USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Systematic Evaluation of Randomized Cache Designs against Cache Occupancy

Anirban Chakraborty, *Max Planck Institute for Security and Privacy;* Nimish Mishra, *Indian Institute of Technology Kharagpur;* Sayandeep Saha, *Indian Institute of Technology Bombay;* Sarani Bhattacharya and Debdeep Mukhopadhyay, *Indian Institute of Technology Kharagpur*

https://www.usenix.org/conference/usenixsecurity25/presentation/chakraborty

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Systematic Evaluation of Randomized Cache Designs against Cache Occupancy

Anirban Chakraborty*† Nimish Mishra*‡ Sayandeep Saha§
Sarani Bhattacharya‡ Debdeep Mukhopadhyay‡

†Max Planck Institute for Security and Privacy, Germany
‡Indian Institute of Technology Kharagpur, India
§Indian Institute of Technology Bombay, India

anirban.chakraborty@mpi-sp.org nimish.mishra@kgpian.iitkgp.ac.in {sarani, debdeep}@cse.iitkgp.ac.in sayandeepsaha@cse.iitb.ac.in

Abstract

Randomizing the address-to-set mapping and partitioning of the cache has been shown to be an effective mechanism in designing secured caches. Several designs have been proposed on a variety of rationales: ① randomized design, ② randomized-and-partitioned design, and ③ psuedo-fully associative design. This work fills in a crucial gap in current literature on randomized caches: currently most randomized cache designs defend only contention-based attacks, and leave out considerations of cache occupancy. We perform a systematic evaluation of 5 randomized cache designs- CEASER, CEASER-S, MIRAGE, ScatterCache, and SassCache against cache occupancy wrt. *both* performance as well as security.

With respect to performance, we first establish that benchmarking strategies used by contemporary designs are unsuitable for a *fair* evaluation (because of differing cache configurations, choice of benchmarking suites, additional implementation-specific assumptions). We thus propose a uniform benchmarking strategy, which allows us to perform a fair and comparative analysis across all designs under various replacement policies. Likewise, with respect to security against cache occupancy attacks, we evaluate the cache designs against various threat assumptions: ① covert channels, ② process fingerprinting, and ③ AES key recovery (to the best of our knowledge, this work is the first to demonstrate full AES key recovery on a randomized cache design using cache occupancy attack). Our results establish the need to *also* consider cache occupancy side-channel in randomized cache design considerations.

1 Introduction

Caches hide the overall memory access latency by keeping data and instructions close to the processor. It leverages the principle of locality of reference by buffering recently used data such that for future references those data can be directly served to the processor, saving multiple clock cycles. Modern processors employ cache memories in slices, sets and ways and are typically based on set-associative design. Depending on the cache hierarchy, either the virtual or the physical address is used to map the incoming memory block into one of the ways of a particular set in the cache. In commercial processors from Intel and AMD, the address to set mapping in the last level cache (LLC) is typically done using *complex addressing* [24] on the virtual address. Due to the limited size of the cache, multiple memory blocks can be allocated to the same set, thereby creating *conflicts*. These conflicting addresses that are mapped to the same cache set are called *congruent* to each other. A number of works in literature exploit this phenomenon to devise a class of cache attacks called *contention-based* attacks [11, 13, 16–18, 23, 26–28, 35, 49, 50]. In such attacks, the attacker tries to fill up particular sets in the cache and then monitors those sets to ascertain whether targeted victim address is allocated to the same set. While other forms of cache attacks exist in literature, the attention garnered by contention-based attacks is overwhelming.

The `Prime+Probe` [1, 2, 18, 23, 28] is the most notable among contention based cache attacks. Here the attacker tries to create a set of congruent addresses, called *eviction set*, that will potentially contend with the victim's address for a particular cache set. The success of the attack depends greatly on the efficiency of eviction set generation by the attacker for an unknown target address. In the last few years, multiple works [7, 23, 32, 45] have proposed efficient creation of eviction sets to increase the success rate of contention-based attacks. Consequently, protecting the cache, especially the LLC, as it is shared among all the cores, against such contention-based attacks has been one of the widely researched directions. A majority of cache protection schemes attempt to scramble the address-to-cache-set mapping by using randomization techniques. Although different designs employ randomization in different ways, the broad idea is to use cryptographic primitives like block ciphers, hash functions, etc., to randomize the mappings in hardware with secret keys.

Contemporary randomized cache schemes can be classified into three categories [8, 29, 37] - (i) *randomized* (e.g.

*Equal Contribution.

CEASER), (ii) *randomized and partitioned* (e.g. CEASER-S, ScatterCache, SassCache) and (iii) *pseudo fully associative* (e.g. MIRAGE). With the sole exception of SassCache [14], the other designs of randomized caches have been subjected to rigorous security evaluations [8, 29, 37, 40] but only from the perspective of a particular attack, the contention-based attack. In other words, the security guarantees of these ingenious cache designs have been envisaged on the basis of the relative degree of resilience they present against the formation of eviction set. To the best of our knowledge, there has been no work in literature that evaluates these schemes with respect to other important cache attack variants[1]. In this work, we take an orthogonal approach to explore the security and performance properties of various state-of-the-art randomized cache designs with respect to an important but hugely overlooked cache attack - *cache occupancy attacks* [25, 34, 39].

