SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization

Mario Werner, Thomas Unterluggauer, Lukas Giner,
Michael Schwarz, Daniel Gruss, Stefan Mangard
Graz University of Technology

Abstract

Cache side-channel attacks can be leveraged as a building block in attacks leaking secrets even in the absence of software bugs. Currently, there are no practical and generic mitigations with an acceptable performance overhead and strong security guarantees. The underlying problem is that caches are shared in a predictable way across security domains.

In this paper, we eliminate this problem. We present SCATTERCACHE, a novel cache design to prevent cache attacks. SCATTERCACHE eliminates fixed cache-set congruences and, thus, makes eviction-based cache attacks unpractical. For this purpose, SCATTERCACHE retrofits skewed associative caches with a keyed mapping function, yielding a security-domain-dependent cache mapping. Hence, it becomes virtually impossible to find fully overlapping cache sets, rendering current eviction-based attacks infeasible. Even theoretical statistical attacks become unrealistic, as the attacker cannot confine contention to chosen cache sets. Consequently, the attacker has to resort to eviction of the entire cache, making deductions over cache sets or lines impossible and fully preventing high-frequency attacks. Our security analysis reveals that even in the strongest possible attacker model (noise-free), the construction of a reliable eviction set for PRIME+PROBE in an 8-way SCATTERCACHE with 16384 lines requires observation of at least 33.5 million victim memory accesses as compared to fewer than 103 on commodity caches. SCATTERCACHE requires hardware and software changes, yet is minimally invasive on the software level and is fully backward compatible with legacy software while still improving the security level over state-of-the-art caches. Finally, our evaluations show that the runtime performance of software is not curtailed and our design even outperforms state-of-the-art caches for certain realistic workloads.

1 Introduction

Caches are core components of today's computing architectures. They bridge the performance gap between CPU cores and a computer's main memory. However, in the past two decades, caches have turned out to be the origin of a wide range of security threats [10, 15, 27, 38, 39, 43, 44, 51, 76]. In particular, the intrinsic timing behavior of caches that speeds up computing systems allows for cache side-channel attacks (cache attacks), which are able to recover secret information.

Historically, research on cache attacks focused on cryptographic algorithms [10, 44, 51, 76]. More recently, however, cache attacks like PRIME+PROBE [44, 48, 51, 54, 62] and FLUSH+RELOAD [27, 76] have also been used to attack address-space-layout randomization [23, 25, 36], keystroke processing and inter-keystroke timing [26, 27, 60], and general purpose computations [81]. For shared caches on modern multi-core processors, PRIME+PROBE and FLUSH+RELOAD even work across cores executing code from different security domains, e.g., processes or virtual machines.

The most simple cache attacks, however, are covert channels [46, 48, 72]. In contrast to a regular side-channel attack, in a covert channel, the "victim" is colluding and actively trying to transmit data to the attacker, e.g., running in a different security domain. For instance, Meltdown [43], Spectre [38], and Foreshadow [15] use cache covert channels to transfer secrets from the transient execution domain to an attacker. These recent examples highlight the importance of finding practical approaches to thwart cache attacks.

To cope with cache attacks, there has been much research on ways to identify information leaks in a software's memory access pattern, such as static code [19, 20, 41, 45] and dynamic program analysis [34, 71, 74, 77]. However, mitigating these leaks both generically and efficiently is difficult. While there are techniques to design software without address-based information leaks, such as unifying control flow [17] and bitsliced implementations of cryptography [37, 40, 58], their general application to arbitrary software remains difficult. Hence, protecting against cache attacks puts a significant burden on software developers aiming to protect secrets in the view of microarchitectural details that vary a lot across different Instruction-Set Architecture (ISA) implementations.

A different direction to counteract cache attacks is to design

more resilient cache architectures. Typically, these architectures modify the cache organization in order to minimize interference between different processes, either by breaking the trivial link between memory address and cache index [22, 55, 67, 69, 70] or by providing exclusive access to cache partitions for critical code [53, 57, 69]. While cache partitioning completely prevents cache interference, its rather static allocation suffers from scalability and performance issues. On the other hand, randomized cache (re-)placement [69, 70] makes mappings of memory addresses to cache indices random and unpredictable. Yet, managing these cache mappings in lookup tables inheres extensive changes to the cache architecture and cost. Finally, the introduction of a keyed function [55, 67] to pseudorandomly map the accessed memory location to the cache-set index can counteract PRIME+PROBE attacks. However, these solutions either suffer from a low number of cache sets, weakly chosen functions, or cache interference for shared memory and thus require to change the key frequently at the cost of performance.

Hence, there is a strong need for a practical and effective solution to thwart both cache attacks and cache covert channels. In particular, this solution should (1) make cache attacks sufficiently hard, (2) require as little software support as possible, (3) embed flexibly into existing cache architectures, (4) be efficiently implementable in hardware, and (5) retain or even enhance cache performance.

Contribution. In this paper, we present SCATTERCACHE, which achieves all these goals. SCATTERCACHE is a novel and highly flexible cache design that prevents cache attacks such as EVICT+RELOAD and PRIME+PROBE and severely limits cache covert channel capacities by increasing the number of cache sets beyond the number of physically available addresses with competitive performance and implementation cost. Hereby, SCATTERCACHE closes the gap between previous secure cache designs and today's cache architectures by introducing a minimal set of cache modifications to provide strong security guarantees.

Most prominently, SCATTERCACHE eliminates the fixed cache-set congruences that are the cornerstone of PRIME+PROBE attacks. For this purpose, SCATTERCACHE builds upon two ideas. First, SCATTERCACHE uses a keyed mapping function to translate memory addresses and the active security domain, e.g., process, to cache set indices. Second, similar to skewed associative caches [63], the mapping function in SCATTERCACHE computes a different index for each cache way. As a result, the number of different cache sets increases exponentially with the number of ways. While SCATTERCACHE makes finding fully identical cache sets statistically impossible on state-of-the-art architectures, the complexity for exploiting inevitable partial cache-set collisions also rises heavily. The reason is in part that the mapping of memory addresses to cache sets in SCATTERCACHE is different for each security domain. Hence, and as our security analysis shows, the construction of a reliable eviction set for PRIME+PROBE in an 8-way SCATTERCACHE with 16384 lines requires observation of at least 33.5 million victim memory accesses as compared to fewer than 103 on commodity caches, rendering these attacks impractical on real systems with noise.

Additionally, SCATTERCACHE effectively prevents FLUSH+RELOAD-based cache attacks, e.g., on shared libraries, as well. The inclusion of security domains in SCATTERCACHE and its mapping function preserves shared memory in RAM, but prevents any cache lines to be shared across security boundaries. Yet, SCATTERCACHE supports shared memory for inter-process communication via dedicated separate security domains. To achieve highest flexibility, managing the security domains of SCATTERCACHE is done by software, e.g., the operating system. However, SCATTERCACHE is fully backwards compatible and already increases the effort of cache attacks even without any software support. Nevertheless, the runtime performance of software on SCATTERCACHE is highly competitive and, on certain workloads, even outperforms cache designs implemented in commodity CPUs.

SCATTERCACHE constitutes a comparably simple extension to cache and processor architectures with minimal hardware cost: SCATTERCACHE essentially only adds additional index derivation logic, *i.e.*, a lightweight cryptographic primitive, and an index decoder for each scattered cache way. Moreover, to enable efficient lookups and writebacks, SCATTERCACHE stores the index bits from the physical address in addition to the tag bits, which adds $< 5\,\%$ storage overhead per cache line. Finally, SCATTERCACHE consumes one bit per page-table entry ($\approx 1.5\,\%$ storage overhead per page-table entry) for the kernel to communicate with the user space.

Outline. This paper is organized as follows. In Section 2, we provide background information on caches and cache attacks. In Section 3, we describe the design and concept of SCATTERCACHE. In Section 4, we analyze the security of SCATTERCACHE against cache attacks. In Section 5, we provide a performance evaluation. We conclude in Section 6.

2 Background

In this section, we provide background on caches, cache side-channel attacks, and resilient cache architectures.

2.1 Caches

Modern computers have a memory hierarchy consisting of many layers, each following the principle of locality, storing data that is expected to be used in the future, e.g., based on what has been accessed in the past. Modern processors have a hierarchy of caches that keep instructions and data likely to be used in the future near the execution core to avoid the latency of accesses to the slow (DRAM) main memory. This cache hierarchy typically consists of 2 to 4 layers, where the

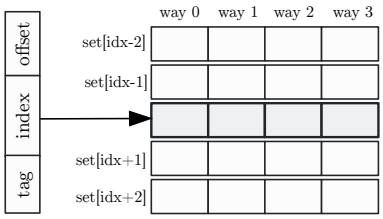

Figure 1: Indexing cache sets in a 4-way set-associative cache.

lowest layer is the smallest and fastest, typically only a few kilobytes. The last-level cache is the largest cache, typically in the range of several megabytes. On most processors, the last-level cache is shared among all cores. The last-level cache is often inclusive, *i.e.*, any cache line in a lower level cache must also be present in the last-level cache.

Caches are typically organized into *cache sets* that are composed of multiple *cache lines* or *cache ways*. The cache set is determined by computing the cache index from address bits. Figure 1 illustrates the indexing of a 4-way set-associative cache. As the cache is small and the memory large, many memory locations map to the same cache set (*i.e.*, the addresses are *congruent*). The replacement policy (e.g., pseudo-LRU, random) decides which way is replaced by a newly requested cache line. Any process can observe whether data is cached or not by observing the memory access latency which is the basis for cache side-channel attacks.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks have been studied for over the past two decades, initially with a focus on cryptographic algorithms [10, 39, 51, 52, 54, 68]. Today, a set of powerful attack techniques enable attacks in realistic cross-core scenarios. Based on the access latency, an attacker can deduce whether or not a cache line is in the cache, leaking two opposite kinds of information. (1) By continuously removing (*i.e.*, evicting or flushing) a cache line from the cache and measuring the access latency, an attacker can determine whether this cache line has been accessed by another process. (2) By continuously filling a part of the cache with attacker-accessible data, the attacker can measure the contention of the corresponding part, by checking whether the attacker-accessible data remained in the cache. Contention-based attacks work on different layers:

The Entire Cache or Cache Slices. An attacker can measure contention of the entire cache or a cache slice. Maurice et al. [46] proposed a covert channel where the sender evicts the entire cache to leak information across cores and the victim observes the cache contention. A similar attack could be mounted on a cache slice if the cache slice function is known [47]. The granularity is extremely coarse, but with statistical attacks can leak meaningful information [61].

