Fast and Efficient Secure L1 Caches for SMT

Lukas Giner, Roland Czerny, Simon Lammer, Aaron Giner, Paul Gollob, Jonas Juffinger, and Daniel Gruss

Graz University of Technology
first.last@tugraz.at

Abstract. Secure randomized caches use the latency budgets of last-level caches to isolate data by security domain. In contrast, L1 caches are very latency- and size-constrained (by cache ways and page size), hindering both the adoption of secure randomized designs and increases in size without losing backward compatibility due to page size changes. We propose a new secure and larger L1 cache design for SMT cores: SMT Cache. SMTCache uses separate, identical L1 caches (slices) to isolate security domains. The overall cache size scales with the number of SMT threads, with individual slices mirroring current designs without changing the page size. SMTCache consumes less power than larger sets and does not increase hit latency. We show that SMTCache is a principled mitigation against L1 cache attacks and fundamentally precludes vulnerabilities like L1TF. Further, we measure that SMTCache improves L1 cache performance compared to current designs and even remains competitive with larger caches. For instance, on a system with SMT-2, SMTCache provides equivalent hit ratios across the SPEC CPU2017 suite to a state-of-the-art L1 cache of comparable size while improving system security and significantly reducing energy costs.

1 Introduction

Caches hide the high access times of main memory by storing recently used data within the CPU. With low latency, limited space, and sharing across security contexts, they are an attractive target for attacks. Attacks range from side channels [7, 16, 22, 35, 46] to severe vulnerabilities like Meltdown [29] and its variants [8, 45, 47, 49, 54, 56]. All of these attacks rely on the cache being a shared resource without security domain isolation.

While recent secure cache proposals address this problem for the large last-level caches [10, 14, 41, 43, 51, 57], low latency is crucial for L1 caches. Hence, we cannot simply apply last-level secure cache designs to the L1 cache. Furthermore, partitioning the L1 cache is costly as it is already very size-constrained. Due to the virtual indexing, the L1 size is determined by the number of ways times the page size, which for commodity laptop, desktop, and server CPUs has been 4 KiB

for over a decade. This limits an 8-way L1 cache to a maximum size of $4\,\text{KiB} \cdot 8 = 32\,\text{KiB}$. Thus, there are currently only two options to increase the L1 cache size, without even taking security considerations into account: First, like some recent Intel server CPUs, the number of ways per set is increased (e.g., from 8 to 12), at the cost of a super-linear increase in energy consumption [1,37]. Second, like recent Apple CPUs, the page size could be increased (e.g., to 16 KiB [12]). However, this is only possible given Apple's firm control of both hardware and software on their machines, reducing the need for backward compatibility. Still, with this change, Apple increased the L1 cache size to 128 KiB. While this shows that *die area near the execution core is available*, it further emphasizes the page size as a limiting factor to efficiently scale the L1 and its lack of security that becomes increasingly interesting for attacks.

This leads us to investigate the following research questions:
How can we prevent L1 cache attacks in a principled way? Is it possible to increase L1 cache size and security without substantial efficiency loss or software-breaking changes? What is the energy cost of scaling the L1 cache?

In this paper, we propose SMTCache, a secure L1 data cache (L1D) design that offers advantages in L1 cache size, security, and energy efficiency on CPUs with simultaneous multithreading (SMT). SMTCache stays within the existing ISA specifications as well as power and latency budgets of commodity off-the-shelf CPUs. We achieve this by creating independent L1D *slices* (like L3 slices) accessed by a memory address and a security domain. Every domain has its own L1 slice, ensuring principled data separation.

In our default configuration, SMTCache does not require operating system support and switches domains based on existing mechanisms, *i.e.*, user mode and kernel mode. This provides out-of-the-box data separation between processes and the operating system. From the point of view of processes, they have their own private L1D cache without interference or data leakage from one SMT thread or process to another. Often, the number of processes active on a core is higher than the number of L1D slices (we evaluate up to 9 slices per core). If a process is scheduled to run on a core where it does not have a slice assigned, the least recently used (LRU) slice is flushed to higher cache levels, and the new process gets this slice exclusively until it is eventually flushed for a different process.

Each slice can be the same size as current L1D caches, as sets are addressed by their domain in addition to the virtual address, thereby sidestepping the page-size limitation while scaling the cache. The maximum active number of slices is limited to the number of SMT threads. For SMT-2, this doubles the effective available L1D cache space for simultaneously running processes. When the operating system schedules only a small number of processes on a core, this markedly increases performance as processes are not competing over cache space. With more slices than SMT threads, inactive slices store currently unused data for different processes that are not running while only drawing static power.

We evaluate the performance of SMTCache in CacheSim [15] and on traces recorded on a native Linux server running different workloads, as well as the functionality via micro-benchmarks with Linux on gem5 [5,33]. Our evaluation

shows that performance scales very well with SMT and often exceeds the performance of an equivalently large standard cache due to the inherent thrashing protection. For SMT-2, and especially SMT-4 (Section 6.1), SMTCache increases the available L1 cache per thread while guaranteeing fairness and security. We find that a number of slices higher than SMT ways + 1 only minimally improves performance, as processes rarely return to an empty L1D cache at that point.

Contributions. In summary, our main contributions are:

- We propose a novel secure L1 cache design, SMTCache, providing strict isolation between security domains on the hardware level.
- SMTCache builds on the synergetic introduction of security and performance enhancements to expand cache sizes without breaking backward compatibility.
- We provide a security argument for SMTCache, showing that it mitigates a range of state-of-the-art attacks in a principled way.
- We evaluate the performance of SMTCache in many different configurations and demonstrate that it offers competitive hit ratios even when compared to larger monolithic L1 caches like Apple's M1.

Outline. Section 2 presents background and Section 3 the design. Section 4 discusses energy and area costs and Section 5 security. Section 6 evaluates the performance. Section 7 presents related and future work. Section 8 concludes.

2 Background

In this section, we discuss caches, limiting factors for their size, traditional and secure L1 designs as well as their attack surfaces.

Caches. CPU caches are buffers close to the CPU, orders of magnitude smaller than main memory. They hide high memory access latencies for recently used data. In modern set-associative caches, addresses are statically mapped to one of many sets of cache lines and occupy any of the ways within that set.

Traditional caches are organized in a 3-level hierarchy, with the cache closest to the CPU (L1) being the smallest and fastest, with acritical impact on CPU performance. Unlike higher-level caches, most L1 caches use the virtual address to index the cache set to reduce latency by already looking for a cache line while translating the address. To not map a physical address to multiple sets, the index is taken only from bits shared with the virtual address, *i.e.*, the page offset, typically 12 bit. This limits the L1 cache size by the page size and number of ways, e.g., 4 KiB·8 ways = 32 KiB. Hence, traditional L1 cache size can only be increased with the page size, the number of ways, or by dropping the virtually indexed design. Intel increased the L1D cache size of the "Core" CPUs from 16 KiB to 32 KiB and 48 KiB with the number of ways, from 4 to 8 and 12. In the Apple M1, the 16 KiB page size allows for a 128 KiB L1D cache and a 192 KiB L1 instruction cache (L1I).

