A Systematic Evaluation of Novel and Existing Cache Side Channels

Fabian Rauscher
Graz University of Technology
fabian.rauscher@iaik.tugraz.at

Carina Fiedler
Graz University of Technology
carina.fiedler@student.tugraz.at

Andreas Kogler
Graz University of Technology
andreas.kogler.0x@gmail.com

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

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

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

I. Introduction

Modern CPUs have multiple cache levels with lower latencies and capacities in lower levels, closer to the execution core, and higher latencies and capacities at higher levels, further away from the core. Cache performance is so crucial that disabling caching on commodity systems effectively slows them down by multiple orders of magnitude. Caches cannot buffer all memory, meaning that some memory locations will be buffered and fast, whereas others have higher access times, introducing the problem of cache side-channel attacks: Which data is cached, is decided based on what was recently used, is frequently used, as well as what is predicted to be used in the near future. Hence, cache side-channel attacks measure the timing to then infer which memory locations were recently used [84], [26], when memory locations are used [26], [77], and which memory locations are predicted to be used [22], [42]. Side-channel attacks then use this information to infer the actual secrets that led to these memory accesses or predictions.

Several generic cache attack techniques have been discussed in the literature, with the most prominent examples being Prime+Probe [53] and Flush+Reload [84]. Flush+Flush [24] and Evict+Reload [26] are variations of Flush+Reload that can be beneficial in some use cases [43], [71]. A promising defense against Prime+Probe is the concept of randomized secure caches, minimizing [81], [57], [61] or even eliminating [19] the chance of priming and probing the cache successfully. However, flush-based attacks are typically excluded, *i.e.*, Flush+Reload and Flush+Flush, and instead, propose to disable `clflush` [81]. Other works try to mitigate flush-based attacks through detection [11], [27], [24]. Instead of introducing a secure cache, Intel decided to move to a non-inclusive L3 cache with their recent Sapphire Rapids microarchitecture. The non-inclusive L3 cache provides no guarantees on inclusiveness towards lower levels, effectively mitigating the possibility to evict cache lines from the private L1 and L2 caches of other cores. Hence, eviction-based cross-core attacks, e.g., Prime+Probe and Evict+Reload, are not possible anymore unless the victim's own activity repeatedly leads to data placement in the L3 cache. Furthermore, the addressing functions and replacement policy for Sapphire Rapids have not been reverse-engineered yet, and even with these steps, the input to these functions is physical addresses, *i.e.*, privileged information typically unavailable to an attacker.

In this paper, we present the first systematic evaluation using 9 characteristics of the 4 most relevant cache attacks, Flush+Reload, Flush+Flush, Evict+Reload, and Prime+Probe, as well as three new attacks that we introduce: Demote+Reload, Demote+Demote, and DemoteContention. We evaluate hit-miss margins, temporal precision, spatial precision, topological scope, attack time, blind spot length, channel capacity, noise resilience, and detectability. In a comprehensive comparison, we show that our new attacks, Demote+Reload and Demote+Demote, perform better on some and worse on other characteristics but yield overall a similar attack performance as known attacks. Demote+Reload has a $60.7\,\%$ smaller blind spot than

Network and Distributed System Security (NDSS) Symposium 2025
23 February - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23253
www.ndss-symposium.org

Flush+Reload, and Demote+Demote has the lowest attack runtime (185.8 cycles). Our blind-spot evaluation also reveals that Flush+Flush and Demote+Demote have no significant blind spot, whereas other attacks have considerable blind-spot lengths compared to their attack runtime, of up to 90.1 %. With a true capacity of 15.48 Mbit/s, Demote+Reload has a 64.3 % higher channel capacity than Flush+Reload, making it slightly faster than the previously fastest CPU covert channel [60]. Similar to Flush+Flush, Demote+Demote doesn't even trigger L1 or L2 misses, making it less visible to state-of-the-art detection mechanisms [11], [27], [24].

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

Our analysis also revealed that the non-inclusive L3 cache of the recent Sapphire Rapids microarchitecture practically poses a significant restriction of what cross-core cache attacks can observe. Our results show that Flush+Reload and Flush+Flush are unaffected as they rely on the `clflush` instruction that evicts a cache line from all caches. However, attacks relying on eviction or `cldemote` do not lead to an eviction of the cache line from the other core's private cache. Hence, without the `clflush` instruction, an attacker can only spy on cache lines that the victim itself brings into the L3 cache. Based on this insight, we present a new cross-core attack, DemoteContention. DemoteContention does not rely on reverse-engineering of addressing functions or cache replacement policy for eviction, neither of which has been reverse-engineered for Sapphire Rapids yet. As DemoteContention does not rely on shared memory between attacker and victim, it can be used as already as a reliable alternative to Prime+Probe and Evict+Reload on the non-inclusive L3 cache, even without privileged information about physical addresses.

In summary, we make the following contributions:

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

Outline. Section II provides background. Section III presents our novel attacks. Section IV presents a systematic evaluation of state-of-the-art cache side-channel attacks. Section V presents Demote+Reload case studies. We discuss mitigations in Section VI and conclude in Section VII.

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

II. Background

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

A. CPU Caches

CPU caches are small and fast memories the CPU uses to store copies of data from main memory to hide the latency of main memory accesses. Modern CPUs have different levels of cache, typically three, varying in size and latency: the L1 cache is the smallest and fastest, while the L3 cache, also called last-level cache, is larger and slower. Modern CPUs are set-associative, *i.e.*, a cache line is stored in a fixed set, as determined by either its virtual or physical address. The last-level cache is physically indexed and shared across cores of the same CPU. On many CPUs from the last decade, the L3 cache was inclusive with respect to L1 and L2, meaning that all data stored in L1 and L2 is also stored in the last-level cache. To maintain this property, every line evicted from the last-level cache is also evicted from L1 and L2 caches. Some Intel CPUs also have an L4 cache, which acts as a victim cache, shared across all cores. However, recent Intel Xeon CPUs, e.g., Xeon Silver 4410T, and some large Intel Core CPUs have a non-inclusive L3 but no L4 cache. In a non-inclusive L3, cache lines present in the L1 and L2 of CPU cores do not have to be present in the L3. However, the L2 in turn is now inclusive with respect to the L1, whereas the L3 now acts as a victim cache. The last-level cache, though shared across cores, is also divided into slices. The undocumented hash function that maps physical addresses to slices in Intel CPUs has been reverse-engineered for older CPUs [48], [85], [30], however, it has not yet been reverse-engineered for Intel Sapphire Rapids CPUs.

B. Cache Attacks

Cache attacks exploit timing differences between cached and non-cached memory. Access-driven attacks are most powerful, where an attacker monitors its own activity to infer activity of a victim, e.g., which cache lines or cache sets the victim accessed. Flush+Reload [84], Evict+Reload [26] and Flush+Flush [24] all use shared memory, which is then

shared in the cache, to infer whether the victim accessed a specific cache line. The attacker evicts data either by using the `clflush` instruction (Flush+Reload and Flush+Flush), or accessing congruent addresses, *i.e.*, cache lines that belong to the same cache set (Evict+Reload). These attacks have a very fine granularity (*i.e.*, a 64-byte cache line) and are most relevant in native environments.

An access-driven attack that does not rely on shared memory and, hence, is widely applicable, is Prime+Probe [53], [46], [33]. As it does not share a cache line with the victim, it cannot use the `clflush` instruction but instead has to access congruent addresses to evict data from the victim. The granularity of the attack is noisier and coarser as an attacker only learns which cache set was accessed. Besides noise from other processes, replacement policies make it hard to guarantee eviction from a cache set [23]. Disselkoen et al. [15] and Gruss et al. [25] use TSX transactions to detect victim-induced evictions instead of the probe step as a variation of the attack (Prime+Abort). Purnal et al. [57] proposed a variant optimized for randomized secure caches called Prime+Prune+Probe and a variant that reduces the observer effect [58] (blind spot) by exploiting the specific replacement policy of the cache and allowing for a single cache access of the victim to be detected with a single cache access by the attacker. It is important to note that these attacks are specializations of Prime+Probe and, beyond the generic attack principle, exploit specific behaviors of the caches and CPUs they attack.

C. Mitigating Cache Attacks

Eliminating resource sharing in the cache can mitigate attacks [55] at substantially increased costs. Abandoning technologies like SMT (Simultaneous Multi-Threading, which shares L1 and L2 caches) would reduce performance by 25 % to 35 % [76]. Consequently, the trend is going in the other direction, towards more sharing on all hardware and software levels. Furthermore, attacks through remote interfaces [8], [3], [1] are playing an increasing role [72], [64], [73], [40], [78] where cross-domain sharing is not the root cause. Resource sharing is unavoidable on personal computers, which serve the purpose of executing third-party code both in the form of native binaries or JavaScript on a website. Hence, researchers try to find defense mechanisms that maintain sharing.

Eliminating Measurable Timing Differences. Bernstein [3] proposed constant-time (*i.e.*, no secret-dependent branches or memory accesses) to mitigate cache attacks, a technique that today is the standard means to protect cryptographic algorithms. However, writing truly constant-time code can still be challenging [86], [56]. Several independent works proposed to manipulate timers to remove the ability to measure timing differences through determinism [2], [47], [38] or fuzziness [74]. Furthermore, even without timing sources, counting threads [82], [43], [63], [62] and timeless methods [73] are viable alternatives. Disabling `clflush` [84], [24] can be circumvented by using eviction [26].

Eliminating cache-line and cache-set sharing. Attack techniques like Flush+Reload require shared memory and could be stopped by removing shared memory [84]. Indeed, one source of shared memory, page deduplication, is more and more restricted to prevent its malicious use [70], [4], [66], [13]. However, other use cases of shared memory, particularly shared libraries, are unaffected and still available for attacks [65]. Mitigating Prime+Probe requires addressing the sharing issue on the level of cache sets, e.g., by coloring cache lines and assigning colors to security domains [68], [36], [20], [12]. Intel CAT [32] provides dedicated software control over cache ways and can be used to separate workloads into different parts of the cache [44]. Several works eliminate sharing via cache flushing during context switches between different domains [89], [20].