Cache Sets. An attacker can also measure the contention of a cache set. For this, additional knowledge may be required, such as the mapping from virtual addresses to physical addresses, as well as the functions mapping physical addresses to cache slices and cache sets. The attacker continuously fills a cache set with a set of congruent memory locations. Filling a cache set is also called cache-set eviction, as it evicts any previously contained cache lines. Only if some other process accessed a congruent memory location, memory locations are evicted from a cache set. The attacker can measure this for instance by measuring runtime variations in a so-called EVICT+TIME attack [51]. The EVICT+TIME technique has mostly been applied in attacks on cryptographic implementations [31, 42, 51, 65]. Instead of the runtime, the attacker can also directly check how many of the memory locations are still cached. This attack is called PRIME+PROBE [51]. Many PRIME+PROBE attacks on private L1 caches have been demonstrated [3, 14, 51, 54, 80]. More recently, PRIME+PROBE attacks on last-level caches have also been demonstrated in various generic use cases [4, 44, 48, 50, 59, 79].

Cache Lines. At a cache line granularity, the attacker can measure whether a memory location is cached or not. As already indicated above, here the logic is inverted. Now the attacker continuously evicts (or flushes) a cache line from the cache. Later on, the attacker can measure the latency and deduce whether another process has loaded the cache line into the cache. This technique is called FLUSH+RELOAD [28, 76]. FLUSH+RELOAD has been studied in a long list of different attacks [4–6, 27, 32, 35, 42, 76, 78, 81]. Variations of FLUSH+RELOAD are FLUSH+FLUSH [26] and EVICT+RELOAD [27, 42].

Cache Covert Channels

Cache covert channels are one of the simplest forms of cache attacks. Instead of an attacker process attacking a victim process, both processes collude to covertly communicate using the cache as transmission channel. Thus, in this scenario, the colluding processes are referred to as sender and receiver, as the communication is mostly unidirectional. A cache covert channel allows bypassing all architectural restrictions regarding data exchange between processes.

Cache covert channels have been shown using various cache attacks, such as PRIME+PROBE [44, 48, 73, 75] and FLUSH+RELOAD [26]. They achieve transmission rates of up to 496 kB/s [26]. Besides native attacks, covert channels have also been shown to work within virtualized environments, across virtual machines [44, 48, 75]. Even in these restricted environments, cache-based covert channels achieve transmission rates of up to 45 kB/s [48].

2.3 Resilient Cache Architectures

The threat of cache-based attacks sparked several novel cache architectures designed to be resilient against these attacks. While fixed cache partitions [53] lack flexibility, randomized

cache allocation appears to be more promising. The following briefly discusses previous designs for a randomized cache.

RPCache [69] and NewCache [70] completely disrupt the meaningful observability of interference by performing random (re-)placement of lines in the cache. However, managing the cache mappings efficiently either requires full associativity or content addressable memory. While optimized addressing logic can lead to efficient implementations, these designs differ significantly from conventional architectures.

Time-Secure Caches [67] is based on standard set-associative caches that are indexed with a keyed function that takes cache line address and Process ID (PID) as an input. While this design destroys the obvious cache congruences between processes to minimize cache interference, a comparably weak indexing function is used. Eventually, re-keying needs to be done quite frequently, which amounts to flushing the cache and thus reduces practical performance. SCATTERCACHE can be seen as a generalization of this approach with higher entropy in the indexing of cache lines.

CEASER [55] as well uses standard set-associative caches with keyed indexing, which, however, does not include the PID. Hence, inter-process cache interference is predictable based on in-process cache collisions. As a result, CEASER strongly relies on continuous re-keying of its index derivation to limit the time available for conducting an attack. For efficient implementation, CEASER uses its own lightweight cryptographic primitive designed for that specific application.

3 ScatterCache

As Section 2 showed, caches are a serious security concern in contemporary computing systems. In this section, we hence present SCATTERCACHE—a novel cache architecture that counteracts cache-based side-channel attacks by skewed pseudorandom cache indexing. After discussing the main idea behind SCATTERCACHE, we discuss its building blocks and system integration in more detail. SCATTERCACHE's security implications are, subsequently, analyzed in Section 4.

3.1 Targeted Properties

Even though contemporary transparent cache architectures are certainly flawed from the security point of view, they still feature desirable properties. In particular, for regular computations, basically no software support is required for cache maintenance. Also, even in the case of multitasking and -processing, no dedicated cache resource allocation and scheduling is needed. Finally, by selecting the cache size and the number of associative ways, chip vendors can trade hardware complexity and costs against performance as desired.

SCATTERCACHE's design strives to preserve these features while adding the following three security properties:

1. Between software defined security domains (e.g., different processes or users on the same machine, different VMs, ...), even for exactly the same physical addresses, cache lines should only be shared if cross-context coherency is required (*i.e.*, writable shared memory).
2. Finding and exploiting addresses that are congruent in the cache should be as hard as possible (*i.e.*, we want to "break" the direct link between the accessed physical address and the resulting cache set index for adversaries).
3. Controlling and measuring complete cache sets should be hard in order to prevent eviction-based attacks.

Finally, to ease the adoption and to utilize the vast knowledge on building efficient caches, the SCATTERCACHE hardware should be as similar to current cache architectures as possible.

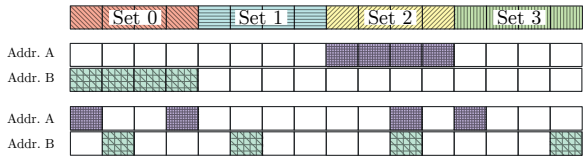

Figure 2: Flattened visualization of mapping addresses to cache sets in a 4-way set-associative cache with 16 cache lines. *Top:* Standard cache where index bits select the cache set. *Middle:* Pseudorandom mapping from addresses to cache sets. The mapping from cache lines to sets is still static. *Bottom:* Pseudorandom mapping from addresses to a set of cache lines that dynamically form the cache set in SCATTERCACHE.

3.2 Idea

Two main ideas influenced the design of SCATTERCACHE to reach the desired security properties. First, addresses should be translated to cache sets using a keyed, security-domain aware mapping. Second, which exact n_{ways} cache lines form a cache set in a n_{ways}-way associative cache should not be fixed, but depend on the currently used key and security domain too. SCATTERCACHE combines both mappings in a single operation that associates each address, depending on the key and security domain, with a set of up to n_{ways} cache lines. In other words, in a generic SCATTERCACHE, any possible combination of up to n_{ways} cache lines can form a cache set.

Figure 2 visualizes the idea and shows how it differs from related work. Traditional caches as well as alternative designs which pseudorandomly map addresses to cache sets statically allocate cache lines to cache sets. Hence, as soon as a cache set is selected based on (possibly encrypted) index bits, always the same n_{ways} cache lines are used. This means that all addresses mapping to the same cache set are congruent and enables PRIME+PROBE-style attacks.

In SCATTERCACHE, on the other hand, the cache set for a particular access is a pseudorandom selection of arbitrary n_{ways} cache lines from all available lines. As a result, there is a much higher number of different cache sets and finding addresses with identical cache sets becomes highly unlikely.

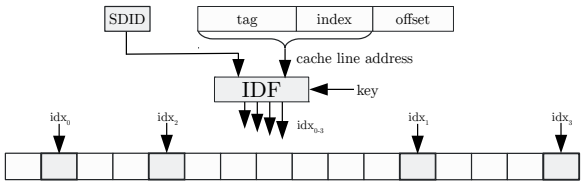

Figure 3: Idea: For an n_{ways} associative cache, n_{ways} indices into the cache memory are derived using a cryptographic IDF. This IDF effectively randomizes the mapping from addresses to cache sets as well as the composition of the cache set itself.

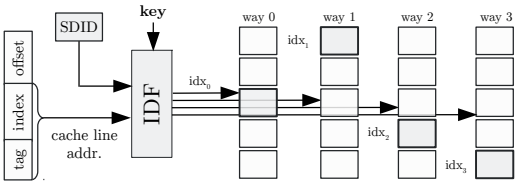

Figure 4: 4-way set-associative SCATTERCACHE where each index addresses exclusively one cache way.

Instead, as shown at the bottom of Figure 2, at best, partially overlapping cache sets can be found (cf. Section 4.3), which makes exploitation tremendously hard in practice.

A straightforward concept for SCATTERCACHE is shown in Figure 3. Here, the Index Derivation Function (IDF) combines the mapping operations in a single cryptographic primitive. In a set-associative SCATTERCACHE with set size n_{ways}, for each input address, the IDF outputs n_{ways} indices to form the cache set for the respective access. How exactly the mapping is performed in SCATTERCACHE is solely determined by the used key, the Security Domain Identifier (SDID), and the IDF. Note that, as will be discussed in Section 3.3.1, hash-based as well as permutation-based IDFs can be used in this context.

Theoretically, a key alone is sufficient to implement the overall idea. However, separating concerns via the SDID leads to a more robust and harder-to-misuse concept. The key is managed entirely in hardware, is typically longer, and gets switched less often than the SDID. On the other hand, the SDID is managed solely by the software and, depending on the implemented policy, has to be updated quite frequently. Importantly, as we show in Section 4, SCATTERCACHE alone already provides significantly improved security in PRIME+PROBE-style attack settings even without software support (*i.e.*, SDID is not used).

3.3 SCATTERCACHE Design

In the actual design we propose for SCATTERCACHE, the indices (*i.e.*, IDF output) do not address into one huge joint cache array. Instead, as shown in Figure 4, each index addresses a separate memory, *i.e.*, an independent cache way.

On the one hand, this change is counter-intuitive as it decreases the number of possible cache sets from $\binom{n_{ways}\cdot 2^{b_{indices}}+n_{ways}-1}{n_{ways}}$ to $2^{b_{indices}\cdot n_{ways}}$. However, this reduction in possibilities is acceptable. For cache configurations with up to 4 cache ways, the gap between both approaches is only a few bits. For higher associativity, the exponential growth ensures that sufficiently many cache sets exist.

On the other hand, the advantages gained from switching to this design far outweigh the costs. Namely, for the original idea, no restrictions on the generated indices exist. Therefore, a massive n_{ways}-fold multi-port memory would be required to be able to lookup a n_{ways}-way cache-set in parallel. The design shown in Figure 4 does not suffer from this problem and permits to instantiate SCATTERCACHE using n_{ways} instances of simpler/smaller memory. Furthermore, this design guarantees that even in case the single index outputs of the IDF collide, the generated cache always consists of exactly n_{ways} many cache lines. This effectively precludes the introduction of systematic biases for potentially "weak" address-key-SDID combinations that map to fewer than n_{ways} cache lines.

