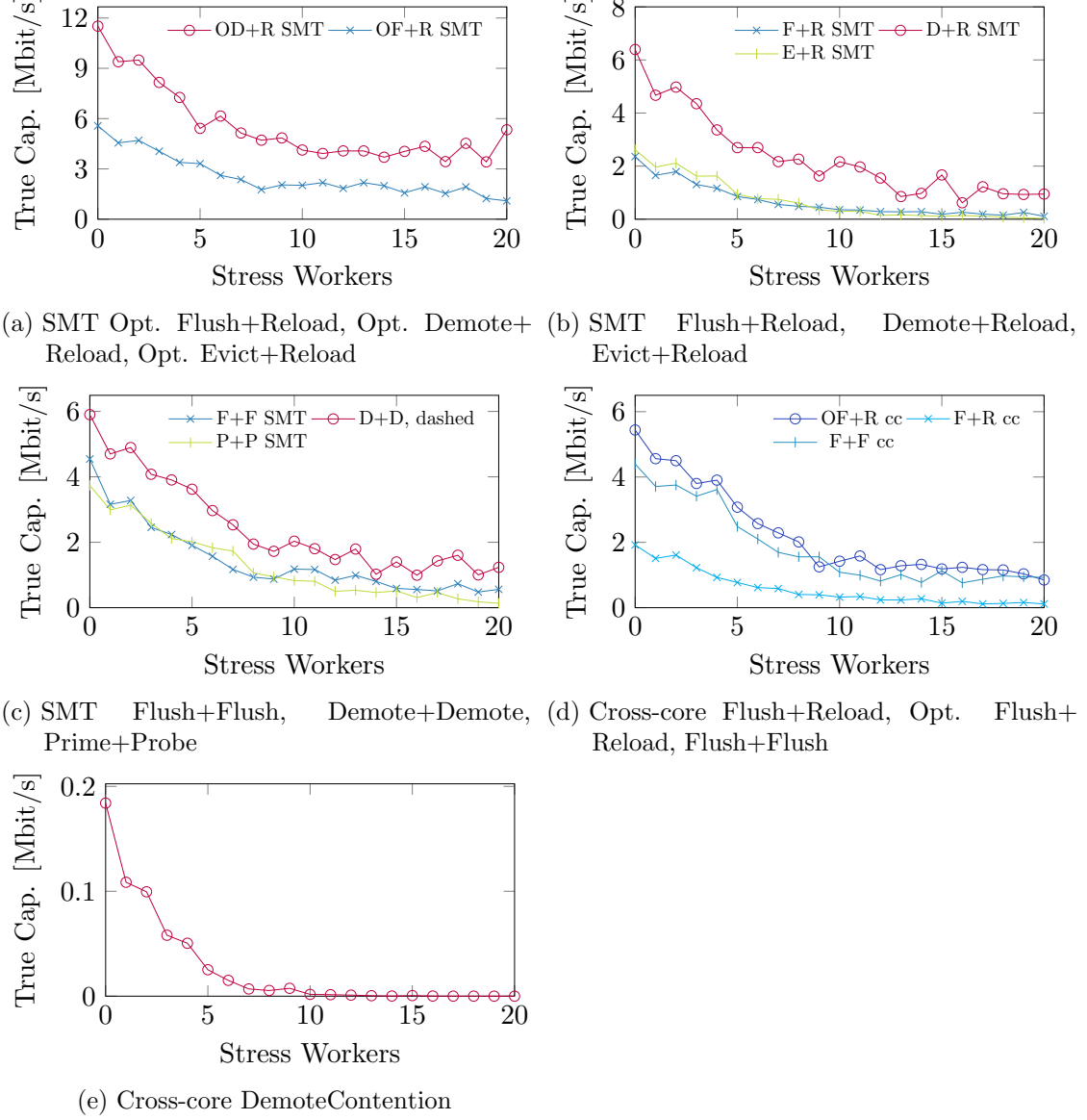


Figure 4.7: Evolution of true capacity of all tested covert channels with an increasing number of memory stress threads (up to number of virtual cores) on our Sapphire Rapids CPU.

