Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks

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

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

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

1. Introduction

Caches hide the memory latency in modern CPUs. Modern Intel CPUs organize caches in slices, sets, and ways, which are selected based on the physical address. Slices are independent caches comprised of sets. Each set has multiple ways, i.e., the $64\,\text{B}$ cache lines. Memory mapping to the same set is called *congruent*. An attacker can use congruent addresses to measure contention or to *evict* cache lines of a victim process. The most notable cache attack exploiting set contention is Prime+Probe [43], [53], [42], [27], [36], [39]. But even without direct contention within one cache set, an attacker can still mount a cache-occupancy attack, where the attacker only observes aggregated cache usage. Cache-occupancy attacks have fewer requirements but contention-based attacks are more powerful and dangerous.

Contention-based attacks [69], [70], [33], [35], [34], [66], [49], [50], [71], [65], [54] are hindered by the scrambling of address-cache-line mappings in secure randomized caches. Recent designs [49], [50], [71], [65], [54] use cryptography to randomize mappings in hardware with a secret key. This is backward-compatible on the software level and maintains certain sharing capabilities.

While randomized caches are a promising solution to eviction-based attacks [46], [11], cache collisions still exist due to limited cache sizes. *Rekeying* alleviates the problem, but the best rekeying interval is difficult to determine for yet unknown attacks and known attacks require high rekeying intervals with a significant performance cost. Furthermore, some previous proposals suffer from cryptanalytic flaws [46], [9]. Hence, we ask the research questions:
Can a secure randomized cache prevent contention between any attacker cache line and a victim cache line in most cases? Which cryptographic constructions provide this property while maintaining high performance?

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

At the core of SassCache is a new two-layered cryptographic construction with configurable output distribution determining reachable cache lines. Inspired by `QARMA` [4], we propose `QARTA`, tailored to our functional (i.e., uncommon bit sizes), latency, and security requirements.

In the recommended configuration, the attacker *cannot* build an eviction set for $99.999\,97\,\%$ of the victim's addresses. For the remaining $0.000\,03\,\%$, SassCache maintains the same security as previous secure skewed caches. Even cache-occupancy attacks are not feasible anymore because the hiding effects affect this channel equally. We evaluate this property and discover that an attacker can observe the occupancy of OpenSSL AES T-Tables in less than $0.0005\,\%$ of cases and the occupancy of secret-dependent cache lines in mbedTLS RSA-4096 in less than $0.0003\,\%$ of cases.

The basic functionality of SassCache is fully compatible with standard caches and, hence, simple to integrate into existing CPU architectures. SassCache provides full compatibility with legacy software, but to fully harness its security, software has to configure and switch security domains. Our main use case for SassCache are multi-tenant cloud systems, as they offer high degrees of parallelism between clearly

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

defined security domains (i.e., tenants). Here, SassCache provides inherent quality-of-service properties. We evaluate this in SPEC2017 with multiple other cache-intense tenants running in parallel, showing competitive performance.

We implemented SassCache in gem5, our own cache simulator[1], and in CacheFX [23]. We evaluate the impact on performance and cache-hit rate. Compared to a set-associative `LRU` cache, SassCache has a performance penalty of only 1.75 % on the cache hit rate in SPEC2017, similar to previous secure caches. In MiBench-small, the average cache hit rate of SassCache is 16 % lower than for a set-associative `LRU` cache; in scimark2, it is 23.6 % higher.

Contributions. In summary, our main contributions are:

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

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

2. Background

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

2.1. Caches

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

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

Modern CPUs have a large (multiple MB) shared last-level cache (LLC). It facilitates the use of code and data on multiple cores simultaneously and simplifies coherency and cache lookups. Typically, on Intel and AMD CPUs, this is the L3 cache, and on ARM CPUs, the L2 cache. The LLC is often *inclusive*, i.e., all cache lines in L1 or L2 caches are also in the LLC. Some CPUs split the LLCs into so-called *slices* [38], which are independent caches, e.g., per (logical) core. The slice is selected based on the physical address, not the core. Each core can access each slice via a ring bus that connects all cores and slices.

2.2. Index Derivation Function

Conventional caches map addresses to cache sets by simply using a part of the physical address as a set index, but more advanced functions can be implemented as well [61]. Werner et al. [71] introduced the term Index Derivation Function (IDF) for these mapping functions, which we will use going forward. An IDF generates a set of possile cache lines for each physical address. A more complex IDF can break the traditionally linear relationship between addresses and sets, and may also change the static sets into dynamic ones, such that sets are not fixed collections of cache lines anymore. For this, a cryptographic function that works on the physical address as well as some secret can be used. IDFs generally need to scale well, incur low overheads, and be fully transparent to the software level.

2.3. Cache Attacks

As the cache state depends on recent memory accesses, an attacker can learn interactions of other programs with memory (e.g., instruction and data accesses). Initial attacks focused on the execution time [32], [44], [67], [8]. More recent techniques interact directly with the shared cache.

Flush+Reload [72] relies on (read-only) shared memory between attacker and victim: It flushes an address and later determines whether a victim accessed it by measuring its load latency. Prime+Probe [43] has no such restriction. It measures contention on a portion of the cache (e.g., a cache set) by filling this portion (prime) and measuring the time this takes (probe). Victim accesses to congruent addresses evict the attacker's lines, influencing the probe time.

Contention-based attacks (e.g., Prime+Probe) on the LLC require profiling to identify *eviction sets*, i.e., sets of congruent addresses. The attacker starts with a large pool of lines and, by timing accesses, discards those that do not contribute to contention [36]. The profiling duration depends on knowledge of the mappings and the number of elements discarded per iteration [68]. Prime+Probe based attacks are still actively being improved [46], [47], [11], [63].

2.4. Secure Caches

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

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

Many of these designs require randomization mappings to be dynamic, i.e., renewed over time. This *rekeying* (or *remapping*) destroys any congruence information an attacker may have learned. More frequent rekeying is more secure but comes with a performance [49] and power penalty.

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

CEASER-S [50], **ScatterCache** [71] and **Phantom Cache** [65] are examples of skewed designs [59] that compute indices on the fly. ScatterCache computes indices to individual cache lines which together form a unique set with random replacement. CEASER-S and PhantomCache randomly select from computed indices to entire sets, which can then use standard replacement policies like LRU. Since computations are done on-the-fly, these designs are scalable and suited for large LLCs. However, they have been shown to be susceptible to recent attacks [46], [11], [62]. **MIRAGE** [54] moves the randomization to the cache directory and uses it to approximate a fully associative cache. So far, none of the randomization-based secure caches protect against cache occupancy attacks.

Random Permutation (RP) Cache [69] precomputes permutation tables per process. **Newcache** [70] proposes an entirely new cache design with a secure table of indirection that tries to approach a directly-mapped cache. These table-based designs require overhead proportional to their size, which can be prohibitve for large LLCs.

Partitioning splits the cache into (fixed) slices by its sets (e.g., cache coloring [15], [73]), or its ways (e.g., Intel Cache Allocation Technology (CAT)). The security depends on the strength of the isolation between domains and how much remains observable to attackers. However, static partitioning reduces performance and lacks flexibility and scalability.