1.1 Motivation

Although the number of proposed secured cache schemes and papers that claim to break such schemes try to outnumber each other, an important observation at this juncture is that most of these schemes focus on thwarting only contention-based attacks. The main rationale behind such schemes is that if address-to-set mapping can be made unpredictable and unobservable, the probability of creating an effective eviction set becomes negligible. In this pursuit, modern state-of-the-art cache protection schemes have incorporated complex structures (like specialized block ciphers, decoupling tag and data stores, localized randomization, etc.) both in software and hardware that beget the following questions:

- How do such schemes, engineered to be resilient against contention-based attacks, impact the performance across various replacement policies? This question becomes especially important because for an uniformly sampled key, the randomizing function has a uniform distribution over *all* sets in the cache, for each cache-line install. Because of this uniform probability distribution, a particular set has higher chances of being chosen for replacement in a randomized LLC than in baseline set-associative cache[2]. Furthermore, various replacement policies are proposed for baseline set-associative caches, but contemporary literature mostly compares baseline set-associative cache with random replacement or plain Least-Recently-Used.
- In an attempt to provide protection against certain types of attacks, do state-of-the-art cache protection schemes make themselves more vulnerable to other types of attacks? Specifically, wrt. randomized caches, can cache occupancy lead to adversarial success against the following threat assumptions: ① covert channels, ② process fingerprinting, and ③ AES key recovery.

In this paper, we answer these questions in the context of three distinct classes of secured caches, with respect to their relative resilience and performance against cache occupancy attacks (a type of cache attack that does not rely on creating eviction sets). In the microarchitectural security research community, it is usually considered that cache occupancy attacks do not pose a significant security threat to real-world critical applications. Moreover, many randomized cache designs (e.g. MIRAGE [36]) do not consider cache occupancy attacks in their perceived threat model. However, the realm of cache occupancy channels and its potential impact on the security of randomized caches has not been thoroughly investigated in literature. In this work, we show that cache occupancy attacks can be used to perform key recovery attacks on AES T-tables.

1.2 Contribution

We make the following contributions in this paper:

- We provide a detailed performance analysis of different randomized cache designs across multiple replacement policies and spurious cache occupancy.
- We provide a comprehensive evaluation of multiple secured cache schemes from different design families, namely CEASER, CEASER-S, ScatterCache and MIRAGE with respect to cache occupancy attack. This work delves into the implications of certain design principles and implementational decisions from an orthogonal perspective that has been largely overlooked in literature.
- We compare the cache candidates based on their relative ease of creating covert channels and adversarial fingerprinting of different workloads of SPEC2017 benchmark suite.
- Finally, we evaluate the candidate cache designs against key recovery attacks on T-table based AES-128, and provide a comparative analysis based on the guessing entropy of key recovery in each case.

2 Background

2.1 Traditional Cache Designs

The classical cache found in most modern commercial processors closely follows set-associative cache design: entire cache is divided into multiple *sets*, each containing a fixed number of blocks, called *ways*. The address-to-set mapping is done through a deterministic function that directly depends on a certain part of the address. This deterministic mapping makes such caches vulnerable to contention-based attacks. In modern processors, the Last-Level Cache (LLC) are typically set-associative caches split into multiple *slices*. This is

[1] Only SassCache defends against occupancy attacks by cryptographically limiting the total number of cache lines available for attacker occupancy.

[2] Because in addition to dependence on cache-line address, the eviction probability in case of randomized caches is also conditioned on the randomizer key and security domain identifier.

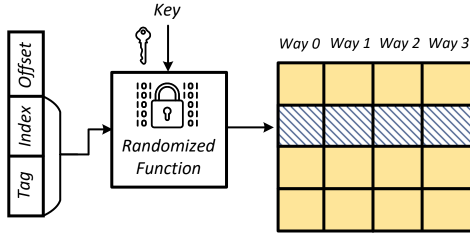


Figure 1: CEASER - randomized using block cipher

analogous to *partitioned caches*, where the cache is logically broken into multiple *partitions* and an incoming address can be mapped to one of the partitions.

2.2 Secured Cache Designs

The majority of the modern cache attacks focus on either performing contention by setting the cache to a known state (`Prime+Probe` [28]) or exploiting the hit-miss ratio for shared addresses (`Flush+Reload` [15]). Naturally, the countermeasures proposed limit the side-channel leakage through contention. We now summarize such design principles:

2.2.1 Randomized Caches

In randomized caches, for every cache-line install, the allocated set is determined by *keyed randomization* of the cache-line address by either a lookup table or a pseudo-random function (like block ciphers). Assuming the randomization key remains private, such design makes contention statistically difficult as the address-to-set mappings become unpredictable to the adversary. Earlier works like RPCache [46], NewCache [47] and RandomFill Cache [21] used permutation tables makes such designs unscalable for large LLCs.[3]. More recent designs like Time-Secured Cache [42], CEASER [31], etc. alleviate the scalability issues by computing the mapping in hardware using cryptographic functions.

CEASER: CEASER, as shown in Fig. 1, uses a block cipher to convert the physical address into an *encrypted address line* and uses it to index into the LLC. The encryption key is randomly generated and periodically refreshed (*rekeying*) to change the address-to-set mapping after a fixed epoch period. Although block ciphers in address-to-set calculation add to the overall access latency, they are suitable for LLCs due to their large latency budget[4]. However, note that although the design of CEASER is referred to as "randomized", it fundamentally employs a pseudo-random mapping with a deterministic outcome [8, 29]. Keeping other variables unchanged (like encryption key in a given epoch), a particular cache-line address always produces the same output.

[3]We thus do not focus on such designs in this work.

[4]LLCs are significantly slower compared to L1 caches.