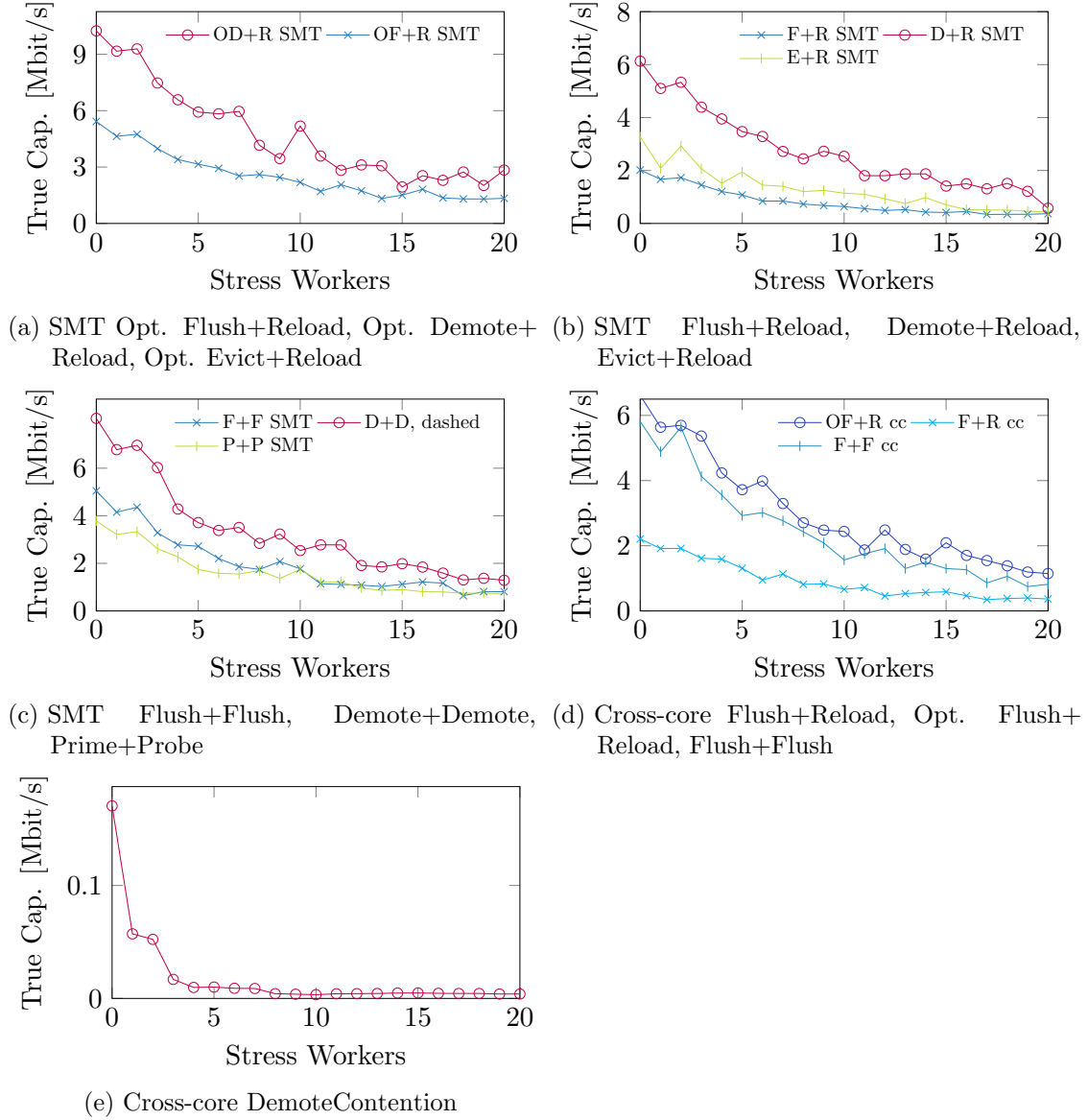


Figure 4.8: Evolution of true capacity of all tested covert channels with an increasing number of memory stress threads (up to number of virtual cores) on our Emerald Rapids CPU.

