Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks

Thomas Unterluggauer*, Austin Harris†, Scott Constable*, Fangfei Liu*, Carlos Rozas*

*Intel Corporation *first.last@intel.com*
†UT Austin *austinharris@utexas.edu*

Abstract—Randomized, skewed caches (RSCs) such as CEASER-S have recently received much attention to defend against contention-based cache side channels. By randomizing and regularly changing the mapping(s) of addresses to cache sets, these techniques are designed to obfuscate the leakage of memory access patterns. However, new attack techniques, e.g., Prime+Prune+Probe, soon demonstrated the limits of RSCs as they allow attackers to more quickly learn which addresses contend in the cache and use this information to circumvent the randomization. To yet maintain side-channel resilience, RSCs must change the random mapping(s) more frequently with adverse effects on performance and implementation complexity.

This work aims to make randomization-based approaches more robust to allow for reduced re-keying rates and presents Chameleon Cache. Chameleon Cache extends RSCs with a victim cache (VC) to decouple contention in the RSC from evictions observed by the user. The VC allows Chameleon Cache to make additional use of the multiple mappings RSCs provide to translate addresses to cache set indices: when a cache line is evicted from the RSC to the VC under one of its mappings, the VC automatically reinserts this evicted line back into the RSC by using a different mapping. As a result, the effects of previous RSC set contention are hidden and Chameleon Cache exhibits side-channel resistance and eviction patterns similar to fully associative caches with random replacement. We show that Chameleon Cache has performance overheads of $< 1\%$ and stress that VCs are more generically helpful to increase side-channel resistance and re-keying intervals of randomized caches.

I. Introduction

Cache side channels have been intensively studied over the past two decades as these allow to circumvent architectural isolation boundaries and reveal sensitive information being processed by applications running on the same system. Over time, the scope of cache side channels has expanded from cryptographic targets [5, 7, 11, 14] to other domains such as AI [31] and the more recent transient execution attacks (e.g., Spectre[9], Meltdown [10]) and thus sparked interest in potential mitigations.

Fundamentally, cache side channels originate from the intrinsic timing difference between cache hits and misses. Attackers can use this timing difference to infer memory access patterns in contention-based cache attacks [14, 23] (e.g., Prime+Probe) by exploiting the limited size of cache (sets), or in shared-memory based cache attacks [6, 32] (e.g., Flush+Reload) by manipulating and learning the cache state of a cache line shared with a victim application. While software strives to mitigate shared-memory based cache attacks (e.g., by disabling memory deduplication and static linking of libraries) generically mitigating contention-based attacks remains difficult for software.

Two main approaches strive to prevent contention-based attacks in hardware: Partition-based approaches [3, 26, 27] split the cache into two or more partitions and allow each partition to be used by a specific security domain only. Cache partitioning does not allow any leakage to occur between different partitions and hence provides relatively strong security, but is difficult to scale for large numbers of security domains and involves software to manage the partitions. Randomization-based approaches, on the other hand, are transparent to software and obfuscate cache side channel leakage rather than prevent it completely to allow for more efficient cache utilization.

Among the randomization-based approaches, cache-set randomization (e.g., CEASER [17], CEASER-S [18], Scatter-Cache [28], and PhantomCache [24]) has recently gained much attention. These proposals encrypt cache line addresses to randomize the mapping of addresses to cache sets and prevent attackers from inferring memory access patterns from cache-set contention. As attackers over time can learn the mapping from observing the cache behavior, the encryption key used by cache-set randomization needs to be regularly changed.

While cache-set randomization is a promising direction, its security is also largely dependent on the state-of-the-art of applicable attack strategies. New approaches to efficiently learn the secret address-to-set mapping [15, 16, 25] pointed out the requirement for higher re-key frequencies that hurt performance. In addition, new analysis techniques [2] have highlighted the possibility to even accumulate leakage across key epochs. Consequently, there is a desire to make cache-set randomization more robust.

A. Contribution

In this work, we improve randomization-based countermeasures and present Chameleon Cache to increase security and achieve practical re-key intervals. Similar to NewCache [12], we start with the observation that for non-partitioned caches,

Fully Associative (FA) caches with random replacement achieve the best side-channel resilience. Namely, FA caches with random replacement allow every cache line address to evict any other cache line irrespective of its address and past usage. Thus, FA caches with random replacement are meant to protect against fine-grained cache-set contention attacks, but the dynamic sharing of cache resources allows for more coarse-grained cache occupancy channels [23]. While the side-channel properties of FA caches are desirable, they are difficult to build within typical power and area constraints. With the design of Chameleon Cache, we aim to approximate the behavior of FA caches with random replacement to obtain their side-channel properties and to simultaneously keep Chameleon Cache's implementation practical.

We further observe that *Randomized Skewed Caches* (RSCs) like ScatterCache [28] and CEASER-S [18] use multiple address-to-set mappings, which effectively increase the likelihood of two addresses contending in the cache. To approximate a FA cache with random replacement, Chameleon Cache thus builds upon RSCs and extends it with the concept of a reinserting victim cache [8] (VC). The VC in Chameleon Cache decouples evictions being observed by an attacker from contention in the RSC: a line that is evicted from the RSC is moved to the VC and then automatically reinserted into the RSC using one of its alternative address-to-set mappings. As a line is moved from the RSC to the VC, lines may get evicted from the VC to memory, but these evictions are unrelated to the original contention in the RSC. Eventually, we demonstrate that Chameleon Cache shows eviction patterns that are similar to FA caches with random replacement, thus aiming to prevent fine-grained cache contention attacks, at $< 1\%$ performance overhead. While Chameleon Cache resembles a FA cache with random replacement to enjoy its security properties, we also stress that victim caches more generally are a convenient tool to improve the security of randomized caches as they help add noise to the attacker's observations without performance degradation.

The paper is organized as follows. Section II gives background about cache attacks and countermeasures. We present Chameleon Cache in Section III and evaluates its security and performance in Section IV and Section V, respectively. We generalize the victim cache idea in Section VI, compare with related work in Section VII, and finally conclude in Section VIII.

II. BACKGROUND

In this Section, we present background on cache attacks and state-of-the-art countermeasures.

A. Cache Attacks

Modern computing systems make extensive use of caches to bridge the performance gap between the CPU and memory. However, cache structures have also been shown to allow for information leakage in side-channel attacks. These cache side-channel attacks make use of the intrinsic timing difference observed depending on whether a memory request hits or misses in the cache. An attacker can use this information about cache hits and cache misses to understand whether a victim application has accessed a memory location. Cache side-channel attacks thus reveal memory access patterns that can be used to infer sensitive information such as user behavior [23] and cryptographic keys [11].