Detecting attacks. Many works researched the detection of cache attacks in binaries [26], [16], [34], [7], or at runtime using cache attacks [88], [26] or a range of performance counters [28], [10], [87], [54].

III. Novel `cldemote`-based Attacks

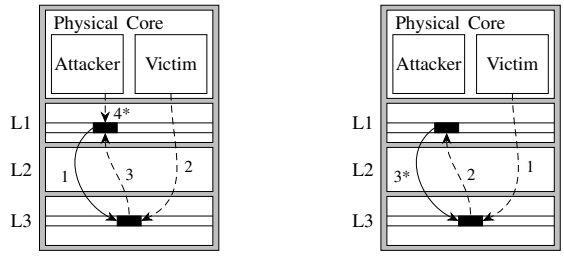
In this section, we present two attacks for same-core attack scenarios, Demote+Reload and Demote+Demote, and one cross-core attack, DemoteContention.

A. Same-Core Attacks

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

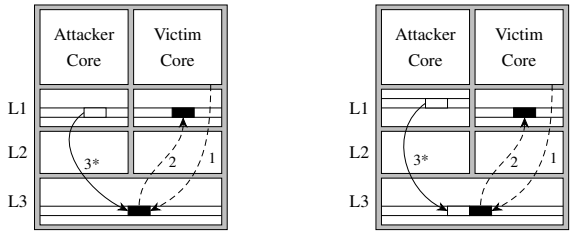
Demote+Reload and Demote+Demote build on the observation that the `cldemote` instruction can evict a cache line from L1 to L3. Subsequent victim accesses load the cache line into the L1 again, which the attacker can observe by timing the reload operation in Demote+Reload or `cldemote` in Demote+Demote.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

The basic principle of our two attack techniques is illustrated in Figure 1. Demote+Reload follows the semantics of Flush+Reload: The attacker first demotes the cache line, then a victim operation possibly accesses this cache line, and afterward, the attacker times reloading the cache line,



(a) Demote+Reload (b) Demote+Demote

Fig. 1. Working principle of Demote+Reload and Demote+Demote. The ∗ denotes the timed operation. Solid lines are `cldemote` executions and dashed lines memory accesses. Both attacks only move a cache line from L1 to L3. The victim activity is visible by timing either the reload (Demote+Reload) or the `cldemote` (Demote+Demote).



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

Fig. 2. Cross-Core DemoteContention. The ∗ denotes the timed operation. Solid lines are `cldemote` executions and dashed lines memory accesses. In both attacks, the victim repeatedly moves a cache line from L1 to L3. The attacker measures a memory access to the exact cache line the victim uses (Figure 2a) or a cache line in the same cache set (Figure 2b).

observing whether the victim accessed it (Figure 1a). Demote+Demote follows the semantics of Flush+Flush: The victim first possibly accesses this cache line, and afterward, the attacker times the `cldemote` instruction, observing whether the victim accessed the cache line in between (Figure 1b). Due to the reliance on a load to the L1 by the victim, both attacks require that the attacker and victim run on the same physical core but potentially different logical cores with SMT.

B. Cross-Core Attacks

Cross-core attacks like Prime+Probe and Evict+Reload on previous systems exploited the inclusiveness of the L3 cache [49], [50]: Recent Xeon processors have a non-inclusive L3. Evicting a cache line from the L3 implies the removal of this cache line from all private L1 and L2 caches of all cores. This is not the case on these new CPUs anymore, where eviction from L3 has no implications on the private L1 and L2 caches of any core. Consequently, Prime+Probe and Evict+Reload can only sense when victim operations reach the L3 but not operations in the victim's private L1 and L2 caches. Beyond this limitation, the cache eviction in Prime+Probe and Evict+Reload brings another significant hurdle for an attacker: Eviction sets are generated based on physical addresses [49], [75], the undocumented cache addressing functions [18], and the undocumented cache replacement policy [23]. Mounting efficient Prime+Probe or Evict+Reload would require to reverse-engineer these functions and additionally to obtain privileged physical address information, e.g., possibly via a timing side channel [22], [14]. The non-inclusive L3 complicates Prime+Probe and Evict+Reload even further as the attacker's accesses to the eviction set need to reach the L3. This is possible by using `cldemote`, as we propose below, or by creating an eviction set that simultaneously evicts L1 and L2 caches, moves the corresponding cache lines to the L3 and thereby evicts the L3 cache set. Our experiments indicate that L1 and L2 eviction on Sapphire Rapids in general does not lead to placement of cache lines in the L3, *i.e.*, the L3 does not act primarily as a victim cache.

As an alternative to cross-core Prime+Probe and Evict+Reload, we present a new cross-core attack, DemoteContention (Figure 2). DemoteContention is a reliable alternative to Prime+Probe and Evict+Reload without reverse-engineering of addressing functions and cache replacement. DemoteContention relies on contention on the L3 cache set by continuously demoting a cache line. The attacker uses their own cache line, *i.e.*, no shared memory, located in the same L3 cache set as the victim cache line, e.g., found via profiling [26]. If the victim performed an operation that reaches the L3 caches, DemoteContention observes a spike in the `cldemote` execution time, even if the attacker-monitored cache line is not in the cache, allowing for continuous execution of `cldemote`.

The reason why DemoteContention works, is that `cldemote` has to interact with the L3 or cache directory, *i.e.*, leading to contention on the corresponding set in the L3 or the cache directory, regardless of the state of the cache line. The L3 and the cache directory have been investigated in previous works [83], [49] and identified as viable cross-core channels. Given that `cldemote`, under regular usage, updates either the L3 or cache directory, we suspect it performs a lookup in the cache directory or L3 even before it knows whether the cache line to be demoted is in the L1 or L2. Consequently, `cldemote` is influenced by concurrent contention on the corresponding set in the L3 or cache directory.

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

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

In this section, we provide a systematic evaluation of state-of-the-art cache attack techniques, including Demote+Reload, Demote+Demote, Flush+Reload [84], Flush+Flush [24], Evict+Reload [26] on the L1, and Prime+Probe [53] on the L1. All measurements are taken on the Sapphire Rapids Xeon Silver 4410T (CPU SR) and the Emerald Rapids Xeon Silver 4514Y (CPU ER) CPUs, as the very new `cldemote` instruction is only supported on Intel Xeon Sapphire Rapids and Emerald Rapids CPUs so far. Our setup runs Ubuntu 22.04 LTS, Linux 6.2.0, and gcc 11.4 on the CPU SR and Ubuntu 24.04 LTS, Linux 6.8.0, and gcc 11.4 on

TABLE I. COMPARISON OF ALL TESTED ATTACKS ON AN INTEL SAPPHIRE RAPIDS XEON SILVER 4410T (SR) AND AN EMERALD RAPIDS XEON SILVER 4514Y (ER). THE HIT-MISS MARGIN AND ATTACK TIME ARE IN CYCLES (C).

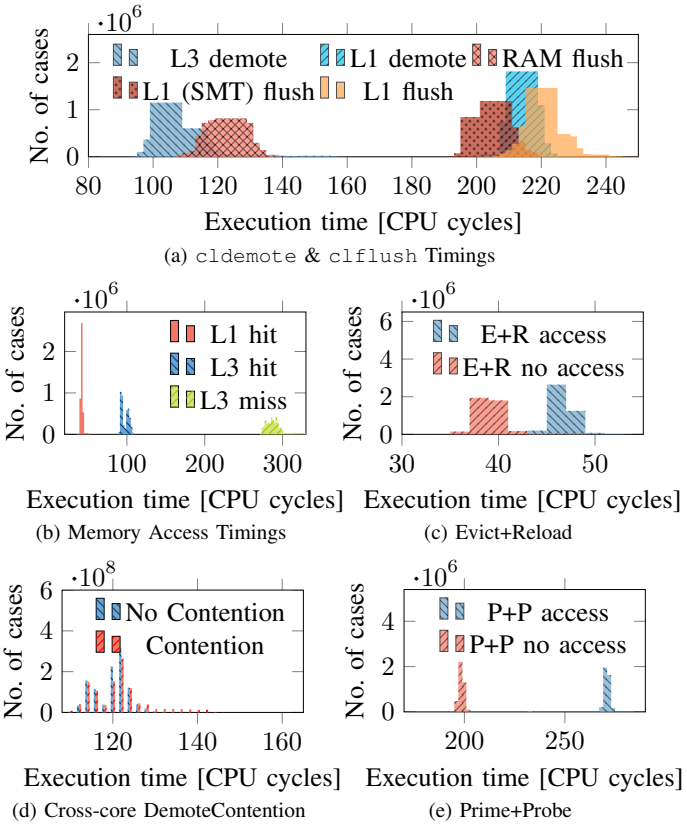
Attack	Topological Constr.	Shared Memory	Attack Margin		Temporal Prec. (SD)		Spatial Prec.	Blind Spot		Attack Time		Channel Cap. [Mbit/s]		Error Ratio	
CPU	SR/ER	SR/ER	SR	ER	SR	ER	SR/ER	SR	ER	SR	ER	SR	ER	SR	ER
Demote+Reload	Core	✓	48 C	34 C	±27 ns	±24 ns	cache line	42.8 %	42.6 %	424.3 C	320.6 C	6.38	6.09	0.7 %	0.7 %
Demote+Demote	Core	✓	86 C	60 C	±17 ns	±16 ns	cache line	0.0 %	0.0 %	185.8 C	137.6 C	8.34	9.36	4.6 %	2.3 %
DemoteContention	CPU	✗	–	–	±16 ns	±24 ns	L3 cache set	90.1 %	89.5 %	185.8 C	137.6 C	0.18	0.19	5.9 %	1.9 %
Flush+Reload	CPU	✓	170 C	124 C	±24 ns	±15 ns	cache line	89.9 %	86.3 %	614.1 C	462.8 C	1.94	2.15	4.9 %	3.6 %
Flush+Reload (SMT)	Core	✓	232 C	166 C	±20 ns	±20 ns	cache line	75.1 %	74.3 %	614.1 C	462.8 C	2.35	2.01	3.4 %	3.2 %
Flush+Flush	CPU	✓	86 C	38 C	±12 ns	±15 ns	cache line	0.0 %	0.0 %	192.0 C	155.2 C	8.80	10.26	1.2 %	1.9 %
Flush+Flush (SMT)	Core	✓	72 C	30 C	±19 ns	±18 ns	cache line	0.0 %	0.0 %	192.0 C	155.2 C	8.55	9.03	5.6 %	2.6 %
Prime+Probe (L1)	Core	✗	78 C	68 C	±33 ns	±33 ns	L1 cache set	23.9 %	24.8 %	281.2 C	245.0 C	3.62	3.78	10.9 %	1.2 %
Evict+Reload (L1)	Core	✓	10 C	4 C	±38 ns	±41 ns	cache line	32.4 %	15.5 %	390.0 C	339.8 C	2.51	3.32	6.8 %	1.4 %