In terms of cache-replacement policy, SCATTERCACHE uses simple random replacement to ensure that no systematic bias is introduced when writing to the cache and to simplify the security analysis. Furthermore, and as we will show in Section 5, the performance of SCATTERCACHE with random replacement is competitive to regular set associative caches with the same replacement policy. Therefore, evaluation of alternative replacement policies has been postponed. Independent of the replacement policy, it has to be noted that, for some IDFs, additional tag bits have to be stored in SCATTERCACHE. In particular, in case of a non invertible IDF, the original index bits need to be stored to facilitate write back of dirty cache lines and to ensure correct cache lookups. However, compared to the amount of data that is already stored for each cache line, the overhead of adding these few bits should not be problematic ($< 5\,\%$ overhead).

In summary, the overall hardware design of SCATTERCACHE closely resembles a traditional set-associative architecture. The only differences to contemporary fixed-set designs is the more complex IDF and the amount of required logic which permits to address each way individually. However, both changes are well understood. As we detail in the following section, lightweight (*i.e.*, low area and latency) cryptographic primitives are suitable building blocks for the IDF. Similarly, duplication of addressing logic is already common practice in current processors. Modern Intel architectures, for example, already partition their Last-Level Cache (LLC) into multiple smaller cache slices with individual addressing logic.

3.3.1 Suitable Index Derivation Functions

Choosing a suitable IDF is essential for both security and performance. In terms of security, the IDF has to be an unpredictable (but still deterministic) mapping from physical addresses to indices. Following Kerckhoffs's principle, even

for attackers which know every detail except the key, three properties are expected from the IDF: (1) Given perfect control over the public inputs of the function (*i.e.*, the physical address and SDID) constructing colliding outputs (*i.e.*, the indices) should be hard. (2) Given colliding outputs, determining the inputs or constructing further collisions should be hard. (3) Recovering the key should be infeasible given input and output for the function.

Existing Building Blocks: Cryptographic primitives like (tweakable) block ciphers, Message Authentication Codes (MACs), and hash functions are designed to provide these kind of security properties (e.g., indistinguishability of encryptions, existential unforgeability, pre-image and collision resistance). Furthermore, design and implementation of cryptographic primitives with tight performance constraints is already a well-established field of research which we want to take advantage of. For example, with PRINCE [13], a low-latency block cipher, and QARMA [8], a family of low-latency tweakable block ciphers, exist and can be used as building blocks for the IDF. Such tweakable block ciphers are a flexible extension to ordinary block ciphers, which, in addition to a secret key, also use a public, application-specific tweak to en-/decrypt messages. Similarly, sponge-based MAC, hash and cipher designs are a suitable basis for IDFs. These sponge modes of operation are built entirely upon permutations, e.g., Keccak-p, which can often be implemented with low latency [7, 11]. Using such cryptographic primitives, we define the following two variants of building IDFs:

Hashing Variant (SCv1): The idea of SCv1 is to combine all IDF inputs using a single cryptographic primitive with pseudo random output. MACs (e.g., hash-based) are examples for such functions and permit to determine the output indices by simply selecting the appropriate number of disjunct bits from the calculated tag. However, also other cryptographic primitives can be used for instantiating this IDF variant.

It is, for example possible to slice the indices from the ciphertext of a regular block cipher encryption which uses the concatenation of cache line address and the SDID as the plaintext. Similarly, tweakable block ciphers allow to use the SDID as a tweak instead of connecting it to the plaintext. Interestingly, finding cryptographic primitives for SCv1 IDFs is comparably simple given that the block sizes do not have to match perfectly and the output can be truncated as needed.

However, there are also disadvantages when selecting the indices pseudo randomly, like in the case of SCv1. In particular, when many accesses with high spatial locality are performed, index collisions get more likely. This is due to the fact that collisions in SCv1 output have birthday-bound complexity. Subsequently, performance can degrade when executing many different accesses with high spatial locality. Fortunately, this effect weakens with increasing way numbers, *i.e.*, an increase in associativity decreases the probability that all index outputs of the IDF collide.

In summary, SCv1 translates the address without distinguishing between index and tag bits. Given a fixed key and SDID, the indices are simply pseudo random numbers that are derived using a single cryptographic primitive.

Permutation Variant (SCv2): The idea behind the permutation variant of the IDF is to distinguish the index from the tag bits in the cache line address during calculation of the indices. Specifically, instead of generating pseudo random indices from the cache line address, tag dependent permutations of the input index are calculated.

The reason for preferring a permutation over pseudo random index generation is to counteract the effect of birthday-bound index collisions, as present in SCv1. Using a tag dependent permutation of the input index mitigates this problem by design since permutations are bijections that, for a specific tag, cannot yield colliding mappings.

Like in the hashing variant, a tweakable block cipher can be used to compute the permutation. Here, the concatenation of the tag bits, the SDID and the way index constitutes the tweak while the address' index bits are used as the plaintext. The resulting ciphertext corresponds to the output index for the respective way. Note that the block size of the cipher has to be equal to the size of the index. Additionally, in order to generate all indices in parallel, one instance of the tweakable block cipher is needed per cache way. However, as the block size is comparably small, each cipher instance is also smaller than an implementation of the hashing IDF (SCv1).

Independently of the selected IDF variant, we leave the decision on the actually used primitive to the discretion of the hardware designers that implement SCATTERCACHE. They are the only ones who can make a profound decision given that they know the exact instantiation parameters (e.g., SDID/key/index/tag bit widths, number of cache ways) as well as the allocatable area, performance, and power budget in their respective product. However, we are certain that, even with the already existing and well-studied cryptographic primitives, SCATTERCACHE implementations are feasible for common computing platforms, ranging from Internet of Things (IoT) devices to desktop computers and servers.

Note further that we expect that, due to the limited observability of the IDF output, weakened (*i.e.*, round reduced) variants of general purpose primitives are sufficient to achieve the desired security level. This is because adversaries can only learn very little information about the function output by observing cache collisions (*i.e.*, no actual values). Subsequently, many more traces have to be observed for mounting an attack. Cryptographers can take advantage of this increase in data complexity to either design fully custom primitives [55] or to decrease the overhead of existing designs.

3.3.2 Key Management and Re-Keying

The key in our SCATTERCACHE design plays a central role in the security of the entire approach. Even when the SDIDs are known, it prevents attackers from systematically constructing

eviction sets for specific physical addresses and thwarts the calculation of addresses from collision information. Keeping the key confidential is therefore of highest importance.

We ensure this confidentiality in our design by mandating that the key of is fully managed by hardware. There must not be any way to configure or retrieve this key in software. This approach prevents various kinds of software-based attacks and is only possible due to the separation of key and SDID.

The hardware for key management is comparably simple as well. Each time the system is powered up, a new random key is generated and used by the IDF. The simplicity of changing the key during operation strongly depends on the configuration of the cache. For example, in a write-through cache, changing the key is possible at any time without causing data inconsistency. In such a scenario, a timer or performance-counter-based rekeying scheme is easily implementable. Note, however, that the interval between key changes should not be too small as each key change corresponds to a full cache flush.

On the other hand, in a cache with write-back policy, the key has to be kept constant as long as dirty cache lines reside in the cache. Therefore, before the key can be changed in this scenario without data loss, all modified cache lines have to be written back to memory first. The x86 Instruction-Set Architecture (ISA), for example, features the `WBINVD` instruction that can be used for that purpose.

If desired, also more complex rekeying schemes, like way-wise or cache-wide dynamic remapping [55], can be implemented. However, it is unclear if adding the additional hardware complexity is worthwhile. Even without changing the key, mounting cache attacks against ScatterCache is much harder than on traditional caches (see Section 4). Subsequently, performing an occasional cache flush to update the key can be the better choice.

3.3.3 Integration into Existing Cache Architectures

ScatterCache is a generic approach for building processor caches that are hard to exploit in cache-based side channel attacks. When hardening a system against cache attacks, independent of ScatterCache, we recommend to restrict flush instructions to privileged software. These instruction are only rarely used in benign userspace code and restricting them prevents the applicability of the whole class of flush-based attacks from userspace. Fortunately, recent ARM architectures already support this restriction.

Next, ScatterCaches can be deployed into the system to protect against eviction based attacks. While not inherently limited to, ScatterCaches are most likely to be deployed as LLCs in modern processor architectures. Due to their large size and the fact that they are typically shared across multiple processor cores, LLCs are simply the most prominent cache attack target and require the most protection. Compared to that, lower cache levels that typically are only accessible by a single processor core, hold far less data and are much harder to attack on current architectures. Still, usage of (unkeyed) skewed [63] lower level caches is an interesting option that has to be considered in this context.

Another promising aspect of employing a ScatterCache as LLC is that this permits to hide large parts of the IDF latency. For example, using a fully unrolled and pipelined IDF implementation, calculation of the required ScatterCache indices can already be started, or even performed entirely, in parallel to the lower level cache lookups. While unneeded results can easily be discarded, this ensures that the required indices for the LLC lookup are available as soon as possible.

Low latency primitives like QARMA, which is also used in recent ARM processors for pointer authentication, are promising building blocks in this regard. The minimal latency Avanzi [8] reported for one of the QARMA-64 variants is only 2.2 ns. Considering that this number is even lower than the time it takes to check the L1 and L2 caches on recent processors (e.g., 3 ns on a 4 GHz Intel Kabylake [2], 9 ns on an ARM Cortex-A57 in an AMD Opteron A1170 [1]), implementing IDFs without notable latency seems feasible.

3.4 Processor Interaction and Software

Even without dedicated software support, ScatterCache increases the complexity of cache-based attacks. However, to make full use of ScatterCache, software assistance and some processor extensions are required.

Security Domains. The ScatterCache hardware permits to isolate different security domains from each other via the SDID input to the IDF. Unfortunately, depending on the use case, the definition on what is a security domain can largely differ. For example, a security domain can be a chunk of the address space (e.g., SGX enclaves), a whole process (e.g., TrustZone application), a group of processes in a common container (e.g., Docker, LXC), or even a full virtual machine (e.g., cloud scenario). Considering that it is next to impossible to define a generic policy in hardware that can capture all these possibilities, we delegate the distinction to software that knows about the desired isolation properties, e.g., the Operating System (OS).