There are two main categories of cache side channels: (1) Contention-based channels make use of contention in the shared cache resource, which reveals information about the cache usage of other applications. E.g., Prime+Probe [14] uses fine-grained cache-set contention in set-associative caches to infer memory access patterns at a high frequency, and cache occupancy [23] attacks analyze coarse-grained contention that reveals how much cache other applications use. (2) Shared-memory based channels make use of cache lines being shared between two applications (e.g., via shared libraries or memory deduplication) and allows attackers to accurately determine whether a specific cache line has been accessed or not, e.g., via Flush+Reload [32].

B. Cache Attack Countermeasures

1) Software: While software can in principle mitigate shared-memory based channels by simple avoidance of shared memory (e.g., through disabling memory deduplication and static linking of libraries) contention-based channels are harder to mitigate from within software. For instance, in cache coloring [33] the operating system (OS) adjusts the virtual-to-physical mapping to map different portions of the cache to different security domains. While cache coloring achieves strong isolation similar to hardware-based cache partitioning, it is difficult to manage, hard to scale for many security domains, and has the undesirable side effect of the binding memory allocation of security domains to their cache allocations. As a result, hardware-based countermeasures seem preferable to mitigate contention-based channels.

2) Hardware: Hardware countermeasures can coarsely be categorized in partition-based and randomization-based countermeasures.

a) Cache Partitioning: Cache partitioning splits the cache into multiple partitions where each partition can be used by its assigned security domains only. For instance, non-monopolizable caches [3] constructs its partitions by assigning each security domain a distinct subset of its cache ways. An alternative approach to partitioning is cache line pinning [27], which provides a software interface to pin specific cache lines in the cache that other security domains can no longer evict. However, while way-based partitioning is difficult to scale for many partitions, cache line pinning requires individual software support and its extensive use can deprive other applications from cache resources.

b) Cache Randomization: While cache partitioning aims to completely stop side-channel leakage, randomization-based approaches allow side-channel leakage between different security domains to occur and obfuscate the side-channel signal to make its exploitation sufficiently hard. An example of an

ideal design is a fully associative cache with random replacement, which, in absence of shared cache lines, can only leak overall cache utilization. While the power demand for large fully associative caches is typically too high, NewCache [12] presents a more efficient implementation variant of a fully-associative cache that uses a two-step lookup process to trade off the properties of a fully-associative design with power and implementation cost. Building upon the state-of-the-art set-associative cache, RPCache [27] performs an indirect cache lookup via random permutation tables in order to randomize the address-to-cache mapping. While hiding the mapping between addresses and cache sets does not stop contention-based channels, it helps to prevent attackers from directly inferring fine-grained memory access patterns. However, implementing RPcache is challenging as it requires software to manage randomization tables for the indirect set lookup.

A different approach is cache-set randomization: CEASER [17] realizes efficient randomization by encrypting physical addresses before extracting the cache-set index and accessing standard set-associative caches. In addition, CEASER needs to regularly change the key to prevent attackers from learning address-to-set mappings and ensure long-term side channel resistance. The successor designs CEASER-S [18] and ScatterCache [28] introduce *Randomized Skewed Caches* (RSCs) that improve security by skewing [22] the cache by its divisions. Each division consists of a few cache ways and RSCs derive a different set index for each division by using a different encryption key. RSCs make it hard for attackers to find a minimal set of addresses that map to exactly the same cache locations as a victim address of interest (i.e., an eviction set), because the probability that two addresses collide in all cache divisions is very low, i.e., $p = s^{-d}$ for s cache sets and d cache divisions. To overcome this low probability, attackers can use more likely partial collisions in the RSC. Such partially conflicting addresses collide with the victim, e.g., in a single division only, but also have smaller probability to evict the victim address (or observe a victim access), i.e., d^{-2} if an address collides with the victim address in a single division. An alternative improvement over CEASER to RSCs is PhantomCache [24], which computes several randomized cache-set indices to look up multiple cache sets in parallel.

Previous designs based on cache-set randomization and RSCs have been shown susceptible to advanced attack strategies [16], such as the Group Elimination Method [18, 25] and Prime+Prune+Probe [15]. For instance, Prime+Prune+Probe (1) primes the cache with a set of candidate attacker addresses, (2) removes candidate addresses that miss in the cache (prune), (3) triggers the victim to access the address of interest, and (4) probes the remaining set of candidate addresses for cache hits / misses. A candidate address missing in the cache has a conflict with the victim in at least one division. While advanced attack strategies such as Prime+Prune+Probe do not break the RSCs' security entirely, they demand for re-keying rates that are significantly higher than originally envisioned. Ultimately, these higher re-keying rates can sig-

nificantly degrade performance and render the overall design impractical. While PhantomCache appears more resilient to these strategies [4], the design features high associativity and thus higher power consumption.

Mirage [19] recognizes the security benefits of fully-associative caches and builds upon the ideas of RSCs to make fully-associative caches more practical to implement. Mirage introduces a level of indirection between a skewed, randomized, over-provisioned tag array and a global data array and allows for relocating tag entries to make sure replacement decisions are made globally and evictions in the tag array become rare. However, the over-provisioning of the tag array and indirection also leads to storage/area requirements 20% over the baseline set-associative cache, which may be prohibitive for some applications.

III. CHAMELEON CACHE

In the following we describe Chameleon Cache, a new randomized cache design to increase the security against contention-based cache attacks and thus reduce re-keying rates.

A. Idea

Caches in modern computing systems leak memory access patterns between several parties sharing the same cache. Fundamentally, this address leakage stems from two sources: First, the organization of caches in cache sets and cache ways uses a deterministic mapping from addresses to sets. This mapping allows to conclude about address information from contention in cache sets. Second, replacement policies like LRU reveal information about the access order and timing.

On the contrary, Fully Associative (FA) caches with random replacement do not leak address information as the selection of physical cache lines for insertion and eviction is entirely random. This effectively reduces the leakage for a cache that is shared between distrusting users. Contrary to partitioned cache designs, such FA caches with random replacement inevitably leak cache occupancy due to sharing the cache resource, but can attain better resource utilization.

While FA caches exhibit desirable security properties, their area and power demand are prohibitive for larger caches. To improve on these implementation aspects, this work presents Chameleon Cache. Chameleon Cache mimics the statistical properties of FA caches to inherit their security properties at lower implementation cost.