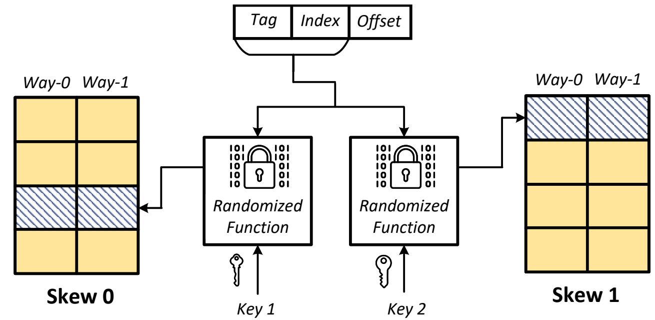


Figure 2: CEASER-S- Randomized-partitioned with two keys

2.2.2 Randomization and Partitioning

Another design rationale is combination of partitioning with randomization to achieve higher degree of non-determinism than randomized caches. Under this rationale, the LLC is divided into fixed number of partitions and uses a randomized mapping function to derive a different cache set in each partition, while the partition is randomly selected. The generic architecture for this type can be realized in two different ways: ① each partition is mapped into by using a separate key and security domain identifier, and ② all partitions are mapped into by using the same key and security domain identifier, but the position of the index bits is dependent on the partition number. Prominent cache designs that follow these principles are CEASER-S [32], ScatterCache [48], and SassCache [14].

CEASER-S: CEASER-S is derived from CEASER by incorporating the concept of static partitioning (skewed cache). While CEASER-S can support multiple partitions (or skews), we consider two partitions for exposition[5]. As shown in Fig. 2, for each incoming cache-line address, every partition has its own independent instance of randomization function with a partition-specific key. The encrypted addresses thus produced are used to access both the skews concurrently. In case of a cache hit in either of the skews, the cache partition returns the data from the line that had the hit. On a miss, one of the skew is chosen randomly to install the line. The victim cache-line to replace is decided by the replacement algorithm in place.

ScatterCache: Unlike CEASER-S, ScatterCache uses a single instantiation of the randomizing function with a single key. The incoming address is processed through the randomizing function, and the encrypted output is used to map into different sets in different partitions. The cache is divided into P partitions (where $1 \leq P \leq$ number of ways) and the different bit-masks of the encrypted address are used to index into each partition. The choice of the partition is random. Fig. 3 shows a representation of ScatterCache with two partitions.

SassCache: SassCache builds upon ScatterCache and introduces keyed randomization function for *each* security domain. It also ensures that the LLC occupancy for any two

[5]The default CEASER-S design contains two partitions.

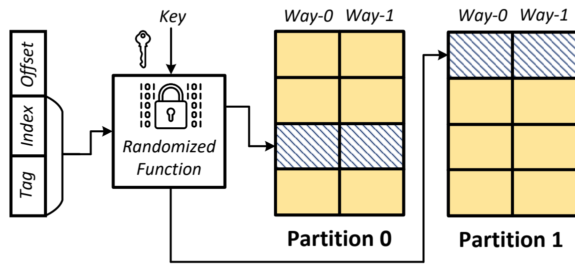


Figure 3: ScatterCache - randomized and skewed.

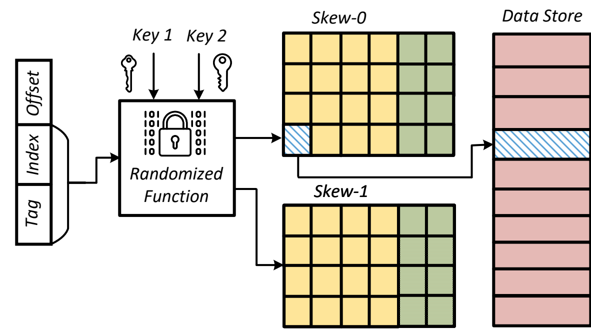


Figure 4: MIRAGE - pseudo fully associative cache.

distinct security domains is only *partially overlapping* by design. This allows SassCache to be resilient against both eviction-based attacks and occupancy-based attacks.

2.2.3 Pseudo Fully Associative Caches

Multiple cache designs [4, 37, 43] have been been proposed that adopt a design rationale orthogonal to the set-associative cache indexing structure by approximating a *pseudo-fully associative randomized design*. The goal of these designs is to provide security of fully associative cache while still having the practical look-up of a set-associative design.

MIRAGE: The key insight in MIRAGE is to separate the tag store and data store (inspired by inspired by V-way caches [33]) such that any replacement of already-installed cache-lines does not leak information about the contents of the cache. To achieve this, MIRAGE uses set-associative lookup in the tag store while maintaining full associativity in the data store. MIRAGE enforces *global replacement* on a cache miss and provisions extra invalid tags in each set in the tag store to prevent set conflicts. Thus, when a new line is installed, an invalid tag can be allocated to it without requiring to evict an already-installed one from the same set. For the data store, a victim entry is chosen *at random*, which ensures global eviction. In addition, MIRAGE also uses skewed caches with load-balancing and randomization of the input address for added security. Fig. 4 shows a schematic depiction of MIRAGE. Following MIRAGE, a few recent schemes like [43] and [4] have been built on different variants of the pseudo fully associative design principle.

2.3 Attacks on Secured Caches

The classical method of generating eviction set occupies a large portion of the cache as a starting point and reduces the eviction set by eliminating addresses one at a time. This method requires $O(n^2)$ accesses [23]. Recent advances in set-eviction creation algorithms however exploit group elimination of cache-lines in $O(n)$ accesses. These attacks [32, 45] trivially break the early randomisation approaches like [31]. Randomisation-with-Partition caches resist these attacks as the degree of non-determinism increases with partitioning as the address-to-set mapping depends on both the randomisation scheme and the selected skew. However, recent attacks like [8, 29, 40] show faster eviction set generation techniques by observing the cache-lines evicted by the victim and probabilistically eliminating non-conflicting cachelines. *Rekeying* is a technique that could alleviate the problem but it imposes a significant performance overhead, and the optimal key refreshment interval is difficult to determine. In addition, some randomization designs suffer from cryptanalytic flaws due to improper implementation and design [5, 29].