ScatterCache Interface. Depending on the targeted processor architecture, different design spaces can be explored before deciding how the current SDID gets defined and what channels are used to communicate the identifier to the ScatterCache. However, at least for modern Intel and ARM processors, binding the currently used SDID to the virtual memory management via user defined bits in each Page Table Entry (PTE) is a promising approach. In more detail, one or more bits can be embedded into each PTE that select from a list, via one level of indirection, which SDID should be used when accessing the respective page.

Both ARM and Intel processors already support a similar mechanism to describe memory attributes of a memory mapping. The x86 architecture defines so-called Page Attribute Ta-

bles (PATs) to define how a memory mapping can be cached. Similarly, the ARM architecture defines Memory Attribute Indirection Registers (MAIRs) for the same purpose. Both PAT and MAIR define a list of 8 memory attributes which are applied by the Memory Management Unit (MMU). The MMU interprets a combination of 3 bits defined in the PTE as index into the appropriate list, and applies the corresponding memory attribute. Adding the SDID to these attribute lists permits to use up to 8 different security domains within a single process. The absolute number of security domains, on the other hand, is only limited by the used IDF and them number of bits that represent the SDID.

Such indirection has a huge advantage over encoding data directly in a PTE. The OS can change a single entry within the list to affect all memory mappings using the corresponding entry. Thus, such a mechanism is beneficial for SCATTERCACHE, where the OS wants to change the SDID for all mappings of a specific process.

Backwards Compatibility. Ensuring backwards compatibility is a key factor for gradual deployment of SCATTERCACHE. By encoding the SDID via a separate list indexed by PTE bits, all processes, as well as the OS, use the same SDID, *i.e.*, the SDID stored as first element of the list (assuming all corresponding PTE bits are '0' by default). Thus, if the OS is not aware of the SCATTERCACHE, all processes—including the OS—use the same SDID. From a software perspective, functionally, SCATTERCACHE behaves the same as currently deployed caches. Only if the OS specifies SDIDs in the list, and sets the corresponding PTE bits to use a certain index, SCATTERCACHE provides its strong security properties.

Implementation Example. In terms of capabilities, having a single bit in each PTE, for example, is already sufficient to implement security domains with process granularity and to maintain a dedicated domain for the OS. In this case, $SDID_0$ can always be used for the OS ID while $SDID_1$ has to be updated as part of the context switch and is always used for the scheduled user space process. Furthermore, by reusing the SDID of the OS, also shared memory between user space processes can easily be implemented without security impact.

Interestingly, SCATTERCACHE fully preserves the capability of the OS to share read-only pages (*i.e.*, libraries) also across security domains as no cache lines will be shared. In contrast, real shared memory has to always be accessed via the same SDID in all processes to ensure data consistency. In general, with SCATTERCACHE, as long as the respective cache lines have not been flushed to RAM, data always needs to be accessed with the same SDID the data has been written with to ensure correctness. This is also true for the OS, which has to ensure that no dirty cache lines reside in the cache, e.g., when a page gets assigned to a new security domain.

A case which has to be explicitly considered by the OS is copying data from user space to kernel space and vice versa. The OS can access the user space via the direct-physical map or via the page tables of the process. Thus, the OS has to select the correct SDID for the PTE used when copying data. Similarly, if the OS sets up page tables, it has to use the same SDID as the MMU uses for resolving page tables.

4 Security Evaluation

SCATTERCACHE is a novel cache design to efficiently thwart cache-based side-channel attacks. In the following, we investigate the security of SCATTERCACHE in terms of state-of-the-art side-channel attacks using both theoretical analysis and simulation-based results. In particular, we elaborate on the complexity of building the eviction sets and explore the necessary changes to the standard PRIME+PROBE technique to make it viable on the SCATTERCACHE architecture.

4.1 Applicability of Cache Attacks

While certain types of cache attacks, such as FLUSH+FLUSH, FLUSH+RELOAD and EVICT+RELOAD, require a particular cache line to be shared, attacks such as PRIME+PROBE have less stringent constraints and only rely on the cache being a shared resource. As sharing a cache line is the result of shared memory, we analyze the applicability of cache attacks on SCATTERCACHE with regard to whether the underlying memory is shared between attacker and victim or not.

Shared, read-only memory. Read-only memory is frequently shared among different processes, e.g., in case of shared code libraries. SCATTERCACHE prevents cache attacks involving shared read-only memory by introducing security domains. In particular, SCATTERCACHE maintains a separate copy of shared read-only memory in cache for each security domain, *i.e.*, the cache lines belonging to the same shared memory region are not being shared in cache across security domains anymore. As a result, reloading data into or flushing data out of the cache does not provide any information on another security domain's accesses to the respective shared memory region. Note, however, that the cache itself is shared, leaving attacks such as PRIME+PROBE still feasible.

Shared, writable memory. Exchanging data between processes requires shared, writable memory. To ensure cache coherency, writing shared memory regions must always use the same cache line and hence the same security domain for that particular memory region—even for different processes. While attacks on these shared memory regions involving `flush` instructions can easily be mitigated by making these instructions privileged, EVICT+RELOAD remains feasible. Still, SCATTERCACHE significantly hampers the construction of targeted eviction sets by skewing, *i.e.*, individually addressing, the cache ways. Moreover, its susceptibility to EVICT+RELOAD attacks is constrained to the processes sharing the respective memory region. Nevertheless, SCATTERCACHE requires writable shared memory to be used only as an interface for data transfer rather than sensitive computations. In addition, PRIME+PROBE attacks are still possible.

Unshared memory. Unshared memory regions never share the same cache line, hence making attacks such as FLUSH+FLUSH, FLUSH+RELOAD and EVICT+RELOAD infeasible. However, as the cache component itself is shared, cache attacks such as PRIME+PROBE remain possible.

As our analysis shows, SCATTERCACHE prevents a wide range of cache attacks that exploit the sharing of cache lines across security boundaries. While PRIME+PROBE attacks cannot be entirely prevented as long as the cache itself is shared, SCATTERCACHE vastly increases their complexity in all aspects. The pseudorandom cache-set composition in SCATTERCACHE prevents attackers from learning concrete cache sets from memory addresses and vice versa. Even if attackers are able to profile information about the mapping of memory addresses to cache-sets in their own security domain, it does not allow them infer the mapping of cache-sets to memory addresses in other security domains. To gain information about memory being accessed in another security domain, an attacker needs to profile the mapping of the attacker's address space to cache lines that are being used by the victim when accessing the memory locations of interest. The effectiveness of PRIME+PROBE attacks thus heavily relies on the complexity of such a profiling phase. We elaborate on the complexity of building eviction sets in Section 4.3.

4.2 Other Microarchitectural Attacks

Many other microarchitectural attacks are not fully mitigated but hindered by SCATTERCACHE. For instance, Meltdown [43] and Spectre [38] attacks cannot use the cache efficiently anymore but must resort to other covert channels. Also, DRAM row buffer attacks and Rowhammer attacks are negatively affected as they require to bypass the cache and reach DRAM. While these attacks are already becoming more difficult due to closed row policies in modern processors [24], we propose to make flush instructions privileged, removing the most widely used cache bypass. Cache eviction gets much more difficult with SCATTERCACHE and additionally, spurious cache misses will open DRAM rows during eviction. These spurious DRAM row accesses make the row hit side channel impractical and introduce a significant amount of noise on the row conflict side channel. Hence, while these attacks are not directly in the scope of this paper, SCATTERCACHE arguably has a negative effect on them.

4.3 Complexity of Building Eviction Sets

Cache skewing significantly increases the number of different cache sets available in cache. However, many of these cache sets will overlap partially, *i.e.*, in $1 \leq i < n_{ways}$ ways. The complexity of building eviction sets for EVICT+RELOAD and PRIME+PROBE in SCATTERCACHE thus depends on the overlap of cache sets.

4.3.1 Full Cache-Set Collisions

The pseudorandom assembly of cache sets in SCATTERCACHE results in $2^{b_{indices} \cdot n_{ways}}$ different compositions. For a given target address, this results in a probability of $2^{-b_{indices} \cdot n_{ways}}$ of finding another address that maps exactly to the same cache lines in its assigned cache set. While dealing with this complexity alone can be considered impractical in a real-world scenario, note that it will commonly even exceed the number of physical addresses available in current systems, rendering full cache-set collisions completely infeasible. A 4-way cache, for example, with $b_{indices} = 12$ index bits yields 2^{48} different cache sets, which already exceeds the address space of state-of-the-art systems.

4.3.2 Partial Cache-Set Collisions

While full cache-set collisions are impractical, partial collisions of cache sets frequently occur in skewed caches such as SCATTERCACHE. If the cache sets of two addresses overlap, two cache sets will most likely have a single cache line in common. For this reason, we analyze the complexity of eviction for single-way collisions in more detail.

Randomized Single-Set Eviction. Without knowledge of the concrete mapping from memory addresses to cache sets, the trivial approach of eviction is to access arbitrary memory locations, which will result in accesses to pseudorandom cache sets in SCATTERCACHE. To elaborate on the performance of this approach, we consider a cache with $n_{lines} = 2^{b_{indices}}$ cache lines per way and investigate the eviction probability for a single cache way, which contains a specific cache line to be evicted. Given that SCATTERCACHE uses a random (re-)placement policy, the probabilities of each cache way are independent, meaning that each way has the same probability of being chosen. Subsequently, the attack complexity on the full SCATTERCACHE increases linearly with the number of cache ways, *i.e.*, the attack gets harder.

The probability of an arbitrary memory accesses to a certain cache way hitting a specific cache line is $p = n_{lines}^{-1}$. Performing $n_{accesses}$ independent accesses to this cache way increases the odds of eviction to a certain confidence level α.

$$\alpha = 1 - (1 - n_{lines}^{-1})^{n_{accesses}}$$

Equivalently, to reach a certain confidence α in evicting the specific cache line, attackers have to perform

$$\mathbb{E}(n_{accesses}) = \frac{\log(1-\alpha)}{\log(1-n_{lines}^{-1})}$$

independent accesses to this cache way, which amounts to their attack complexity. Hence, to evict a certain cache set from an 8-way SCATTERCACHE with 2^{11} lines per way with $\alpha = 99\,\%$ confidence, the estimated attack complexity using this approach is $n_{accesses} \cdot n_{ways} \approx 2^{16}$ independent accesses.