B. Concept

Prior work on RSCs has demonstrated a significant security improvement over set-associative caches by making it very unlikely that two addresses map to the same set of cache lines. However, observation of eviction patterns using techniques like Prime+Prune+Probe[15] still allows to learn about contention between addresses. To overcome this issue, conceptually, Chameleon Cache combines RSCs with a small, fully associative victim cache (VC) that automatically reinserts elements that have been evicted from the RSC to the VC. This
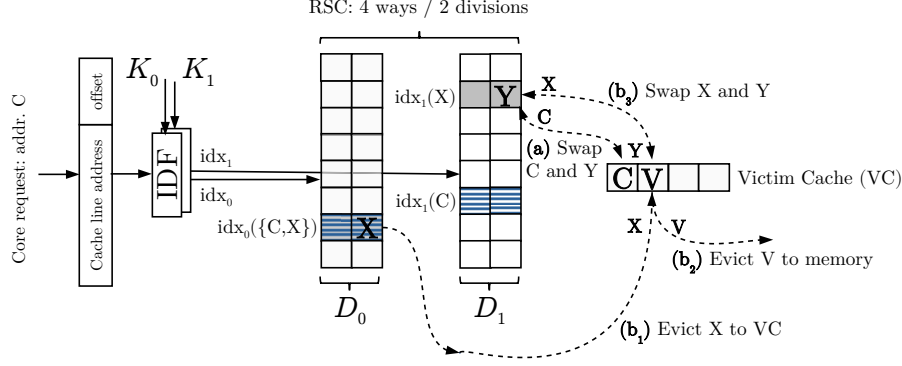
Fig. 1. Concept of Chameleon Cache for handling a core request C. Request C and address X translate to the blue striped and the gray shaded lines, respectively. Flows **(a)** and **(b)** show the cases for a hit in the VC and eviction from the RSC, respectively.

VC breaks the link between evictions from Chameleon Cache and contention in the RSC, results in eviction patterns similar to a FA cache with random replacement, and thus makes it harder for attackers to successfully learn RSC contention.

The example in Figure 1 depicts the concept in more detail: When the core issues a request to address C, Chameleon Cache first computes the indices the request maps to in each RSC division via an index derivation function (IDF) and performs look-ups to both the RSC and the VC in parallel.

1) If the request hits in the RSC, the line is simply returned.
2) If the request hits in the VC, the line is returned and also reinserted in one of the RSC sets previously determined for line C. If this reinsertion of C conflicts with another line Y in the RSC, **(a)** Y is put in place of C in the VC, i.e., C and Y are swapped.
3) If the request to address C misses in both the RSC and the VC, the line C is fetched from memory and inserted into one randomly chosen RSC division, where the concrete set index has been determined for line C before.
   Upon insertion of line C into the RSC, division D_0 in Figure 1, **(b₁)** this line C may conflict with some line X stored in the RSC before, which results in the line X to be moved to the VC. Moving the line X to the VC may cause **(b₂)** eviction of a line V previously present in the VC to the memory.

Later, the VC tries to reinsert X into the RSC, likely in a different cache way or division. If this reinsertion of X conflicts with another line Y in the RSC, **(b₃)** Y is put in place of X in the VC, i.e., the lines X and Y are swapped.

C. Specification

Chameleon Cache uses an RSC with s sets, w ways and $N = s \cdot w$ lines that are organized in $1 \leq d \leq w$ divisions. The divisions $D_0, D_1, \ldots, D_{d-1}$ each consist of $\frac{w}{d}$ ways, where each RSC way is mapped to a single division only. The RSC is skewed by its divisions: when accessing the cache, a different index idx_i is used to select the set S_{i,idx_i} in each division D_i. The RSC uses an IDF to compute the divisions' set indices $idx_0, idx_1, \ldots, idx_{d-1}$ from the requested cache line address.

Algorithm 1 Init

Input: RSC with w ways, s set indices and d divisions, VC
Output: Initialized RSC and VC

1: **for** $0 \leq i < d, 0 \leq j < s, 0 \leq k < \frac{w}{d}$ **do**
2: $\quad RSC[i][j][k] \leftarrow \perp$
3: **end for**
4: $VC[i] \leftarrow \perp \ \forall \ 0 \leq i < w_{VC}$
5: $idx_{VC,insert} \leftarrow 0$
6: $idx_{VC,reinsert} \leftarrow 0$

The pseudo-random mapping given by the IDF is regularly changed, e.g., by changing its keys.

Chameleon Cache uses a fully associative victim cache (VC) with w_{VC} ways. Cache lines evicted from the RSC are moved to the VC, which performs automatic reinsertion into the RSC to increase security and the interval for changing the IDF mapping. The VC maintains two indices $idx_{VC,insert}$ and $idx_{VS,reinsert}$ to keep track of the last item that has been inserted into the VC and re-inserted into the RSC, respectively. In addition, requests that hit in the VC result in automatic reinsertion of the respective line into the RSC.

Chameleon Cache uses a set of different algorithms to perform its operations. **Init** in Algorithm 1 initializes the cache. **IDF** in Algorithm 2 performs the mapping of cache line addresses to RSC sets. **Lookup** in Algorithm 3 describes the lookup of a cache line address in Chameleon Cache. **RSC Insert** in Algorithm 4 specifies the insertion of a new cache line into Chameleon Cache. **RSC Reinsert** in Algorithm 5 performs reinsertion of a cache line from the VC to the RSC. **Automatic RSC Reinsert** in Algorithm 6 is periodically triggered to initiate automatic reinsertion of lines from the VC to the RSC. Note that the respective counters $idx_{VC,insert}$ and $idx_{VC,reinsert}$ automatically wrap around when they reach w_{VC}. For simplicity, all algorithms omit the wrap-around logic.

a) Index Derivation Function: Algorithm 2 implements the IDF using a cryptographic block cipher E. However, the IDF may also be implemented based on other primitives, such

Algorithm 2 Index Derivation Function (IDF)

Input: address A, keys $K_0, ..., K_{d-1}$
Output: indices $idx_0, ..., idx_{d-1}$ for d RSC divisions

1: **for** $0 \leq i < d$ **do**
2:    $A_{enc,i} \leftarrow E_{K_i}(A)$
3:    $idx_i \leftarrow \lceil A_{enc,i} \rceil^{\log_2 s}$ // Slice out $\log_2 s$ bits
4: **end for**
5: **return** $idx_0, ..., idx_{d-1}$

Algorithm 3 Lookup

Input: address A, keys $K_0, ..., K_{d-1}$
Output: data at address A

1: $idx_0, ..., idx_{d-1} \leftarrow IDF(A, K_0, ..., K_{d-1})$
2: Hit $\leftarrow false$
3: **for** $0 \leq i < d$ **do**
4:    **for** $0 \leq j < \frac{w}{d}$ **do**
5:       **if** $RSC[i][idx_i][j].tag = A$ **then**
6:          Data $\leftarrow RSC[i][idx_i][j].data$
7:          Update $RSC[i][idx_i].lru\_state$ if required
8:          Hit $\leftarrow true$
9:       **end if**
10:    **end for**
11: **end for**
12: **for** $0 \leq i < w_{VC}$ **do**
13:    **if** $VC[i].tag = A$ **then**
14:       Data $\leftarrow VC[i].data$
15:       Hit $\leftarrow true$
16:       $RSCReinsert(i)$
17:    **end if**
18: **end for**
19: **if** $\overline{VCHit}$ **and** $\overline{RSCHit}$ **then**
20:    Data $\leftarrow memory[A]$
21:    $RSCInsert(Data, idx_0, ..., idx_{d-1})$
22: **end if**
23: **return** Data