Cross-core optimized Flush+Reload retains slightly less than the SMT version with 15.6 % (0.85 Mbit/s) (SR) and 17.4 % (1.15 Mbit/s) (ER). Flush+Reload performs considerably worse than its SMT equivalent with 6 % (0.12 Mbit/s) (SR) and 16.7 % (0.37 Mbit/s) (ER). Flush+Flush outperforms its SMT version on our Sapphire Rapids CPU with 21 % (0.93 Mbit/s) while achieving similar results on our Emerald Rapids CPU with 14.1 % (0.82 Mbit/s). DemoteContention has a negligible capacity under maximum stress.

Takeaway: All attacks suffer from noise. Optimized Demote+Reload still performs the best under maximum stress, followed by Demote+Demote and optimized Flush+Reload. Among the cross-core attacks, Flush+Flush and optimized Flush+Reload are the most noise-resilient.

4.3.5 Limitations

The process of optimizing the window offsets is non-trivial to automate in a fast manner since small transmission sizes lead to unreliable results due to noise. There is no defined upper limit for the transmission window, especially when transmitting a large number of bits in parallel. Hence, stepping through the entire possible range in large steps is infeasible. Instead, we increase the transmission window and terminate when we have reached a local optimum, in addition to some sanity checks regarding the error rate to ensure we are not terminating too early. However, we cannot guarantee that this is the global optimum. We use a granularity of 10 cycles. For different runs of the automated script, we obtain offsets that vary but mostly obtain similar capacities. The range of reasonable parallelism to test varies for the different attack techniques. Flush+Flush and Demote+Demote perform well with many bits transmitted in parallel. Performing window size optimization for each increment of parallelism is infeasible.

We only transmit each bit once (except for DemoteContention) and perform a single measurement in the receiver per bit. An alternative approach would be to continuously retransmit and remeasure the bit value during the entire timeslice of the transmission window, averaging the result. However, for optimal true capacity, the window sizes are likely reduced to allow only a single transmission in practice. Hence, a majority vote of repeated measurements in the receiver is unlikely to achieve a higher capacity. Therefore, we did not extensively compare the two approaches for each attack technique.

For some attack techniques, we observed varying capacities depending on which cores the sender and receiver were running and for different offsets of the target address used for communication within a page. In particular, the optimal offset within a page changed upon rebooting the machine. We counteracted these variations by testing 64-byte (cache line size) aligned offsets within a page and selecting the optimal offset for the experiments. In addition, for Demote+Demote, we sometimes observed a considerable number of iterations during which the transmission capacity only amounted to complete randomness before the transmission started to resume at optimal capacity. We discarded these measurements.

For setup performance reasons, we implemented a second script in which the sender and receiver are implemented via threads instead of a forked program. Curiously, here the parallelization of optimized Demote+Reload and Flush+Reload did not achieve higher frequencies, but most attack techniques achieved slightly higher capacities with the second script. We selected the best performance from the two scripts for each attack technique.

4.4 Detectability

As discussed in Section 2.3, cache timing side channels are difficult to prevent entirely, but an ongoing attack can be detected via performance counters. In this section, we compare the detectability of the tested attacks. We run 10^6 experiments executing the attack loop and record the changes in performance counter values. They are listed in Table 4.7 and Table 4.8. We repeat the measurements with an additional victim access. The results are summarized in Table 4.9 and Table 4.10. A large increase in performance counter values compared to the base case indicates that the attack is detectable by observing performance counters.