Randomized Multi-Set Eviction. Interestingly, eviction of multiple cache sets using arbitrary memory accesses has

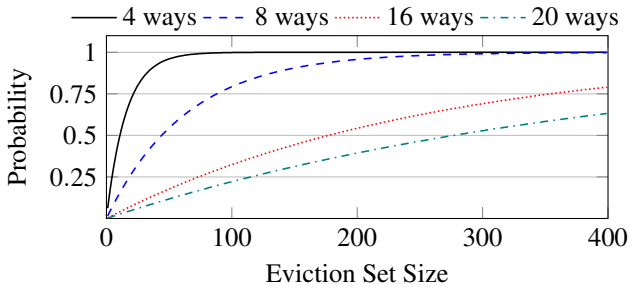

Figure 5: Eviction probability depending on the size of the eviction set and the number of ways.

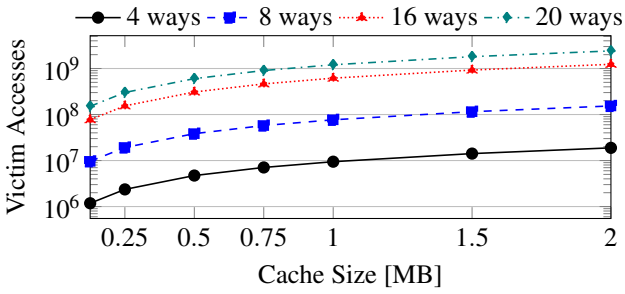

Figure 6: Number of required accesses to the target address to construct a set large enough to achieve 99 % eviction rate when no shared memory is available (cache line size: 32 bytes).

similar complexity. In this regard, the *coupon collector's problem* gives us a tool to estimate the number of accesses an attacker has to perform to a specific cache way to evict a certain percentage of cache lines in the respective way. In more detail, the coupon collector's problem provides the expected number of accesses $n_{accesses}$ required to a specific cache way such that n_{hit} out of all n_{lines} cache lines in the respective way are hit.

$$\mathbb{E}(n_{accesses}) = n_{lines} \cdot (H_{n_{lines}} - H_{n_{lines}-h_{hit}})$$

Hereby, H_n denotes the n-th Harmonic number, which can be approximated using the natural logarithm. This approximation allows to determine the number of cache lines n_{hit} that are expected to be hit in a certain cache way when $n_{accesses}$ random accesses to the specific way are performed.

$$\mathbb{E}(n_{hit}) = n_{lines} \cdot (1 - e^{-\frac{n_{accesses}}{n_{lines}}}) \tag{1}$$

Using n_{hit}, we can estimate the number of independent accesses to be performed to a specific cache way such that a portion β of the respective cache way is evicted.

$$\mathbb{E}(n_{accesses}) = -n_{lines} \cdot \ln(1-\beta)$$

For the same 8-way SCATTERCACHE with 2^{11} lines per way as before, we therefore require roughly 2^{16} independent accesses to evict $\beta = 99\,\%$ of the cache.

Profiled Eviction for PRIME+PROBE. As shown, relying on random eviction to perform cache-based attacks involves significant effort and yields an overapproximation of the eviction set. Moreover, while random eviction is suitable for attacks such as EVICT+RELOAD, in PRIME+PROBE settings random eviction fails to provide information related to the concrete memory location that is being used by a victim. To overcome these issues, attackers may profile a system to construct eviction sets for specific memory addresses of the victim, *i.e.*, they try to find a set of addresses that map to cache sets that partially overlap with the cache set corresponding to the victim address. Eventually, such sets could be used to speed up eviction and to detect accesses to specific memory locations. In the following, we analyze the complexity of finding these eviction sets. In more detail, we perform analysis w.r.t. eviction addresses whose cache sets overlap with the cache set of a victim address in a single cache way only.

To construct a suitable eviction set for PRIME+PROBE, the attacker needs to provoke the victim process to perform the access of interest. In particular, the attacker tests a candidate address for cache-set collisions by accessing it (prime), waiting for the victim to access the memory location of interest, and then measuring the time when accessing the candidate address again (probe). In such a profiling procedure, after the first attempt, we have to assume that the cache line belonging to the victim access already resides in the cache. As a result, attackers need to evict a victim's cache line in their prime step. Hereby, hitting the right cache way and index have probability ${n_{ways}}^{-1}$ and $2^{-b_{indices}}$, respectively. To be able to detect a collision during the probe step, the victim access must then fall into the same cache way as the candidate address, which has a chance of ${n_{ways}}^{-1}$. In total, the expected number of memory accesses required to construct an eviction set of t colliding addresses hence is

$$\mathbb{E}(n_{accesses}) = {n_{ways}}^2 \cdot 2^{b_{indices}} \cdot t.$$

The number of memory addresses t needs to be chosen according to the desired eviction probability for the victim address with the given set. When the eviction set consists of addresses that collide in the cache with the victim in exactly one way each, the probability of evicting the victim with an eviction set of size t is

$$p(\text{Eviction}) = 1 - \left(1 - \frac{1}{n_{ways}}\right)^{\frac{t}{n_{ways}}}.$$

Figure 5 depicts this probability for the size of the eviction set and different numbers of cache ways. For an 8-way SCATTERCACHE with 2^{11} cache lines per way, roughly 275 addresses with single-way cache collisions are needed to evict the respective cache set with 99 % probability. Constructing this eviction set, in the best case, requires profiling of approximately $8^2 \cdot 2^{11} \cdot 275 \approx 2^{25}$ (33.5 million) victim accesses. Figure 6 shows the respective number of PRIME+PROBE experiments needed to generate sets with 99 % eviction probability for different cache configurations. We were able to empirically confirm these numbers within a noise-free standalone simulation of SCATTERCACHE.

For comparison, to generate an eviction set on a commodity cache, e.g., recent Intel processors, for a specific victim memory access, an attacker needs fewer than 103 observations of that access in a completely noise-free attacker-controlled scenario. Hence, our cache increases the complexity for the attacker by factor 325 000. In a real-world scenario the complexity is even higher.

Profiled Eviction for EVICT+RELOAD. For shared memory, such as in EVICT+RELOAD, the construction of eviction sets, however, becomes easier, as shared memory allows the attacker to simply access the victim address. Hence, to build a suitable eviction set, the attacker first primes the victim address, then accesses a candidate address, and finally probes the victim address. In case a specific candidate address collides with the victim address in the cache way the victim access falls into , the attacker can observe this collision with probability $p = {n_{ways}}^{-1}$. As a result, the expected number of memory accesses required to build an eviction set of t colliding addresses for EVICT+RELOAD is

$$\mathbb{E}(n_{accesses}) = n_{ways} \cdot 2^{b_{indices}} \cdot t.$$

For an 8-way SCATTERCACHE with 2^{11} lines per way, constructing an EVICT+RELOAD eviction set of 275 addresses (*i.e.*, 99 % eviction probability) requires profiling with roughly $8 \cdot 2^{11} \cdot 275 = 2^{22}$ memory addresses. Note, however, that EVICT+RELOAD only applies to writable shared memory as used for Inter Process Communication (IPC), whereas SCATTERCACHE effectively prevents EVICT+RELOAD on shared read-only memory by using different cache-set compositions in each security domain. Moreover, eviction sets for both PRIME+PROBE and EVICT+RELOAD must be freshly created whenever the key or the SDID changes.

4.4 Complexity of PRIME+PROBE

As demonstrated, SCATTERCACHE strongly increases the complexity of building the necessary sets of addresses for PRIME+PROBE. However, the actual attacks utilizing these sets are also made more complex by SCATTERCACHE.

In this section, we make the strong assumption that an attacker has successfully profiled the victim process such that they have found addresses which collide with the victim's target addresses in exactly 1 way each, have no collisions with each other outside of these and are sorted into subsets corresponding to the cache line they collide in.

Where in normal PRIME+PROBE an attacker can infer victim accesses (or a lack thereof) with near certainty after only 1 sequence of priming and probing, SCATTERCACHE degrades this into a probabilistic process. At best, one PRIME+PROBE operation on a target address can detect an access with a probability of ${n_{ways}}^{-1}$. This is complicated further by the fact that any one set of addresses is essentially single-use, as the addresses will be cached in a non-colliding cache line with a probability of $1 - {n_{ways}}^{-1}$ after only 1 access, where they cannot be used to detect victim accesses anymore until they themselves are evicted again.

Given the profiled address sets, we can construct general probabilistic variants of the PRIME+PROBE attack. While other methods are possible, we believe the 2 described in the following represent lower bounds for either victim accesses or memory requirement.

Variant 1: Single collision with eviction. We partition our set of addresses, such that one PRIME+PROBE set consists of n_{ways} addresses, where each collides with a different way of the target address. To detect an access to the target, we prime with one set, cause a target access, measure the primed set and then evict the target address. We repeat this process until the desired detection probability is reached. This probability is given by $p(n_{accesses}) = 1 - (1 - {n_{ways}}^{-1})^{n_{accesses}}$. The eviction of the target address can be achieved by either evicting the entire cache or using preconstructed eviction sets (see Section 4.3.2). After the use of an eviction set, a different priming set is necessary, as the eviction sets only target the victim address. After a full cache flush, all sets can be reused. The amount of colliding addresses we need to find during profiling depends on how often a full cache flush is performed. This method requires the least amount of accesses to the target, at the cost of either execution time (full cache flushes) or memory and profiling time (constructing many eviction sets).

Variant 2: Single collision without eviction. Using the same method but without the eviction step, the detection probability can be recursively calculated as

$$p(n_{acc.}) = p(n_{acc.} - 1) + (1 - p(n_{acc.} - 1))(\frac{2 \cdot n_{ways} - 1}{{n_{ways}}^3})$$

with $p(1) = {n_{ways}}^{-1}$. This variant provides decreasing benefits for additional accesses. The reason for this is that the probability that the last step evicted the target address influences the probability to detect an access in the current step. While this approach requires many more target accesses, it has the advantage of a shorter profiling phase.

These two methods require different amounts of memory, profiling time and accesses to the target, but they can also be combined to tailor the attack to the target. Which is most useful depends on the attack scenario, but it is clear that both come with considerable drawbacks when compared to PRIME+PROBE in current caches. For example, achieving a 99 % detection probability in a 2 MB Cache with 8 ways requires 35 target accesses and 9870 profiled addresses in 308 MB of memory for variant 1 if we use an eviction set for every probe step. Variant 2 would require 152 target accesses and 1216 addresses in 38 MB of memory. In contrast, regular PRIME+PROBE requires 1 target access and 8 addresses while providing 100 % accuracy (in this ideal scenario). Detecting non-repeating events is made essentially impossible; to measure any access with confidence requires either the knowledge that the victim process repeats the same access pattern for long periods of time or control of the victim in a way that

allows for repeated measurements. In addition to the large memory requirements, variant 1 also heavily degrades the temporal resolution of a classical PRIME+PROBE attack because of the necessary eviction steps. This makes trace-based attacks like attacks on square-and-multiply in RSA [76] much less practical. Variant 2 does not suffer from this drawback, but requires one PRIME+PROBE set for each time step, for as many high-resolution samples as one trace needs to contain. This can quickly lead to an explosion in required memory when thousands of samples are needed.