3 Performance Analysis of Randomized Cache Designs under Spurious Occupancy

We now investigate and compare the performance of different randomized cache designs. While such comparisons are already performed in individual papers, there exist two main problems. First, different works consider different benchmarks, making it difficult to evaluate *all* designs against a common baseline. Secondly, certain cache designs make inherent assumptions in their implementation which contemporary benchmarking strategies violate. This complicates getting a realistic view of the *actual* performance of these designs.

To make a fair comparative evaluation, we begin by describing the benchmarking strategy employed by each cache design under consideration- ① baseline set-associative, ② CEASER, ③ CEASER-S, ④ ScatterCache, ⑤ MIRAGE, and ⑥ SassCache. We then discuss how the implementation of MIRAGE in particular makes assumptions that are not satisfied by benchmarks like SPEC2017. We then propose a benchmarking strategy- *benchmarking under spurious occupancy*- that removes this implementation assumption of MIRAGE and provides a fair baseline for each cache design. Using this strategy, we provide performance evaluation of all cache designs against SPEC2017. Finally, we also evaluate the performance of the aforementioned designs under five different replacement policies- RandomRP, TreePLRURP, WeightedLRURP, RRIPRP, and FIFORP.

Design	Platform	Benchmark	LLC size
CEASER	pin	SPEC 2006, GAP	8 MB
ScatterCache	gem5	GAP, MiBench, lmbench, scimark2	512 KB / 2 MB
ScatterCache	Custom simulator	SPEC 2017	512 KB / 2 MB
MIRAGE	gem5	SPEC 2006	8 MB
SassCache	gem5	GAP, MiBench, lmbench, scimark2	1 MB
SassCache	Custom simulator	SPEC 2017	1 MB

Table 1: Comparison of simulation platforms and benchmark suites used to evaluate performance of different cache designs.

3.1 Contemporary Benchmarking Strategy

Contemporary benchmarking strategy involves defining *cache configurations* for a given design, and then measuring slowdown with respect to a given benchmark suite. Such cache configuration involves defining overall LLC size, number of ways, choice of randomization function (and its associated latency), L1 size, associativity, number of skews (wherever applicable), and other factors. The configurations are then simulated, and performance statistics are recorded for the considered benchmarks. Usually, one simulation comprises a single benchmark run. In some cases (like gem5), benchmarking may involve copying the simulation over multiple cores to account for process-level fluctuations in measurements.

Table 1 summarizes the different cache designs, the benchmark suites evaluated, and the simulation platform. For want of conciseness, we only capture the LLC size and omit other factors considered in the respective cache configurations. It is immediately clear that a *common* baseline for evaluating these cache designs is not available. We thus resort to the following benchmarking strategy (hereafter referred as **B1**) in this work for a fair and comparative evaluation of these designs:

- **Platform**: gem5
- **Workload**: SPEC2017
- **L1 size**: L1d : 512 KB, L1i : 32 KB
- **LLC size**: 16 MB
- **LLC replacement policy** (wherever applicable): random replacement
- **Skews** (wherever applicable): 2
- **Encryption latency**: 3
- **CPU model**: TimingSimpleCPU

The gem5 implementation of MIRAGE and ScatterCache are open-sourced as part of [37][6]. For CEASER and SassCache[7], we created in-house implementations in gem5. We parameterize the index derivation function such that SassCache's coverage is at (recommended) 63% [14]. Finally, we consider all rate benchmarks of SPEC2017 since they are better suited to estimating system *throughput* than speed benchmarks. Throughput is directly affected by LLC performance: more LLC misses incur higher miss penalties and directly affect the throughput. Our choice of CPU model is not out-of-order, thereby giving us a clear correlation between higher LLC miss penalties and decreased system throughput.

3.2 Assumptions on Distribution of Memory Accesses across Benchmark Lifetime

Before we present our results, we first discuss an additional assumption in the implementation[8] of MIRAGE which is not satisfied by general programs (including SPEC2017). This is crucial as the said assumption has performance implications which need accounting to ensure fairness.

The goal of MIRAGE [37] is to attain a negligible probability of creation of eviction sets for mounting contention based attacks. On every cache miss, **two types of evictions** are possible in MIRAGE by its specification in [37]. If the incoming address finds an invalid tag, a candidate is selected *at random* from the entire data store for eviction and a *reverse-pointer* based invalidation of the corresponding tag entry is performed. This is termed as *global eviction* (or GLE). In contrast, if no invalid entry is found in the calculated sets in both the skews, a *set associative eviction (SAE)* is performed.

However, on evaluating MIRAGE's artifact, we found an additional possibility for a cache-line install (other than GLE and SAE). Annotated as "`Scenario-A`" in the gem5 implementation, it first checks if the tag to be replaced is invalid and *if there are empty slots in the datastore*. If both the conditions are satisfied, the new incoming address is put in place of the victim tag and one of the empty slots of the data cache. Moreover, to keep track of the empty slots in the large LLC data cache (which is implemented as fully-associative), the gem5 implementation uses a software-based queue named `datarepl_vacant_queue`. We highlight important diversions from the original design proposed in MIRAGE. First, this particular scenario finds no mention in the original MIRAGE paper. Secondly, use of such a software queue is not practical for LLC and is not mentioned in the original paper. In fact, authors in MIRAGE argue that designs like RPCache [46], NewCache [22], HybCache [10] that perform fully-associative table-based look-ups are impractical for LLCs, whereas MIRAGE performs random replacement of data from data-cache for every *invalid* tag replacement. Moving forward, we found the other two scenarios annotated as "`Scenario-B`" that implements SAE and "`Scenario-C`" that implements actual GLE were as per the original MIRAGE design, as proposed in the paper [37]. For each benchmark in SPEC2006, MIRAGE fast-forwards execution by 10 billion instructions before beginning the actual benchmarking.