the CPU ER. We fixed the frequency for all experiments in this section, except for the covert channel and noise resilience tests, to reduce the noise of this comparison through frequency scaling. For the measurements, we use `rdtsc`, like prior work, providing a sub-nanosecond resolution timestamp. Similarly, we use `lfence` and `mfence` instructions to ensure that the instructions are ordered with respect to other instructions and memory operations. Our findings are summarized in Table I. While we attempted to mount Prime+Probe or Evict+Reload on the L3, our experiments indicate that L1 and L2 eviction does not reliably lead to placement of the cache line in the L3. Furthermore, the addressing functions published by Gerlach et al. [18] did not yield reliable eviction on the more recent Sapphire Rapids microarchitecture either, possibly due to non-inclusiveness or a change in addressing functions. The only approach we found to place cache lines in the L3 reliably was `cldemote`. Additionally, physical addresses (for the addressing functions) are privileged information, whereas all other attacks in our comparison work with unprivileged information only. Hence, we also evaluate DemoteContention but not Prime+Probe or Evict+Reload on the L3.

A. Hit-Miss Margins

We define the hit-miss or attack margin for an attack as the difference between the 95th percentile of the faster and the 5th percentile of the slower case. The difference between percentiles is a more meaningful metric than a difference between averages, as the deviation around the mean would be ignored otherwise. The results of our measurements for all attacks are shown in Figure 3 (CPU SR) and Figure 13 (CPU ER). We conducted 10^6 measurements for each case.

The L1 hit, L3 hit, and DRAM access timings are shown in Figure 3b and Figure 13b. For Demote+Reload, we distinguish between L1 hits (victim access) and L3 hits (after `cldemote`). The resulting hit-miss margins are 48 cycles (CPU SR) and 34 cycles (CPU ER). SMT Flush+Reload uses L1 hits (victim access) and L3 misses (after `clflush`) with hit-miss margins of 232 cycles (CPU SR) and 166 cycles (CPU ER). Cross-core Flush+Reload uses L3 hits and L3 misses with hit-miss margins of 170 cycles (CPU SR) and 124 cycles (CPU ER). The margin is significantly larger for Flush+Reload than Demote+Reload due to the long time it takes to fetch memory from DRAM.



(a) `cldemote` & `clflush` Timings

(b) Memory Access Timings

(c) Evict+Reload

(d) Cross-core DemoteContention

(e) Prime+Probe

Fig. 3. Timing histograms for all tested attacks on our Xeon Silver 4410T.

The relevant timings for Demote+Demote and Flush+Flush are shown in Figure 3a and Figure 13a. For Demote+Demote, we distinguish between a `cldemote` on a cache line in the L1 (victim access) and a cache line in the L3 (after `cldemote`) with attack margins of 86 cycles (CPU SR) and 60 cycles (CPU ER). The similar SMT Flush+Flush attack distinguishes between a `clflush` on a cache line in the L1 (victim access) and a DRAM access (after `clflush`) with attack margins of 72 cycles (CPU SR) and 30 cycles (CPU ER). Cross-core Flush+Flush distinguishes between a `clflush` on a cache line in an L1 of a different core (victim access) and a not-present cache line with an attack margin of 86 cycles (CPU SR) and 38 cycles (CPU ER). We can observe that the margin for our Demote+Demote is significantly larger than for

Flush+Flush. It takes slightly longer to demote a cache line from the L1 to the L3 than to flush a cache line accessed only by the same core. This difference is most likely the result of `cldemote` ensuring that the cache line is written to the L3 while `clflush` on an unmodified cache line only evicts the data from the cache.

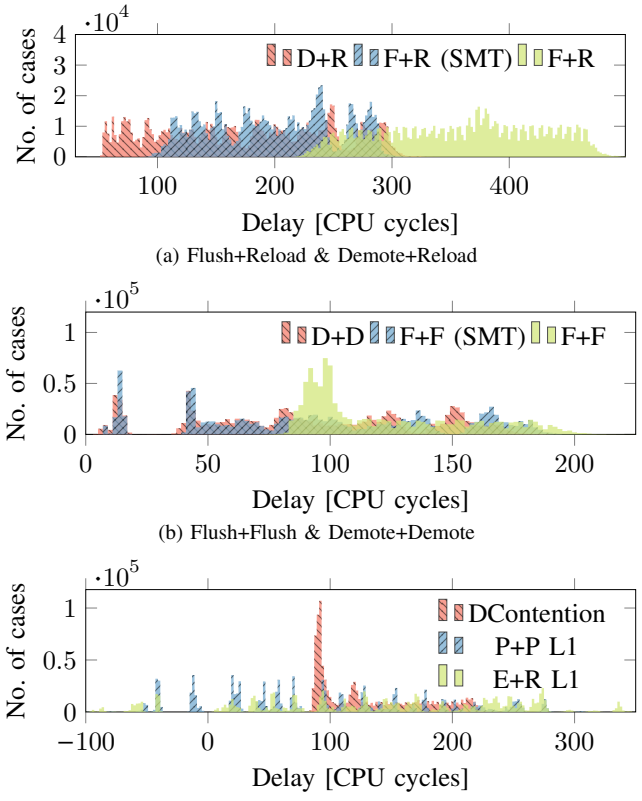
For Prime+Probe and Evict+Reload, we use an eviction-set size of 12 as the Xeon Silver 4410T has a 12-way set-associative L1 cache. L1 Prime+Probe (Figure 3c and Figure 13c) has attack margins of 78 cycles (CPU SR) and 68 cycles (CPU ER). L1 Evict+Reload (Figure 3e and Figure 13e) has attack margins of 10 cycles (CPU SR) and 4 cycles (CPU ER). While the timings seem noise-free for Prime+Probe and Evict+Reload, both attacks are highly susceptible to noise from unrelated memory accesses as they monitor accesses to whole cache sets in the relatively small L1.

Figure 3d shows the hit-miss histogram for DemoteContention. The contention case of DemoteContention (victim access) takes 121.4 cycles (CPU SR) and 101.4 cycles (CPU SR), and the no contention case (no victim access) takes 119.8 cycles (CPU SR) and 100.3 cycles (CPU SR). The two cases overlap almost entirely due to a large blind spot, which can not be easily counteracted, as further discussed in Section IV-D. Due to this, we do not have a realistic attack margin for Demote-Contention. Despite this, the contention case has a significant number of measurements above 124 cycles (CPU SR) and 116 cycles (CPU ER), which are not present without contention, making the two cases distinguishable.

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

B. Temporal Precision

We measure the temporal precision by triggering accesses at a low frequency and measuring the time it takes for the attacker to detect them. The victim thread triggers an access at random intervals. All attacks are performed with a minimal attack loop, including the attack and a store to a memory location if an access is detected. We define the temporal difference as the timing difference between the start of the victim access, measured with `rdtsc` directly before the access, and the time the attacker detects the access. We specifically use the start of the attacker's measurement period that detects the access. We use the start of the measuring period instead of the end, as the end includes the attack time, which can result in more noise and, therefore, a higher standard deviation, an essential metric for attacks such as inter-keystroke timing attacks. Furthermore, we evaluate the attack time in a separate experiment (Section IV-D). Using the start of an attack's measurement period can result in a negative temporal difference, as accesses that occur directly after the measurement period starts can be detected by some attacks,



(a) Flush+Reload & Demote+Reload

(b) Flush+Flush & Demote+Demote

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

Fig. 4. The delay between memory access and the start of the detection period detected the access for all attacks on our Xeon Silver 4410T.

as seen in Figure 4c. Only successfully detected accesses are included in this experiment, as blind spots are separately evaluated in Section IV-D. The results are shown in Figure 4 and Figure 16. We conducted 10^6 measurements for each case, and the standard error for all measurements is ≤ 0.1 cycles.

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

most, which is ideal for variance-critical attacks such as inter-keystroke timing attacks.