Algorithm 4 RSC Insert

Input: data D, address A, indices $idx_0, ..., idx_{d-1}$

1: $\hat{d} \xleftarrow{\$} \{0, ..., d-1\}$
2: In set $RSC[\hat{d}][idx_{\hat{d}}]$: select victim line index v according to replacement policy
3: **if** $RSC[\hat{d}][idx_{\hat{d}}][v].tag \neq \bot$ **then**
4:    $idx_{VC,insert} \leftarrow idx_{VC,insert} + 1$
5:    **if** $VC[idx_{VC,insert}].tag \neq \bot$ **then**
6:       Evict line at $VC[idx_{VC,insert}]$ to memory
7:    **end if**
8:    $VC[idx_{VC,insert}] \leftarrow RSC[\hat{d}][idx_{\hat{d}}][v]$
9: **end if**
10: $RSC[\hat{d}][idx_{\hat{d}}][v].data \leftarrow D$
11: $RSC[\hat{d}][idx_{\hat{d}}][v].tag \leftarrow A$
12: Update replacement bits in set $RSC[\hat{d}][idx_{\hat{d}}]$ if necessary

---

Algorithm 5 RSC Reinsert

Input: VC line index idx_{vc} to be reinserted into the RSC

1: $idx_0, ..., idx_{d-1} \leftarrow IDF(VC[idx_{vc}].tag)$
2: $\hat{d} \xleftarrow{\$} \{0, ..., d-1\}$
3: In set $RSC[\hat{d}][idx_{\hat{d}}]$: select victim line index v according to replacement policy
4: Swap $RSC[\hat{d}][idx_{\hat{d}}][v]$ and $VC[idx_{vc}]$
5: Update replacement bits in $RSC[\hat{d}][idx_{\hat{d}}]$ if necessary

---

as (keyed) hash functions H. A suitable IDF must guarantee that (1) the keys remain secret as attackers observe addresses mapping to the same index (*collisions*), (2) addresses that have index collisions under one key K_A have an index collision with a different key K_B only with negligible probability. Moreover, the IDF should be efficient to implement to ensure low access latencies.

D. Indistinguishability

A main security requirement for Chameleon Cache is the *indistinguishability* of RSC and VC. Namely, implementations of Chameleon Cache must ensure that attackers cannot distinguish whether a line is in the RSC or in the VC. Otherwise, attackers would be able to recognize contention in the RSC by monitoring when lines are (temporarily) moved to the VC. The requirement of indistinguishability implies that implementations must make sure that (a) RSC and VC show the same

access latency for cache hits and (b) there are no observable side effects when lines transition between the RSC and VC and vice versa. Thus, a first step to achieve indistinguishability is a cache pipeline that returns the results from VC and RSC in the same pipeline stage as this can provide the same access latency for both RSC and VC. However, this list is non-exhaustive and a concrete implementation may require additional measures to be taken to guarantee indistinguishability.

IV. SECURITY ANALYSIS

As Section III showed, Chameleon Cache extends RSCs with a VC and automatic reinsertion to improve the security and thus reduce required re-keying intervals. In the following, we analyze the security of Chameleon Cache and compare it to other works. Our analysis consists of a qualitative analysis of the cache's eviction behavior, a probabilistic analysis that formalizes the relative difficulty of contention-based attacks with Chameleon Cache, and a quantitative empirical analysis using a cache attack simulation framework. While our specification of Chameleon Cache is agnostic to the replacement strategy of the RSC, we note that stateful replacement strategies like LRU come with additional side-channel leakage [30] and hence we focus our analysis on random replacement.

A. Victim Cache

Chameleon Cache extends RSCs with a VC to break the direct link between cache conflicts in the RSC and the cache

Algorithm 6 Automatic RSC Reinsert

1: **while** $idx_{VC,reinsert} < idx_{VC,insert}$ **do**
2:    $RSCReinsert(idx_{VC,reinsert})$
3:    $idx_{VC,reinsert} \leftarrow idx_{VC,reinsert} + 1$
4: **end while**

---

misses that may be observed, e.g., in Prime+Prune+Probe. As Figure 1 shows, when a cache line C is inserted into the RSC and evicts another line X to the VC, the cache conflict is hidden as both lines C and X will hit in the cache afterwards. In addition, a line V potentially being evicted from the VC to the memory is uncorrelated to the cache conflict in the RSC.

Without reinsertion of the evicted line, however, the attacker may be able to inspect the contents of the VC and yet learn about the conflict. For instance, an attacker could access a set of random, uncached addresses to force cache lines being moved to the VC. This would eventually flush the lines previously stored in the VC to the memory, giving the attacker a measurable side-channel about the VC. Note however that profiling RSC cache contention via flushing the VC adds noise to the attacker's measurements, i.e., the attacker will observe cache misses on lines that were previously present in the VC but are unrelated to the RSC contention introduced by accessing C. Intuitively, the number of false positives grows with the size of the VC.

To prevent attackers from learning about RSC contention via the VC, Chameleon Cache automatically reinserts cache lines, which have been evicted from the RSC to the VC, back into the RSC (cf. Algorithm 6). As a cache line X, which has been evicted by line C from the RSC to the VC, is reinserted into the RSC, two possible situations can arise:

1) **Reinsertion into a different cache way**: Reinsertion of a line X to the RSC results in another line Y being placed into the VC, i.e., X and Y are swapped. This makes the RSC conflict between C and X invisible, as eventually both will be stored in the RSC again. Cache line Y in the VC, on the other hand, is either invalid or does not directly relate to the contention between C and X in the RSC.

2) **Reinsertion into the same cache way**: X is reinserted into the same cache way it was evicted from, i.e., X and C are swapped. As a result, X is stored in the RSC and C in the VC, thus making the previous RSC contention invisible.
   Note that the attacker may be able to re-access C in order to bring C back into the RSC. In this case, a line Z is swapped with C, which results in Z and C being in the VC and RSC, respectively. This line Z, which may be the original line X, directly contends with C in the RSC. While an attacker might be able to learn Z by flushing the VC, using Z meaningfully remains hard. Namely, whenever Z evicts C from the RSC, the automatic reinsertion mechanism will move C back into the RSC and hence make the conflict invisible.

B. Second-Order Collisions

The principle of automatic reinsertion still affects the cache state and intuitively bears second-order leakage. More concretely, and as Figure 1 shows, accessing a line C that conflicts with a line X can lead to another line Y ending up in the VC, potentially making this line Y visible to attackers who are able to flush the VC. While Y is not directly conflicting with the line C, C and Y are connected via the conflicting line X that goes to the VC and back to the RSC. X conflicts both with C and Y in at least one division each and may thus serve as a proxy in a cache attack. Note, however, since there are multiple divisions in the RSC, this does not imply that C and Y conflict. We in the following denote such addresses Y second-order addresses.