The drawback of increasing the associativity is a rising energy cost per access due to two factors: Firstly, the number of tags that need to be searched to determine a cache hit increases proportionally to the number of ways. Implementations may also load the data of all lines in a set at the time of the tag compar-

ison [36], further adding to the increased energy demand. Secondly, super-linear components to the power draw grow with the size of a cache set [1, 37].

Cache Coherence. When multiple cores access the same memory location on a CPU with private, per-core caches, writes on one core need to become visible to other cores as soon as possible. There are various coherence protocols that ensure this memory consistency. In the simplest case, caches need to know if a local copy of a cache line is modified, shared, or invalid (MSI). There are two common methods to implement coherence protocols, *snooping* and *cache directories* [38]. Snooping protocols work by broadcasting each memory request to all caches but are only viable for a low number of caches. Directories can solve this problem by centralizing the protocol's state information at a point of coherence. In inclusive cache hierarchies, the last-level cache (LLC) stores all cache lines found in lower levels and can, therefore, also act as the directory.

Cache Attacks. As caches are shared and introduce timing differences, they have been a popular target for side-channel research. In a cache attack, an attacker observes different access times to their data to infer a victim's behavior. With knowledge about cache architecture, refined cache attacks are possible.

Attacks on Cache Metadata. The simplest form of these attacks are time-driven attacks, such as Bernstein's attack [4] or **Evict+Time** [39]. The latter, for example, evicts an AES T-Table entry by filling the cache set with attacker memory. By timing the victim's execution the attacker can infer if this entry was used. A more noise-resilient evolution is **Prime+Probe**, where the attacker first *primes* a set by filling it, and then *probes* it by timing accesses. If the victim used a line in this set in between, the attacker measures a longer access and can observe the victim's accesses at the granularity of cache sets. Prime+Probe requires a set of addresses that map to the same cache set, called an *eviction set*. The **Flush+Reload** [60] attack enables cache line accuracy if attacker and victim share memory, by not relying on set conflicts but measuring the target line directly. The attacker uses the `clflush` instruction to evict the targeted address precisely, and later measures it again to see if the victim has brought it into the cache. Achieving shared memory with a victim is more challenging than co-location, and `clflush` might be unavailable. **Evict+Reload** [17, 27] removes the need for `clflush` by replacing it with set eviction like Prime+Probe.

Attacks with Caches. Meltdown, Spectre, etc. [28, 49, 54] use caches for their covert-channel to recover data encoded during speculative execution. This is possible because the state of the caches is not reversed when a speculatively execution is aborted. Meltdown variants leaking from the L1 Cache (or the Line Fill Buffer) exploit caches that does not check permissions when data is served.

Secure Cache Designs and Related Work. With some of the attacks known for decades, many secure cache designs have been proposed, generally based on two methods: randomization or partitioning. The former tries to obscure access patterns by making them seemingly random, while the later tries to make accesses unobservable. Many designs require complex functions whose latency is too large for the L1 and only target the LLC [10, 14, 30, 40–43, 51–53, 57] assuming the other caches are secure. In Section 7 we detail these secure caches

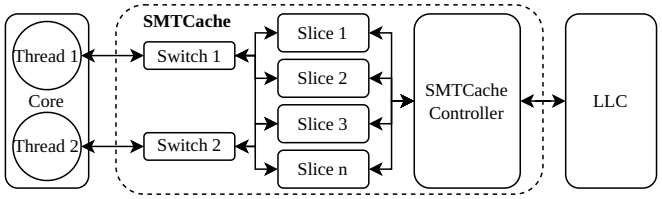


Fig. 1: SMTCache abstract design for n slices. At most 2 slices are active at the same time, one per SMT thread. The SMTCache controller ensures coherence between slices and that SMTCache appears like a normal cache to higher cache levels.

and highlight how SMTCache is orthogonal to many of them and discuss how SMTCache can complement them for improved security and performance.

3 The SMTCache Architecture

At the heart of SMTCache are a number of n identical *slices*; complete L1 data caches with standard parameters, e.g., 8 ways, 32 KiB size as shown in Figure 1. At each context switch (security domain switch), one slice is assigned to the process. Until the next context switch, requests from the SMT thread are statically routed to this slice by a switch. From the perspective of the core, cache hits on this slice behave identical to a standard L1 design cache hit. The communication with the higher-level caches, however, runs through the SMT Cache controller (Section 3.3), presenting SMTCache as a standard L1 cache. This, of course, adds extra latency. While our design could also be used for instruction caches, we focus on L1 data caches to limit the scope.

3.1 Domains

An important aspect of isolation-based designs is how security domains are derived. We propose a basic in-hardware implementation augmented with optional software control. The default configuration changes the slice assignment when the process (PCID/CR3) or the protection ring change. When the protection ring is 3, the CR3 register represents the domain ID, when it is less than 3, it is considered the kernel domain, regardless of the CR3 value. All kernel threads therefore share one slice, while userspace processes are isolated. This ensures security boundaries in line with standard OS process isolation. This is the backward-compatible mode of the design that works regardless of OS version.

With OS support, this could be enhanced to be more or less precise via MSRs. A process might, e.g., want to isolate its threads to maximize its L1D cache size, while another might want to share one slice among an entire process group. As this is highly workload dependent, we do not evaluate OS support in this work.
Hypervisors and SGX. With a hypervisor, we can simply consider ring -1 the only mandatory domain in the default configuration. Thus, the hypervisor and guest can never share an L1. Intel SGX [21] has the unique situation that

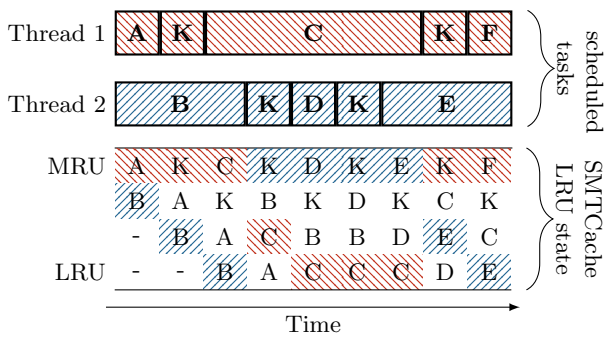


Fig. 2: Domain swapping with modified LRU for SMTCache with 4 slices. Context switches cannot replace active caches and bring the last active slice to the second-most recently used position. As the context switches are performed by the kernel (**K**) its slice is always most or second-most recently used automatically and can never be evicted.

the hardware is generally under the control of the untrusted OS, yet SGX must be secure. We can accommodate this by always treating each enclave as a unique domain, irrespective of any configuration the OS might have chosen.

3.2 Slice Swapping

When a new domain not currently associated with a slice is assigned to a core, one of the slices is chosen to be evicted. On eviction, the hardware flushes all modified (dirty) cache lines to the higher cache levels, unmodifed (clean) cache lines can simply be dropped. When the number of slices equals the number of SMT ways $n = n_{SMT}$, the slices can be statically assigned to logical cores, and the current slice will be reused. For $n = n_{SMT} + 1$, the additional slice is always used for the kernel.

When $n > (n_{SMT} + 1)$, SMTCache chooses the slice to be evicted with a modified LRU algorithm. Domains scheduled often are therefore likely to keep their data in an inactive slice while they are descheduled, ready to resume work when they are scheduled again (see Section 6.2).