4.5 Challenges with Real-World Attacks

We failed at mounting a real-world attack (*i.e.*, with even the slightest amounts of noise) on SCATTERCACHE. Generally, for a PRIME+PROBE attack we need to (1) generate an eviction set (cf. Section 4.3), and (2) use the eviction set to monitor a victim memory access. If we assume step 1 to be solved, we can mount a cache attack (*i.e.*, step 2) with a complexity increases by a factor of 152 (cf. Section 4.4). For some real-world attacks this would not be a problem, in particular if a small fast algorithm is attacked, e.g., AES with T-tables. Gülmezoglu et al. [29] recovered the full AES key from an AES T-tables implementation with only 30 000 encryptions in a fully synchronized setting (that can be implemented with PRIME+PROBE as well [26]), taking 15 seconds, *i.e.*, 500 µs per encryption. The same attack on SCATTERCACHE takes $4.56 \cdot 10^6$ encryptions, *i.e.*, 38 minutes assuming the same execution times, which is clearly viable.

However, the real challenge is solving step 1, which we did not manage for any real-world example. In particular, even if AES would only perform a single attacker-chosen memory access (instead of 160 to the T-tables alone, plus additional code and data accesses), which would be ideal for the attacker in the profiling during step 1, we would need to observe 33.5 million encryptions. In addition to the runtime reported by Gülmezoglu et al. [29] we also need a full cache flush after each attack round (*i.e.*, each encryption). For a 2 MB cache, we need to iterate over a 6 MB array to have a high probability of covering all cache lines. The time for an L3-cache access is e.g., for Kaby Lake 9.5 ns [2]. The absolute minimum number of cache misses here is 65536 (=4 MB), but in practice it will be much higher. A cache miss takes around 50 ns, hence, the full cache eviction will take at least 3.6 ms. Consequently, with 33.5 million tests required to generate the eviction set and a runtime of 4.1 ms per test, the total runtime to generate the eviction set is 38 hours.

This number still only considers the theoretical setting of a completely noise-free and idle system. The process doing AES computations must not be restarted during these 38 hours. The operating system must not replace any physical pages and, most importantly, our hypothetical AES implementation only performs a single memory access. In any realistic setting with only the slightest amount of activity (noise) on the system, this easily explodes to multiple weeks or months. With a second memory access, these two memory accesses can already not be distinguished anymore with the generated eviction set, because the eviction set is generated for an invocation of the entire victim computation, not for an address.

4.6 Noise Sampling

The previous analysis considered a completely noise-free scenario, where the attacker performs PRIME+PROBE on a single memory access executed by the victim. However, in a real system, an attacker will typically not be able to perform an attack on single memory accesses, but face different kinds of noise. Namely, on real systems cache attacks will suffer from both systematic and random noise, which reduces the effectiveness of profiling and the actual attack.

Systematic noise is introduced, for example, by the victim as it executes longer code sequences in between the attacker's prime and probe steps. The victim's code execution intrinsically performs additional memory accesses to fetch code and data that the attacker will observe in the cache deterministically. In SCATTERCACHE, the mappings of memory addresses to cache lines is unknown. Hence, without additional knowledge, the attacker is unable to distinguish the cache collision belonging to the target memory access from collisions due to systematic noise. Instead, the attacker can only observe and learn both simultaneously. As a result, larger eviction sets need to be constructed to yield the same confidence level for eviction. Specifically, the size of an eviction set must increase proportionally to the number of systematic noise accesses to achieve the same properties. While this significantly increases an attackers profiling effort, they may be able to use clustering techniques to prune the eviction set prior to performing an actual attack.

Random noise, on the other hand, stems from arbitrary processes accessing the cache simultaneously or as they are scheduled in between. Random noise hence causes random cache collisions to be detected by an attacker during both profiling and an actual attack, *i.e.*, produces false positives. While attackers cannot distinguish between such random noise and systematic accesses in a single observation, these random noise accesses can be filtered out statistically be repeating the same experiment multiple times. Yet, it increases an attackers effort significantly. For instance, when building eviction sets an attacker can try to observe the same cache collision multiple times for a specific candidate address to be certain about its cache collision with the victim.

Random noise distributes in SCATTERCACHE according to Equation 1 and hence quickly occupies large parts of the cache. As a result, there is a high chance of sampling random noise when checking a candidate address during the construction of eviction sets. Also when probing addresses of an eviction set in an actual attack, random noise is likely to be sampled as attacks on SCATTERCACHE demand for large

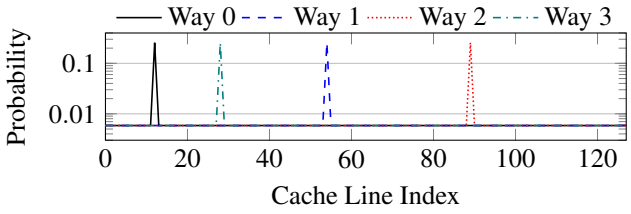

Figure 7: Example distribution of cache indices of addresses in profiled eviction sets ($n_{ways} = 4$, $b_{indices} = 7$).

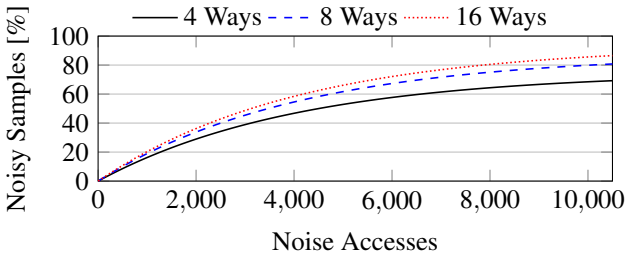

Figure 8: Expected percentage of noisy samples in an eviction set for a cache consisting of 2^{12} cache lines.

eviction sets. As our analysis shows, for a single cache way the distribution of cache line indices corresponding to the memory accesses of profiled eviction sets (cf. Section 4.3) adheres to Figure 7. Clearly, due to profiling there is a high chance of roughly $1/n_{ways}$ to access the index that collides with the victim address. However, with $p = (n_{ways} - 1)/n_{ways}$ the index adheres to an uniformly random selection from all possible indices and hence provides a large surface for sampling random noise. Consequently, for a cache with $n_{lines} = 2^{b_{indices}}$ lines per way and n_{noise} lines being occupied by noise in each way, the probability of sampling random noise when probing an eviction set address is

$$p(\text{Noise}) \approx \frac{n_{ways} - 1}{n_{ways}} \frac{n_{noise}}{n_{lines}}.$$

Figure 8 visualizes this effect and in particular the percentage of noisy samples encountered in an eviction set for different cache configurations and noise levels. While higher random noise clearly increases an attackers effort, the actual noise level strongly depends on the system configuration and load.

4.7 Further Remarks

In the previous analysis, the SDIDs of both attacker and victim were assumed to be constant throughout all experiments for statistical analysis to be applicable. Additionally, systematic and random noise introduced during both profiling and attack further increase the complexity of actual attacks, rendering attacks on most real-world systems impractical.

Also note that the security analysis in this section focuses on SCv1. In a noise-free scenario, SCv2 may allow to construct eviction sets slightly more efficiently since its IDF is a permutation. This means that, once a collision in a certain cache way is found, there will not be any other colliding address for that cache way in the same index range, *i.e.*, for the same address tag. Considering the expected time to find the single collision in a given index range, this could give an attacker a benefit of up to a factor of two in constructing eviction sets. However, in practice multiple cache ways are profiled simultaneously, which results in a high chance of finding a collision in any of the cache ways independent of the address index bits, *i.e.*, the n_{ways} indices for a certain memory address will very likely be scattered over the whole index range. Independent of that, the presence of noise significantly hampers taking advantage of the permuting property of SCv2.

5 Performance Evaluation

SCATTERCACHE significantly increases the effort of attackers to perform cache-based attacks. However, a countermeasure must not degrade performance to be practical as well. This section hence analyzes the performance of SCATTERCACHE using the gem5 full system simulator and GAP [9], MiBench [30], lmbench [49], and the C version of scimark2 [1] as micro benchmarks. Additionally, to closer investigate the impact of SCATTERCACHE on larger workloads, a custom cache simulator is used for SPEC CPU 2017 benchmarks. Our evaluations indicate that, in terms of performance, SCATTERCACHE behaves basically identical to traditional set-associative caches with the same random replacement policy.

5.1 gem5 Setup

We performed our cache evaluation using the gem5 full system simulator [12] in 32-bit ARM mode. In particular, we used the CPU model TimingSimpleCPU together with a cache architecture such as commonly used in ARM Cortex-A9 CPUs: the cache line size was chosen to be 32 bytes, the 4-way L1 data and instruction caches are each sized 32 kB, and the 8-way L2 cache is 512 kB large. We adapted the gem5 simulator such as to support SCATTERCACHE for the L2 cache. This allows to evaluate the impact of six different cache organizations. Besides SCATTERCACHE in both variants (1) SCv1 and (2) SCv2 and standard set-associative caches with (3) LRU, (4) BIP, and (5) random replacement, we also evaluated (6) skewed associative caches [63] with random replacement as we expect them to have similar performance characteristics as SCv1 and SCv2.

On the software side, we used the Poky Linux distribution from Yocto 2.5 (Sumo) with kernel version 4.14.67 after applying patches to run within gem5. We then evaluated the performance of our micro benchmarks running on top of Linux. In particular, we analyzed the cache statistics provided by

[1] https://math.nist.gov/scimark2/

gem5 after booting Linux and running the respective benchmark. Using this approach, we reliably measure the cache performance and execution time for each single application, *i.e.*, without concurrent processes. Since only the L2-cache architecture (*i.e.*, replacement policy, skewed vs. fixed sets) changed between the individual simulation runs, execution performance is simply direct proportional to the resulting cache hit rate. To enable easier comparison between the individual benchmarks as well as with related work we therefore mainly report L2-cache hit results.