[6] The artifact-evaluated gem5 implementation is available at http://github.com/gururaj-s/MIRAGE.

[7] The authors of SassCache generously provided us with their custom simulator for SassCache, which we ported to our setting in gem5: https://anonymous.4open.science/r/randomized_caches-112E/

[8] The artifact-evaluated gem5 implementation is available at http://github.com/gururaj-s/MIRAGE.

The intention is to allow the *first* 10 billion instructions to initialize the micro-architecture state of the LLC (i.e. exhaust all "Scenario-A" executions), post which MIRAGE's performance can be clearly deduced[9].

However, this approach has an inherent assumption: *the distribution of **load/store** instructions across the lifetime of every benchmark under consideration is uniform*. If this assumption is indeed true, then fast-forwarding instructions that exhaust "Scenario-A", and then performing the benchmarking is a sound strategy. However, if distribution of **load/store** across the lifetime of the benchmark is non-uniform, then *using the initial portion of the benchmark to exhaust "Scenario-A" is an unfair strategy*. Worse, if the **load**s/**store**s are clustered during the initial stages of the benchmark, then such a strategy implies that majority of **load**s/**store**s are never evaluated under GLE ("Scenario-B") or SAE ("Scenario-C"). Our experiments captured in Appendix A establish that this assumption is unfounded, leading us to make the following observation:

Takeaway: MIRAGE's gem5 implementation assumes uniform distribution of **load**s/**store**s across the lifetime of the benchmark. Not all SPEC2017 workloads (and by extension, not all user-defined custom workloads) satisfy this criterion. MIRAGE thus provides an optimistic view of the performance overhead due to global evictions (GLE).

3.3 Benchmarking under Spurious Occupancy

Given the discussion in Sec. 3.2, performing a comparative performance evaluation of all cache designs under benchmarking strategy **B1** is unfair as **B1** provides an optimistic view on MIRAGE's actual performance. We thus modify **B1** to a new strategy: Benchmarking under Spurious Occupancy, which adds *spurious* (random) **load**s/**store**s to fill the entire LLC before the actual benchmark execution begins[10], and ensures fairness in our evaluations. Our modified strategy (marked in blue) of benchmarking under spurious occupancy (hereafter referred to as **B2**) is as follows:

- **Platform**: gem5
- **Workload**: Spurious occupancy followed by SPEC2017
- **L1 size**: L1d : 512 KB, L1i : 32 KB
- **LLC size**: 16 MB
- **LLC replacement policy** (wherever applicable): random replacement
- **Skews** (wherever applicable): 2
- **Encryption latency**: 3
- **CPU model**: TimingSimpleCPU

[9] https://github.com/gururaj-s/mirage#steps-for-gem5-evaluation

[10] For MIRAGE in specific, this strategy ensures that the *spurious* **load**s/**store**s are used to exhaust "Scenario-A", while *every* **load/store** in the actual benchmark is vulnerable to GLE ("Scenario-C"). Strategy **B2** thus negates any advantages MIRAGE gains from "Scenario-A" executions.

Note that **B2** is followed for *all* cache design to ensure fairness. We choose the number of LLC misses as our metric for this evaluation[11]. Fig. 5 captures the results of our evaluation. We follow the default replacement policy (RandomRP) for all cache designs under consideration.

Policy	CEASER	CEASER-S	MIRAGE	ScatterCache	SassCache
RandomRP	+6.7%	+1.061%	−0.02%	+1.064%	+13.86%
TreePLRURP	+10.31%	+9.1489%	+4.00%	+9.1485%	N/A
WeightedLRURP	+15.78%	−4.3028%	+5.03%	−4.3027%	N/A
RRIPRP	+15.58%	+0.535750%	+6.42%	+0.535757%	+57.45%
FIFORP	+13.10%	−1.3548%	+8.22%	−1.3546%	+69.21%

Table 2: Comparison of different cache designs against evaluated replacement policies through reported *averages of LLC misses* on SPEC2017, normalized against baseline set-associative cache, and expressed as a percentage. Performance statistics are averaged over 300 copies of SPEC2017 runs. A positive $+x\%$ (alt. $-x\%$) implies the design is $x\%$ slower (alt. faster) than baseline set-associative cache.

3.4 Effect of Replacement Policies

One important factor in a set-associative cache design is the choice of replacement policy invoked during set-associative eviction. Other than RandomRP, we hence conduct performance evaluation of SPEC2017 under other replacement policies: TreePLRURP, WeightedLRURP, RRIPRP, and FIFORP. Our choice of replacement policies is motivated by our intention of covering different *rationales* of replacement policy design in our evaluation. For instance, TreePLRURP and WeightedLRURP are variants of Least Recently Used policy (LRU) that use specialized structures like binary trees and 1-bit pointers to decide candidates for eviction. In complete contrast, RRIPRP is a variant of *not* recently used policy, and uses a re-reference prediction value to estimate whether a cache block will be used in near-future (which educates whether the replacement policy chooses the block for eviction). Finally, FIFORP is textbook first-in, first-out replacement.

Our results are summarized in Tab. 2[12]. Note that the gem5 implementation of SassCache does not support TreePLRURP and WeightedLRURP. Also note that in all experiments, among all designs, MIRAGE has no effect of the policy because of its practically negligible occurrences of set-associative eviction. The variations observed for the the normalized score of MIRAGE stem for varying LLC misses for baseline set-associative, and not from variations in misses for MIRAGE. We summarize the **main takeaways**: (1) For default replacement policy RandomRP, MIRAGE, CEASER-S, and ScatterCache perform almost comparable wrt. baseline set-associative cache.