Figure 2 shows an example of our modified LRU for 4 slices and SMT-2. A thread is moved to the MRU position the moment it is newly scheduled on the core. Because the process scheduling is always performed by the kernel (**K**) it can always only be at the most or second-most recently used position. This ensure that the slice of the kernel is "reserved" and never evicted, guaranteeing fast kernel entries when there are more slices than SMT ways. We modify standard LRU such that active threads can never have their slice taken from them, regardless of their LRU position. Additionally, swapped-out threads are placed in the second-most recent position. This prevents long-running threads from immediately being the new eviction candidate (see switch from B to D or C to F in Figure 2). For a configuration of 5 slices, this means that 4 slices will be

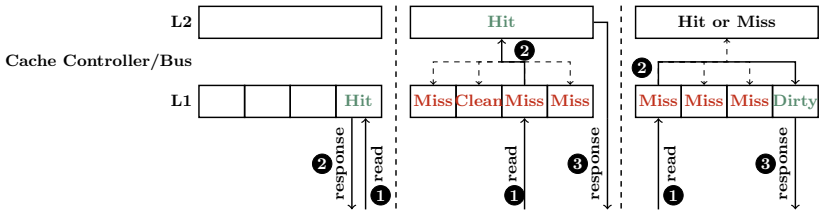


Fig. 3: A read request can be satisfied by the same slice (left), by the L2 (middle), and by a sibling slice (right). Hits on clean lines in sibling core are served from upper levels to prevent side-channel leakage.

available for user space domains. If the kernel is scheduled on both SMT threads at the same time, they share one cache slice similar to normal CPUs where both SMT threads share the L1 cache.

3.3 SMTCache Controller and Coherence

Multi-core processing with several simultaneously executing threads and shared, writeable memory requires caches to implement a coherence protocol that ensures all threads work with consistent copies of modified data. Information about changes to a location in one cache is propagated to other caches as soon as possible. SMTCache includes an extra coherency controller that facilitates security-aware snooping for the L1 slices (see Figure 1) to support shared memory, stay coherent between threads and processes, and curb high lookup latency for writeable shared memory. It handles misses from currently active L1 caches and requests from higher cache levels.

The snooping protocol works like the standard coherence between L1I and L1D caches. It avoids moving the attack surface from the L1 cache slice one layer higher to a directory [59], as there are no evictions from underprovisioning. Contrary to caches on different cores, the slices are also in much closer physical proximity, which reduces the cost of snooping. To reduce the energy costs of querying all slices for data, we propose a dual-mode line lookup for each cache line's tag and state data. When answering a request from the local core, only the currently active slice's set is searched, and tag, state, and data can be loaded in parallel. In response to a sibling-slice or remote miss, tag and state information from all slices is requested in parallel, without loading cache line data simultaneously.

Requests from the core first go to the assigned slice, then, for a miss, are forwarded to other slices and L2 cache at the same time (Figure 3). The cache controller can also aggregate cache line states w.r.t. upper levels, it can distinguish between a total miss in SMTCache and a hit on a clean or dirty line in a sibling slice. When a miss occurs in the controller, the request is served from the L2 cache. Likewise, when the data is found but is clean, the request is still served from the L2 cache to prevent Flush+Reload (see Section 5). When a sibling slice

Table 1: Area and power overheads estimated with McPAT [26] and CACTI [37].

Number of Ways	8-way	12-way	16-way	2 slices (8-way)	24-way	3 slices (8-way)	32-way	4 slices (8-way)	40-way	5 slices (8-way)
Total L1 Cache Size	32 KiB	48 KiB	64 KiB	64 KiB	96 KiB	96 KiB	128 KiB	128 KiB	160 KiB	160 KiB
Number of SMT Cores†	1	1	2	2	2	2	4	4	4	4
Bus Area [mm^2]	0.37	0.37	0.38	0.38	0.39	0.40	0.41	0.42	0.53	0.54
Bus Peak Dynamic [W]	2.04	2.08	2.11	2.13	2.15	2.13	2.27	2.14	2.41	2.45
Bus Subthreshold Leakage [W]	0.05	0.05	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
Bus Runtime Dynamic [W]	2.04	2.08	2.11	2.13	2.15	2.13	2.27	2.14	2.41	2.45
L1 Dynamic read energy [nJ]	2.53	6.87	11.22	2.53	29.16	2.53	47.09	2.53	81.55	2.53
L1 Dynamic write energy [nJ]	2.58	7.01	11.45	2.58	29.73	2.58	48.01	2.58	82.72	2.58
L1 Standby Leakage [mW]	42.50	64.41	86.33	85.83	132.47	127.49	178.61	169.98	276.95	212.48
L1 Area [mm^2]	3.77	9.26	14.75	7.63	36.52	11.32	58.30	15.10	100.54	18.87
L1 Max. Total Leak. (2 loads+stores/cycle) [W]	10.26	27.84	45.42	10.30	117.90	10.34	190.38	10.38	328.83	10.43
L1 Max. Total Leak. (4 loads+stores/cycle) [W]	-	-	90.77	20.52	235.70	20.56	380.63	20.60	657.47	20.64
L1 Max. Total Leak. (8 loads+stores/cycle) [W]	-	-	-	-	-	-	761.04	41.02	1314.59	41.07

† For a fair comparison, we adjusted the number of SMT cores to reflect the L1 cache sizes: 1 SMT core below 64 KiB, 2 SMT cores for the 64 KiB to 96 KiB range, and 4 SMT cores above. We simulate the results for 3 different configurations for the load and store ports from 2 to 8 loads and store per cycle. The maximum total leakage significantly changes with the number of SMT cores and the number of loads and stores per cycle. As the slices of SMTCache act as independent caches, they scale almost linearly in the maximum total leakage.

contains the requested data and it is dirty, it can be served directly from there with limited security concerns. Since the position of the line is already known from the lookup request, the corresponding set does not need to be searched again, saving time and energy.

The slices together with the controller also keep track of copies and only forward modified data when the last copy is evicted or a coherence message from the upper level requires it. This avoids generating unnecessary traffic up the hierarchy when a line is evicted from one slice but still present in others. From a top-down view, SMTCache presents as a standard cache controller within the larger coherency protocol while maintaining its own internal state. Upon a request from a remote core, the controller can locate the address in the slices and adjust the cache lines accordingly, *i.e.*, changing ownership, responding with data, or flushing lines. Again, finding an address via the first broadcast already includes the location in the slice's set, so an extra lookup is unnecessary.

4 Energy and Area Estimation

Estimating energy and area overheads for commercial large-scale CPUs is difficult as CPU vendors do not open-source competitive state-of-the-art designs. Therefore, we follow the methodology of prior work [43, 52] and use McPAT [26] with CACTI [37] to estimate energy and area overheads, close to the actual hardware costs for commercial large-scale CPUs [26]. Like Townley et al. [52], we use the most recent Intel Xeon that McPAT supports. For the cache, we configure CACTI [37] directly, providing more fine-grained configuration and detailed information. The slices of SMTCache behave like separate caches that each contribute to the static power consumption of the CPU. We interpolate unsupported non-power-of-two values.