ScatterCache IDF Instantiations. Both ScatterCache variants have been instantiated using the low-latency tweakable block cipher QARMA-64 [8]. In particular, in the SCv1 variant, the index bits for the individual cache ways have been sliced from the ciphertext of encrypting the cache line address under the secret key and SDID. On the other hand, due to the lack of an off-the-shelf tweakable block cipher with the correct block size, a stream cipher construction was used in the SCv2 variant. Namely, the index is computed as the XOR between the original index bits and the ciphertext of the original tag encrypted using QARMA-64. Note, however, that, although this construction for SCv2 is a proper permutation and entirely sufficient for evaluating the performance of SCv2, we do not recommend the construction as pads are being reused for addresses having the same tag bits.

While the majority of the following results are latency agnostic LLC hit rates, all following results are reported for the zero cycle latency case. For QARMA-64 with 5 rounds, ASIC implementation results with as little as 2.2 ns latency have been reported [8]. We are therefore confident that, if desired, hiding the latency of the IDF by computing it in parallel to the lower level cache lookup is feasible.

However, we still also conducted simulations with latency overheads between 1 and 5 cycles by increasing the `tag_latency` of the cache in gem5. The acquired results show that, even for IDFs which introduce 5 cycles of latency, less than 2 % performance penalty are encountered on the GAP benchmark suite. These numbers are also in line with Qureshi's results reported for CEASER [55].

5.2 Hardware Overhead Discussion

ScatterCache is designed to be as similar to modern cache architectures as possible in terms of hardware. Still, area and power overheads have to be expected due to the introduction of the IDF and the additional addressing logic. Unfortunately, while probably easy for large processor and SoC vendors, determining reliable overhead numbers for these two metrics is a difficult task for academia that requires an actual ASIC implementation of the cache. To the best of our knowledge, even in the quite active RISC-V community, no open and properly working LLC designs are available that can be used as foundation. Furthermore, for merely simulating such a design with a reasonably large cache, commercial EDA tools, access to state-of-the-art technology libraries, and large memory macros with power models are required. As the result, secure cache designs typically fail to deliver hardware implementation results (see Table 6 in [18]).

Because of these problems, similar to related work, we can also not provide concrete numbers for the area and power overhead. However, due to the way we designed ScatterCache and the use of lightweight cryptographic primitives, we can assert that the hardware overhead is reasonable. For example, the 8-way SCv1 ScatterCache with 512 kB that is simulated in the following section, uses two parallel instances of QARMA-64 with 5 rounds as IDF. One fully unrolled instance has a size of 22.6 kGE [8] resulting in an IDF size of less then 50 kGE even in case additional pipeline registers are added. The added latency of such an IDF is the same as the latency of the used primitive which has been reported as 2.2 ns. However, this latency can (partially or fully) be hidden by computing the IDF in parallel to the lower level cache lookup. Interestingly, with similar size, also a sponge-based SCv1 IDF (e.g., 12 rounds of Keccak[200] [11]) can be instantiated. Finally, there is always the option to develop custom IDF primitives [55] that demand even less resources.

For comparison, in the BROOM chip [16], the SRAM macros in the 1 MB L2 cache already consume roughly 50 % of the 4.86 mm^2 chip area. Assuming an utilization of 75 % and a raw gate density of merely 3 $MGate/mm^2$ [21] for the used 28 nm TSMC process, these 2.43 mm^2 already correspond to 5.5 MGE. Subsequently, even strong IDFs are orders of magnitude smaller than the size of a modern LLC.

In terms of overhead for the individual addressing of the cache ways, information is more sparse. Spjuth et al. [64] observed a 17 % energy consumption overhead for a 2-way skewed cache. They also report that skewed caches can be built with lower associativity and still reach similar performance as traditional fixed set-associative caches. Furthermore, modern Intel architectures already feature multiple addressing circuits in their LLC as they partition it into multiple smaller caches (*i.e.*, cache slices).

5.3 gem5 Results and Discussion

Figure 9 visualizes the cache hit rate of our L2 cache when executing programs from the GAP benchmark suite. To ease visualization, the results are plotted in percentage points (pp), *i.e.*, the differences between percentage numbers, using the fixed set-associative cache with random replacement policy as baseline. All six algorithms (*i.e.*, `bc`, `bfs`, `cc`, `pr`, `sssp`, `tc`) have been evaluated. Moreover, as trace sets, both synthetically generated `kron` (`-g16 -k16`) and `urand` (`-u16 -k16`) sets have been used. As can be seen in the graph, the BIP and LRU replacement policies outperform random replacement on average by 4.6 pp and 4 pp respectively. Interestingly, however, all random replacement based schemes, including the skewed variants, perform basically identical.

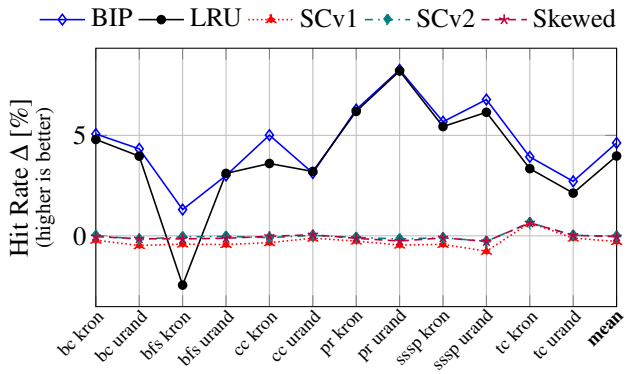

Figure 9: Cache hit rate, simulated with gem5, for the synthetic workloads in the GAP benchmark suite with random replacement policy as baseline.

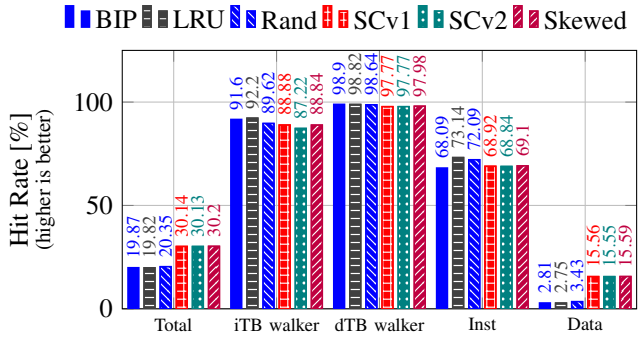

Figure 10: Cache hit rate, simulated with gem5, for scimark2.

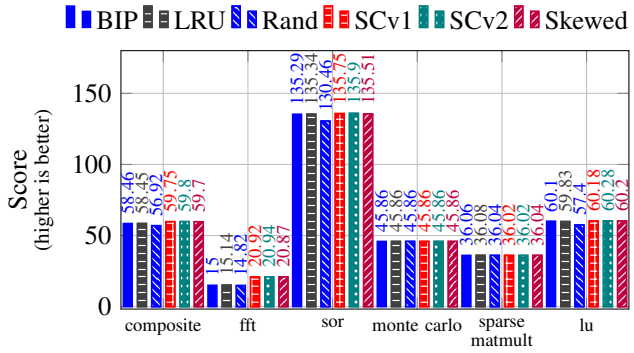

Figure 11: Scimark2 score simulated with gem5.

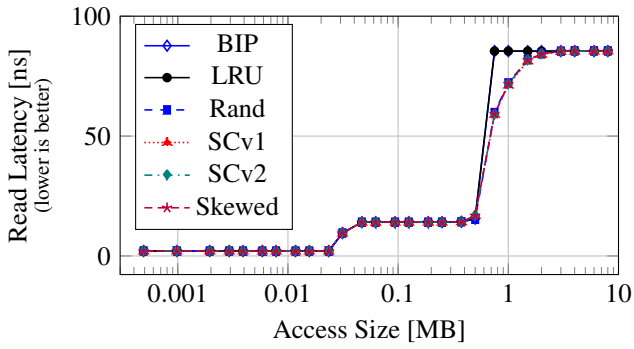

Figure 12: Memory read latency, simulated with gem5, with 32 byte stride (*i.e.*, one access per cache line).

The next benchmark, we visualized in Figure 10, is scimark2 (`-large 0.5`). This benchmark shows an interesting advantage of the skewed cache architectures over the fixed-set architectures, independent of the replacement policy, of approximately 10 pp for the total hit rate. This difference is mainly caused by the 5x difference in hit rate for data accesses. Comparing the achieved benchmark scores in Figure 11 further reveals that the `fft` test within scimark2 is the reason for the observed discrepancy in cache performance.

To investigate this effect in more detail, we measured the memory read latency using using `lat_mem_rd 8M 32` from lmbench in all cache configurations. The respective results in Figure 12 feature two general steps in the read latency at 32 kB (L1-cache size) and at 512 kB (L2-cache size). Notably, configurations with random replacement policy feature a smoother transition at the second step, *i.e.*, when accesses start to hit main memory instead of the L2 cache.

Even more intersting results, as shown in Figure 13, have been acquired by increasing the stride size to four times the cache line size. Skewed caches like ScatterCache break the strong alignment of addresses and cache set indices. As a consequence, a sparse, but strongly aligned memory access pattern such as in `lat_mem_rd`, which in a standard set-associative caches only uses every 4$^{\text{th}}$ cache index, gives high cache hit rates and low read latencies for larger memory ranges due to less cache conflicts. This effect becomes visible in Figure 13 as shift of the second step from 512 kB to 2 MB for the skewed cache variants.

Finally, as last benchmark, MiBench has been evaluated in small and large configuration. The individual results are visualized in Figure 14 and Figure 15 respectively. On average, the achieved performance results in MiBench are very similar to the results from the GAP benchmark suite. Again, caches with BIP and LRU replacement policy outperform the configurations with random replacement policy by a few percent. However, in some individual benchmarks (e.g., qsort in small, `jpeg` in large), skewed cache architectures like ScatterCache outmatch the fixed set appraoches.

In summary, our evaluations with gem5 in full system simulation mode show that the performance of ScatterCache, in terms of hit rate, is basically identical to contemporary fixed set-associative caches with random replacement policy. Considering that we employ the same replacement strategy, this is an absolutely satifying result by itself. Moreover, no tests indicated any notable performance degradation and in some tests ScatterCache even outperformed BIP and LRU replacement policies.