In the no-victim-access case, Flush+Reload induces approximately 10^6 L1, L2 and L3 misses compared to the base case. In each iteration, the attack loop accesses an address which has previously been flushed to the main memory. Demote+Reload induces a similar number of L1 and L2 misses and no L3 misses since the data is demoted to the L3 cache before being accessed in each iteration. L1 Evict+Reload causes approximately $13 \cdot 10^6$ L1 misses compared to the base case. As above, we have one L1 miss per iteration for the measurement access on the evicted target address. In addition, this access evicts one of the eviction addresses. Assuming an LRU replacement policy, the eviction address, accessed first in the loop, is evicted by the target address access. Then, this effect ripples down to all eviction addresses as the first accessed eviction address produces an L1 miss and the load into the L1 cache evicts the second eviction address. Hence, for the 12-way associative L1 cache of our Sapphire Rapids and Emerald Rapids CPUs, this explains the approximate $12 \cdot 10^6$ additional L1 misses.

While Prime+Probe performs L1 accesses, it omits the target address access of Evict+Reload. Therefore, the attack loop of Prime+Probe only generates misses with additional memory access noise on the cache set. The attack loops of Flush+Flush, Demote+Demote and DemoteContention do not induce any memory accesses. Therefore, we do not record a higher number of L1, L2 or L3 misses in the no-victim access case.

In the victim access case, the performance counter values for SMT Flush+Reload, Demote+Reload and Evict+Reload are comparable to those in the no-victim-access case. In each iteration, there is an additional victim access inducing an L1 miss (and L2 and L3 miss accordingly). However, the subsequent attacker access on the target address is a

hit compared to a miss in the no-victim access case. Hence, the total number of cache misses is equal in both scenarios.

In contrast, the performance counter values for Flush+Flush, Demote+Demote and Prime+Probe differ significantly from the no-victim-access case. Flush+Flush now displays approximately 10^6 L1, L2 and L3 misses induced by the victim access in each iteration like in Flush+Reload. Analogously, Demote+Demote induces approximately 10^6 L1 and L2 misses like Demote+Reload and Prime+Probe causes approximately $13 \cdot 10^6$ L1 misses like Evict+Reload.

The cross-core version of Flush+Reload has an even bigger impact on the measured L1 and L2 misses since accesses are issued on both the attacker and victim core, doubling the number of L1 and L2 misses. Cross-core Flush+Flush only induces L1 and L2 cache misses on the victim's core and hence the number of L1, L2 and L3 cache misses is comparable with SMT Flush+Flush. DemoteContention does not induce a substantial amount of additional cache misses.

The measured performance counters of retired branches differ between our two CPUs. On our Emerald Rapids CPU, all SMT attacks show a small increase of around approximately $2.5 \cdot 10^5$ retired branches, both with and without a victim access. In contrast, on our Sapphire Rapids CPU, all SMT attacks show a larger number of retired branches in the victim-access case with SMT Flush+Reload and SMT Flush+Flush inducing approximately 10^6 retired branches with a victim access.

Takeaway: DemoteContention is the only technique that cannot be detected in both the no-victim-access case as well as in the victim-access case. All other attacks are detectable by at least an increase in L1 misses in the victim access case. Flush+Flush, Demote+Demote and Prime+Probe cannot be detected in the no-victim-access case.

Table 4.7: Changes in performance counter values after 10^6 attack iterations **without** a victim access on our Sapphire Rapids CPU. The given values are relative to the base measurement.

	Attack	L1 misses	L2 misses	L3 misses	Retired Branches
	SMT Base	2,544,058	846,906	1713	11,334,083
SMT	Flush+Reload	+1,223,771	+977,567	+999,994	+347,032
	Demote+Reload	+1,102,440	+827,485	+122	+582,176
	Evict+Reload	+12,467,912	-839,654	+62	+161,526
	Flush+Flush	-413,853	-137,328	-73	-552,494
	Demote+Demote	-240,919	-129,447	-147	-177,470
	Prime+Probe	-431,491	-840,369	+405	+21,577
	CC Base	4,004,011	11,758	1834	12,101,850
CC	Flush+Reload	+960,548	+998,644	+999,977	+87,403
	Flush+Flush	-164,235	+954	+89	+25,456
	DemoteContention	+3298	-2783	-143	+53,339

Table 4.8: Changes in performance counter values after 10^6 attack iterations **without** a victim access on our Emerald Rapids CPU. The given values are relative to the base measurement.