Attackers may be able to learn about second-order addresses and use them in an attack to measure RSC contention with C. In Figure 1, C may evict line Y to the VC via the proxy X and flushing Y from the VC as described in Section IV-A may then allow an attacker to observe the contention. Yet, second-order addresses are hard to exploit in practice for multiple reasons.

1) **Indistinguishability:** For profiling strategies like Prime+Prune+Probe, an attacker observing a cache miss after flushing the VC cannot determine whether they sampled a second-order address or a completely unrelated address. This effectively increases the number of addresses needed for an eviction set and hence noise.

2) **Unknown proxy address:** Attackers do not know the proxy address X, because it cannot be observed. However, the second-order address Y is only valuable for attackers if they know and insert X before, i.e., X is a proxy for Y and is required to evict C.
   Moreover, since X is unknown, a second-order address Y that is collected by the attacker is as good as a random address: Without X, Y has the same probability as a randomly chosen address to evict C.

3) **Prevalence of proxy addresses:** An attacker observing a miss on the second-order address Y via proxy X is unlikely to find another proxy X' which collides with both C and Y in different divisions. A randomly chosen address is a proxy for C and Y with a probability $p \approx \frac{w^2}{s^2}$ if $d = w$, e.g., 1 in 16384 addresses are a suitable proxy in a cache with 2048 sets and 16 ways. However, an arbitrarily chosen address itself has higher probability of directly featuring a partial collision with C, roughly $p = 1 - \frac{(s-1)^w}{s^w} \approx 2^{-8}$ for the same cache configuration. Note that that when X contends with Y and C in the same division, Y may directly evict C from the RSC, but automatic reinsertion of C will make this contention invisible.

4) **Success probability:** Even if attackers know the proxy X, the probability of evicting the line C from the RSC to the VC using the second-order address Y and vice versa is low. Namely, this approach requires (1) C and X to reside in the correct cache divisions before, (2) Y
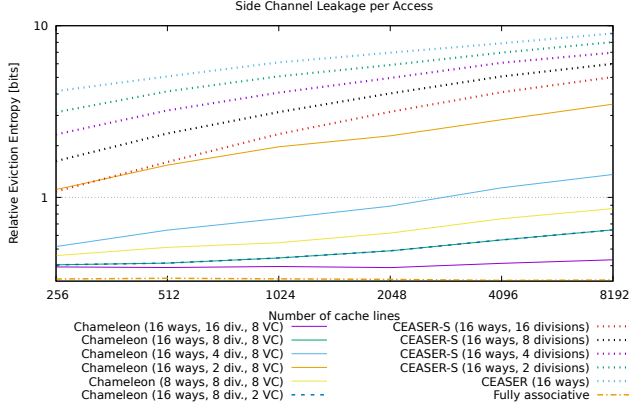
18

Fig. 2. Relative eviction entropy for Chameleon Cache compared to CEASER, CEASER-S, and a fully associative cache.
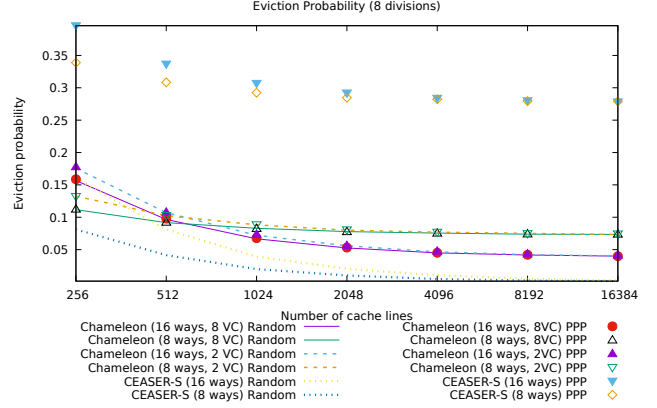

Fig. 3. Eviction success rates of eviction sets constructed via Prime+Prune+Probe compared to randomly selected addresses. All caches are operated with 8 divisions
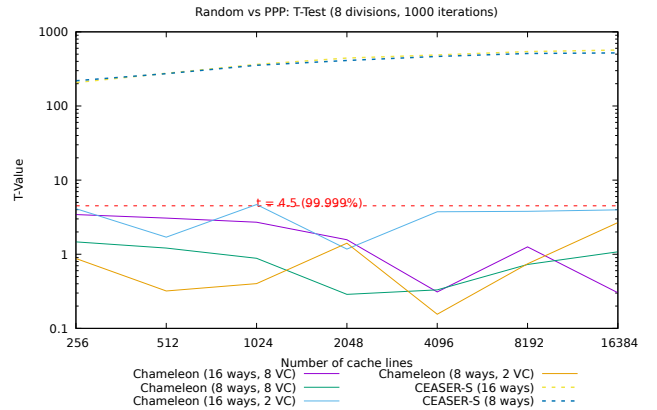

Fig. 4. T-values for the eviction success rates of eviction sets constructed via Prime+Prune+Probe compared to randomly selected addresses.

needs to be inserted such as to evict X, and (3) X must be reinserted such as to evict C to the VC. For random replacement, RSC eviction using second-order addresses thus has a success probability of only w^{-4}, e.g., 2^{-16} for a 16-way cache. For a 16-way cache with 2048 lines, it would hence require more addresses than fit in the cache to with high probability evict the target address into the VC by using second-order addresses. Moreover, once C has been moved to the VC, attackers further need to flush the VC with random addresses, which adds more noise through contention required in the RSC.

C. Relative Eviction Entropy

We used the cache security framework CacheFX [4] to implement a model for Chameleon Cache and comprehensively test and compare it to state-of-the-art cache designs. In Figure 2, we evaluated the relative eviction entropy of Chameleon Cache, CEASER, CEASER-S, and a fully associative cache for increasing cache sizes and with all these caches using random replacement. Figure 2 shows that the information leakage is significantly lowered for Chameleon Cache compared to prior cache randomization techniques. For instance, adding a victim cache to 16-way 8192-line CEASER-S with 16 divisions reduces information leakage per eviction from 5 to 0.4 bits. Further note that the relative eviction entropy is the same for instances of Chameleon Cache that only differ in their VC size.

D. Eviction Set Success Rate

To demonstrate the security of Chameleon Cache w.r.t. Prime+Prune+Probe, we compare the eviction success rate of eviction sets constructed with Prime+Prune+Probe on Chameleon Cache to the eviction success rate for a set of randomly chosen addresses. Using CacheFX [4], we run Prime+Prune+Probe and sample a random set of addresses $M = 1000$ times to form eviction sets of $4 \cdot w$ addresses for a random target and evaluate the success rate of each eviction set. The success rate of each eviction set is determined by trying to

evict the target address 1000 times and compute the mean over all experiments. We show the mean eviction success rates for different configurations of Chameleon Cache and CEASER-S in Figure 3 and the M experiments. In this evaluation, we operate all caches with 8 divisions and experimented with 2 and 8 victim cache lines for Chameleon Cache as well as 8 and 16 cache ways. While CEASER-S shows a strong difference in the eviction success rate for eviction sets constructed via random sampling and Prime+Prune+Probe, Chameleon Cache does not yield an observable difference.