Non-Monopolizeable (Nomo) [19] cache reserves some ways per set to be only writeable by one domain, but this leaves reserved ways observable. **Vantage** [55] partitions *most* of the cache while maintaining associativity. Partitions can outgrow their allocated size into a small, unpartitioned space. Additional cache tag bits determine the number of partitions. **AutoLock** [24] prevents cross-core evictions by locking cache lines on ARM CPUs. **Hybcache** [17] uses a hybrid approach between a set-associative and a fully-associative cache. Full associativity is realized only in a small number of ways used for security-critical applications, whereas the rest uses the cache set-associatively. This assumes secure and insecure domains, which differs from the usecase for SassCache. **Jigsaw** [6] and **Jumanji** [58] partition the cache dynamically by splitting it into shares. Software defines capacity and mapping by assigning identifiers to the Page Table entries (PTEs). Jumanji has a lower latency, and higher performance and security than Jigsaw.

He et al. [26] analyzed static partitioning, Nomo, NewCache, RP Cache, and others, and found that all are, to some degree, vulnerable to at least 2 of 4 studied attacks.

In summary, later analyses [46], [11], [26] of randomized and partition-based caches that provide probabilistic security have shown that these can achieve relatively high performance, but are often not as secure as first thought. Static and total partitioning, on the other hand, provides strong security guarantees at the cost of flexibility and performance.

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

2.5. Attacking Secure Caches

Profiling secret-dependent cache lines and finding addresses congruent to them, is an important prerequisite for successful exploitation. The first proposals [49], [66] were bypassed by optimized eviction set algorithms [68], [50], allowing the exploitation phase to proceed as in traditional caches. Consequently, they are practically broken since they require impractical rekeying periods to maintain security [50].

Randomized caches with a probabilistic component [71], [50], [65] preclude traditional eviction by design. Obtaining fully congruent addresses, mapped to the same set in every partition, is theoretically infeasible. In particular, the notion of eviction sets needs generalization (i.e., weakening) if the attacker is to succeed at all. Werner et al. [71] propose eviction sets with addresses congruent in at least one cache way, which was later generalized to partitions [46].

While finding generalized eviction sets is more difficult, a resourceful attacker can still find them by observing which lines are evicted by victim execution. To maximize the chance of observing such evictions, Purnal et al. [48], [46] propose Prime+Prune+Probe (`PPP`). It extends Prime+Probe with a *pruning* step, enabling occupying a large portion of the cache before transferring control to the victim. `PPP` was originally applied to CEASER-S and ScatterCache and reduced the complexity of finding eviction sets by orders of magnitude. However, it also applies to other randomized caches, e.g., those that skew across sets instead of ways [65]. Song et al. [63] propose to flush the attacker's own lines to speed up `PPP` by avoiding costly full cache evictions.

Given a rekeying period, the attacker needs to split resources between profiling (i.e., gaining spatial information) and exploitation (i.e., the actual attack). Bourgeat et al. [11] propose a methodology to navigate this tradeoff.

At one extreme, an attacker spends no time profiling and just measures *cache occupancy* [60], i.e., cache utilization. While less accurate than congruence-based channels (i.e., no spatial information), it is unaffected by rekeying. Current secure LLC designs, even those approximating fully-associative caches [17], [54], have the property that victim accesses are reflected in the observable cache utilization, leaving the cache occupancy channel unmitigated. Some designs are also vulnerable to so-called *shortcut* attacks that exploit weaknesses in randomization to bypass the protection altogether [46]. Thus, it is crucial that the randomization mapping uses well-designed cryptographic primitives.

3. Threat Model and Mitigation Goals

In this section, we describe our threat model and mitigation goals for secure caches, forming the basis of our secure cache design, SassCache. As shown in Section 2.5, even modern secure caches are affected, e.g., by Prime+Prune+Probe [46] or due to weak cryptographic constructions [46].

3.1. Threat Model

Our threat model is in line with prior work [71], [65], [46] but takes more recent and advanced attack techniques into account (cf. Section 2.5). In this threat model, Sass

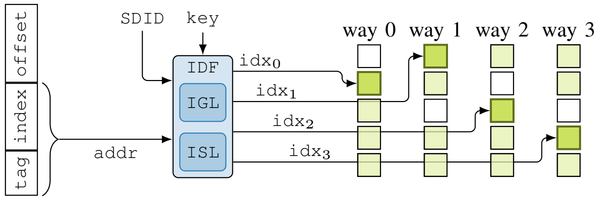

Figure 1: Our two-layer Index Derivation Function (IDF): The first Index Generation Layer (IGL) is a cryptographically randomized mapping of addresses to indices, like in ScatterCache; the second Index Spacing Layer (ISL) reduces the set of reachable cache lines through another cryptographically randomized mapping (cf. Section 5.1).

Cache constitutes the cache level that is shared between attacker and victim. For our evaluation, we do not consider self-eviction in the victim, because in randomized skewed caches, *reliable* self-eviction only occurs with substantial amounts of memory accesses as part of the secret-dependent operations or active victim participation.

SassCache uses a function that maps physical addresses to cache sets by generating indices $\mathtt{idx}_i$, where i counts the ways. (Figure 1). We assume the function is known to the attacker but uses a random secret key and a security domain (SDID) to separate security contexts. The key is inaccessible to the attacker, whereas the address is fully controlled. Attacker and victim are separate tenants on a multi-tenant system, where each has their own SDID. Consequently, they are located in different security contexts, and the attacker only has control over few contexts (c.f. Section 7.5). While the attacker may be able to read the SDID, it cannot set it; only privileged software (e.g., the hypervisor) is allowed to set it. The generated indices $\mathtt{idx}_i$ are not observable directly but only via cache contention. Physical attacks on the function, its intermediate values, or secret parameters and bugs in the privileged software are out of scope.

3.2. Required Attributes

Functionally, a secure cache should be mostly transparent to software but maintain performance that is comparable to standard caches. On the security side, we extend the security attributes of ScatterCache [71] as follows to address more recent attacks [46] and attacks commonly considered out-of-scope (e.g., the cache occupancy channel [71], [65], [54]):

- Software-defined security domains (based on properties like virtual machine (VM), tenant, user, or process ID) must not share cache lines unless cross-domain coherency is explicitly required, e.g., writable shared memory.
- It must be hard to find congruent addresses in the cache, i.e., it should be hard for adversaries to infer a connection between accessed physical addresses and cache set index.
- Partially accessible addresses should become hidden with a high probability to reduce their observability.
- It must be impossible to evict, measure or control all cache lines from another security domain.

4. The SassCache Architecture

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

Purnal et al. [46] show that randomized secure caches can still be attacked both with old attacks and new attack variations, albeit at a lower attack performance. The underlying problem is that the full cache is still accessible to the attacker. On the other hand, approaches based on cache partitioning such as cache coloring [15], [74] and Intel's Cache Allocation Technology (CAT) offer strict security by splitting the cache into fixed allocations but lack flexibility and scalability. To overcome the drawbacks of previous randomized and partitioned caches, we combine the two principles in SassCache. Hence, to mitigate even statistical and occupancy attacks, the idea for SassCache is to cryptographically limit the total number of cache lines accessible to an attacker, in addition to the permutation performed by ScatterCache. We refer to the number of accessible cache lines (=share of the total cache) as *coverage*.

SassCache follows a two-layered approach for its IDF (Figure 1): First, the Index Generation Layer (IGL) is a permutation of cache sets as in ScatterCache. This breaks the link between cache set and physical address across different security domains and makes it very hard to profile the cache for an attack [46]. If this layer were a known, non-skewing mapping (such as a bit mask on the physical address), the number of unique sets would be limited. A small number of fixed sets not only makes profiling trivial, it also makes self-eviction deterministic and more likely. This is because some addresses would always share the same set, and therefore compete for the same unobservable ways (Section 7.1). Additionally, with potentially millions of profiling attempts per successful attack (Section 7) and minutes per attempt [46], the IGL ensures high costs for attackers. In short, the IGL provides important support for the security of the second layer and defense-in-depth properties.