	Attack	L1 misses	L2 misses	L3 misses	Retired Branches
	SMT Base	47,121	11,299	1496	10,630,295
SMT	Flush+Reload	+1,028,165	+1,004,819	+1,000,035	+252,268
	Demote+Reload	+1,005,858	+1,000,316	-159	+253,894
	Evict+Reload	+12,975,982	-353	-28	+251,063
	Flush+Flush	-6412	-1577	-50	+253,820
	Demote+Demote	-2882	-2163	-129	+255,864
	Prime+Probe	-5633	-1816	+257	+244,627
	CC Base	2,895,991	2,886,008	1412	11,744,772
CC	Flush+Reload	+1,146,501	+1,140,887	+1,000,101	-31,445
	Flush+Flush	-100,893	-101,813	+369	-30,212
	DemoteContention	+92,784	+89,428	-74	+36,516

Table 4.9: Changes in performance counter values after 10^6 attack iterations with a **victim access** on our Sapphire Rapids CPU. The given values are relative to the base measurement.

	Attack	L1 misses	L2 misses	L3 misses	Retired Branches
	SMT Base	2,463,241	852,141	1457	11,210,752
SMT	Flush+Reload	+1,522,511	+1,139,838	+1,000,359	+1,029,568
	Demote+Reload	+1,292,874	+959,524	+266	+681,630
	Evict+Reload	+12,547,320	-845,456	+723	+619,805
	Flush+Flush	+1,512,337	+1,133,265	+1,000,201	+1,002,832
	Demote+Demote	+1,322,684	+975,733	+194	+559,293
	Prime+Probe	+12,547,209	-845,455	+605	+436,735
	CC Base	4,010,026	10,842	1797	12,130,563
CC	Flush+Reload	+2,875,273	+2,891,790	+999,963	+77,215
	Flush+Flush	+1,900,492	+1,891,532	+1,000,180	+71,113
	DemoteContention	+2427	-1647	+91	+9305

Table 4.10: Changes in performance counter values after 10^6 attack iterations with a **victim access** on our Emerald Rapids CPU. The given values are relative to the base measurement.

	Attack	L1 misses	L2 misses	L3 misses	Retired Branches
	SMT Base	55,672	13,004	1582	10,629,320
SMT	Flush+Reload	+1,012,583	+999,942	+999,809	+255,874
	Demote+Reload	+997,487	+998,062	-173	+256,066
	Evict+Reload	+12,965,693	-1816	-13	+256,679
	Flush+Flush	+995,793	+998,169	+999,729	+258,162
	Demote+Demote	+990,042	+998,046	-57	+253,159
	Prime+Probe	+12,964,919	-709	-112	+252,015
	CC Base	2,933,579	2,922,245	1452	11,682,440
CC	Flush+Reload	+2,511,172	+2,507,127	+999,966	+65,809
	Flush+Flush	+1,358,337	+1,358,670	+999,578	+59,821
	DemoteContention	+45,586	+41,914	+16	+63,115

Chapter 5

Discussion

In the previous chapters, we presented a new family of side-channel attacks built on the `cldemote` instruction. We systemized existing and novel timing side-channel attacks according to nine metrics, in most of which our new attacks rank at the top. With our novel attacks, we are extending and improving the existing family of cache timing side channels. Our optimized Demote+Reload channel is at least 69 % faster than the best existing SMT channel technique and has the lowest error rate. The channel also retains the highest capacity under memory access noise. In the SMT scenario, our novel Demote+Demote attack outperforms existing techniques concerning temporal precision and no victim access attack time. Demote+Demote and is on par with the best existing blind spot and spatial precision. Since the `cldemote` instruction does not affect the L1 and L2 caches of different cores, the topology of the novel attacks is more constrained. With DemoteContention we demonstrate a cross-core attack. However, it is not as powerful and generic slot-in replacement for Flush+Reload as in the SMT scenario. The attack technique assumes that the victim performs the `cldemote` instruction in addition to a memory access, posing higher requirements than the existing attacks. DemoteContention is the only attack not detectable in the victim access case, Demote+Demote is not detectable in the no victim access case on par with Flush+Flush.