C. Spatial Precision and Topological Scope

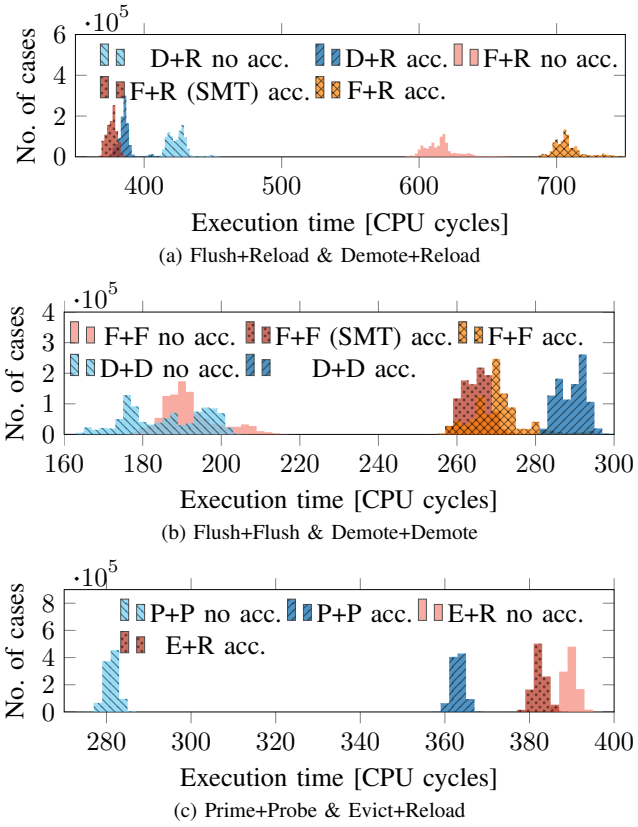
The spatial precision of the side channels we evaluate is defined by the microarchitectural element they attack. We provide the spatial precision for all attacks in Table I. While the attacks have theoretical spatial precision, it is essential to note that practically, targeting multiple locations within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

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

D. Attack Times and Blind Spot

The attack time is essential in determining the throughput an attack can achieve. To determine the attack time, we evaluated the attack round length and provide the results in Figure 5 and Figure 14. An attack round consists of the minimal code for the tested attacks, a check whether the resulting timing value is above or below a threshold and an access is detected, and a store of the result in an atomic variable for further processing in a separate thread. We conducted 10^6 measurements for each case. The standard error of the average for all results is ≤ 0.1 cycles. On both CPUs, the tested attacks perform similarly to each other, with the main change being a lower cycle count for all measurements on our CPU ER due to a different clock frequency compared to our CPU SR.

Demote+Demote has the lowest attack times, with 185.8 cycles (CPU SR) and 137.6 cycles (CPU ER) with no victim access, and 289.2 cycles (CPU SR) and 216.5 cycles



(a) Flush+Reload & Demote+Reload

(b) Flush+Flush & Demote+Demote

(c) Prime+Probe & Evict+Reload

Fig. 5. Execution time in cycles of a single attack iteration for all tested attacks on our Xeon Silver 4410T.

(CPU ER) with a victim access (see Figure 5b and Figure 14b). The similar SMT Flush+Flush has attack times of 192.0 cycles (CPU SR) and 146.5 cycles (CPU ER) with no victim access and 264.9 cycles (CPU SR) and 197.7 cycles (CPU ER) with a victim access. The attack time without a victim access for Demote+Demote is slightly lower than for Flush+Flush. The attack time after a victim access for SMT Flush+Flush is slightly lower than for Demote+Demote; this is expected, as `cldemote` on an L1 cache line is slightly slower than a `clflush` on the cache line in the L1 of the attacking core (see Section IV-A). We consider the case without a victim access more relevant, as it is more common in most attack scenarios. Both Demote+Demote and Flush+Flush consist of measuring a single instruction without further setup, unlike Flush+Reload and Demote+Reload, resulting in significantly lower attack times. Flush+Reload performs the worst with attack times of 614.1 cycles (CPU SR) and 462.8 cycles (CPU ER) with no victim access (see Figure 5a and Figure 14a).

The attack time for cross-core DemoteContention is the same as for Demote+Demote in the case of no victim access. The victim access case is challenging to evaluate as cross-core DemoteContention is based on contention with a huge blind spot and high noise. Despite this, the attack time with a victim access is only slightly longer than in the case of no victim access, as shown in Figure 3d and Figure 13d.

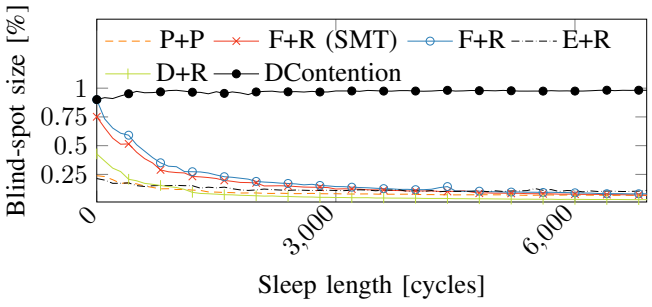


Fig. 6. Blind-spot size of all tested attacks that have a blind-spot for varying delay lengths after each attack iteration. The blind-spot size is given in the percent of a single attack loop iteration (including the delay) on our Xeon Silver 4410T.

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

Cache side-channel measurements are typically destructive in the sense that they destroy the previous state of the microarchitectural element. Restoring the previous state takes time, during which a victim's operation may be missed. Therefore, this observer effect is called "blind spot" and has been studied as a limiting factor for attacks [24], [58]. To measure the size of the blind spot, we let one thread continuously execute the attack. After a random number of cycles, a second thread accesses the memory location, which the first thread monitors. We log whether the attack detected the memory access. We repeat this measurement $1\,000$ times, resulting in a percentage of successfully detected memory accesses. We use this result as an approximation for the size of the blind spot a given attack has relative to its execution time. Furthermore, to show the effect the wait time after each attack iteration can have on the blind spot, we ran our evaluation for different sleep or delay periods after each attack execution. We additionally combine these results with the previously measured values for the attack time with no victim access to compute an estimate for the absolute blind-spot size in cycles. Similar to previously discussed metrics, the blind spot for all attacks is very similar for both of our tested CPUs. The sample size for all measurements is 200, and the standard error is $\leq 1\%$.

Our blind-spot evaluation results are shown in Figure 6 and Figure 15. We use a blind-spot percentage instead of a cycle estimate, as it directly represents the percentage of victim accesses that an attacker might miss and visualizes the effect a delay after each attack iteration has on the blind spot. Demote+Demote and Flush+Flush are excluded from this plot, as we did not observe a blind spot for both of them, resulting in a blind-spot size of $\approx 0\,\%$ regardless of the delay length. The blind spot for all attacks, except for cross-core DemoteContention, decreases with an increase in the delay length, approaching $\approx 0\,\%$. This is the expected behavior as a longer delay decreases the likelihood of the victim accessing the memory in the blind spot of the attack. Cross-core DemoteContention is the only exception, as it relies on contention with the victim. For cross-core DemoteContention, the victim and attacker have to access the same L3 cache set at roughly the same time. An increase in the delay length after each attack iteration decreases the likelihood for the attacker to trigger the contention, increasing the blind-spot size, as shown in Figure 6 and Figure 15. Ideally, the blind-spot size for cross-core DemoteContention would approach 1. However, the small attack margin, as discussed in Section IV-A of cross-core DemoteContention, results in a high amount of noise, which, in turn, results in a high amount of false positives compared to the other tested attacks. These false positives lead to an underestimation of the blind spot. Also, the relative blind-spot size for Prime+Probe and Evict+Reload only slightly decreases over time as they are more susceptible to noise.

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

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

E. Channel Capacity and Noise Resilience

The channel capacity is a standard evaluation metric for side channels. While prior work reported capacities for most of these side channels, comparing covert-channel capacities across different systems can be misleading as the channel capacity can be significantly influenced by the general performance of a CPU or its memory subsystem. Consequently, we take a different approach where we construct a simple covert channel that can be instantiated generically with any of the side channels we discuss. The basic construction uses time

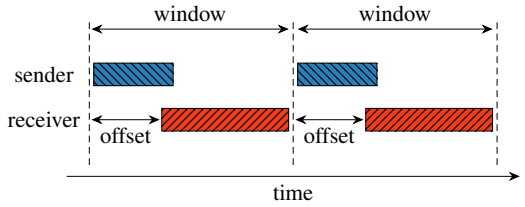


Fig. 7. Our basic covert channel construction and the dimensions to optimize. The `window` parameter defines the size of a single transmission window, and `offset` defines the head start the sender gets to prepare, e.g., access memory locations.

TABLE II. SINGLE-BIT (1-BIT) AND MULTI-BIT (N-BIT) COVERT-CHANNEL TRUE CAPACITY IN **MBIT/S** AND ERROR RATIO OF ALL TESTED ATTACKS ON AN INTEL XEON SILVER 4410T.

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

slices to transmit one or more bits through the channel. To provide a fair comparison, we assume perfect synchronization of the first time slice used by the channel. We use two threads for each channel, one receiver, and one sender thread. Each channel capacity listed in this section has a sample size of 100 and a standard error of the mean of ≤0.2 Mbit/s.

This simple channel has two dimensions to optimize (Figure 7): First, the window length. Reducing the window length increases how many slices fit in a time frame, *i.e.*, increasing the transmission rate. Second, the offset the receiver has from the sender. The offset prevents receiving too early before the sender has sent the data, which could result in incorrect detections. While the error ratio can also be a parameter to optimize for, we avoid this additional dimension by working with the true capacity computed from the raw capacity and the error ratio. This way, we optimize for the optimal trade-off between error ratio and raw capacity. Furthermore, we can adjust the number of bits sent within a window to further optimize the true capacity.