Area. The main area overhead of SMTCache is storage area, closely resembling that of L1 caches in recent Apple CPUs. An increase from 32 KiB to a 128 KiB

cache (like Apple's) comes with a proportional area growth of factor 4. The bus area increase is entirely negligible compared to the storage. SMTCache has a about 1 % area overhead from a basic 8-way cache due to additional complexity and tag bits added. However, SMTCache scales much better than a naive extension of current cache designs with a higher number of ways.

Energy. The dynamic read and write energy for a single operation (2.53 nJ and 2.58 nJ respectively) stays at the level of the initial cache design (see Table 1). While standby leakage increases significantly it is negligible compared to overall power consumption. For the maximum total leakage, we use the metrics of a current CPU, *i.e.*, a throughput of 0.5 cache reads and writes per cycle. On a CPU with a 4 cycle cache latency, two load and two store ports, and 4 GHz clock, the upper bound for the throughput is 2 billion cache reads and cache writes each per second, which we also empirically tested on an Intel i7-8565U CPU.

For a fair comparison across all designs, we compute the maximum total leakage for 1 SMT core for all caches with less than 64 KiB, 2 SMT cores for all caches from 64 KiB to 96 KiB, and 4 SMT cores for 128 KiB or more. As SMT Cache slices act as entirely separate L1 caches, their energy consumption only increases linearly with the number of slices. The two slice variant of SMTCache has twice as much maximum total leakage, as both caches can be fully utilized by the two SMT threads. However, even at this point the maximum total leakage is lower than the 12-way L1 cache without SMT and significantly lower than the 16-way L1 cache with two SMT threads. This trend continues for the 96 KiB to 160 KiB caches. The energy costs for the 40-way L1 cache are particularly prohibitive, whereas SMTCache with SMT-4 support stays below the maximum total leakage of the 16-way L1 cache.

5 Security

SMTCache provides strong isolation guarantees for the L1 cache. Therefore, we discuss how different cache contention and cache utilization channels are mitigated by our design. However, equally importantly, we show how SMTCache is a defense-in-depth against data leakage attacks.

Data Leakage (Defense in Depth). The strict separation of L1 slices ensures that the L1 cache can no longer be a source for leakage of data at rest, such as Meltdown [29, 49] and L1TF [49, 54, 56]. As requests from one domain are never directly routed to the slice of a different domain, the active L1 slice can never respond with data outside its domain. The request to other domains is only issued with the request to the L2, which happens after the permission check on Meltdown-affected hardware. Though orthogonal to SMTCache, a similar separation (or static partitioning) of the line fill buffer (LFB) could be implemented to additionally prevent leaking data in use, as seen in several microarchitectural data sampling (MDS) attacks [8, 45, 47]. Though these vulnerabilities have been mitigated in current CPU generations, designs with clear isolation boundaries provide defense in depth against possible future leakage from similar sources. We conclude that had these processors already followed a design like SMTCache,

Meltdown [29, 49] and L1TF [49, 54, 56] would have had very little security impact.

Kernel Domain. As mentioned in Section 3.1, the kernel shares a single domain. This is in line with standard process isolation but leaves open the possibility of (transient) confused deputy attacks. We weigh this against the significant overhead of providing each process with a separate kernel slice. We consider this an acceptable tradeoff, primarily because confused deputy attacks in the case of SMTCache require both a disclosure gadget in the victim's kernel code and a leakage gadget in the attacker's. Additionally, this attack surface is known, and gadgets have been systematically reduced in recent years.

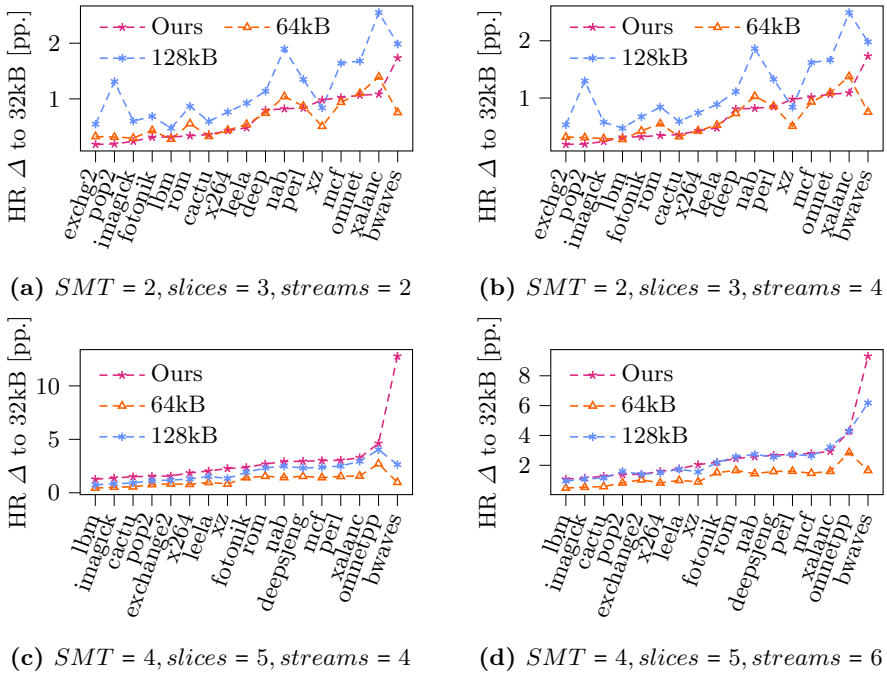
OpenSSL AES. The AES T-Table implementation in OpenSSL is often considered as a benchmark for side channels. The typically page-aligned block of T-Tables (`Te` and `Td`) is accessed during the encryption, e.g., in the first round with a byte-wise xor of plaintext and key. With SMTCache, the initial `prefetch256` call loads the tables into the L1 cache, *i.e.*, they are placed in separate slices of SMTCache. Consequently, we cannot observe any contention.

mbedTLS RSA. Another side-channel attack commonly used as a benchmark is the mbedTLS RSA implementation. mbedTLS uses a windowed square-and-multiply implementation. However, prior attacks [32,48] exploited that a window size of 1 results in a simple square-and-multiply where the buffer containing the exponent is used in different ways, allowing to observe different contention patterns With SMTCache, the buffer is first loaded into the L1 cache, *i.e.*, again in separate slices of SMTCache where we cannot observe any contention.

Generic Side Channels. In general, Prime+Probe builds on the foundational assumption that an attacker can find the set that the victim process' targeted address is cached in and interact with it. Specifically, the *Prime* step fills the entire set, thereby evicting the victim cache line. The *Probe* step then measures how many of the attacker's own addresses are still cached after the victim has executed some code. If an address has been replaced, the attacker infers that, with some likelihood, an address from the victim was loaded. SMTCache cuts this primitive off at the root, as two different security domains cannot interact with each other's cache line allocation anymore. As the sets are separated in both the slice and the L1 directory, the victim's set contents are unaffected by Prime+Probe or other attacks that manipulate the replacement algorithm.

Flush+Reload and Flush+Flush rely on shared memory between victim and attacker. However, with SMTCache, sibling slices do not respond to requests for unmodified data (see Figure 3 middle). Thus, neither Flush+Reload nor Flush+Flush on unmodified data are possible on SMTCache.