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

We propose SassCache as an inclusive or non-inclusive last-level cache (LLC). We opt for a set-associative base design with W ways, i.e., W cache arrays, exactly like existing caches deployed in current CPUs. Each cache array with a size S is indexed individually by one of the W indices. Because the sets are dynamic, we use a random replacement policy. This approach results in S^W possible cache sets after the first layer, similar to ScatterCache [71]. The second layer ISL restricts the possible indices in each way. While this reduces the number of sets per domain, it brings a novel security property: certain cache lines cannot be evicted by an attacker. Our cryptographic design is the basis that makes it improbable (cf. Section 7) for an attacker to evict a target cache line or measure cache occupancy.

5. Cryptographic Design

For SassCache, we need a function to generate cache-set indices from addresses to skew the cache, i.e., the Index Derivation Function (IDF). Additionally, the IDF must limit the number of accessible cache lines per security domain, i.e., the coverage. Hence, in this section, we introduce the two-layered cryptographic primitive at the core of Sass Cache. We design a low-latency IDF that maps the address to W indices $\mathtt{idx}_i$, where the mapping is controlled by the SDID and the key (Figure 1). The IDF consists of two layers: an Index Generation Layer (IGL) that maps the address to W independent intermediate identifiers $\mathtt{id}_i$ and an Index Spacing Layer (ISL) that maps each identifier to the final index $\mathtt{idx}_i$ in the index space. This index $\mathtt{idx}_i$ is selected uniformly from a subset of the index space defined by the SDID, key, and way i. The ISL is designed such that the subset is expected to cover a defined share of the full index space that we refer to as *coverage*.

5.1. Design of the Index Derivation Function

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

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

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

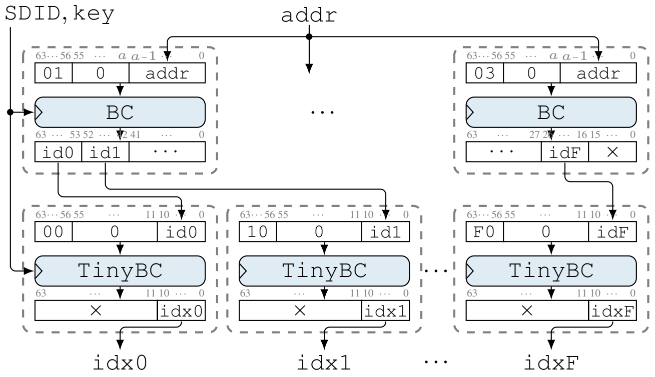

Figure 2: The address-to-index mapping function IDF with 63 % coverage, for $W = 16$ ways and 11-bit indices ($\ell = n = 11$). The top layer is the IGL, the bottom ISL.

Index Spacing Layer. To generate the final indices $\mathtt{idx}_i$ from $\mathtt{id}_i$, the ISL uses W parallel TinyBC invocations. Prepending the counter i before $\mathtt{id}_i$ in the input and truncating the output of TinyBC effectively gives us a family of ℓ-bit to n-bit pseudorandom functions, keyed with SDID, key. For TinyBC, a cipher smaller than BC also suffices, with a block size of at least $\max(n+\varepsilon, \ell+\log_2 W)$ bits for some small integer ε.

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

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

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

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

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

$$C_b = 1 - \frac{\binom{2^b - 2^{b-n}}{2^\ell} \cdot 2^\ell! \cdot (2^b - 2^\ell)!}{2^b!} = 1 - \frac{\prod_{i=0}^{2^{b-n}-1} (2^b - 2^\ell - i)}{\prod_{i=0}^{2^{b-n}-1} (2^b - i)}$$

$$= 1 - \prod_{i=0}^{2^{b-n}-1} \left(1 - \frac{2^\ell}{2^b - i}\right) \approx 1 - \left(1 - \frac{1}{2^{b-n-t}}\right)^{2^{b-n}} .$$

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

5.2. Instantiation with QARMA and QARTA

We want to instantiate this design with efficient cryptographic functions BC and TinyBC. Several low-latency block ciphers [10], [12] and tweakable block ciphers (TBCs) [4], [7] have been published, though some provide insufficient security [20], which share several design ideas with PRINCE [10]. We propose an instantiation using (parts of) QARMA [4], a TBC used for ARM pointer authentication.

Conservative instantiation. A conservative instantiation is to use the 64-bit variant of QARMA, QARMA$_7$-64, for both BC and TinyBC. This TBC encrypts 64-bit plaintext blocks with a 128-bit key K and 64-bit tweak T, fitting with the dimensions given in Figure 2. The 192-bit combined tweakey (K, T) is available for key material from the SDID and key, fitting, e.g., a 128-bit key as K and 64-bit SDID as T, or the XOR of two 128-bit values as K with $T = 0$. The same key can be used for BC and TinyBC. The expected coverage C for this construction can be derived with the model for large block ciphers.

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

QARTA$_4$-16 operates on one 16-bit column of a QARMA state and key using the QARMA 4-bit S-box layer S (SubCells), mixing layer M (MixColumns), and round tweakey addition (AddRoundTweakey), without applying an equivalent of the permutation layer τ (ShuffleCells). Four parallel instances of one QARTA$_4$-16 round correspond to one round of QARMA without the τ operation (cf. Figure 4). The key (w^0, k^0) and tweak T are again derived from the SDID and key. In this instantiation, the key material for TinyBC should be independent of that of BC. Figure 5 shows which parts of the key material influence which index computations. Notice that the tweak schedule implies that the tweak material influences several columns, and acts differently on each tweak column. The QARMA round constants $\mathfrak{c}_i$ are also different for each column.

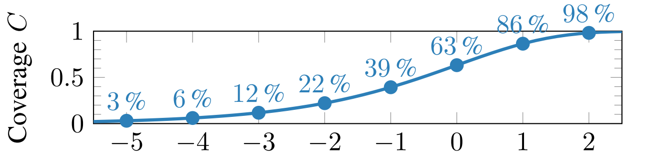

Figure 3: Expected coverage $C \approx 1 - e^{-2^t}$ for $t \in \{-5, \ldots, 2\}$.

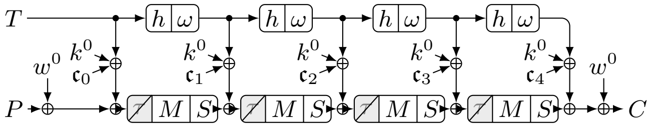

Figure 4: Four parallel invocations of QARTA$_4$-16 (with different, but partially related keys).

QARTA$_4$-16 is not a generically secure tweakable block cipher due to its small size and low-latency design. It is tailored for the proposed application, where an attacker has little control and never learns the cipher inputs except for a few index and padding bits. Since each column depends on at least 8 cells of the tweak T (Figure 5), the effective tweakey size for QARTA$_4$-16 is at least 64 bits. Regarding its cryptanalytic properties, QARTA$_4$-16 is expected to reach its full algebraic degree after 3 rounds since its S-box has an algebraic degree of 3 and $3^3 > 15$. The MixColumns matrix has a branch number of $\mathcal{B} = 4$; there are several truncated differential and linear patterns with a total of 8 active S-boxes that are compatible with the input format, e.g., the iterative pattern $(0, 0, *, *)$, where $*$ denotes active cells. Since the maximum differential probability and absolute linear bias of the S-box are 2^{-2}, the maximum achievable probability for differential characteristics is 2^{-16}. Even with potential clustering effects, this is hard to exploit for largely unknown cipher inputs. Still, an attacker that observes a large number of cipher outputs might succeed in recovering a few key bits by exploiting the few known cipher input bits and the partially overlapping key material between indices. However, since the key material is independent of the key used in BC, there is little information to derive from this potential knowledge beyond the image set of TinyBC, which is easily obtained through direct observation rather than cryptanalysis. The main criterion for security is, however, the statistical behavior, which we analyze next.