[11] This is a reasonable choice since increased LLC misses directly correlate with slowdown in execution time, since each LLC miss incurs a huge miss penalty in being serviced by the memory controller.

[12] We refer to the full version of this work at https://arxiv.org/pdf/2310.05172 for comparison at benchmark granularity.

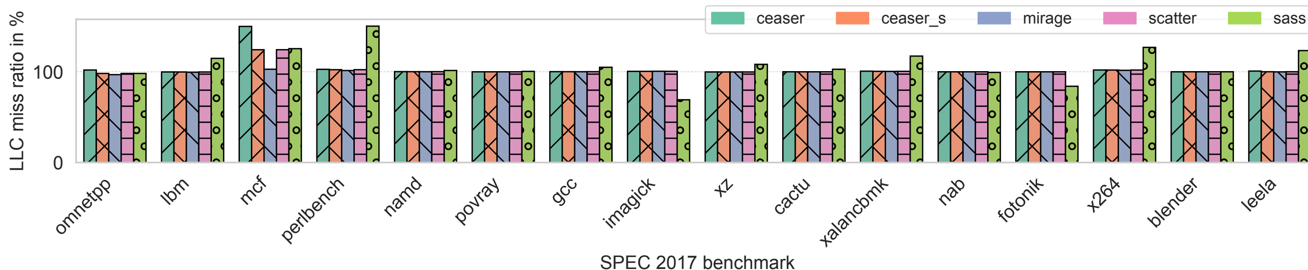


Figure 5: Performance evaluation of considered cache designs with RandomRP replacement policy (normalized against baseline set-associative, and expressed as a %). Performance statistics are averaged over 300 copies of SPEC2017 runs.

(2) For WeightedLRURPand FIFORP, CEASER-S and ScatterCache outperform baseline set-associative cache considerably. WeightedLRURP and FIFORP are the only instances where choice of a replacement policy allows randomized cache designs to *outperform* baseline set-associative cache. In other words, WeightedLRURPand FIFORP are the only instances of replacement policies where *randomized* caches trump baseline set-associative caches in *both* performance as well as resilience against contention based attacks.

(3) MIRAGE is comparable to baseline set-associative cache *only* for RandomRP. Because of practically no set-associative evictions, there is no perceivable effect of changes in replacement policies on MIRAGE. Hence, when baseline set-associative cache takes advantage of replacement policies other than RandomRP, MIRAGE performs much worse.

(4) CEASER and SassCache perform considerably worse than all their counterparts on *all* evaluations.

(5) There is *no* cache design which, for *any* of the replacement policies, trumps baseline set-associative cache in *all* of the following parameters: ① better performance, ② resilience against contention based cache attacks, and ③ resilience against occupancy based attacks (cf. Sec. 4).

We attribute the poor performance of CEASER to its lack of skews; this observation is helped by the fact that performance of CEASER does not improve with changing replacement policies. Likewise, we attribute the poor performance of SassCache to its design choice of security-domain based isolation of LLC occupancy which limits occupancy for a benchmark even if it has genuine requirement of LLC occupancy[13]. As with CEASER, we observe no considerable improvement in SassCache even on varying replacement policies.

[13] As the next sections establish, it is this same design choice that allows SassCache to resist our occupancy based attack vectors, thereby establishing an interesting perspective where a design decision leads to *both* worse performance but increased security wrt. contention *and* occupancy attacks.

Takeaway: Benchmarking in contemporary literature is extremely diverse (wrt. choice of platform, choice of benchmarking suite, LLC configuration etc). Once we harmonize all cache designs to a uniform benchmarking strategy, we have an interesting insight: each cache design has differing affinities for replacement policies. Concretely, while most prior works are evaluated mostly with random replacement, we show that different cache designs have varied effects of replacement policies. In some cases, randomized caches actually outperform baseline set-associative cache.

4 Systematic Evaluation of Randomized Cache Designs against Cache Occupancy Attacks

An important criterion for contention-based attacks like `Prime+Probe`, `Evict+Time` and similar variants, is that the attacker must fill all the ways in the target cache set to leak information about the victim. However, a separate class of cache attacks exist in literature that do not depend on the granularity of extracted cache set information. These attacks, broadly called *cache occupancy attacks*, focus on the cache footprint of the victim in terms of cache occupancy, rather than relying on activity within specific cache sets. Such attacks are powerful as they have been used in literature to perform website fingerprinting [38, 39] through javascript-supported browsers as well as covert channels for Spectre [44]. For victim processes that leave memory footprint in terms of cache occupancy depending on some secret bit, an attacker can recover such information simply by filling up the cache and observing the cache occupancy. Earlier attacks like [9] constructed covert channels using L1 cache occupancy whereas [35] showed how occupancy could even detect keystroke timing and network load in co-located virtual machines on cloud servers. More recently, the authors in [34] showed that cache occupancy attacks can be used to differentiate between secret keys of ciphers such as AES and RSA [14]. However, one of the major requirements, which in fact is a drawback of such an attack, is

[14] In this paper, we consider a stronger adversarial attack: key recovery on AES rather than just distinguishing keys.

that the attacker process needs to fill up a significant portion of the cache in order to enforce cache occupancy channel. Previous works [19, 24, 25] that demonstrated covert channels and side channels using cache occupancy techniques have noisy data transfer and can be easily detected by the operating system or watchdog applications when a particular program routinely occupies a large portion of the cache.