Cache side channels on writeable shared memory are still possible. However, this is a special case that was not handled by prior work on secure last-level caches either, as writeable shared memory already requires trust between victim and attacker for these shared memory regions. Hence, we also conclude that given the lack of a plausible threat model it is no case that SMTCache should cover.



(a) $SMT = 2, slices = 3, streams = 2$ **(b)** $SMT = 2, slices = 3, streams = 4$

(c) $SMT = 4, slices = 5, streams = 4$ **(d)** $SMT = 4, slices = 5, streams = 6$

Fig. 4: Average simulator hit ratios over SPEC-speed 2017 benchmark combinations of different L1D cache configurations compared to a standard 32 KiB cache. Each datapoint represents the average hitrate of that benchmark measured in all combinations with other benchmarks. Base hit ratios are around 90-99%. Benchmarks sorted by ascending SMTCache hit ratio.

6 Performance Evaluation

As gem5 lacks SMT support, we cannot use it to test SMTCache performance, as its benefits only materialize with SMT. Instead, we evaluate performance in CacheSim [15] and on an Intel CPU, both with SMT, using the SPEC benchmark. We evaluate real-world single-threaded and SMT switching behaviour on Linux server workloads over several hours, resulting in data for SMTCache performance estimates for different numbers of slices.

6.1 CacheSim Hit Ratio Simulation

We use CacheSim [15] to evaluate hit ratios in SMTCache in different SMT configurations and two levels of cache. Like prior work [11, 14, 57], we use a representative sample of 250 million instructions from SPECspeed CPU 2017 benchmarks. We use a standard 8 way, 32 KiB L1 instruction cache, combined with the different L1 data caches we evaluate.

SMT workloads are simulated by interleaving memory accesses of the currently active workloads. We test all 153 pairwise combinations (with repetition)

of 17 SPEC workloads. To fill up to 8 SMT threads, we use multiples of the pairs to create up to 8 workload streams and avoid an explosion of simulation time. We shift the recorded addresses of streams such that no two workloads share memory addresses. To simulate context switches by the operating system, threads change their workload in regular intervals of 3 000 000 accesses, which roughly equals 500 Hz on a 3 GHz machine, assuming 2 memory accesses per cycle. Between each switch, the implementation of SMTCache briefly loads a fictitious kernel domain. In addition to context switches, we also add the option to simulate a number of syscalls in every context switch interval, e.g., 5 syscalls for every context switch. A syscall here is simulated simply by loading the kernel domain and switching back to the last workload.

We examine the hit ratio of these combinations in Figures 4 to 5. In Figure 4, we plot the averages for each benchmark combination. We simulate configurations where the number of workloads is equal or higher than the number of slices. This shows an ideal and non-ideal case for SMTCache. SMTCache performs about on par with a standard cache of equivalent size to the maximum active number of slices, *i.e.*, the number of SMT ways. We only see a significant deviation for the benchmark combinations that include *bwaves* (and, to a minor extent, *xz*), as this workload seems to use a particularly large working set. The example of *bwaves* also demonstrates the thrashing resistance of SMTCache, as thrashing can only spill over to the second thread via evictions caused by inclusivity in higher caches. The combination of 2 *bwaves* workloads (Figure 4a) produces a 1.73 pp higher hit ratio on SMTCache than a standard cache on SMT-2 with 3 slices, compared to 1.98 pp for the 128 KiB standard cache with twice the concurrently available cache memory. This becomes even more pronounced for SMT-4 (Figure 4c) with hit ratio increases of 12.78 pp vs 2.64 pp for SMTCache (5 slices) vs. a 128 KiB cache.

Figure 5 reinforces the result that thrashing resistance becomes increasingly more pronounced with more logical cores. In this graph, the size of SMTCache increases with the number of SMT ways. While SMTCache starts with hit ratios very similar to the standard cache with the corresponding size, we can see that the hit ratio of standard caches quickly drop as the 8 workloads start to interfere more and more, while SMTCache remains somewhat static.

As Figure 6 shows, the increase in hit ratio for each extra slice beyond $SMT+1$ is fairly small, compared to the benefit from increasing the effective cache size. This coincides with our observations in other benchmarks (cf. Section 6.2) that returning to an empty cache is not a significant cost when the uninterrupted runtime is significantly larger than the time it takes to refill the cache. The overhead of the full-flush mitigation is mostly small, but some applications see a significant loss in performance [25]. In our tests, when we add a number of simulated syscalls per context switch similar to what we find in Section 6.2, we see that the gap from 2 to 3 slices grows slightly. Specifically, this occurs when the number of slices is not higher than the number of SMT ways, as then each syscall results in a full cache eviction. For example, in the depicted configuration with 2 slices, 6 workloads and SMT-2, we see the average hit ratio drop from 94.16 pp

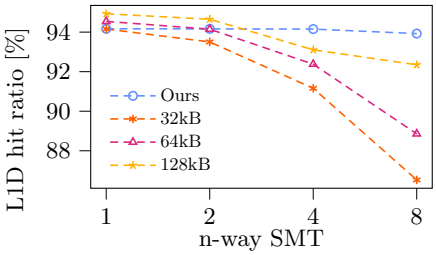


Fig. 5: Mean simulator hit ratio over SPEC combinations for different number of slices with standard designs for reference. 1,2,4,8-way SMT. 8 workloads. $n = n_{SMT} + 1$ for SMTCache.

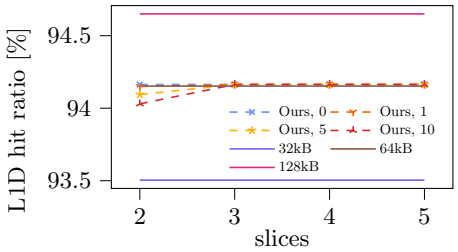


Fig. 6: Mean simulator hit ratio over SPEC combinations for different number of slices with different numbers of syscalls per context switch. Standard designs for reference. SMT-2, 6 workloads.

to 93.50 pp when we increase number of syscalls per context switch from 0 to 10. Therefore, the number of slices in our proposed default configuration of SMT Cache is the number of SMT ways + 1. This ensures that applications always return to a full cache from a syscall.

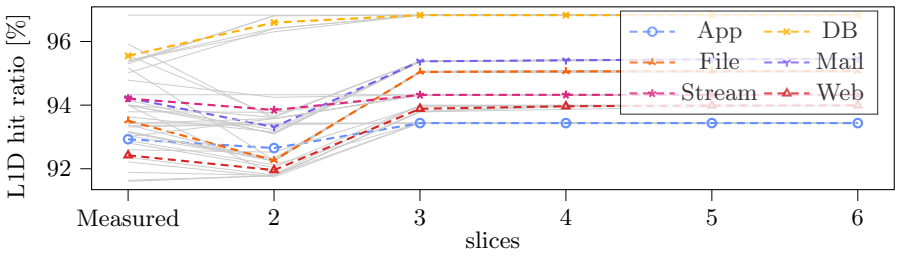


Fig. 7: Measured hit ratios from conventional cache architecture compared to expected hit ratios for different numbers of slices in SMTCache. SMT-2.