Coverage evaluation. Figure 6 shows the observed distribution of the coverage for both instantiations of TinyBC for 100 random keys with 16 counter values i each. Both QARMA$_7$-64 and QARTA$_4$-16 behave as expected, with

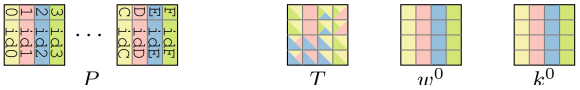

Figure 5: Mapping of QARMA-64 and QARTA-16 states.

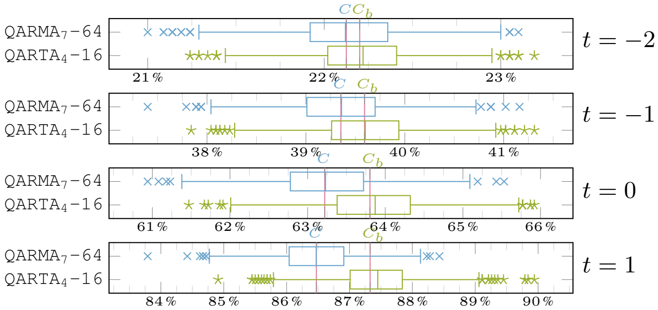

Figure 6: Experimental coverage for $t \in \{-2, \ldots, 1\}$ for 100 random keys and 16 ways i, with QARMA or QARTA$_4$-16, and expected values C and C_b, respectively.

average coverages C and C_b ($b = 16$), respectively. For example, for $t = -1$, the coverage for QARMA$_7$-64 ranges from 37 % to 41 %, with an average of $C \approx 39\,\%$. The coverage for QARTA$_4$-16 ranges from 37 % to 41 %, with an average close to $C_b \approx 39\,\%$.

6. Implementation of SassCache

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

6.1. Hardware Modification

While recent advances in the RISC-V community lead to the first RISC-V CPUs with experimental L2 caches [75], [13], they are far from state-of-the-art high-performance L3 LLCs used in server-grade CPUs and thus not yet suited for estimations on the hardware modification costs that Intel, AMD or ARM would see. Hence, we estimate SassCache's hardware costs by determining the hardware costs for the building blocks in terms of chip-area and latency.

Area. The cryptographic primitives for the IDF make up the main cost in hardware. We propose QARTA$_4$-16, a custom low latency instantiation for 16 ways (cf. Section 5.2) corresponding to roughly four QARMA$_7$-64 instances with 34.4kGE each [4]. Therefore, in total, the IDF requires less than 140kGE. Previous work [71] estimated that the open-source BROOM core's LLC takes up about 5.5MGE. Hence, our design should result in less than 3 % of that.

The additional area required to skew the cache is specific to the overall design. In general, Djordjalian [18] noted that additional decoders and wiring are required for each way. For a 2-way skewed cache, Spjuth et al. [64] saw a 17 % increase in energy consumption. However, Sardashti et al. [56] show that cache skewing only has 1.5 % to 15.3 % overhead. Furthermore, most Intel cache architectures already feature cache slices; thus, they partition the cache into smaller caches with multiple addressing circuits already.

Latency. SassCache's latency depends on the IDF's latency for generating the set indices. We use QARMA, which is already used for latency-critical applications like pointer authentication [3] or memory tagging [41]. Notably, QARMA achieves a latency as low as 2.20 ns when fully unrolled at a 7 nm process [4]. For our low-latency instantiation (cf. Section 5.2), the combined latency of QARMA$_5$-64 and QARTA$_4$-16, used for IGL and ISL, is comparable to that of QARMA$_7$-64 at about 3.25 ns. In comparison, this is still lower than L2 cache access on current CPUs (e.g., Intel Xeon 8280 at 5.18 ns [16], AMD Epyc 7742 [16] at 3.86 ns, and ARM Ampere Altra Q80-33 at 4.11 ns [22]). Since Sass Cache is designed for shared LLCs, most if not all of the latency is hidden in lower-level cache lookups, and our IDF becomes viable for practical implementation. In line with other skewed cache designs, the skewing itself does not introduce additional latency [54], [49].

6.2. SassCache Interactions

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

Backward Compatibility. To make SassCache backward compatible, we suggest to initially deactivate the spacing layer of the IDF. While booting, privileged software (e.g., the hypervisor) aware of SassCache and security domains activates the second layer via a configuration register. Thus, legacy software still benefits from a cache similar in functionality to ScatterCache without security domains, which already protects against some cache attacks [71].

SassCache Software Interface. The software must supply the correct SDID to SassCache. Depending on the architecture and what constitutes a security domain (we focus on the cloud use case), this can be achieved in numerous ways. On x86-64, ARM, or RISC-V systems, we can use unused bits in the PTEs to supply the SDID either directly or indirectly via an additional lookup table. Using the SDID directly only allows for a limited number of security domains, which we therefore do not recommend. Intel x86-64 has 14 bits [31], ARM64 4 bits [2], and RISC-V 10 bits [52] reserved for future use in the PTEs that can be used for this.

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

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

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

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

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

6.3. Key Management and Rekeying

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

SassCache uses a boot-time hardware-generated key, like CEASER-S [49] and ScatterCache [71]. The key is stored in a hidden CPU register and is inaccessible to software. This prevents an attacker from predicting the resulting cache set from a physical address and vice-versa. Only the SDID may be set by the software to provide different security contexts. This keeps the required software and hardware changes low and modular, decreasing the effort to implement SassCache.

CEASER-S and ScatterCache require rekeying [49], [71]. The frequency of rekeying is a security parameter to adapt to improved attacks and security margins [46]. Key changes work well on write-through caches as no data inconsistencies can occur. However, rekeying costs performance as data is invalidated, causing cache misses. Thus, changing the key corresponds to a cache flush in terms of performance. Implementing rekeying for write-back caches is more costly: The key must not change before all dirty cache lines were written to memory.

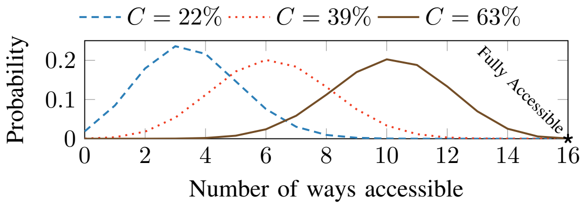

Figure 7: Accessibility distribution of cache lines ($W = 16$) for different cache coverages C. *Fully accessible* lines are observable to an attacker; all others become unobservable.

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

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

7. Security Evaluation

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

1) For t=-1, W=16 SassCache prevents full eviction of the target cache line in 99.999 97 % of cases (cf. Section 7.1). Thus, on average, the attacker can try to construct an eviction set for 1 in 3 000 000 cache lines, or 64 B in 185 MB of memory. On CPUs with 20 ways per cache set, this increases 1 in 125 000 000 cache lines, or 64 B in 7.6 GB of memory.
2) For attackable cache lines, security reduces to that of ScatterCache, with attack times from prior work [46].