Rauscher et al. [64] employ the novel attack techniques as a drop-in replacement for Flush+Reload in several case studies, including OpenSSL AES T-table attacks, inter-keystroke timing attacks, and KASLR breaks. While we do not present any new attack use cases that rely on the novel cache side-channel techniques, we showed that the novel attacks show an advantage over existing attacks in several metrics. Shorter attack times, high temporal precision and higher covert channel throughput provide the potential for future attacks that rely on these characteristics. For inter-keystroke timing attacks, the temporal precision of Flush+Reload is already sufficient since the speed of a human is orders of magnitude higher. Hence, the advantages of the novel attacks do not provide a significant benefit. Future work could include identifying use cases that require short attack time or high temporal precision for success.

Since Demote+Reload and Demote+Demote follow the same working principle as Flush+Reload and Flush+Flush, the discussed mitigations in Section 2.3 including limiting high-resolution timer access, restricting shared memory, removing the `clflush` or `cldemote`

Chapter 5 Discussion

instruction, respectively, utilizing a random fill cache architecture, detecting attacks via cache misses and following best practices in software development apply equally to these novel attacks. However, DemoteContention does not require shared memory and in contrast to all other tested attacks, it does not induce any cache misses. Hence, current detection approaches are not applicable to DemoteContention.

Chapter 6

Conclusion

We presented the SMT attacks Demote+Reload and Demote+Demote that are suitable drop-in replacements for Flush+Reload and Flush+Flush. In the cross-core case, the applications of the `cldemote`-based attacks are more restricted since `cldemote` does not reach cross-core private caches. However, with DemoteContention we introduced a new type of attack on non-inclusive LLC. In contrast to previous attacks on the cache directory, DemoteContention does not require reverse engineering of the structure and eviction strategy of the extended directory.

In a second step, we evaluated novel and existing cache timing attacks with respect to spatial and topological constraints, temporal metrics and covert channel metrics. We tested all attacks in a standardized setup. Our novel attacks rank at the top in most metrics. We showed that Flush+Reload has the largest hit-miss margin (124 - 232 cycles), but performs poorly with respect to attack time, detection delay, blind spot length and covert channel capacity due to its time-consuming attacker access. Our evaluation demonstrates that our novel technique Demote+Reload outperforms Flush+Reload in several metrics with a 30 % faster no-victim-access attack time, a 60 % smaller absolute blind spot length and a 2.7 - 3 times higher covert channel capacity. We demonstrated the fastest optimized multi-bit covert channel with our novel Demote+Reload attack with a channel capacity of 15.5 - 17.0 Mbit/s. We discovered that Flush+Reload and Demote+Demote are the only attacks with zero blind spot lengths since the attacker does not induce any memory accesses that can hide victim behavior. Both attacks have fast attack times (138 - 289 cycles), high temporal precision (34 - 45 ns, SD 15 - 19 ns) and fast covert channels (8.3 - 10.3 Mbit/s). Our measurements demonstrated that DemoteContention has the smallest hit-miss margin (not measurable), the largest relative blind spot length (90 %) and consequently the lowest channel capacity (0.2 Mbit/s). We showed that Flush+Reload and Flush+Flush are easily adaptable to the cross-core scenario, while Prime+Probe and DemoteContention are the only attacks with cache set granularity and do not require shared memory. In conclusion, our evaluation reveals that our novel attacks extend and improve upon the family of existing cache side-channel attacks and provides insights into the advantages and drawbacks of the individual attacks.