6.2 Server Context-Switch Evaluation

We analyze real-world switching behavior with SMT-2 by running several server workloads in different configurations on a native Linux system and simulate the impact of our design. We use the applications proposed by prior work ([19,58]) to evaluate the performance of SMTCache in a realistic cloud scenario. These benchmarks include the following server applications combined in pairs of two to form our server workloads: Apache Tomcat (application server), MySQL (DB server), Postfix (mail server), Samba (file server), FFserver (streaming), and Apache (http server). We run all experiments on an Intel i7-6700K CPU with 4 cores and SMT. We isolate one physical core to eliminate interference from unrelated tasks and execute the workloads on the two SMT cores.

We modify a Linux v5.13 Kernel to record all context switches and syscalls with tracepoints in the `context_switch` and `do_syscall_64` functions. In addition to information about the current and next process, we also record L1D performance counters of hit ratios for all applications. The context switching and syscall information now lets us simulate LRU replacement for varying numbers of slices for each application in different workload combinations. The EER indicates how often a process receives a cleared cache upon being scheduled. A higher number of slices results in a lower EER. With the performance counters, we create two L1 cache hit ratio baselines for each server application on an isolated core. The first baseline is standard switching without flushing, yielding the highest possible hit ratio for each application (HR_{high}). For the second baseline, we evict the cache in every context switch and syscall, producing each application's lowest possible hit ratio (HR_{low}). We assume that the hit ratio decreases linearly from an EER of 0% (HR_{high}) to an EER of 100% (HR_{low}), that a process with a dedicated cache performs roughly the same as a process running on an isolated core, and that kernel threads only interfere minimally on isolated cores. Based on these assumptions, a process receiving a cleared cache each time it is scheduled (EER 100%) has the hit ratio of the process operating on a dedicated core, where the cache is flushed at every context switch and syscall (HR_{low}) and vice versa (EER 0%, HR_{high}).

For the evaluation, we record the application's L1 hit ratios, context switches, and syscalls in all workload combinations. We use the information about context switches and syscalls to compute the EER for each application and varying numbers of slices. We then use the EER values to interpolate between the hit ratio baselines. This yields the expected hit ratios of each application in a SMT Cache architecture with different numbers of slices.

Figure 7 shows the expected hit ratios for all workload combinations with different numbers of slices. The leftmost values represent the measured hit ratios for each application in all workload combinations in a conventional cache architecture. The other values are the expected hit ratios for the respective number of slices in a SMTCache architecture. The grey lines represent specific workload combinations. The colored lines show the average hit ratio for each application. Our evaluation shows an expected performance improvement for SMT workloads when there are *SMT ways + 1* slices compared to the measured value in a conventional cache architecture. We observe a decrease in performance when using as many slices as *SMT ways* in most workload combinations. The cause for this expected performance decrease roots in syscalls. Since syscalls constantly refresh the kernel to be the most recently used cache domain, only one slice is left for parallel tasks. Table 2 shows that, on average, between 4 and 46 syscalls occur during a scheduled period, depending on the application.

Figure 8 depicts each server application's average expected eviction ratios in a SMTCache architecture for different numbers of slices. We see a high eviction ratio when using as many slices as *SMT ways*. For some workloads, the computed eviction ratio is almost 100 % when using 2 slices on our machine with 2 *SMT ways*. Moreover, we see that using more than *SMT ways + 1* slices brings almost

Table 2: Comparison of measured syscall and scheduling metrics.

	App	DB	File	Mail	Stream	Web
Syscalls per scheduled period	6.86	4.84	10.56	10.98	46.34	4.12
Avg. scheduled period (ms)	0.033	0.088	0.066	0.298	9.843	0.028
Avg. time to context switch or syscall (ms)	0.004	0.015	0.006	0.025	0.208	0.005

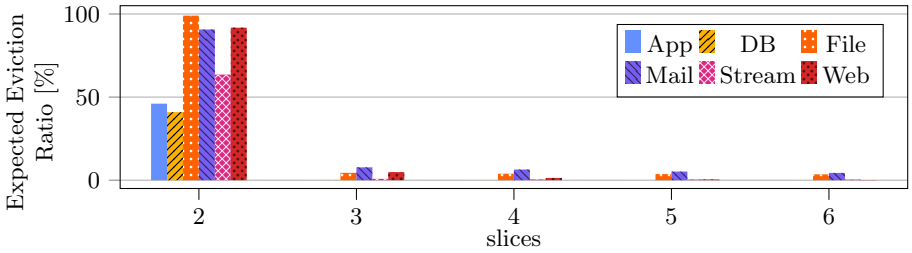


Fig. 8: Expected eviction ratios computed from context switch and syscall information for different numbers of slices. SMT-2.

no performance improvement, given that the eviction ratio is already almost as low as 0 % for 3 slices for all applications.

The average time for a full L1D cache flush in our applications is 3836 cycles (2332 cycles to 10 272 cycles, median 2401 cycles). We flush the L1D cache upon every syscall and context switch to record this value. We observe a higher duration of 10 272 cycles for the stream testcase compared to the other server applications. The average L1D flush duration correlates with the average time between syscalls and context switches for each application. The stream testcase runs uninterrupted for 0.208ms between syscalls and context switches on average (Table 2), allowing a longer time for data to be written to the L1D cache. As all dirty cache lines are flushed to higher cache levels, the L1D cache flush duration increases with the number of writes.

To confirm these L1 flush delays, we also micro-benchmark the full-cache-flush duration in gem5. The results show 350 writebacks on average taking an average of 1700 cycles, comparable to our real-world results.

7 Related and Future Work

Many secure cache designs have been proposed to curb these attacks. We can divide these designs into two groups: designs based on randomization and on partitioning. The former tries to obscure access patterns by making them seemingly random to an attacker, while the latter tries to make accesses unobservable. Many designs require complex functions whose latency is too large for implementation in an L1 cache or simply target the LLC because they assume the underlying caches are secured in a different way. These designs therefore only target the LLC [10, 14, 30, 40–43, 51–53, 57].

While prior partition-based designs may be applicable to the L1, they have so far come at a reduced cache utilization or available cache size. Only way-based partitioning even has the option to increase cache size, though as examined in Section 4, may come with increased energy needs. In this sense, SMTCache achieves an orthogonal goal of offering security and an increase in the overall L1 size, which is complementary to the partition-based designs. We anticipate that for a fully secure system memory subsystem, SMTCache will be combined with one or more of the secure cache approaches for L2 and L3 caches.

Wang et al. [55] presented *PLCache* and *RPCache*. *PLCache* has the ability to lock critical cache lines dynamically in the cache. While less wasteful than static partitioning, the programmer has to mark secrets. Instead, *Random Permutation Cache* tries to prevent observable interference between cache lines of different processes by randomizing their locations with a permutation table. Both *PLCache* and *RPCache* have low-overhead implementations, though Kong et al. [24] point out security-related shortcomings of both. Further approaches have been proposed that offer fine-grained specification of cache partitions, on a cache-line and cache bank granularity respectively [3, 44]. Still, all these designs are size-limited, where SMTCache offers an orthogonal approach to increase the overall L1 size.