Like previous designs [71], [54], SassCache precludes attack techniques based on shared read-only memory (e.g., Flush+Reload) by placing them separately in the cache per security domain. Thus, we evaluate SassCache against the remaining attack classes, i.e., contention- and occupancy-based attacks.

7.1. Properties of SassCache

Partially Accessible Lines. In the worst case, a victim line falls within the attacker's coverage in each of the W ways. The probability $P_{\mathrm{W}} = C^W$ is low for practical con-

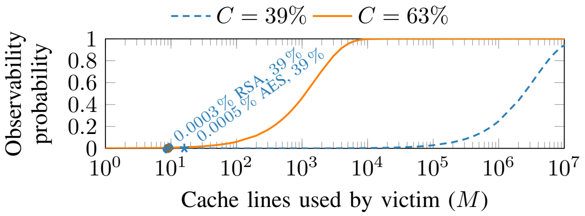

Figure 8: Observability probability ($W = 16$, $C = 39\,\%$ & $C = 63\,\%$), increases with cache lines. A covert channel requires thousands of cache lines. The probability of a successful occupancy-based attack is extremely low. Example points are openSSL AES T-Tables and mbedTLS RSA-4096.

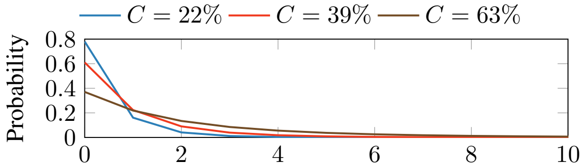

Figure 9: Probability to observe partially accessible victim line N times before it is hidden ($W = 16$).

figurations (e.g., $P_{\mathrm{W}} = 0.06\,\%$ in a 16-way SassCache with $C = 63\,\%$ or $P_{\mathrm{W}} = 0.000\,033\,\%$ with $C = 39\,\%$). For such *fully accessible* lines, the security of SassCache is equivalent to that of ScatterCache with the same security domain size (i.e., W ways, $C \cdot S$ sets). All other lines are *partially accessible*, i.e., they can hide in the victim region, where they are not observable nor controllable by the attacker. Figure 7 highlights the abundance of partially accessible lines in the distribution. The exponential dependence of P_{W} on the number of indices W motivates SassCache to build on ScatterCache instead of other skewed designs.

For victim programs where M lines encode the same information (e.g., AES T-Tables, $M = 16$), the probability that at least one of them is fully accessible is $P_{\mathrm{M}} = 1 - (1 - C^W)^M$. Figure 8 illustrates P_{M} for a 16-way cache.

Repeated Observations. Cache attacks typically require many observations of sensitive lines, both for contention-based attacks [36], [68], [46], [36], [57] and for cache occupancy attacks [37], [60]. In SassCache, accesses to victim lines are only *observable* to the attacker when they evict an attacker's line. The probability for a partially accessible victim line to be observable N times before being hidden is $\frac{P[N]}{(1-C^W)}$, where $P[N] = \sum_{i=0}^{W-1}[(\frac{i}{W})^N \cdot \frac{W-i}{W} \cdot \binom{i}{W} \cdot C^i \cdot (1-C)^{(W-i)}]$ (cf. Figure 9). The expected number of observable victim accesses until the line is hidden can be computed as $E[N] = \sum_{n=0}^{\infty} n \cdot P[N = n]$. In a 16-way SassCache with $C = 39\%$, this is already after $E[N] \approx 0.72$ accesses on average, for $C = 63\%$ after $E[N] \approx 2$ accesses. This reflects the number of potentially observable victim accesses to the cache line, i.e., assuming that the attacker occupies its full share of the cache ($C \cdot S \cdot W$ lines).

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

7.2. State-of-the-Art Attack Evaluation

We implement SassCache in CacheFX [23], which models attack strategies on real systems with attacker and victim code, which the framework invokes accordingly. Attacker and victim issue requests to cache implementations that respond with hits and misses. The framework analyzes attack success statistically and offers several knobs to test caches configured with different parameters, e.g., number of sets, ways, and replacement policy. With the framework we test the applicability of state-of-the-art profiling (Single Holdout Method [49], Group Elimination Method [50], and Prime+Prune+Probe [46]) on SassCache, to find partially congruent addresses, i.e., lines congruent with the target in one or more ways. We configure SassCache with 16 ways, and $t = -1$ (i.e., $C = 39\%$) and test each profiling algorithm 500 times. We sample a random victim address, let the algorithm find an eviction set, and test the quality of the found set by (a) determining the share of truly conflicting addresses (*True Positive Rate* (TPR)) in the eviction set, and (b) computing the success rate (SR) of evicting the victim address. We compute the minimum, maximum, mean, and median for TPR and SR over all runs to statistically analyze profiling on SassCache.

We observe that all techniques fail with overwhelming probability, as expected (cf. Section 7.1). Our analysis of the algorithms' progress shows that after a few victim evictions, the victim address is not observable to the attacker anymore and hidden in the cache, as expected (cf. Figure 9). Hence, both the minimum and median of TPR and SR are 0. Maximum TPR and SR vary depending on the algorithm, the eviction set size, and the concrete cache parameters, but occur infrequently when the victim address is fully attacker-reachable (and SassCache falls back to ScatterCache). For instance, the most efficient technique Prime+Prune+Probe achieves a maximum TPR of 1 and a SR according to Figure 5 in [71]. Mean TPR and SR are skewed according to the maximum and the probability of fully accessible lines.

Note that our empirical analysis considers a single target victim line. Figure 8 covers the probability for profiling to succeed on at least one out of M redundant lines, which is low for real-world attack targets. Advanced profiling methods [46] enlarge eviction sets (of sufficient size) without relying on further victim accesses. This does not affect Sass Cache, as the address is mostly not reachable in the first place. Furthermore, once a line is hidden, the attacker cannot proceed without self-eviction by the victim (cf. Section 3).

7.3. Cache Occupancy Leakage

The cache occupancy channel is arguably the most primitive cache side-channel. When the number of cache lines a victim uses depends on a secret bit, an attacker can recover the bit by simply measuring cache utilization. Traditional shared caches as well as secure randomized caches [49], [71], [50], [54] cannot completely close the cache occupancy channel. This can easily be seen when looking at a fully-associative cache: While there is no cache-set information to gain, the occupancy of a different number of cache lines can still be observed. SassCache improves over previous secure randomized caches, as only part of the cache occupancy of the victim is visible to the attacker. As the cache occupancy channel covers a significant amount of irrelevant cache lines, all cache occupancy attacks so far [37], [60] require a large number of repetitions. Unless the victim's cache line is fully coverable by the attacker, it will quickly be hidden in an attacker-unreachable part of the cache (cf. Section 7.1). Furthermore, self-eviction of a specific cache line is generally unreliable and unlikely (cf. Section 3). If a victim occupies large amounts of memory, self-eviction can occur, but with adverse effects as it reduces the overall number of cache lines occupied by the victim as compared to the non-self-eviction case, reducing exposure to the attacker. This is particularly relevant for the covert channel case where the attacker occupies a large fraction of the cache. To obtain worst-case numbers, we assume that in this case the attacker is completely lucky and there is no self-eviction that conceals some of the fully coverable visible cache lines. Consequently, the formula from Section 7.1 also applies to the cache occupancy channel: The probability that a cache occupancy channel encoding '0' and '1' into M cache lines works successfully is the probability that at least one of these cache lines is fully coverable (cf. P_M in Section 7.1).