With the sole exception of SassCache, cache occupancy attacks are considered outside the threat model of state-of-the-art randomized cache designs. In this work, we reason that omitting cache occupancy attacks from design considerations of randomized caches is not prudent. Recall that design rationales for randomized caches aim to attain a uniform distribution over all LLC sets for every cache line install. While a uniform distribution of address-to-set mapping helps defending against contention attacks, on the other hand, it amplifies opportunities for a cache occupancy attack simply because every set in the LLC has a *higher* probability of being selected for a cache-line install (compared to baseline set-associative cache). Formally, we ask the following question:

In an attempt to achieve uniform distribution over the sets for a cache-line install, do existing randomized cache designs inadvertently end up ***amplifying*** *cache occupancy attacks?*

Only SassCache claims to defend against *both* contention attacks and occupancy attacks. SassCache sacrifices achieving a uniform distribution over *all* sets of the LLC in favor of achieving uniformity over a *subset* of the LLC. Every security domain is assigned such a *subset*, and there is little overlap between two subsets. As a result, even though a given set in SassCache has a higher probability (compared to that of baseline set-associative cache) of being picked for address-to-set mapping, by not allowing the *subsets* to significantly overlap, it ensures security-domain based isolation of LLC occupancy. Thereby it evades both attacks at the same time.

We evaluate the randomized cache designs under considerations under three different threat assumptions, ordered from strongest to weakest:

- **Covert Channel**: This use-case has the strongest assumption: two processes *collude* to covertly transmit a bit-stream over a LLC occupancy channel. It is well understood that occupancy covert channels can be created over baseline set-associative caches too. Our objective is to rather understand whether randomized caches make it *easier* to do so.

- **Process Fingerprinting**: This use-case is a cache-occupancy based side-channel where an attacker attempts to profile victim execution.

- **Key Recovery on AES**: This use-case has the weakest assumption and the most difficult objective: from a cache occupancy side-channel, an attacker attempts to perform key recovery of AES T-Table victim. Prior work like [12, 34] has shown cache occupancy side-channel to *distinguish* between two AES keys. Full key recovery was not an objective of their attack.

Our consideration of varied threat assumptions is to establish the potency of cache occupancy attacks, as well as to demonstrate the *amplification* of such attacks when combined with the design rationale of certain randomized caches.

5 Case 1: Covert Channel

We consider two non-privileged processes installing addresses in the cachelines by accessing different memory locations belonging to their own address space. In line with assumptions of prior attacks on randomized caches [7, 8, 29, 40], we assume the randomized LLC is shared between the two processes. We also assume that the randomization function is pseudo-random and cryptanalytically secure.

We use a Python-based functional LLC simulator for the covert channel. We configure MIRAGE with 16MB size and $8+6$ ways in each tag set with 2 skews. Each incoming address is randomized using PRESENT [6] block cipher with two fixed and distinct keys. Likewise, for ScatterCache we use 16MB cache with 2 partitions and 4 ways in each partition[15]. Similar configurations were also maintained for CEASER and CEASER-S, with CEASER-S having two partitions (skews)[16]. We assume random replacement policy for all the cache designs. The adversary only uses `rdtsc`-based timing measurements on its own memory accesses to determine cache hits and misses.

5.1 Channel Setup

We describe the channel setup to transmit a single bit. For transmitting an entire bit-string, we repeat this setup over the length of the bit-string.

➊ **Receiver:** Allocates sufficient memory space and randomly accesses ℓ number of addresses. In order to ensure that each address occupies an entire cacheline, the addresses are accessed in a sequential manner with an interval of $1,000$ (without loss of generality) between each address. For example, if the first address accessed is p, then the next address to be accessed will be $1000+p$. This ensures that none of the addresses experiences a cache hit.

➋ **Sender:** The sender is assumed to be executing on a separate physical core, completely agnostic of the addresses accessed by the receiver. The sender allocates a memory range and depending on the value of the bit (0 or 1) it wants to transmit, either makes x or y memory accesses. Additionally,

[15] To make the comparison fair, we configure ScatterCache and MIRAGE with 2 partitions/skews and total 8 (4 in each partition for ScatterCache) ways available for each incoming address.

[16] We skip SassCache because its non-overlapping, security domain dependent LLC subsets make it improbable to setup the covert channel.

the addresses accessed are at an interval of $1,000$ (wlog.) to ensure no two accesses are mapped to the same cache-line.

❸ **Receiver:** Once the sender has completed its operation, the receiver re-accesses all the ℓ addresses it has previously installed in the cache. It is worth mentioning that the receiver, at this point, does not have any knowledge about the number of its own addresses being evicted from the cache. Nor does it know the number of addresses from the sender installed presently in the cache. The receiver re-accesses the addresses and measures the access times for each of them individually. The ones that are still in the cache will experience a cache hit, whereas the ones evicted due to contention with the sender addresses will experience a cache miss.

For synchronization between the processes, a busy wait without any memory accesses is sufficient. The sender sleeps for the time duration it takes the receiver to access l addresses. The receiver sleeps for the time duration it takes the sender to access the maximum of two accesses (i.e. $\mathtt{max}(x, y)$).

5.2 Evaluation: Different Occupancy Rates

We perform a characterization of cache occupancy channel on all variants (under consideration) to estimate the optimal occupancy amount for each variant. We vary the number of addresses accessed by the receiver to install in the cache such that the cache occupancy rate covers the entire range of 1% to 40%. For simplicity, we fix the number of sender accesses to $x = 1000$ for transmitting '0' and $y = 2000$ for transmitting '1'. The receiver only considers number of misses observed in its occupancy. For a given % of cache occupancy, if the number of misses observed by the receiver are statistically distinguishable between transmission of '0' or '1', we consider the channel to be successfully established.