Bibliography

- [1] Andreas Abel and Jan Reineke. “uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures”. In: *ASPLOS*. 2019.
- [2] Alfred Brauer. “On Addition Chains”. In: *Bulletin of the American Mathematical Society* (1939).
- [3] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M Moya. “Cache misses and the recovery of the full AES 256 key”. In: *Applied Sciences* 9.5 (2019), p. 944.
- [4] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. “CaSym: Cache aware symbolic execution for side channel detection and mitigation”. In: *S&P*. 2019.
- [5] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *USENIX Security*. 2003.
- [6] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat”. In: *AsiaCCS*. 2020.
- [7] Marco Chiappetta, Erkey Savas, and Cemal Yilmaz. “Real time detection of cache-based side-channel attacks using Hardware Performance Counters”. In: *Cryptology ePrint Archive, Report 2015/1034* (2015).
- [8] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. “Real-time detection for cache side channel attack using performance counter monitor”. In: *Applied Sciences* 10.3 (2020), p. 984.
- [9] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. “The last mile: An empirical study of timing channels on seL4”. In: *CCS*. 2014.
- [10] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. “On the effectiveness of same-domain memory deduplication”. In: *EuroSec*. 2022.
- [11] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
- [12] Paul Dale. *AES: Make the No-ASM Constant Time Code Path Not the Default #17600*. 2022. URL: <https://github.com/openssl/openssl/pull/17600>.
- [13] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *USENIX Security*. 2013.

Bibliography

- [14] Luís Fiolhais, Manuel Goulão, and Leonel Sousa. “CoDi \$: Randomized Caches Through Confusion and Diffusion”. In: *IEEE Access* (2023).
- [15] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. “LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization”. In: *RAID*. 2017.
- [16] Lukas Gerlach, Simon Schwarz, Nicolas Faröß, and Michael Schwarz. “Efficient and generic microarchitectural hash-function recovery”. In: *S&P*. 2024.
- [17] 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.
- [18] Michael Misiu Godfrey and Mohammad Zulkernine. “Preventing Cache-Based Side-Channel Attacks in a Cloud Environment”. In: *IEEE Transactions on Cloud Computing* (2014).
- [19] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR”. In: *ESSoS*. 2017.
- [20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *CCS*. 2016.
- [21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.
- [22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security*. 2015.
- [23] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *S&P*. 2011.
- [24] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning”. In: *arXiv:1907.03651* (2019).
- [25] Nishad Herath and Anders Fogh. “These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security”. In: *Black Hat USA*. 2015.
- [26] Wei-Ming Hu. “Reducing Timing Channels with Fuzzy Time”. In: *S&P*. 1991.
- [27] Ralf Hund, Carsten Willems, and Thorsten Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *S&P*. 2013.
- [28] Intel. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. 2022. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html>.
- [29] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual Volume 1*. 2024.

Bibliography

- [30] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. 2024.
- [31] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. 2024.
- [32] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2024.
- [33] Intel. *Intel Pentium 4 and Intel Xeon Processor Optimization - Reference Manual*. 2001.
- [34] Intel. *Persistent Memory FAQ*. 2020. URL: <https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/persistent-memory-faq.html>.
- [35] Intel. *What Is the Difference in Cache Memory Between CPUs for Intel Xeon Scalable Processors?* 2024. URL: <https://www.intel.com/content/www/us/en/support/articles/000027820/processors/intel-xeon-processors.html>.
- [36] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "MASCAT: Stopping Microarchitectural Attacks Before Execution". In: *CODASPY* (2017).
- [37] 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.
- [38] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES". In: *RAID*. 2014.
- [39] D Jayasinghe, Jayani Fernando, R Herath, and Roshan Ragel. "Remote Cache Timing Attack on Advanced Encryption Standard and Countermeasures". In: *IEEE Information and Automation for Sustainability*. 2010.
- [40] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. "Side Channel Cryptanalysis of Product Ciphers". In: *ESORICS*. 1998.
- [41] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "StealthMem: system-level protection against cache-based side channel attacks in the cloud". In: *USENIX Security*. 2012.
- [42] 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.
- [43] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO*. 1996.
- [44] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times". In: *USENIX Security*. 2016.