With the default configuration of $39\,\%$ coverage, for very low values, the success probability for the occupancy channel is close to $0\,\%$ (cf. Figure 8). If the difference between a secret bit '0' and a secret bit '1' is reflected in the access to more than two million cache lines, the probability that the attacker can observe the occupancy channel is $50\,\%$. For an observability probability of $95\,\%$, more than 10 million cache lines must encode a '1'. This number of cache lines is far beyond normal cache attack targets: OpenSSL AES T-Table encryption encodes key-bit information in 16 cache lines resulting in an observability probability below $0.0005\,\%$. mbedTLS RSA-4096 signature computation encodes equivalent key-bit information in up to 9 cache lines [57], resulting in an observability probability below $0.0003\,\%$. Hence, SassCache also closes the cache occupancy channel in many attack scenarios.

7.4. Asymmetric Domain Sizes

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

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

7.5. Multi-Domain Attacks and the Choice of t

Ristenpart et al. [53] showed that achieving co-location in the cloud is possible. Inci et al. [28] also showed that, while rare, co-location of more than 1 VM on the same system is possible. Fang et al. [21] also show container scheduling (e.g., Kubernetes) is vulnerable to co-location.

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

7.6. Trusted Execution Environments

The threat model of Intel SGX [14] and AMD SEV-SNP [1] explicitly allows malicious privileged software and hypervisors. Without further modification for these TEEs, this goes beyond our threat model because it contradicts our goal of backwards compatibility. If the ISL is disabled, SassCache falls back to the security of a secure randomized cache.

8. Performance Evaluation

In this section, we analyze SassCache's performance in a default configuration with $39\,\%$ coverage (unless stated otherwise). We use gem5 to run MiBench [25], lmbench [40], scimark2 [45], and the GAP benchmark [5], in line with previous works [71], [49], [50] to show the skewed cache characteristics of SassCache. With our custom simulator we run extensive workloads (e.g., SPEC CPU 2017) efficiently and evaluate the cache's behavior based on real-world recorded memory access traces, e.g., for a multi-tenant cloud scenario. We show that SassCache performs similar to traditional set-associative caches of the same size, and can increase in relative performance as multiple tenants compete for the cache. All caches are evaluated without rekeying, as this would be highly implementation dependant.

8.1. gem5 Test Setup

To evaluate SassCache in the gem5 full system simulator, we run gem5 as a 32-bit ARM DerivO3 3 GHz CPU. We configure a two-level cache system. The L1 is made up of 2

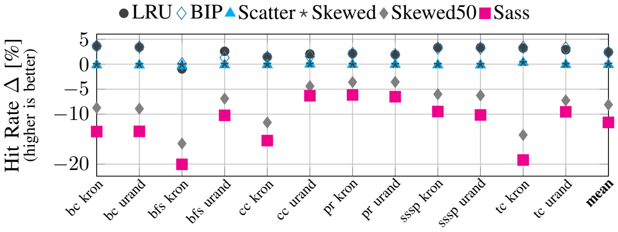

Figure 10: gem5 cache hit rate in the GAP benchmark. Comparison against `Rand` as a baseline.

32 kB, 8-way caches for instructions and data. As this is the same for all configurations, the differences in performance stem from the respective L2 implementations only. All L2 caches are configured with 1 MB and 16 ways. The L1 and L2 cache line size is 64 B. This is similar to typical server and desktop configurations today with slices functioning as independent caches (i.e., 1 MB per cache slice and 64 B cache line size), since when all cores on a system are in use, the average per-core share of the LLC amounts to the size of 1 slice. We test 6 cache designs: (1) Bimodal Insertion Policy (`BIP`), (2) Least Recently Used (`LRU`), and (3) random replacement (`Rand`), for regular caches; (4) ScatterCache (hash-based), (5) skewed associative caches, and (6) Sass Cache, for skewed caches. For SassCache, we evaluate the default security level with the coverage parameter $t = -1$, i.e., a single-tenant coverage $C \approx 39\,\%$. We also include Skewed50, a skewed cache at $50\,\%$ capacity, as this is the nearest size to $39\,\%$ SassCache for a standard configuration without changing W.

We deploy the same software setup as prior work [71], with poky Linux (19.0.2) in Yocto 2.5 (“sumo”) and Linux 4.14.67 patched for compatibility with gem5. For the evaluation, we use the cache statistics from the gem5 simulator. We configure the `QUARTA` instances for our IDF with the exact distribution and latency properties described in Section 6.

8.2. gem5 Results

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

The GAP benchmark consists of 6 workloads (bc, bfs, cc, pr, sssp, tc) with 2 input graphs, `kron` (`-g27 -k16`) and `urand` (`-u27 -k16`). `Rand` is the baseline for the hit rate (cf. Figure 10). As expected, SassCache’s smaller effective cache size lowers the hit rate on average by 11.6 p.p. (6.1 p.p. to 20 p.p.). On average, SassCache’s hit rate is 11.5 p.p. lower than ScatterCache. For GAP, (the best performing) `LRU` has an average 14.1 p.p. higher hit rate than SassCache. Skewed50 approximates SassCache’s effective size and closely follows its hit rate, which supports that the delta stems mostly from the smaller effective size.

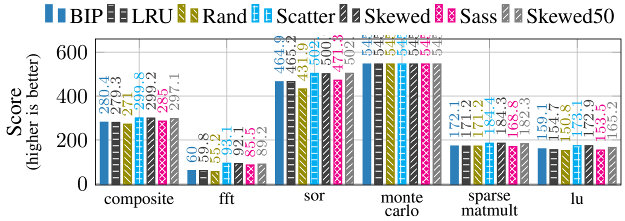

Figure 11: Result score for the scimark2 benchmarks for the gem5 simulator with different L2 replacement policies.

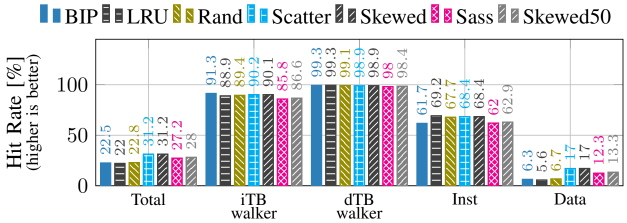

Figure 12: Cache hit rate by origin of cache requests for the scimark2 benchmark in gem5 for different caches.

To evaluate the impact of the smaller effective cache size, we use scimark2 [45] in the `-large 1` configuration. The cache hit rate (Figure 12) shows that SassCache performs similar to other skewed caches for the total hit rate and data accesses. All skewed caches show a notably higher hit rate for the `fft` and `composite` benchmark than the non-skewed `BIP`, `LRU`, and `Rand` policies. Consequently, for scimark2, skewed caches like SassCache outperform the non-skewed caches due to these benchmarks (cf. Figure 11) [71]. As expected, SassCache’s hit rate in this benchmark lies slightly below the Skewed50 configuration, due to SassCache’s smaller effective size of $\sim 39\,\%$. Here, ScatterCache and the skewed cache have the highest total hit rate, with SassCache being 4 p.p. lower, due to the data (-4.7 p.p.) and instruction (-6.3 p.p.) hit rates. The total hit rate of `LRU` is 5.2 p.p. lower than SassCache. Although `LRU`’s instruction hit rate outperforms SassCache by 7.1 p.p., its data hit rate is 6.7 p.p. lower. The reason for this is a high code locality but a weaker data locality in scimark2.

Caches with random replacement policy, e.g., our skewed caches, show a smoother roll-off after the L2 cache [71]. We verified this for SassCache using the lmbench `lat_mem_rd` benchmark [40] for 8 MB size and 64 B (i.e., cache line size) strides, cf. Figure 13. SassCache closely follows Skewed50, due to its smaller effective size.