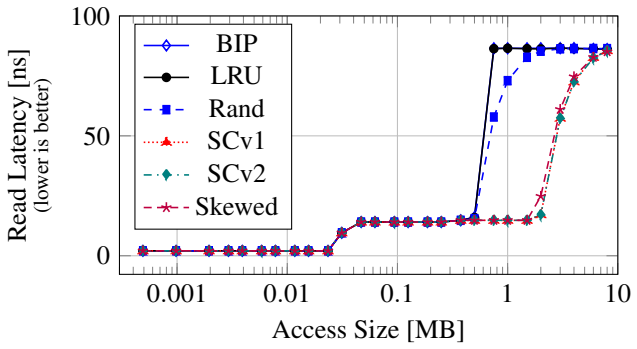

Figure 13: Memory read latency, simulated with gem5, with 128 byte stride (*i.e.*, one access in every fourth cache line).

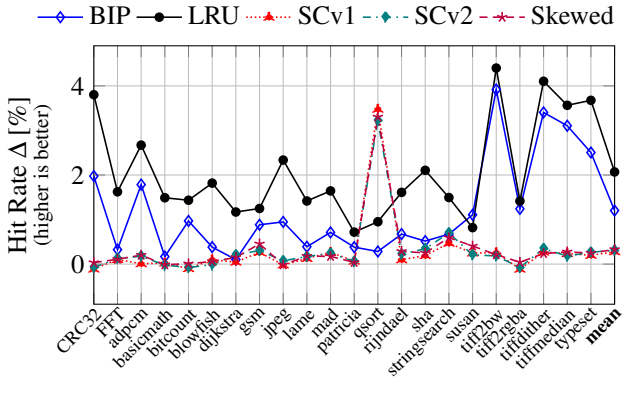

Figure 14: Cache hit rate, simulated with gem5, for MiBench in small configuration compared to random replacement.

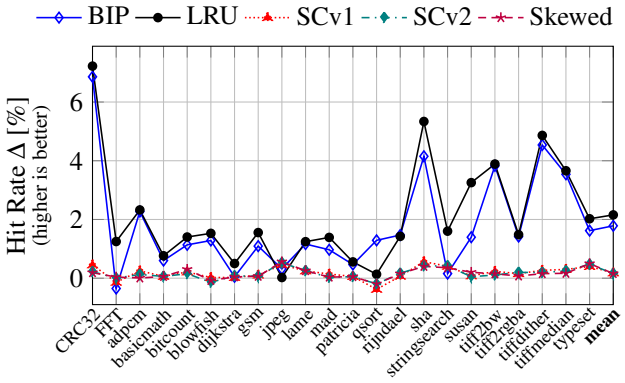

Figure 15: Cache hit rate, simulated with gem5, for MiBench in large configuration compared to random replacement.

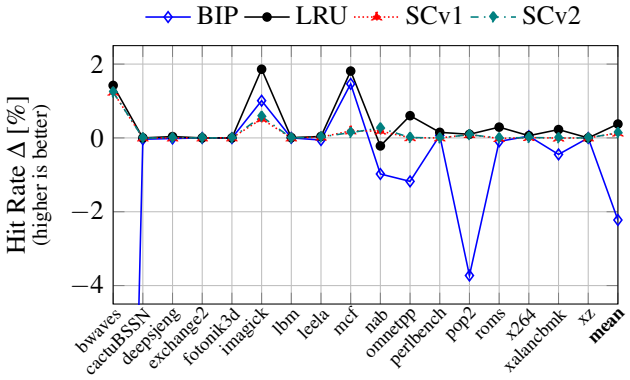

Figure 16: Average cache hit rate for SPEC CPU 2017 benchmarks compared to random replacement over 10 runs.

5.4 Cache Simulation and SPEC Results

Lastly, we evaluated the performance of SCATTERCACHE using the SPEC CPU 2017 [66] benchmark with both the "SPECspeed 2017 Integer" and "SPECspeed 2017 Floating Point" suites. We performed all benchmarks in these suites with the exception of `gcc`, `wrf` and `cam4`, as these failed to compile on our system. Because these benchmarks are too large to be run in full system simulation, we created a software cache simulator, capable of simulating different cache models and replacement policies. Even so, the benchmarks proved to be too large to run in full, so we opted to run segments of 250 million instructions from each, following the methodology of Qureshi et al. [56]. We made an effort to select parts of the benchmarks that are representative of their respective core workloads. To be able to run the benchmarks with our simulator, we recorded a trace of all instruction addresses and memory accesses with the Intel PIN Tool [33]. We then replayed this access stream for different cache configurations. The simulator implements the set-associative replacement policies Pseudo-LRU (Tree-PLRU), LRU (ideal), BIP as described in [56], and random replacement, as well as the two SCATTERCACHE variants. The number of ways per set, total cache size, number of slices, and cache line size are fully configurable. Additionally, the simulator supports multiple levels of inclusive caches, as well as a cache that is split for data and instructions. All simulations were run on an inclusive two level cache, where the L1 was separated into instruction and data caches, both of which use LRU replacement. Figure 16 shows results for the cache configuration, as described in Section 5.1, as the difference in percentage points for last-level hit rates when compared to random replacement. While we can see large differences in individual tests, the mean shows that both versions of SCATTERCACHE perform at least as well as random replacement and very similar to LRU. Using the same cache configuration but with 64 B cache lines, we actually observe a mean advantage of 0.23 ± 0.76 pp of SCATTERCACHE over random replacement, where LRU sees a marginally worse result of -0.21 ± 1.02 pp. On a larger configuration with 64 B cache lines, 32 kB 8-way L1 and 2 MB 16-way LLC, the results show a slim improvement of 0.035 ± 0.10 pp for SCATTERCACHE and 0.37 ± 1.14 pp for LRU over random replacement.

6 Conclusion

In this paper, we presented SCATTERCACHE, a novel cache design to eliminate cache attacks that eliminates fixed cache-set congruences and, thus, makes eviction-based cache attacks unpractical. We showed how skewed associative caches when retrofitted with a keyed mapping function increase the attack complexity so far that it exceeds practical scenarios Furthermore, high-frequency attacks become infeasible. Our evaluations show that the runtime performance of software is not curtailed and SCATTERCACHE can even outperform state-of-the-art caches for certain realistic workloads.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Yossi Oren, for their comments and suggestions that substantially helped in improving the paper. This project has received funding from the European Research Council (ERC) under Horizon 2020 grant agreement No 681402. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] 7-cpu. ARM Cortex-A57. www.7-cpu.com/cpu/Cortex-A57.html.

[2] 7-cpu. Intel Skylake. www.7-cpu.com/cpu/Skylake.html.

[3] O. Aciiçmez, B. B. Brumley, and P. Grabher. New Results on Instruction Cache Attacks. In *CHES*, 2010.

[4] G. I. Apecechea, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *S&P*, 2015.

[5] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID*, 2014.

[6] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Lucky 13 Strikes Back. In *CCS*, 2015.

[7] V. Arribas, B. Bilgin, G. Petrides, S. Nikova, and V. Rijmen. Rhythmic Keccak: SCA Security and Low Latency in HW. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018.

[8] R. 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 Trans. Symmetric Cryptol.*, 2017.

[9] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP Benchmark Suite. *arXiv abs/1508.03619*, 2015.

[10] D. J. Bernstein. Cache-Timing Attacks on AES. Technical report, University of Illinois at Chicago, 2005.

[11] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. Keccak implementation overview, 2012.

[12] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comp. Arch. News*, 39, 2011.

[13] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *ASIACRYPT*, 2012.

[14] B. B. Brumley and R. M. Hakala. Cache-Timing Template Attacks. In *ASIACRYPT*, 2009.

[15] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[16] C. Celio, P. Chiu, K. Asanovic, B. Nikolic, and D. A. Patterson. BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS. *MICRO*, 39, 2019.

[17] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P*, 2009.

[18] S. Deng, W. Xiong, and J. Szefer. Analysis of Secure Caches and Timing-Based Side-Channel Attacks. *ePrint 2019/167*.

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

[20] G. Doychev and B. Köpf. Rigorous analysis of software countermeasures against cache attacks. In *PLDI*, 2017.

[21] Europractice. TSMC Standard cell libraries. http://www.europractice-ic.com/libraries_TSMC.php.

[22] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. M. Austin. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *ASPLOS*, 2019.

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

[24] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

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

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

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

[28] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*, 2011.

[29] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In *COSADE*, 2015.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

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

[32] M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*, 2016.

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

[34] G. Irazoqui, K. Cong, X. Guo, H. Khattri, A. K. Kanuparthi, T. Eisenbarth, and B. Sunar. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *arXiv abs/1709.01552*, 2017.

[35] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross Processor Cache Attacks. In *CCS*, 2016.

[36] Y. Jang, S. Lee, and T. Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[37] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES*, 2009.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[39] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[40] R. Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In *CT-RSA*, 2008.

[41] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic Quantification of Cache Side-Channels. In *CAV*, 2012.

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

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

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

[45] H. Mantel, A. Weber, and B. Köpf. A Systematic Study of Cache Side Channels Across AES Implementations. In *ESSoS*, 2017.

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

[47] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*, 2015.

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

[49] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[50] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, 2015.

[51] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[52] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *ePrint 2002/169*.

[53] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *ePrint 2005/280*.

[54] C. Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

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

[56] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[57] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

[58] C. Rebeiro, A. D. Selvakumar, and A. S. L. Devi. Bitslice Implementation of AES. In *CANS*, 2006.

[59] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[60] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*, 2018.

[61] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. NetSpectre: Read Arbitrary Memory over Network. *arXiv abs/1807.10535*, 2018.

[62] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[63] A. Seznec. A Case for Two-Way Skewed-Associative Caches. In *ISCA*, 1993.

[64] M. Spjuth, M. Karlsson, and E. Hagersten. Skewed caches from a low-power perspective. In *Computing Frontiers – CF*, 2005.

[65] R. Spreitzer and T. Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *COSADE*, 2013.

[66] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[67] D. Trilla, C. Hernández, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *DAC*, 2018.

[68] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In *CHES*, 2003.

[69] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.

[70] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *MICRO*, 2008.

[71] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security*, 2018.

[72] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*, 2012.

[73] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud. *IEEE/ACM Trans. Netw.*, 23, 2015.

[74] Y. Xiao, M. Li, S. Chen, and Y. Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS*, 2017.

[75] Y. Xu, M. Bailey, F. Jahanian, K. R. Joshi, M. A. Hiltunen, and R. D. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *CCSW*, 2011.

[76] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[77] A. Zankl, J. Heyszl, and G. Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *CARDIS*, 2016.

[78] X. Zhang, Y. Xiao, and Y. Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *CCS*, 2016.

[79] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P*, 2011.

[80] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[81] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*, 2014.