To investigate the properties of Chameleon Cache further, we also determine the statistical variance of the M eviction success rates and compute the t-value [21]. We reject the hypothesis that the mean success rates for eviction sets from Prime+Prune+Probe and random sampling are equal with a confidence of 99.999% if $|t| > 4.5$.

Figure 4 shows that the t-value stays largely below the threshold of 4.5 for Chameleon Cache, suggesting that attackers do not have a clear advantage from constructing eviction
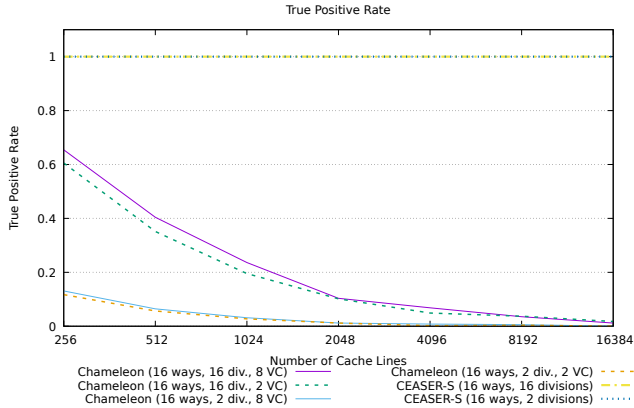
19

Fig. 5. Rate of addresses in the eviction set that are truly conflicting with the victim address.

Fig. 6. Number of memory accesses required by the attacker to find one truly conflicting address.

System	OS	Redhat 8 with Linux kernel 5.4.49
	Processor	4 x86 OoO Cores at 3GHz
Core	Predictor	LTAGE and Indirect Predictor, 512-entry BTB
	Fetch	5 wide Fetch, Decode, Rename, 224-entry ROB
	Dispatch	8 wide Dispatch, Issue, Writeback, 97-entry IQ
	Exec	4 INT ALUs, 3 INT VectU, 2 FP FMAs, 168/180 Phys. Reg., 72/56-entry Ld/St Buffer
Memory	L1-I/D	32kB, 8-way, 2/4 cycles, 16-entry MSHR, Random Replacement
	L2	256kB, 4-way, 10 cycles, 20-entry MSHR, Random Replacement
	Shared L3	16MB, 16-way, 40 cycles, 256-entry MSHR, stride prefetch
	DRAM	8GB 4 Channel DDR4-2400, 38.4GB/s Peak Bandwidth, Latency from DRAMSim2

TABLE I
GEM5 FULL-SYSTEM SIMULATION CONFIGURATIONS [29]

sets with Prime+Prune+Probe over random sampling. On the other hand, eviction sets built with Prime+Prune+Probe for CEASER-S have success rates clearly distinguishable to success rates of randomly assembled eviction sets. Note however that for Chameleon Cache and a larger number of different experiments M there is statistically measurable difference between eviction sets constructed via Prime+Prune+Probe and random sampling according to the t-statistics. Yet, we argue that this difference is small enough not to be relevant in practice.

E. Eviction Set Profiling

We evaluated the properties of Prime+Prune+Probe, the currently most-efficient algorithm to construct eviction sets for skewed caches, in more detail. Figure 5 shows the fraction of addresses found by Prime+Prune+Probe in a noise-free setting that are truly conflicting in at least one division with the victim address. While for CEASER-S this True Positive Rate (TPR) in the absence of noise is consistently 1, the TPR is clearly lower for all configurations of Chameleon Cache and decreases with cache size. The TPR is generally smaller for 2 divisions than for 8 divisions, because more divisions increase the probability of random conflicts in any of the divisions.

Figure 6 shows the number of read accesses that need to be done by the attacker in order to find one address that is truly conflicting with the victim address, i.e., not noise. The effort for finding such address is one order of magnitude higher for Chameleon Cache compared to CEASER-S with larger instances of Chameleon Cache having even higher relative profiling cost. Overall, Chameleon Cache increases the cost of profiling and significantly decreases the value of the eviction sets found via Prime+Prune+Probe.

F. Cache Occupancy

Recent work [23] demonstrated scenarios that exploit side-channel leakage stemming from cache occupancy. Like fully associative caches with random replacement, Chameleon Cache does not protect against this cache occupancy leakage.
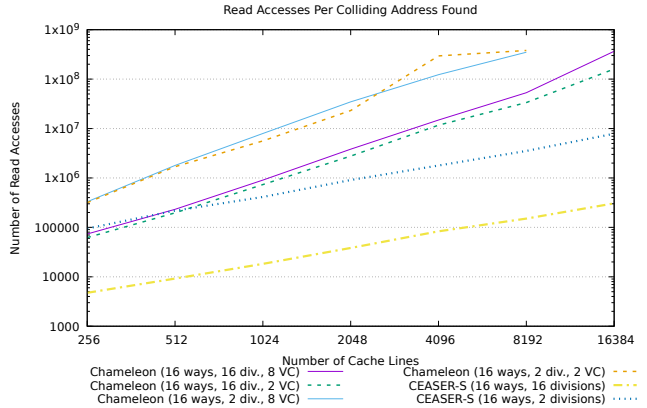
Note however that cache occupancy leakage is very coarse-grained, has only limited temporal resolution, and is inevitable for any shared cache design. We thus believe that the security properties offered by Chameleon Cache will be sufficient in most cases.

V. EVALUATION

We use the cycle-accurate gem5 [1, 13] simulator to evaluate Chameleon Cache. Table I shows the baseline simulator configuration based on a Skylake processor. We run SPECRate2017 with 4 copies in full-system mode, and gather statistics using a sampling methodology based on pFSA [20] due to the long execution time of SPEC reference inputs. In this sampling methodology, we execute each benchmark using hardware virtualization (i.e., the gem5 KVM CPU) to record the total instructions and generate 1000 random samples. Next, we quickly fast-forward to each sample using KVM and then perform functional warm-up of the caches for 10 million instructions with the atomic CPU, switch to detailed warm-up for 6 million instructions, and finally record detailed statistics for 5 million instructions with the Skylake CPU.