Some works explored way-based partitioning [11,23] similar to Intel CAT [18, 20] with additional security by disabling cross-domain cache hits and moderate performance costs. We believe that compared to our work, these way-split designs could not benefit from power savings in the way SMTCache does because of the dynamic nature of the designs. *Hybcache* [9] proposes selective cache partitioning that incurs only a low overhead and only for protected code. It does so by combining random replacement with a small but fully-associative sub-set of the cache for a trusted execution environment. *Jumanji* [50] partitions the L3 cache dynamically by splitting it into software-defined shares. Still, partitioning reduces the effective cache size, which is unsuitable for the size-limited L1 cache. *Newcache* [31] is a pseudo-fully-associative cache with random replacement, that maps address and domain ID of a load to a possible random location in the cache, at moderate performance, area, and energy costs.

TEE-SHirT [2] is a design with partitioned L3 caches and private L2 caches, and non-partitioned private L1 caches. To secure the L1 cache, they simply flush the cache on context switches, which is not overly expensive, given that refills from L2 and L3 are possible. Ge et al. [13] estimated the overhead for L1 flushing to be as low as 1 % on the L4 kernel. However, benchmarks on the Linux kernel showed a significantly higher cost of 10 % [25] on commodity CPUs. SMTCache complements *TEE-SHirT* from a security perspective while offering better performance than L1 flushing. Similarly, for *MI6* [6], SMTCache offers a better alternative to simple L1 flushing.

Future Work. Our experiments have shown that while scaling the number of slices with the number of SMT threads provides a performance boost very similar to an equivalent increase in cache size, going beyond has quickly diminishing returns. The impact of context switches and syscalls, however, shows that

an extra domain for the kernel is useful. An open question for future work is, therefore, if a separate but smaller slice dedicated to the operating system would be a good tradeoff between performance and chip area.

To maintain energy consumption on par with current designs, we assumed the same bandwidth between the core and SMTCache as in standard caches. SMT Cache supports twice that bandwidth for SMT-2. Future work could investigate dynamic scaling of the amount of issued loads and stores by the core to optimally fit power budgets and provide increased performance.

8 Conclusion

We proposed SMTCache, a secure L1D cache increasing cache size and thrashing resistance while being energy efficient. SMTCache achieves strong domain isolation, as security critical memory accesses from one domain are never served from another. With CacheSim and a simulation based on traces from native Linux benchmarks, we also showed that increasing the cache size with multiple slices provides not only the performance boost from simply increasing the cache size, but also from preventing interference between workloads. Lastly, our CACTI power simulation revealed that SMTCache design is significantly more energy-efficient than a traditional design of comparable size. We conclude that the SMTCache design shows promising results in terms of security, performance, and energy efficiency.

Acknowledgments

This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85 and FWF project NeRAM 10.55776/I6054). Additional funding was provided by generous gifts from Intel, Red Hat and Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

A Implementation of SMTCache in gem5

To demonstrate the functionality of SMTCache we implemented it in gem5. At the moment, the gem5 simulator does not support simultaneous multithreading in full system mode. Therefore, we cannot use it to estimate full system-level performance overheads for SMTCache, as it scales with simultaneous multithreading. We still modelled the additional latencies caused by our design realistically, allowing for micro-benchmarks of specific operations. Our implementation aims to functionally represent the features of the design described in Section 3, while working within the limitations of the gem5 codebase.

A.1 Implementation Overview

The gem5 framework simulates a freely configurable set of CPU cores, caches, crossbars (XBar), peripheral devices, etc. connected through ports on with each other. To avoid a complete overhaul of the memory subsystem, our implementation works within the system as much as possible, only swapping the default cache configuration with our SMTCache implementation.

Because all the SMTCache L1 slices behave like independent caches, we can build SMTCache on top of the existing L1 cache implementation. More specifically, we add functionality to perform a full cache flush (Appendix A.3). In a typical CPU, the gem5 CPU core is directly connected to a L1 data cache. For SMTCache, we instead add multiple L1 caches and connect all of them to the CPU core through a custom SMTCache-XBar that implements the switch, as shown in Figure 1. Additionally, this XBar also simulates the SMTCache coherence behavior. The design of the XBar is described in detail in Appendix A.2. The L2 cache in our system is shared between cores and the point of coherence. Usually gem5 connects all L1 caches of all cores to the shared L2 cache through the L2XBar. For SMTCache we do exactly the same, with all L1 data slices of all cores connected to the L2XBar. Finally, we customize the move-into-control-register instruction implementation (`MOV_C_R`) to inform our custom SMTCache-XBar about a `CR3` change.

A.2 SMTCache-XBar Coherence Controller

The CPU core communicates with its SMTCache-XBar by writing to a special address, whenever the `CR3` register is written. This communication is necessary to allow the SMTCache-XBar to respond to a switch in the active domain. The SMTCache-XBar implements the LRU slice eviction and causes a full flush of all lines in the slice about to be assigned to a new domain.

Finally, the SMTCache-XBar also simulates the snooping coherence behavior. In a real implementation, every memory access would go to the active L1 slice, which may then forward the request to the controller if it is a miss. The controller then forwards the request to a slice that contains the cache line if there is one, or the L2 cache. In our gem5 implementation, the SMTCache-XBar directly checks all connected L1 slices and forwards the request to the correct one, if appropriate (*i.e.*, the line is found in the current slice or is modified in a different slice). By adding the correct latencies differentiating a cache hit vs a miss in the active L1 slice, our implementation can simulate the correct overhead. For the tag-matching in the slices, we budget one extra cycle. With this, we implement the behavior of the SMTCache coherence controller without requiring a separate component.

A.3 Flushing

Whenever a process without an associated slice is scheduled, the least recently used cache slice must be flushed and write back dirty data into higher cache levels

or the main memory. Because we only have to write back dirty data, the flush latency is dependent on the number of dirty cache lines. Intel Skylake and later CPUs have a bandwidth of 1 cache line per cycle between the L1 and L2 [34]. This gives a lower bound of 512 cycles for a full flush if every single line is dirty. The flushing can take longer if, e.g., the L2 has to write data into the main memory to make space for the flushed data from the L1 slice. We implement our cache flushing to simulate this behavior and latency.

In gem5, caches can tell the CPU that they are blocked for various reasons. We use this mechanism to block the cache while flushing, as this can take many clock cycles. The CPU waits for the flushing to be finished, treating it as a fully serializing operation. This is important to avoid speculative loads or stores to the wrong slice during this step.