With a 256 B stride size the step between the L2 and higher memory levels is shifted to a larger access size, cf. Figure 14. This shift results from skewed caches breaking the alignment of addresses and the cache set indices. In traditional set-associative caches, `lat_mem_rd`’s 256 B stride size performs sparse but aligned memory accesses that use every fourth cache index. With skewed caches the indices are random and, hence, there are fewer cache conflicts. Thus, for this type of access pattern, skewed caches generally improve the hit rate and lead to lower read latencies for

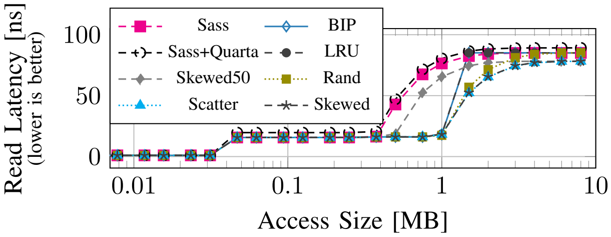

Figure 13: Memory read latency with `lat_mem_rd` for different cache configurations with 64 B strides.

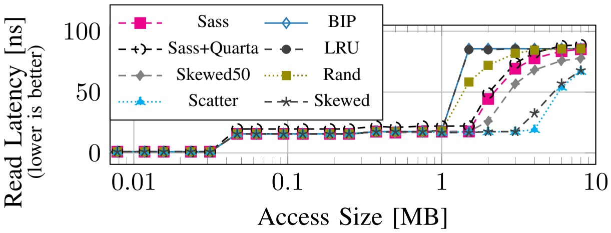

Figure 14: Memory read latency with `lat_mem_rd` for different cache configurations with 256 B strides.

larger ranges of memory [71]. However, due to SassCache's smaller effective cache size, this effect is less pronounced than for, e.g., ScatterCache. Still, the apparent size shifts from 1 MB to about 1.5 MB ($\approx 4 * 0.39$).

We also run the MiBench benchmark (small and large setting) on gem5 with the `Rand` cache as a baseline. The average hit rate in MiBench (small dataset) is 11.8 p.p. (2.4 p.p. to 22.5 p.p.) lower than `Rand`, cf. Figure 15. For MiBench with the large dataset, cf. Figure 16, this average improves to be 7.5 p.p. (0.8 p.p. to 20.2 p.p.) lower than `Rand`. Sass Cache has on average a 12.1 p.p. and 7.6 p.p. lower hit rate for MiBench small and large, respectively, than Scatter Cache. SassCache's hit rate is on average 13.6 p.p.(MiBench small) and 9.2 p.p.(MiBench large) lower than for `LRU` and `BIP` caches, which have the highest hit rates. In both cases, the Skewed50 consistently has a higher hit rate than Sass Cache but strongly correlates with it, confirming skewed cache characteristics for SassCache and lower hit rates in the single-tenant scenario with the lower effective cache size.

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

`QUARTA` Latency. Though we expect to hide latency in lower level accesses (see Section 6.1), we also evaluate a 12c (3.25 ns@3 GHz) overhead in our L2 (see Figures 13 and 14). This is a costly 30 % increase at our LLC base latency of 40c. Compared to SassCache without overhead, hitrates remain virtually the same (<±0.2 p.p. avg. difference), while scimark scores drop 4.1 % on average.

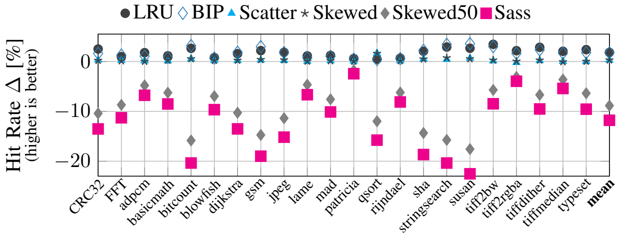

Figure 15: MiBench cache hit rate in gem5 with a small dataset. Percentage points over `Rand`.

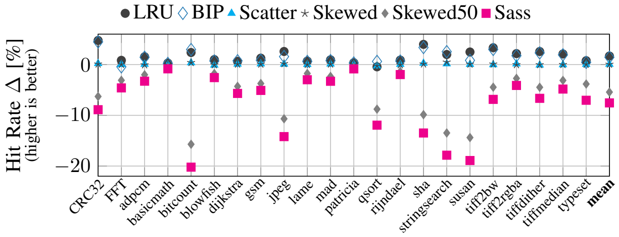

Figure 16: MiBench cache hit rate in gem5 with a large dataset. Percentage points over `Rand`.

8.3. Custom Cache Simulator Setup

We build a cache simulator based on the model by Purnal et al. [46]. We evaluate SassCache against ScatterCache, CEASER-S, standard `LRU`, `Rand`, and a way-split cache (an approximation of Intel CAT [30]), in SPEC CPU 2017.

To run benchmarks in our simulator, we collect real memory access traces (including instructions) with the Intel PIN tool [29]. While this is much faster than gem5 emulation, we still need to limit the size of our traces. Like prior works [71], [50], [51], we collect a representative sample trace over 250 million instructions for each benchmark.

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

8.4. Custom Cache Simulator Results

We run our recorded traces through the cache simulator to measure LLC hit rates in SPECspeed 2017. Since average hit rates vary between benchmarks, we compare the ratio of hit rates for 4 cache implementations to the `Rand` cache in Figure 17. Owing to its reduced size per security domain, SassCache performs worse than other caches in most tests, which is more pronounced in benchmarks with larger

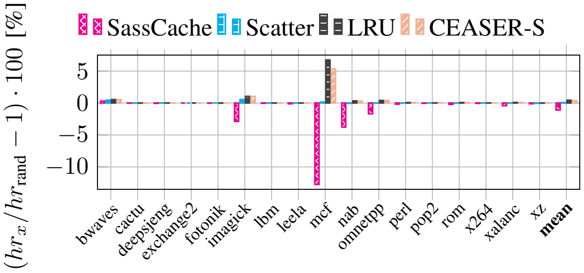

Figure 17: Change in LLC hit rate over Rand for SPECspeed 2017. Std.dev. $< 0.03\%$ for all tests over 10 runs.

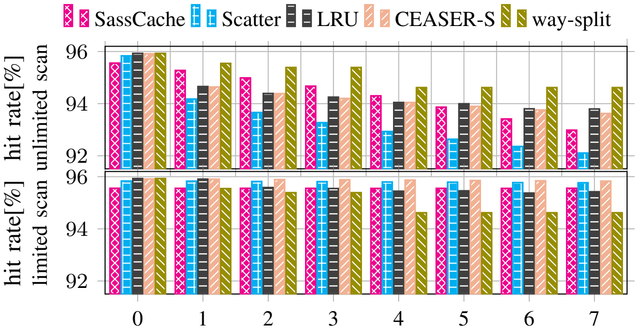

Figure 18: Average simulated hit rates of SPECspeed 2017 with 0-7 parallel domain workloads. Size of way-split partions: 0: 100%, 1: 50%, 2-3: 25%, 4-7: 12.5%. Top: adversarial workload with 64 B stride, unlimited size. Bottom: 1024 B stride repeating 512 addresses.

working sets or those optimized for LRU replacement. As in Section 8.2, we see that SassCache is at a disadvantage in single-threaded performance evaluations. When compared to the overall best, LRU, we see average hit rate drops of 1.75 %, 0.46 %, 0.08 %, and 0.52 % for SassCache, Scatter Cache, CEASER-S, and Rand, respectively.