Bibliography

- [45] 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.
- [46] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “NetCAT: Practical Cache Attacks from the Network”. In: *S&P*. May 2020.
- [47] Butler W. Lampson. “A Note on the Confinement Problem”. In: *Communications of the ACM* (1973).
- [48] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Cl  mentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security*. 2016.
- [49] 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.
- [50] Fangfei Liu and Ruby B. Lee. “Random Fill Cache Architecture”. In: *MICRO*. 2014.
- [51] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *S&P*. 2015.
- [52] Robert Martin, John Demme, and Simha Sethumadhavan. “TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks”. In: *ACM SIGARCH Computer Architecture News* (2012).
- [53] Cl  mentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aur  lien Francillon. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *RAID*. 2015.
- [54] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. “NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters”. In: *HASP*. 2018.
- [55] 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.
- [56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: the Case of AES”. In: *CT-RSA*. 2006.
- [57] Dan Page. “Partitioned Cache Architecture as a Side-Channel Defence Mechanism”. In: *Cryptology ePrint Archive, Report 2005/280* (2005).
- [58] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 2011.
- [59] Matthias Payer. “HexPADS: A Platform to detect “Stealth” Attacks”. In: *ESSoS*. 2016.
- [60] Colin Percival. “Cache Missing for Fun and Profit”. In: *BSDCan*. 2005.

Bibliography

- [61] Moinuddin K Qureshi. “CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping”. In: *MICRO*. 2018.
- [62] Moinuddin K Qureshi. “New Attacks and Defense for Encrypted-Address Cache”. In: *ISCA*. 2019.
- [63] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. “Resource Management for Isolation Enhanced Cloud Services”. In: *CCSW*. 2009.
- [64] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. “A Systematic Evaluation of Novel and Existing Cache Side Channels”. In: *NDSS*. 2025.
- [65] Gururaj Saileshwar and Moinuddin K. Qureshi. “MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design”. In: *USENIX Security*. 2021.
- [66] 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.
- [67] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *DIMVA*. 2017.
- [68] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clementine Lucie Noemie Maurice, Raphael Spreitzer, and Stefan Mangard. “KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks”. In: *NDSS*. 2018.
- [69] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *FC*. 2017.
- [70] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. “NetSpectre: Read Arbitrary Memory over Network”. In: *ESORICS*. 2019.
- [71] Martin Schwarzl, Erik Kraft, and Daniel Gruss. “Layered Binary Templating”. In: *ACNS*. 2023.
- [72] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. “Remote Page Deduplication Attacks”. In: *NDSS*. 2022.
- [73] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. “Limiting Cache-based Side-Channel in Multi-tenant Cloud using Dynamic Page Coloring”. In: *Dependable Systems and Networks Workshops (DSN-W)*. 2011.
- [74] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. “Timing Analysis of Keystrokes and Timing Attacks on SSH”. In: *USENIX Security*. 2001.
- [75] 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.

Bibliography

- [76] 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.
- [77] Leif Uhsadel, Andy Georges, and Ingrid M. R. Verbauwhede. “Exploiting Hardware Performance Counters”. In: *FDTC* (2008).
- [78] 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.
- [79] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. “Eliminating Fine Grained Timers in Xen”. In: *CCSW*. 2011.
- [80] Zhenghong Wang and Ruby B. Lee. “A Novel Cache Architecture with Enhanced Performance and Security”. In: *MICRO*. 2008.
- [81] 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.
- [82] 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.
- [83] John C Wray. “An Analysis of Covert Timing Channels”. In: *Journal of Computer Security* (1992).
- [84] 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.
- [85] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security*. 2014.
- [86] Kehuan Zhang and XiaoFeng Wang. “Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems”. In: *USENIX Security*. 2009.
- [87] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. “CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds”. In: *RAID*. 2016.
- [88] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. “HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis”. In: *S&P*. 2011.
- [89] Yinqian Zhang and MK Reiter. “Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud”. In: *CCS*. 2013.