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

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

The results for each side channel are provided in Table II (CPU SR) and Table IX (CPU ER). With one bit per window, optimized Demote+Reload performs the best with 11.47 Mbit/s (CPU SR) and 11.10 Mbit/s (CPU ER). Demote+Reload and Demote+Demote perform similarly on our CPU SR with 6.38 Mbit/s and 6.04 Mbit/s, respectively. On our CPU ER, Demote+Demote is faster with 8.17 Mbit/s compared to Demote+Reload 6.09 Mbit/s, presumably due to microarchitectural changes. Optimized cross-core Flush+Reload has capacities of 5.42 Mbit/s (CPU SR) and 6.51 Mbit/s (CPU ER). SMT Flush+Flush performs significantly worse than Demote+Demote using one bit per window at 4.56 Mbit/s (CPU SR) and 5.03 Mbit/s (CPU ER). DemoteContention performs the worst with 0.18 Mbit/s (CPU SR) and 0.19 Mbit/s (CPU ER). However, without reverse-engineering L3 addressing functions and replacement policies, this is the only cross-core attack available that does not require shared memory.

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

We measure the noise resilience by starting on a completely idle system and measuring the true capacity of the optimized 1-bit covert channels, representing a typical attack scenario. We gradually increase the system load from 0 to one worker per logical CPU core, using cache thrashing worker threads from the stress-ng test suite. As shown in Figure 11 and Figure 12, all attacks suffer from noise.

Optimized Demote+Reload drops significantly until 10 workers and afterward decreases slowly to 5.24 Mbit/s

(≈−56 %) on our CPU SR (see Figure 11a) and 2.23 Mbit/s (≈−80 %) on our CPU ER (see Figure 12a), performing the best. Cross-core DemoteContention performs the worst, dropping drastically, reaching almost 0 kbit/s at 10 (≈−100 %) on both tested CPUs. The remaining attacks perform similarly to each other, dropping slowly until they lose ≈80 % to 95 % of their channel capacity with the maximum number of workers on both CPUs, as shown in Figure 11 and Figure 12.

While the covert channel capacity is a metric to show how fast information can be transmitted covertly using a given attack, it also demonstrates how much information each channel can theoretically leak in a given time. The 1-bit scenario is the typical attack scenario, where an attacker wants to monitor a single memory location. When monitoring a single memory location, Demote+Demote performs the best with a capacity similar to Demote+Reload on our CPU SR and outperforming it on CPU ER. This makes Demote+Demote the better choice in such a scenario over other attacks when attacking over other attacks such as Flush+Flush and Flush+Reload when targeting a victim over SMT. When monitoring many memory locations, Flush+Flush performs slightly better than Demote+Demote while also working across cores.

F. Detectability

Cho et al. [11] recently developed a detection scheme using `mem_load_retired.l1_miss` (L1 miss), `mem_load_retired.l2_miss` (L2 miss), `mem_load_retired.l3_miss` (L3 miss), and `br_inst_retired.all_branches` (retired branches). This is similar to the approaches of Gulmezoglu et al. [27] and Gruss et al. [24]. To evaluate the detectability of the different attacks, we focused on the performance counters selected by Cho et al. [11].

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

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

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

V. Demote+Reload Attack Case Studies

In this section, we evaluate our attacks in multiple case studies: First, we compare Demote+Reload and Demote+Demote to existing attacks on an AES T-tables attack and an inter-keystroke timing attack. Second, we demonstrate that `cldemote` does not only result in leaks through access times but also through power consumption [37]. Finally, we show that `cldemote` can break KASLR, even on systems that do not officially support `cldemote`.

A. Attacking OpenSSL AES T-tables

A standard benchmark for cache side-channel attacks is OpenSSL AES T-tables. While the T-table implementation is not used anymore by OpenSSL, it is a common means to compare cache attacks in a realistic cryptographic attack scenario, *i.e.*, repeatable high-frequency attacks to accumulate leakage. We mounted a last-round attack following the approach by Irazoqui et al. [35]. All attacks were run in a same-core scenario, *i.e.*, the attacker triggers the encryption (with an inaccessible key) and then mounts the cache attack after the encryption. For statistical significance, we perform 1 000 key recoveries, corresponding to at least 10 million runs of each of the attacks. The success rate of the key recovery increases with the number of samples, *i.e.*, even with noisy channels the

TABLE III. COMPARISON OF ATTACK TECHNIQUES ON SAPPHIRE RAPIDS. DEMOTE+RELOAD RESULTS IN THE LOWEST OVERALL ATTACK RUNTIME.

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

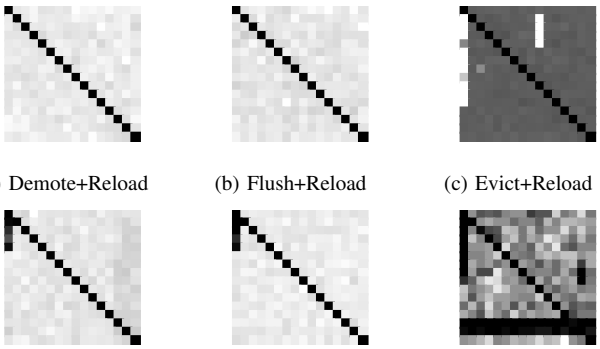
TABLE IV. COMPARISON OF ATTACK TECHNIQUES ON EMERALD RAPIDS. DEMOTE+RELOAD RESULTS IN THE LOWEST OVERALL ATTACK RUNTIME. YIELDING DID NOT IMPROVE THE ATTACK PERFORMANCE FOR ANY OF THE ATTACKS.

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

attack typically converges to the correct key with sufficient samples. An attacker can minimize the attack runtime by minimizing the number of encryptions until the success rate is not 100 % anymore. Hence, this is a good metric for comparison: We minimize the number of encryptions for each attack until we reach a 97 % to 99 % correct key guess range. As seen in Table III, `sched_yield`, while costing > 100 cycles, can help improve the attack performance, as it indicates to the operating system that this is a cooperative thread: With default configuration, the kernel then less frequently preempts (interrupts) the thread. Furthermore, it increases the chance of running an idle thread, reducing power dissipation and thermal emissions, and hence, throttling effects.

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

As shown in Table III, Demote+Reload results in the overall lowest attack runtime, with 14.5 ms and 98.7 % of the key guesses correct. The number of encryptions required to reach this percentage is the same as with Flush+Reload, 11 000. This is on par with state-of-the-art key recovery attacks on AES that require, e.g., 6 000 to 10 000 encryptions [67], [35], [6]. However, Flush+Reload is 24 % slower than Demote+Reload. Only Demote+Demote with `sched_yield` worked with a lower number of encryptions, namely 9 000, yet had a higher overall



(a) Demote+Reload (b) Flush+Reload (c) Evict+Reload

(d) Demote+Demote (e) Flush+Flush (f) Prime+Probe

Fig. 8. Comparison of different attack techniques over 10 000 encryptions using a zero key. The y-axis is the plaintext byte value, and the x-axis is the T-table cache lines. A darker shade means more cache line accesses. A visible diagonal shows that the attack correctly identified the zero key.

runtime due to the runtime overhead of `sched_yield`. Without `sched_yield`, there is more noise, requiring 15 000 encryptions to get to the same correct key rate. This is not unexpected as Demote+Demote has virtually no blind spot, and any interrupt will result in a timing deviation that is noise for the attack. For Flush+Flush, we observe a similar situation, where adding a `sched_yield` allows us to reduce the number of encryptions from 18 000 to 16 000, but this is not enough to compensate the runtime cost of `sched_yield`, resulting in the highest attack runtime in our test. Evict+Reload and Prime+Probe (on the L1) required the highest number of traces for key recovery. As we focus on the L1, the timing difference between hit and miss is very low (< 10 cycles), and thus, it is more susceptible to noise due to subtle timing variations, bringing up the attack runtime. Without `sched_yield`, the number of encryptions required is higher, but the overall runtime is lower. For a visual comparison of the three attacks, we provide a zero-key first-round attack matrix in Figure 8. The x-axis corresponds to the cache lines making up the T-Table, and the y-axis corresponds to the plaintext byte. A darker color means more detected accesses. A visible diagonal means that an attack identified the zero key correctly. The less visible the diagonal, the harder it is to identify the key with a given attack.

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

B. Inter-Keystroke Timing Attack

Another standard benchmark for cache side channels is to mount an inter-keystroke timing attack, which can be used to infer the actual key press values and written text [69], [51]. Keystroke attacks are an excellent means to compare cache

attacks in scenarios of low-frequency, non-repeatable attacks, where leakage cannot be easily accumulated, but the accuracy of a single attack is more relevant. For our attack, we first used a template attack to identify leaky offsets in a shared library using Flush+Reload [26]. Afterward, we use the same offset in the shared library for Flush+Reload and Demote+Reload to compare their temporal precision and noise resilience.

For both our Flush+Reload attack and our Demote+Reload attack, we let a human perform 1000 keystrokes, typing into a program over multiple minutes to obtain the ground truth for the timings. In parallel, we run one of the two attacks, which have exactly the same implementation except for swapping the `clflush` and `cldemote` instructions and the corresponding hit-miss threshold. We observe that for both Flush+Reload and Demote+Reload, there are no false positive detections. This highlights the noise resilience of both attacks. With Flush+Reload, we ran ≈2.8 million, and with Demote+Reload ≈3.6 million attacks per second, with not a single measurement resulting in a false positive detection. For Demote+Reload, we observed no false negatives, and for Flush+Reload, only a single one, *i.e.*, Flush+Reload detected 999 out of 1000 keystrokes correctly. That is, in terms of the F-Score measure, which is often used to assess the accuracy of keystroke detection [26], [24], [52], we achieve an F-Score of 1 in this experiment for Demote+Reload and 0.999 5 for Flush+Reload, showing that the attacks are on par. For the temporal precision, we observe a mean delta between the ground truth and the recovered timing of 288.5 ns (n=1000, $\sigma_{\bar{x}}$=3.95 ns) with Flush+Reload and 233.5 ns (n=1000, $\sigma_{\bar{x}}$=2.73 ns) with Demote+Reload. While an attacker can account for the mean delta, the standard error $\sigma_{\bar{x}}$ remains an inaccuracy. Thus, we can conclude that Demote+Reload has roughly 30 % more temporal precision in this attack scenario. It is essential to highlight that timing variations of humans are orders of magnitude higher (*i.e.*, in the millisecond range) [41], than either of the two attacks.