References

1. Al-Tarawneh, M.: An investigation of the impact of instruction cache (i-cache) organization on power-performance trade-offs in the design of scalar processors. European Journal of Scientific Research **115**, 7–26 (11 2013)
2. Arıkan, K., Farrell, A., Cen, W.Z., McMahon, J., Williams, B., Liu, Y.D., Abu-Ghazaleh, N., Ponomarev, D.: TEE-SHirT: Scalable Leakage-Free Cache Hierarchies for TEEs. In: NDSS (2024)
3. Beckmann, N., Sanchez, D.: Jigsaw: Scalable software-defined caches. In: PACT (2013)
4. Bernstein, D.J.: Cache-Timing Attacks on AES (2005), `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
5. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH Computer Architecture News (2011)
6. Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Devadas, S.: MI6: Secure enclaves in a speculative out-of-order processor. In: MICRO (2019)
7. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT (2017)
8. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., Yarom, Y.: Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS (2019)
9. Dessouky, G., Frassetto, T., Sadeghi, A.R.: HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In: USENIX Security (2019)
10. Dessouky, G., Gruler, A., Mahmoody, P., Sadeghi, A.R., Stapf, E.: Chunked-cache: On-demand and scalable cache isolation for security architectures. In: NDSS (2022)
11. Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., Ponomarev, D.: Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. ACM TACO **8**(4) (2011)
12. Frumusanu, A.: Apple Announces The Apple Silicon M1: Ditching x86 - What to Expect, Based on A14 (11 2020), `https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive`
13. Ge, Q., Yarom, Y., Chothia, T., Heiser, G.: Time Protection: The Missing OS Abstraction. In: EuroSys (2019)

14. Giner, L., Steinegger, S., Purnal, A., Eichlseder, M., Unterluggauer, T., Mangard, S., Gruss, D.: Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In: USENIX Security (2023)
15. Giner, Lukas: CacheSim Cache Simulator (2023), https://github.com/isec-tugraz/CacheSim
16. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache Attacks on Intel SGX. In: EuroSec (2017)
17. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security (2015)
18. Herdrich, A., Verplanke, E., Autee, P., Illikkal, R., Gianos, C., Singhal, R., Iyer, R.: Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In: HPCA (2016)
19. Huang, D., Ye, D., He, Q., Chen, J., Ye, K.: Virt-LM: a benchmark for live migration of virtual machine. In: ACM/SPEC ICPE (2011)
20. Intel: Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Cache (2015), https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf
21. Intel: Intel Software Guard Extensions (Intel SGX) (2024), https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html
22. Jiang, Z.H., Fei, Y.: A novel cache bank timing attack. In: ICCAD (2017)
23. Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., Emer, J.: DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: MICRO (2018)
24. Kong, J., Acıiçmez, O., Seifert, J.P., Zhou, H.: Deconstructing new cache designs for thwarting software cache-based side channel attacks. CSAW p. 25 (2008)
25. Larabel, M.: An Early Look At The L1 Terminal Fault "L1TF" Performance Impact On Virtual Machines (2018), https://www.phoronix.com/review/l1tf-early-look
26. Li, S., Chen, K., Ahn, J.H., Brockman, J.B., Jouppi, N.P.: Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In: ICCAD (2011)
27. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security (2016)
28. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security (2018)
29. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., Strackx, R.: Meltdown: Reading Kernel Memory from User Space. Commununications of the ACM **63**(6) (5 2020)
30. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: HPCA (2016)
31. Liu, F., Wu, H., Mai, K., Lee, R.B.: Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. IEEE Micro **36**(5), 8–16 (2016)
32. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P (2015)
33. Lowe-Power, J., Ahmad, A.M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Beckmann, B., Bharadwaj, S., Black, G., Bloom,

G., Bruce, B.R., Carvalho, D.R., Castrillon, J., Chen, L., Derumigny, N., Diestelhorst, S., Elsasser, W., Escuin, C., Fariborz, M., Farmahini-Farahani, A., Fotouhi, P., Gambord, R., Gandhi, J., Gope, D., Grass, T., Gutierrez, A., Hanindhito, B., Hansson, A., Haria, S., Harris, A., Hayes, T., Herrera, A., Horsnell, M., Jafri, S.A.R., Jagtap, R., Jang, H., Jeyapaul, R., Jones, T.M., Jung, M., Kannoth, S., Khaleghzadeh, H., Kodama, Y., Krishna, T., Marinelli, T., Menard, C., Mondelli, A., Moreto, M., Mück, T., Naji, O., Nathella, K., Nguyen, H., Nikoleris, N., Olson, L.E., Orr, M., Pham, B., Prieto, P., Reddy, T., Roelke, A., Samani, M., Sandberg, A., Setoain, J., Shingarov, B., Sinclair, M.D., Ta, T., Thakur, R., Travaglini, G., Upton, M., Vaish, N., Vougioukas, I., Wang, W., Wang, Z., Wehn, N., Weis, C., Wood, D.A., Yoon, H., Éder F. Zulian: The gem5 simulator: Version 20.0+ (2020)

34. Mandelblat, J.: Technology Insight: Intel's Next Generation Microarchitecture Code Name Skylake (2015), `https://en.wikichip.org/w/images/8/8f/Technology_Insight_Intel%E2%80%99s_Next_Generation_Microarchitecture_Code_Name_Skylake.pdf`
35. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: How SGX amplifies the power of cache attacks. In: CHES (2017)
36. Mohammad, B.: Embedded Memory Design for Multi-Core and Systems on Chip, Analog Circuits and Signal Processing, vol. 116. Springer (2014)
37. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: CACTI 6.0: A Tool to Model Large Caches. HP Laboratories **27**, 28 (2009)
38. Nagarajan, V., Sorin, D.J., Hill, M.D., Wood, D.A.: A primer on memory consistency and cache coherence. Springer Nature (2020)
39. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)
40. Qureshi, M.K.: CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: MICRO (2018)
41. Qureshi, M.K.: New attacks and defense for encrypted-address cache. In: ISCA (2019)
42. Saileshwar, G., Kariyappa, S., Qureshi, M.: Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. In: SEED (2021)
43. Saileshwar, G., Qureshi, M.K.: MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In: USENIX Security (2021)
44. Sanchez, D., Kozyrakis, C.: Vantage: scalable and efficient fine-grain cache partitioning. In: ISCA (2011)
45. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-flight Data Load. In: S&P (2019)
46. van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In: S&P (2021)
47. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS (2019)
48. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA (2017)
49. Schwarzl, M., Schuster, T., Schwarz, M., Gruss, D.: Speculative Dereferencing of Registers: Reviving Foreshadow. In: FC (2021)
50. Schwedock, B.C., Beckmann, N.: Jumanji: The Case for Dynamic NUCA in the Datacenter. In: MICRO (2020)
51. Tan, Q., Zeng, Z., Bu, K., Ren, K.: PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In: NDSS (2020)

52. Townley, D., Arıkan, K., Liu, Y.D., Ponomarev, D., Ergin, O.: Composable Cachelets: Protecting Enclaves from Cache {Side-Channel} Attacks. In: USENIX Security. pp. 2839–2856 (2022)
53. Unterluggauer, T., Harris, A., Constable, S., Liu, F., Rozas, C.: Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks. In: SEED (2022)
54. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security (2018)
55. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. ACM SIGARCH Computer Architecture News **35**(2), 494 (2007)
56. Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution (2018), `https://foreshadowattack.eu/`
57. Werner, M., Unterluggauer, T., Giner, L., Schwarz, M., Gruss, D., Mangard, S.: ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security (2019)
58. Wu, H., Liu, F., Lee, R.B.: Cloud Server Benchmark Suite for Evaluating New Hardware Architectures. IEEE CAL **16**(1), 14–17 (2017)
59. Yan, M., Sprabery, R., Gopireddy, B., Fletcher, C., Campbell, R., Torrellas, J.: Attack directories, not caches: Side channel attacks in a non-inclusive world. In: S&P (2019)
60. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security (2014)