Figure 7 depicts the relative Instructions-Per-Cycle (IPC) for CEASER-S and Chameleon Cache with varying division counts. Note that ScatterCache can be viewed as an instance of CEASER-S with 16 divisions. It shows that on average the
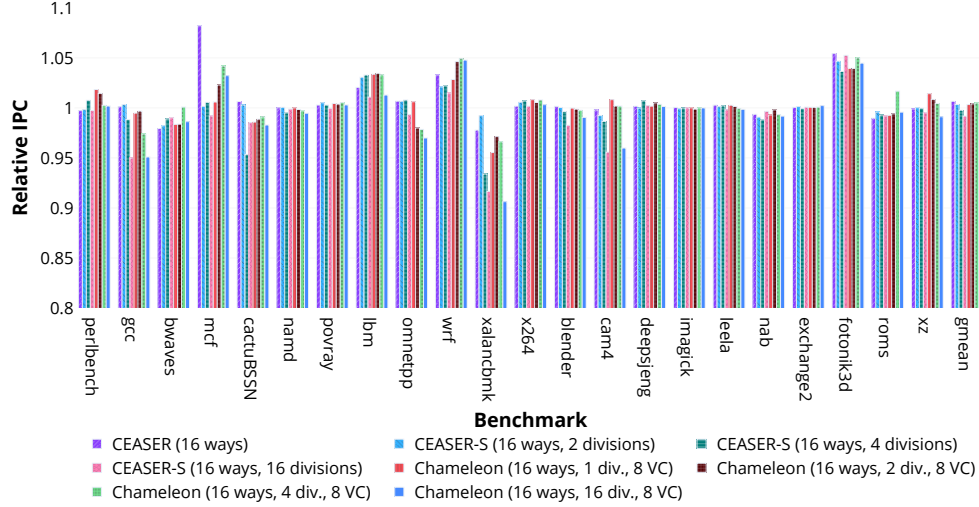
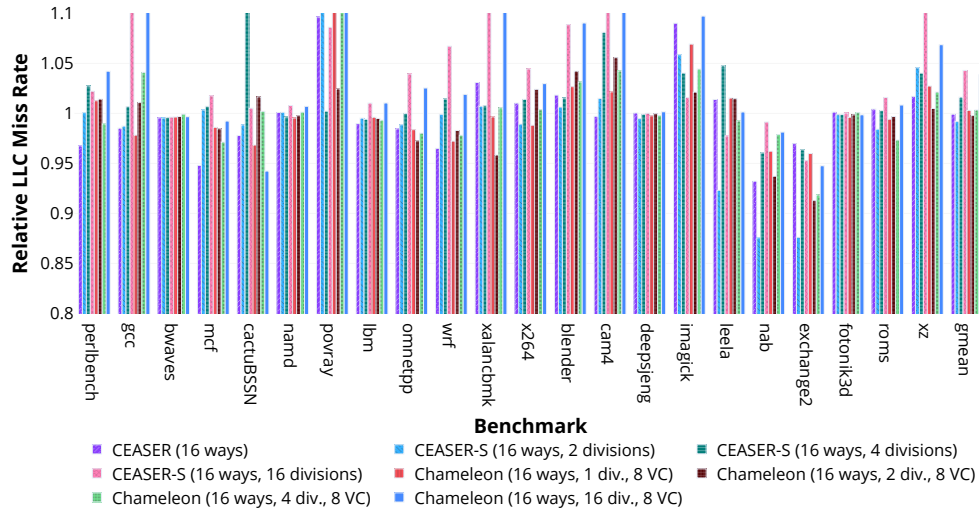Fig. 7.   Relative Instructions Per Cycle (IPC) for SPECRate2017 with 4 copies.

Fig. 8.   Relative LLC Miss Rate for SPECRate2017 with 4 copies.

relative IPC drops for CEASER-S with a higher number of divisions, whereas Chameleon Cache, except for 16 divisions, is less sensitive to the division count. This results in better average performance than for previous RSC proposals and helps some workloads, e.g., wrf and mcf, to perform even better than the baseline design from Table I. Generally, the relative performance impact is very small, i.e., <1% on average, and ranges between -10% and +5% for individual workloads. Figure 8 further shows the relative miss rate for the shared Chameleon Cache L3/LLC cache. Except for 16 divisions, Chameleon Cache on average features a miss rate and relative

IPC equal to the baseline. For individual workloads, miss rate and relative IPC range between -10% and +5%. In some cases the victim cache reinsertion can improve performance by helping spread hot sets out to other partitions. In addition, we measured the frequency of contention events between the reinsertion of victim cache entries and incoming cache requests, but saw no conflicts in our experiments. In terms of area, Chameleon Cache with an 8-way VC must maintain 8 additional cache lines per cache slice, which amounts to <0.1% additional storage for the architecture specified in Table I.

VI. VARIANTS

While Chameleon Cache aims to mimic a fully associative cache with random replacement and its security properties, the design principle of adding a victim cache to decouple contention in a RSC from evictions observed by users is applicable more generally. In the following, we thus lay out several design variants that can as well increase security over a baseline RSC.

A. Chameleon Cache without Reinsertion

A first simplification of Chameleon Cache is to omit its automatic reinsertion functionality, i.e., to simply extend a RSC with a fully associative VC. This fully associative victim cache can use first-in first-out replacement like Chameleon Cache, or implement random replacement to add more noise to the design's eviction patterns. The disadvantage of omitting automatic reinsertion is that contention in the RSC is more easily observable if attackers can flush the VC, e.g., by creating contention in the RSC. However, as indicated in Section IV-A and as depicted in Figure 9, this adds noise and thus profiling cost proportional to the size of the VC and hence helps reduce re-key rates accordingly.
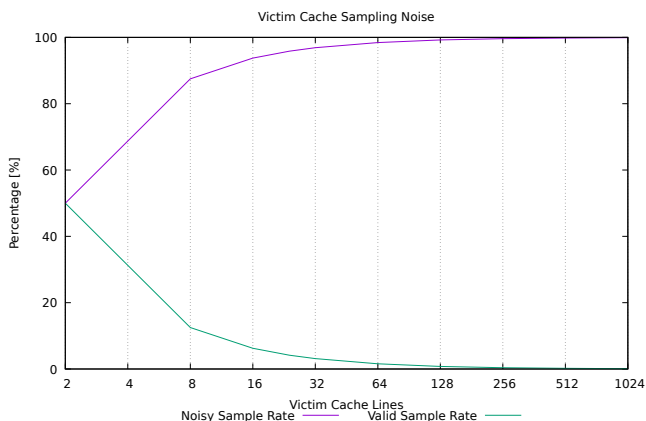

Fig. 9. Rate of noisy samples and truly conflicting samples when performing eviction set profiling [16] in randomized set-associative caches with various victim cache sizes.

We expect random accesses stemming from the system's background noise to be equivalent to an attacker creating pseudo-random contention in the RSC in terms of their ability to flush the VC. Consequently, some open questions are (1) to what extent system noise and the attacker's specific behavior compose, (2) whether system noise itself is sufficient to make RSC contention visible, and (3) what impact system noise has on security overall.

B. Cache-Set Randomization with a Victim Cache

Another simplification is to omit the complexity of cache skewing and extend cache-set randomization like CEASER [17] with a fully associative VC. As before, the VC will decouple evictions being observed from contention in the randomized cache and thus add noise proportional to the VC size and help reduce re-key intervals. One additional drawback is the smaller number of effective cache sets in this design, which may allow the attacker to exhaustively build eviction sets for all the cache sets and more easily find patterns of interest. More importantly, once the attacker has crafted one eviction set for one set in the cache, the attacker can use these to accurately create contention in the randomized, set-associative cache that will more reliably flush the VC and help any profiling or attack further on.