We also examine SassCache when several competing workloads run concurrently, e.g., in a multi-tenant cloud. Here we simulate only the LLC, as the L1 is not shared among cores. We run SPEC traces as before, but interleave them with up to 7 adversarial workloads. We test 2 different types of adversarial workloads: a linear scan over an infinite range with 64 B stride (e.g., streaming a lot of data), and a scan over a limited set of 512 addresses with a stride of 1024 B, both shown in Figure 18. We configure our way-split cache such that it always has at least as many separate domains as workloads, i.e., 1/2/4/8, which results in the visible performance plateaus. In this particular test, we run only the memory accesses of the benchmark traces, leaving out the instruction accesses. This accentuates the different behaviors of the caches for these parallel workloads. As Figure 18 shows, caches like SassCache and ScatterCache suffer under workloads that use many cache lines without re-referencing them because of their random replacement, which can cause evictions even on recently used lines. For SassCache, this effect is at first counteracted by its isolation property. With few parallel workloads, many cache lines are exclusive to each domain. LRU-based caches fare better in general, as often-referenced cache lines can consistently survive streaming. The way-split cache leads under this workload, because the complete domain separation provides excellent thrashing protection as long as the workloads fit within the reduced cache size. The second workload is more adversarial to LRU-based caches, as not all sets are filled optimally. Here, SassCache can provide higher hit rates than a way-split cache while offering almost the same security.

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

Finally, we compare the hit rates for different C of Sass Cache for the 2-level setup. The average SPEC hit rate for SassCache (cf. Figure 17) is 88.36 % at 39 % coverage. For coverages of 12 %, 22 %, 63 %, 86 %, and 98 %, this changes to 86.18 %, 87.29 %, 88.89 %, 89.17 %, and 89.28 %. This almost reaches ScatterCache's performance for 98 % coverage and clearly drops towards 12 % coverage as expected.

9. Conclusion

In this paper, we proposed SassCache, a novel secure cache design based on a low-latency cryptographic function tailored to this use case. SassCache eliminates the attacker's capability of building an eviction set in 99.999 97 % of the cases. We found that the *hiding* property allows for a higher share of the cache than static partitioning, while providing virtually the same security. Furthermore, we showed that SassCache also mitigates the cache occupancy channel, e.g., a cache occupancy attack on OpenSSL AES T-Tables or mbedTLS RSA-4096 can succeed in less than 0.0005 % of cases. Our performance evaluation revealed that SassCache only has an overhead of 1.75 % on average on the last-level cache hit rate in the SPEC2017 and an average decrease of 11.7 p.p. in hit rate for MiBench, GAP, and Scimark benchmark compared to a set-associative LRU cache. Hence, we conclude that SassCache is a promising design for use in appropriate, security-critical contexts.

Acknowledgments

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

References

[1] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. URL: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[2] ARM. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.

[3] ARM Connected blog. Armv8-A architecture: 2016 additions, 2016. URL: https://www.community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions.

[4] Roberto Avanzi. The QARMA block cipher family: Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, 2017.

[5] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP Benchmark Suite. *arXiv:1508.03619*, 2015.

[6] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In *PACT*, 2013.

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

[8] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[9] Rahul Bodduna, Vinod Ganesan, Patanjali Slpsk, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[10] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçın. PRINCE – A low-latency block cipher for pervasive computing applications. In *ASIACRYPT*, 2012.

[11] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *MICRO*, 2020.

[12] Dušan Božilov, Maria Eichlseder, Miroslav Kneževic, Baptiste Lambin, Gregor Leander, Thorben Moos, Ventzislav Nikov, Shahram Rasoolzadeh, Yosuke Todo, and Friedrich Wiemer. PRINCEv2 – more security for (almost) no overhead. In *SAC*, 2020.

[13] Pi-Feng Chiu, Christopher Celio, Krste Asanović, David Patterson, and Borivoje Nikolić. An out-of-order risc-v processor with resilient low-voltage operation in 28nm cmos. In *IEEE Symposium on VLSI Circuits*, 2018.

[14] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

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

[17] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security Symposium*, 2019.

[18] Andrés Djordjalian. Minimally-skewed-associative caches. In *Symposium on Computer Architecture and High Performance Computing. Proceedings.*, 2002.

[19] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2011.

[20] Maria Eichlseder and Daniel Kales. Clustering Related-Tweak Characteristics: Application to MANTIS-6. *IACR Transactions on Symmetric Cryptology*, 2018(2):111–132, 2018.

[21] Chongzhou Fang, Han Wang, Najmeh Nazari, Behnam Omidi, Avesta Sasan, Khaled N Khasawneh, Setareh Rafatirad, and Houman Homayoun. Repttack: Exploiting cloud schedulers to guide co-location attacks. *arXiv:2110.00846*, 2021.

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

[23] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. Cachefx: A framework for evaluating cache security. *arXiv:2201.11377*, 2022.

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

[25] Matthew R. Guthaus, Jeff Ringenberg, Dan Ernst, Todd Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

[26] Zecheng He and Ruby B Lee. How secure is your cache against side-channel attacks? In *MICRO*, 2017.

[27] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[28] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-location detection on the cloud. In *COSADE*, 2016.

[29] Intel. Pin - A Dynamic Binary Instrumentation Tool, 2012. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

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

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

[32] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[33] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA*, 2009.

[34] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[35] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *MICRO*, 2014.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[37] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *DIMVA*, 2015.

[38] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In *RAID*, 2015.

[39] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[40] Larry McVoy and Carl Staelin. Lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*, 1996.

[41] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan Mangard. Cryptag: Thwarting physical and logical memory vulnerabilities using cryptographically colored memory. *arXiv:2012.06761*, 2020.

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

[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[44] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *Cryptology ePrint Archive, Report 2002/169*, 2002.

[45] Roldan Pozo and Bruce R. Miller. Scimark 2.0, 2004. URL: https://math.nist.gov/scimark2/.

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

[47] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, 2021.

[48] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. *arXiv:1908.03383*, 2019.

[49] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*, 2018.

[50] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In *ISCA*, 2019.

[51] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381, 2007.

[52] RISC-V Foundation. The RISC-V Instruction Set Manual, Vol. II: Privileged Architecture, Version 20190608-Priv-MSU-Ratified, 2019.

[53] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.

[54] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: mitigating conflict-based cache attacks with a practical fully-associative design. In *USENIX Security Symposium*, 2021.

[55] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ISCA*, 2011.

[56] Somayeh Sardashti, André Seznec, and David A Wood. Skewed compressed caches. In *MICRO*, 2014.

[57] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity*, 3(1):2, 2020.

[58] Brian C. Schwedock and Nathan Beckmann. Jumanji: The case for dynamic NUCA in the datacenter. In *MICRO*, 2020.

[59] André Seznec. A case for two-way skewed-associative caches. *ACM Computer Architecture News*, 1993.

[60] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through The Cache Occupancy Channel. In *USENIX Security Symposium*, 2019.

[61] Alan Jay Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. Software Eng.*, 4(2), 1978.

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

[63] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *S&P*, 2021.

[64] Mathias Spjuth, Martin Karlsson, and Erik Hagersten. Skewed caches from a low-power perspective. In *Conf. Computing Frontiers*, 2005.

[65] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *NDSS*, 2020.

[66] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *DAC*, 2018.

[67] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 2003.

[68] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.

[69] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494, 2007.

[70] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *MICRO*, 2008.

[71] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[72] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[73] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium*, 2009.

[74] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *EuroSys*, 2009.

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