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

C. Collide+Power

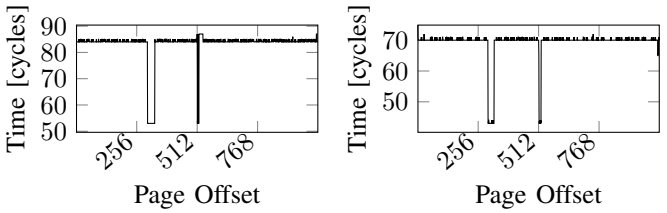
Collide+Power [37] exploits the power leakage of data collisions in the memory subsystem between security domains. These microarchitectural data collisions occur since the memory subsystem is shared between the attacker and victim domain, including caches and buses connecting the distinct cache levels. Therefore, data blocks traveling over the shared components immediately after another expose their Hamming distance, *i.e.*, the number of different bits, in the power domain, resulting in an exploitable signal to recover the actual data. We focus on a specific leakage effect of Collide+Power, the so-called *self-leakage* [37]: Here, the Hamming distance between the upper and lower 32 B of a cache line leak to one another when the cache line is moved into the L3 cache, reflecting the behavior of `cldemote`.

We show that `cldemote` amplifies the *self-leakage* effect for software-based power-analysis attacks when data is frequently demoted into the L3 cache, e.g., to make the data used in a multithreaded context visible to other physical cores faster as recommended by the Intel guidelines [31]. We measure the power consumption via the package domain of the Running Average Power Limit (RAPL) interface. We repeat a single store instruction once with and once without the added optimization in a loop for ≈ 12 ms to record a single measurement sample. The analysis framework of Collide+Power uses the recorded power samples to fit the power model $P(U, L) = x \cdot hd(U, L)$, where the coefficient x indicates the strength of the Hamming distance leakage between the upper (U) and lower (L) 32 B of the cache line.

We record 1.2 million samples per case and estimate a power leakage of x = 310 µW per bit difference between U and L when using `cldemote` on our Xeon Silver 4410T. For our Xeon Silver 4514Y, we record 3.5 million samples with an estimated power leakage of x = 119 µW. In contrast, we do not observe a significant power leakage without the optimization. Finally, we compute the Pearson correlation coefficient of the power model and the measurements, resulting in correlations of 0.012 (Xeon Silver 4410T) and 0.004 (Xeon Silver 4514Y) for the case with `cldemote` and no significant correlation without the instruction. Therefore, using `cldemote` as an optimization to improve cross-core access latencies, as recommended by Intel [31], increases the attack surface for software-based power-analysis attacks.

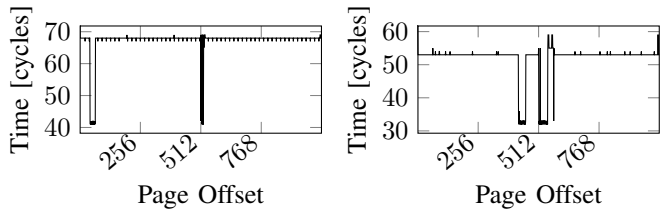
D. Breaking KASLR

In this section, we demonstrate a KASLR break using the `cldemote` instruction on Ubuntu 22.04 LTS (Linux 6.2.0) on our Xeon Silver 4410T and Ubuntu 24.04 LTS (Linux 6.8.0) on our Xeon Silver 4514Y. We exploit the absence of faults by `cldemote` and its' TLB-dependent timing behavior, *i.e.*, `cldemote` needs to translate an address before determining if the demote operation is valid. KASLR is a low-cost security mechanism, randomizing kernel code and data addresses. Due to many KASLR breaks [29], [22], [21], [9], [39], the security value may be limited, but it is now a typical benchmark for new microarchitectural attacks. The TLB-induced timing



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

Fig. 9. Execution time of `cldemote` over the Kernel Text segment KASLR range for all 2 MB pages. The mapped regions show a significant drop in execution time, revealing the KASLR offset to an attacker at page offset 302 on our Xeon Silver 4410T and offset 298 on our Xeon Silver 4514Y.



(a) i7-1260P (b) i9-13900K

Fig. 10. Execution time of `cldemote` over the Kernel Text segment KASLR range for all 2 MB pages on two CPUs that do not support the instruction. The mapped regions show a significant drop in execution time on both CPUs, even though the instruction should be interpreted as a `nop`.

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

As large parts of the kernel are regularly accessed through interrupts, syscalls, context switches, and other kernel activities, they have TLB entries. The `cldemote` timing for the first cache line of each 2 MB page starting at `0xffffffff80000000` covering the kernel binary mapping region is shown in Figure 9. For our Xeon Silver 4410T (Sapphire Rapids), shown in Figure 9a, the execution time for invalid addresses is ≈85 cycles. The dips to ≈53 cycles (starting at offset 302, virtual address `0xffffffffa5c00000`) indicate mapped pages. For our Xeon Silver 4514Y (Emerald Rapids), shown in Figure 9b, the execution time for invalid addresses is ≈70 cycles. The dips to ≈43 cycles (starting at offset 298, virtual address `0xffffffffa5400000`) indicate mapped pages. We verified these mappings for both CPUs through `/proc/kallsyms`. Scanning the kernel binary KASLR range multiple times to account for noise to find the base address takes <10 ms on both CPUs.

The `cldemote` is currently only supported on recent Xeon microarchitectures. On all other CPUs, `cldemote <register>` is interpreted as a `nop` that dereferences the register. Indeed, `cldemote` does not exhibit any unexpected timing behavior on any 10th-generation or older Intel Core CPUs we tested. However, for our Alder Lake i7-1260P CPU and Raptor Lake i9-13900K CPU, the instruction has a timing behavior similar to the Xeon Silver 4410T when used on mapped but inaccessible kernel addresses. Even though this timing behavior for `cldemote` exists on these unsupported CPUs, `cldemote` does not trigger a cache line demotion on valid memory locations. The instruction triggers only a memory translation on these CPUs and does not further influence program behavior. This behavior seems specific to the `cldemote` op-code, as we could not find another `nop` instruction that exhibits the same timing behavior. The `cldemote` timings for both CPUs are provided in Figure 10. Similar to the Sapphire Rapids CPU, there are clear drops in execution time at the page offsets where the kernel is mapped, which is offset 43 (virtual address `0xffffffff85600000`) for our i7-1260P in Figure 10a, and offset 427 (virtual address `0xffffffffb5600000`) for our i9-13900K in Figure 10b.

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

VI. Discussion and Mitigations

Cache attacks can be *prevented* at three levels: at the hardware level, at the system level, and finally, at the application level. At the hardware level, several solutions have been proposed to prevent cache attacks, either by removing cache interferences, or randomizing them. The solutions include new secure cache designs [79], [80], [45] or altering the prefetcher policy [17]. However, hardware changes are not applicable to commodity systems. At the system level, page coloring provides cache isolation in software [59], [36]. Zhang et al. [89] proposed a more relaxed isolation like repeated cache cleansing. These solutions cause performance issues, as they prevent optimal use of the cache. Application-level countermeasures seek to find the source of information leakage and patch it [5]. However, application-level countermeasures are bounded and cannot prevent cache attacks such as covert channels. In contrast to prevention solutions that incur a loss of performance, using performance counters does not prevent attacks but rather detects them without overhead.

VII. Conclusion

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

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our anonymous shepherd for their guidance, comments, and suggestions. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I6054). Additional funding was provided by a generous gifts from Red Hat, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] O. Acıiçmez, W. Schindler, and C. K. Koc, "Cache Based Remote Timing Attack on the AES," in *CT-RSA*, 2006.

[2] A. Aviram, S. Hu, B. Ford, and R. Gummadi, "Determinating timing channels in compute clouds," in *CCSW*, 2010.

[3] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, 2016.

[5] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," *Cryptology ePrint Archive, Report 2006/052*, 2006.

[6] S. Briongos, P. Malagón, J.-M. de Goyeneche, and J. M. Moya, "Cache misses and the recovery of the full AES 256 key," *Applied Sciences*, vol. 9, no. 5, p. 944, 2019.

[7] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation," in *S&P*, 2019.

[8] D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical," in *USENIX Security*, 2003.

[9] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *AsiaCCS*, 2020.

[10] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using Hardware Performance Counters," *Cryptology ePrint Archive, Report 2015/1034*, 2015.

[11] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Applied Sciences*, vol. 10, no. 3, p. 984, 2020.

[12] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on seL4," in *CCS*, 2014.

[13] A. Costi, B. Johannesmeyer, E. Bosman, C. Giuffrida, and H. Bos, "On the effectiveness of same-domain memory deduplication," in *European Workshop on Systems Security*, 2022.

[14] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript," in *USENIX Security*, 2021.

[15] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security*, 2017.

[16] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," in *USENIX Security*, 2013.

[17] A. Fuchs and R. B. Lee, "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs," in *SYSTOR*, 2015.

[18] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and generic microarchitectural hash-function recovery," *S&P*, 2024.

[19] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks," in *USENIX Security*, 2023.

[20] M. M. Godfrey and M. Zulkernine, "Preventing Cache-Based Side-Channel Attacks in a Cloud Environment," *IEEE Transactions on Cloud Computing*, 2014.

[21] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU." in *NDSS*, 2017.

[22] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

[23] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[24] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[25] D. Gruss, F. Schuster, O. Ohrimenko, I. Haller, J. Lettner, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *USENIX Security*, 2017.

[26] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security*, 2015.

[27] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning," *arXiv:1907.03651*, 2019.

[28] N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security," in *Black Hat USA*, 2015.

[29] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *S&P*, 2013.

[30] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *CHES*, 2016.