C. Randomized (Skewed) Cache with a Set-Associative Victim Cache

As Figure 9 shows, the noise level introduced by the VC increases proportional to its size. As this can lead to smaller re-keying rates, a larger VC is desirable. However, there is a practical size limit for fully associative (victim) caches due to power and area constraints. To yet realize large VC sizes, a more aggressive design variant is to replace the fully associative VC with a set-associative VC that, like the randomized (skewed) main cache, uses a secret mapping to derive the set index. This randomized, set-associative VC would still be smaller than the randomized main cache and could as well be subject to re-keying. Interestingly, if the main cache uses pure set-randomization like CEASER, a randomized, set-associative VC adds a second-level mapping similar to cache skewing, but without the necessity to touch the baseline cache's set-associative structure. It seems a relevant question if this two-level design is indeed equivalent to RSCs or has different security properties and, e.g., allows for new attacks.

VII. COMPARISON

The objective and design of Chameleon Cache bears some similarity to Mirage [19]: both aim to mimic the behavior of a fully-associative cache with random replacement and extend the concept of RSCs. Moreover, the reinsertion policy of Chameleon Cache behaves similar to cuckoo relocation as presented for Mirage, but serves a different purpose: Chameleon Cache uses reinsertion to increase effective associativity, whereas Mirage implements cuckoo relocation to reduce the likelihood of conflicts in the skewed, over-provisioned tag array.

However, there are also some major differences between Chameleon Cache and Mirage: Mirage is designed to make global replacement decisions as in a fully-associative cache, whereas Chameleon Cache uses RSC reinsertion to obtain eviction patterns that look similar to fully associative caches. Moreover, Chameleon Cache features lower area overheads as it does not require indirection pointers nor over-provisioning of the tags, but may introduce higher cache activity from reinsertion. Both Chameleon Cache and Mirage show low overheads of $<1\%$ and 2% on spec2017, respectively. Last, Chameleon Cache introduces the concept of victim caches to RSCs to hide contention in the cache eviction pattern and stage re-insertions. As highlighted in Section VI, this victim cache is a versatile tool that thus may also be used to extend Mirage and improve its properties and trade-offs.

VIII. Conclusion

Recent analysis of cache randomization has demonstrated new approaches, e.g., Prime+Prune+Probe, to more efficiently learn how addresses map to cache sets and calls for more frequent re-keying to maintain side-channel resilience. However, increasing the re-keying rate comes with higher performance overheads and implementation cost. As a result, this work presented Chameleon Cache to make cache-set randomization more robust and facilitate reduced re-keying intervals. With the aim to mimic FA caches with random replacement, Chameleon Cache extends RSCs with a fully associative victim cache (VC) and automatic reinsertion. This additional VC hides contention occurring in the Randomized Skewed Caches (RSC) by decoupling the RSC contention from evictions being observed by system users. More importantly, Chameleon Cache leverages the multiple mappings available in skewed caches to automatically reinsert lines moved to the VC back into the RSC in a potentially different division (with a different mapping). This automatic reinsertion mechanism is designed to revert the original RSC contention and pick an alternative eviction candidate and seeks to obtain eviction patterns that are similar to eviction patterns of fully-associative (FA) caches with random replacement. Thus, Chameleon Cache can resist fine-grained contention-based attacks and reduce its attack surface to cache utilization channels as in FA caches with rand. replacement. We evaluated the performance of Chameleon Cache in gem5 showing overheads of $< 1\%$ and highlighted the versatility of the VC in alternative designs to increase side-channel resilience and reduce re-keying rates in randomized caches.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. S. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020.* IEEE, 2020, pp. 1110–1123. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00092

[3] L. Domnitser, A. Jaleel, J. Loew, N. B. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *TACO*, vol. 8, no. 4, pp. 35:1–35:21, 2012.

[4] D. Genkin, W. Kosasih, F. Liu, A. Trikalinou, T. Unterluggauer, and Y. Yarom, "Cachefx: A framework for evaluating cache security," *CoRR*, vol. abs/2201.11377, 2022. [Online]. Available: https://arxiv.org/abs/2201.11377

[5] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, Gauss, and reload - a cache attack on the BLISS lattice-based signature scheme," in *CHES*, 2016, pp. 323–345.

[6] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *DIMVA*, 2016, pp. 279–299.

[7] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *IEEE SP*, 2011, pp. 490–505.

[8] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, May 1990.

[9] 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 *IEEE SP*, 2019, pp. 1–19.

[10] 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, pp. 973–990.

[11] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015, pp. 605–622.

[12] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[13] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," *arXiv:2007.03152 [cs]*, Sep. 2020.

[14] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006, pp. 1–20.

[15] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE SP*, 2021.

[16] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+scope: Overcoming the observer effect for high-precision cache contention attacks," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security,*

Virtual Event, Republic of Korea, November 15 - 19, 2021, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 2906–2920. [Online]. Available: https://doi.org/10.1145/3460120.3484816

[17] M. K. Qureshi, "CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018, pp. 775–787.

[18] ——, "New attacks and defense for encrypted-address cache," in *ISCA*, 2019, pp. 360–371.

[19] G. Saileshwar and M. Qureshi, "{MIRAGE}: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021, pp. 1379–1396.

[20] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed," in *2015 IEEE International Symposium on Workload Characterization*, Oct. 2015, pp. 183–192.

[21] T. Schneider and A. Moradi, "Leakage assessment methodology - A clear roadmap for side-channel evaluations," in *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, ser. Lecture Notes in Computer Science, T. Güneysu and H. Handschuh, Eds., vol. 9293. Springer, 2015, pp. 495–513. [Online]. Available: https://doi.org/10.1007/978-3-662-48324-4_25

[22] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: Association for Computing Machinery, May 1993, pp. 169–178.

[23] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security*, 2019, pp. 639–656.

[24] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating cache conflicts with localized randomization," in *NDSS*, 2020.

[25] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE SP*, 2019, pp. 39–54.

[26] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, pp. 74:1–74:6. [Online]. Available: https://doi.org/10.1145/2897937.2898086

[27] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007, pp. 494–505.

[28] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *USENIX Security*, 2019, pp. 675–692.

[29] WikiChip, "Skylake (server) - microarchitectures - intel," https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server), May 2017.

[30] W. Xiong and J. Szefer, "Leaking Information Through Cache LRU States," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2020, pp. 139–152.

[31] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2003–2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yan

[32] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.

[33] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, W. Schröder-Preikschat, J. Wilkes, and R. Isaacs, Eds. ACM, 2009, pp. 89–102. [Online]. Available: https://doi.org/10.1145/1519065.1519076