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

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

[33] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES," in *S&P*, 2015.

[34] ——, "MASCAT: Stopping Microarchitectural Attacks Before Execution," *Cryptology ePrint Archive, Report 2016/1196*, 2017.

[35] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, Cross-VM attack on AES," in *RAID*, 2014.

[36] T. Kim, M. Peinado, and G. Mainar-Ruiz, "StealthMem: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security*, 2012.

[37] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+power: Leaking inaccessible data with software-based power side channels," in *USENIX Security*, 2023.

[38] D. Kohlbrenner and H. Shacham, "Trusted Browsers for Uncertain Times," in *USENIX Security*, 2016.

[39] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, "TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs," in *EuroS&P*, 2020.

[40] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical Cache Attacks from the Network," in *S&P*, 5 2020.

[41] P.-M. Lee, W.-H. Tsui, and T.-C. Hsiao, "The Influence of Emotion on Keyboard Typing: An Experimental Study Using Auditory Stimuli," *PLOS ONE*, vol. 10, pp. 1–16, 2015.

[42] M. Lipp, D. Gruss, and M. Schwarz, "AMD Prefetch Attacks through Power and Time," in *USENIX Security*, 2022.

[43] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security*, 2016.

[44] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[45] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *MICRO*, 2014.

[46] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[47] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *ACM SIGARCH Computer Architecture News*, 2012.

[48] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *DIMVA*, 2015.

[49] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.

[50] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[51] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.

[52] J. V. Monaco, "Feasibility of a Keystroke Timing Attack on Search Engines with Autocomplete," in *IEEE Security and Privacy Workshops (SPW)*, 2019.
[53] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.
[54] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*, 2016.
[55] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.
[56] C. Pereida García, B. B. Brumley, and Y. Yarom, "Make Sure DSA Signing Exponentiations Really Are Constant-Time," in *CCS*, 2016.
[57] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic Analysis of Randomization-based Protected Cache Architectures," in *S&P*, 2021.
[58] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.
[59] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource Management for Isolation Enhanced Cloud Services," in *CCSW*, 2009, pp. 77–84.
[60] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *ASPLOS*, 2021.
[61] G. Saileshwar and M. K. Qureshi, "MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design," in *USENIX Security*, 2021.
[62] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.
[63] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.
[64] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.
[65] M. Schwarzl, E. Kraft, and D. Gruss, "Layered Binary Templating," in *ACNS*, 2023.
[66] M. Schwarzl, E. Kraft, M. Lipp, and D. Gruss, "Remote Page Deduplication Attacks," in *NDSS*, 2022.
[67] M. Seddigh and H. Soleimany, "Enhanced Flush+Reload Attack on AES," *ISC International Journal of Information Security*, 2020.
[68] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Dependable Systems and Networks Workshops (DSN-W)*, 2011.
[69] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security*, 2001.
[70] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory Deduplication as a Threat to the Guest OS," in *EuroSys*, 2011.
[71] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *USENIX Security*, 2017.
[72] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *CCS*, 2015.
[73] T. Van Goethem, C. Pöpper, W. Joosen, and M. Vanhoef, "Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections," in *USENIX Security*, 2020.
[74] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen," in *CCSW*, 2011.
[75] P. Vila, B. Köpf, and J. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.
[76] S. Walton, "How Screwed is Intel without Hyper-Threading?" 2019. [Online]. Available: https://www.techspot.com/article/1850-how-screwed-is-intel-no-hyper-threading/
[77] D. Wang, A. Neupane, Z. Qian, N. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries," in *NDSS*, 2019.
[78] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86," in *USENIX Security*, 2022.
[79] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 494, 2007.
[80] ——, "A Novel Cache Architecture with Enhanced Performance and Security," in *MICRO*, 2008.
[81] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security*, 2019.
[82] J. C. Wray, "An Analysis of Covert Timing Channels," *Journal of Computer Security*, 1992.
[83] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *S&P*, 2019.
[84] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.
[85] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," *Cryptology ePrint Archive, Report 2015/905*, 2015.
[86] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *JCEN*, 2017.
[87] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds," in *RAID*, 2016.
[88] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis," in *S&P*, 2011.
[89] Y. Zhang and M. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *CCS*, 2013.

APPENDIX

A. Covert Channel and Noise Resilience

To determine the noise resilience, we run the 1-bit covert channel for all attacks with a varying number of background workers that perform cache-heavy operations. The results of our measurements are shown in Figure 11 and Figure 12. Optimized Demote+Reload (Figure 11a and Figure 12a) performed the best, while DemoteContention (Figure 11b) performs the worst, dropping to roughly 0 Mbit/s capacity. All other attacks lose $\approx 80\% - 95\%$ of their capacity with the maximum number of worker threads.

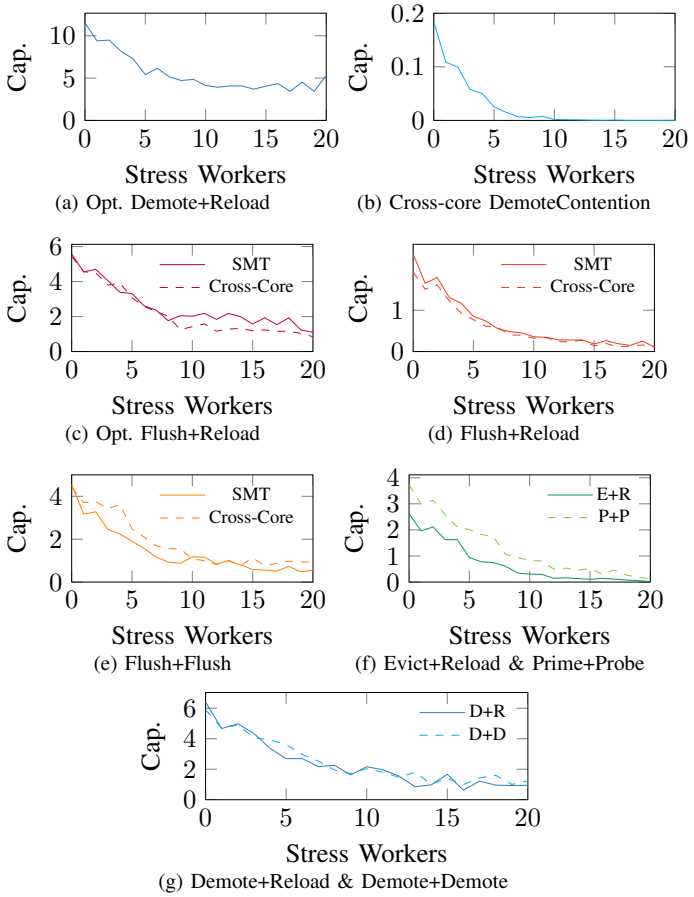
B. Performance Counter Values

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

C. Further Emerald Rapids Data

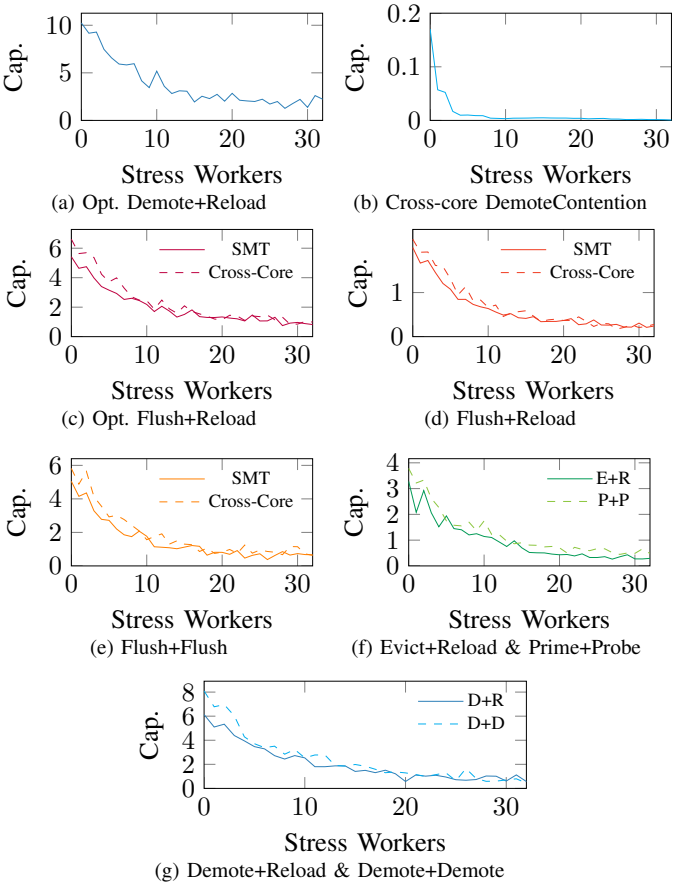
In this section, we present visualizations and summarizations of further data collected on our Xeon Silver 4514Y. Histograms of all the attack's hit-miss timings are shown in Figure 13. Similar as on our Sapphire Rapids CPU, SMT Flush+Reload has the largest margin at 166 cycles as it distinguishes between a fast L1 hit and a slow L3 miss (Figure 13b). Evict+Reload has the lowest margin at only 4 cycles, which is the difference between an L1 hit and an L2 hit (Figure 13c).

The attack times for all attacks are visualized in Figure 14. The results on our Xeon Silver 4514Y are very similar to the measurements on our Xeon Silver 4410T, with Flush+Reload having the highest attack time at 462.8 cycles, and Demote+Demote and DemoteContention having the lowest attack times at 137.6 cycles.



(a) Opt. Demote+Reload (b) Cross-core DemoteContention (c) Opt. Flush+Reload (d) Flush+Reload (e) Flush+Flush (f) Evict+Reload & Prime+Probe (g) Demote+Reload & Demote+Demote

Fig. 11. Noise resilience of single-bit covert channels of all tested attacks. We increase the system noise with stress-ng worker threads until the number of cores of our Xeon Silver 4410T and provide the true capacity in **Mbit/s**. The capacity deteriorates for all attacks, indicating their respective noise resilience.



(a) Opt. Demote+Reload (b) Cross-core DemoteContention (c) Opt. Flush+Reload (d) Flush+Reload (e) Flush+Flush (f) Evict+Reload & Prime+Probe (g) Demote+Reload & Demote+Demote

Fig. 12. Noise resilience of single-bit covert channels of all tested attacks. We increase the system noise with stress-ng worker threads until the number of cores of our Xeon Silver 4514Y and provide the true capacity in **Mbit/s**. The capacity deteriorates for all attacks, indicating their respective noise resilience.

TABLE V. PERFORMANCE COUNTER VALUES FOR 10^6 RUNS OF EACH TESTED ATTACK WITHOUT A VICTIM ACCESS ON OUR XEON SILVER 4410T.

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

TABLE VI. PERFORMANCE COUNTER VALUES FOR 10^6 RUNS OF EACH TESTED ATTACK WITH A VICTIM ACCESS ON OUR XEON SILVER 4410T.

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

The blind-spot size and the effect of a delay between each attack iteration are shown in Figure 15. The relative blind spots are almost identical to the ones measured on our Sapphire Rapids CPU, with DemoteContention having the largest blind spot at 89.5 %, closely followed by cross-core Flush+Reload with 86.3 %. Flush+Flush and Demote+Demote have no measurable blind spot.

The temporal difference is visualized in Figure 16. Similar to the other metrics, the temporal differences are very similar to our measurements on our other Sapphire Rapids CPU, with the only exception being cross-core Flush+Reload. The standard deviation for cross-core Flush+Reload is significantly lower at 15 ns compared to the 24 ns measured on our Sapphire Rapids CPU. This is most likely the result of some microarchitectural changes in the CPU.

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

TABLE VII. PERFORMANCE COUNTER VALUES FOR 10^6 RUNS OF EACH TESTED ATTACK WITHOUT A VICTIM ACCESS ON OUR XEON SILVER 4514Y.

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

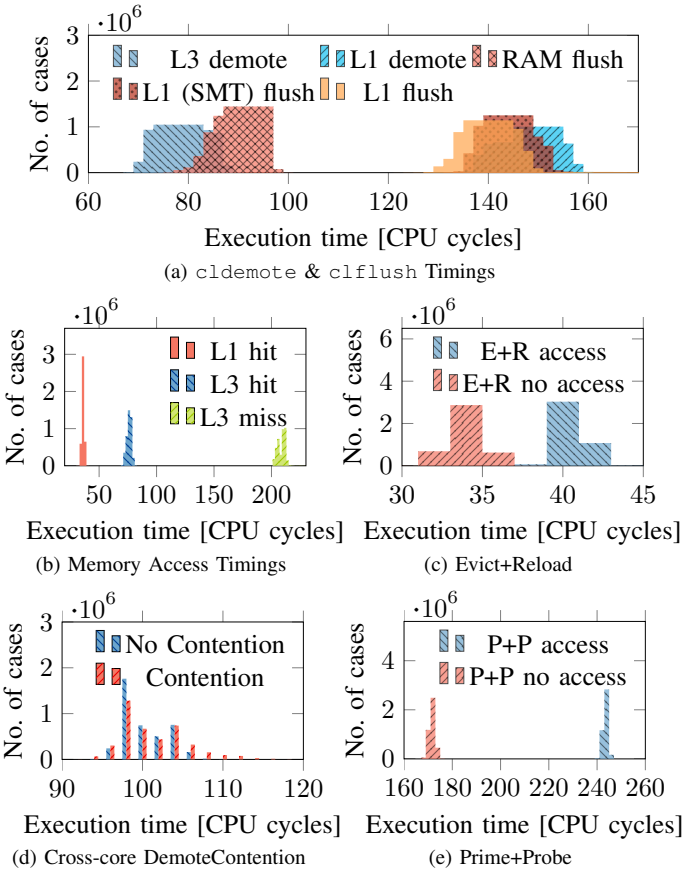
TABLE VIII. PERFORMANCE COUNTER VALUES FOR 10^6 RUNS OF EACH TESTED ATTACK WITH A VICTIM ACCESS ON OUR XEON SILVER 4514Y.

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

TABLE IX. SINGLE-BIT (1-BIT) AND MULTI-BIT (N-BIT) COVERT-CHANNEL TRUE CAPACITY IN **MBIT/S** AND ERROR RATIO OF ALL TESTED ATTACKS ON AN INTEL XEON SILVER 4514Y.

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

and cross-core Flush+Flush has the best multi-bit performance at 10.26 Mbit/s.



(a) `cldemote` & `clflush` Timings

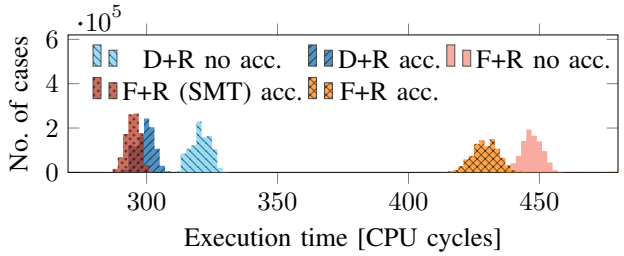
(b) Memory Access Timings

(c) Evict+Reload

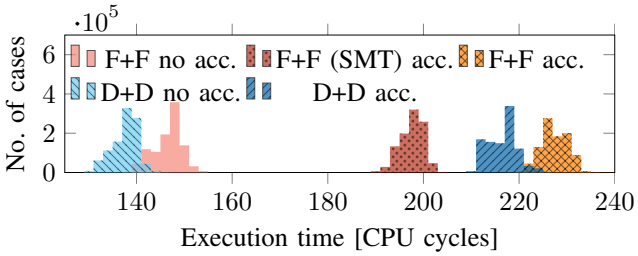
(d) Cross-core DemoteContention

(e) Prime+Probe

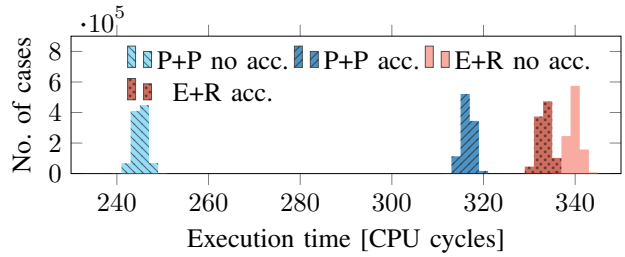
Fig. 13. Timing histograms for all tested attacks on our Xeon Silver 4514Y.



(a) Flush+Reload & Demote+Reload



(b) Flush+Flush & Demote+Demote



(c) Prime+Probe & Evict+Reload

Fig. 14. Execution time in cycles of a single attack iteration for all tested attacks on our Xeon Silver 4514Y.

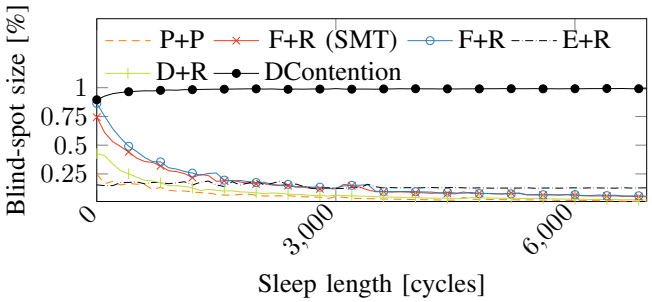
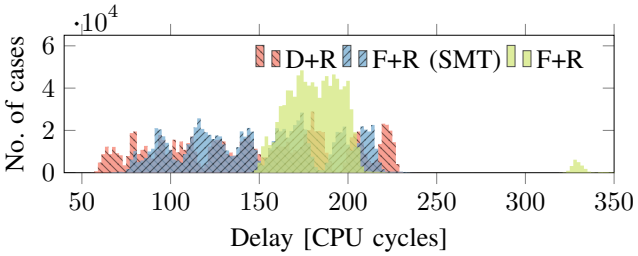
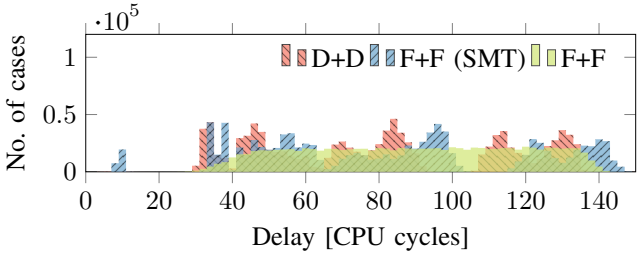


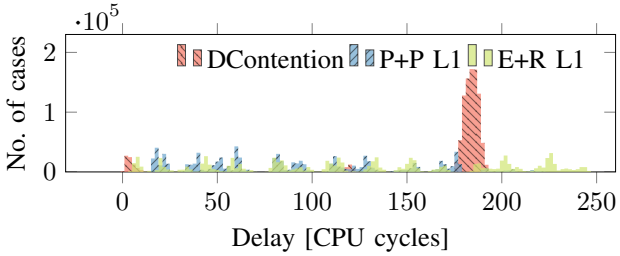
Fig. 15. Blind-spot size of all tested attacks that have a blind-spot for varying delay lengths after each attack iteration on our Xeon Silver 4514Y. The blind-spot size is given in the percent of a single attack loop iteration (including the delay).



(a) Flush+Reload & Demote+Reload



(b) Flush+Flush & Demote+Demote



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

Fig. 16. The delay between memory access and the start of the detection period that detected the access for all attacks on our Xeon Silver 4514Y.