Fig. 6a and Fig. 6b show the comparative analysis of all variants of caches under consideration with respect to cache occupancy attack on a 16MB cache. Each data-point reported is averaged over 100 runs of same experiment. For baseline cache, the receiver requires at least $\ell = 30,000$ accesses, while for ScatterCache it requires around $\ell = 40,000$ accesses to observe a statistically significant difference in the number of cache misses between x and y accesses from sender (cf. Fig. 6a). For CEASER-S, the receiver requires around $\ell = 40,000$ accesses, which is expected as the design principle of CEASER-S is similar to ScatterCache; whereas, for CEASER, the receiver requires more than $\ell = 40,000$ accesses (cf. Fig. 6b). The higher resilience of CEASER could be attributed to the monolithic design approach undertaken by the designers which requires the receiver to fill up all the ways (8 in our case) in any set for the sender to successfully evict one of the entries. Furthermore, one can observe that the number of misses corresponding to two access patterns (signifying '0' and '1') for MIRAGE start diverging at $10k$ accesses while for both baseline and ScatterCache it's much higher. Note that we were not able to observe any misses for baseline, ScatterCache, CEASER and CEASER-S till $25k$ accesses.

5.3 Evaluation: Fixed *low* Occupancy Rate

It is clear than pseudo fully associative design rationale (i.e. MIRAGE) allows a statistically distinguishable distribution of cache misses between "0" and "1" in much *lesser* receiver accesses than other designs based upon set-associative design rationale. To investigate this further, we repeat the previous experiment, but with a deliberately fixed *low* LLC occupancy rate. Conservatively, we fix this rate at 10%; refer Fig. 6a where MIRAGE shows signs of divergence at this rate. We thus fix $10k$ accesses from the receiver and and $3k$ accesses from sender. As before, the receiver first makes $\ell = 10,000$ memory accesses, installing them in the cache and then waits for the sender to make its own accesses. All evaluation numbers are reported as averages over 100 repetitions of the experiment.

Observations for '0': To send bit '0', the sender makes $x = 1000$ memory accesses. For MIRAGE we observed that around 90 addresses of the receiver that were already installed get evicted due to random replacement in the global data-store. The receiver, when re-accessing its own addresses, observes cache misses (higher latency) for around 490 of them[17]. Note that we did *not* observe any cache misses for the receiver in CEASER, CEASER-S, ScatterCache and baseline cache.

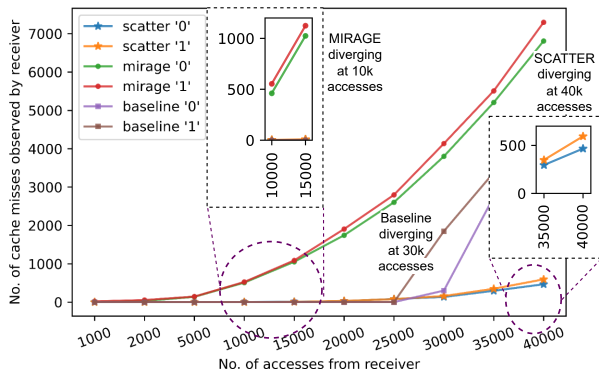
Observations for '1': To send bit '1', the sender makes $y = 2000$ memory accesses. In the case of MIRAGE we observed that 120 addresses of the receiver are evicted due to this operation. The receiver now re-accesses all its $10,000$ addresses and observed cache misses for around 540 of them. However, for other variants such as CEASER, CEASER-S, ScatterCache and baseline cache, we did not observe any misses from the receiver side.

Takeaway: In the context of sender's accesses evicting a statistically significant portion of receiver's accesses, set-associative based design principles such as CEASER, CEASER-S, ScatterCache show higher resilience than fully associative cache based designs (like MIRAGE). In other words, while *all* design variants show statistically distinguishable distributions for "0" and "1", pseudo fully associative design principle requires the *least* occupancy among all designs under consideration, making it more vulnerable than baseline set associative cache wrt. occupancy attacks.

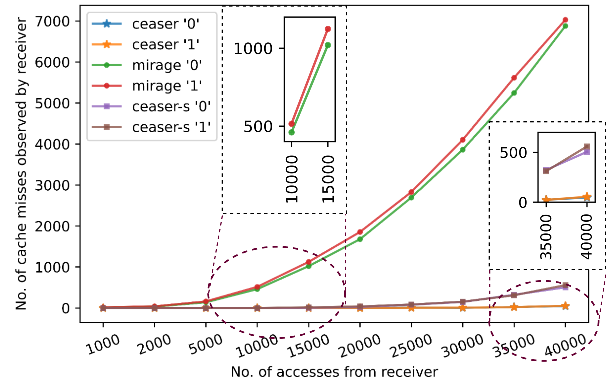
5.4 Root Cause

Prior work [8, 30, 37] has established that pseudo-fully associative design rationale like MIRAGE is more resilient wrt.

[17] Note that while the operation of the sender amounts of 90 evictions, we observe far higher (i.e. 490) cache misses. We attribute this to *self-evictions*: for any two cache-lines $i, j \in \{1,2,3,\cdots,l\} : j > i$ installed by the receiver, the installation of line j evicts line i due to global eviction.



(a) Comparative analysis of cache occupancy channel in MIRAGE, ScatterCache and baseline set-associative cache.



(b) Comparative analysis of cache occupancy channel in MIRAGE, CEASER and CEASER-S cache designs.

Figure 6: Comparative analysis of cache occupancy based covert channel in different cache designs.

contention-based attacks compared to other contemporary designs like ScatterCache, CEASER-S, etc. Our analysis, on the other extreme, shows that the same psuedo-fully associative design principle makes MIRAGE *more* vulnerable to occupancy attacks than other variants. We now analyze the root cause of this observation and explore further this amplification of cache occupancy attacks on MIRAGE.

The rationale behind pseudo-fully associative design principle is to balance between security of fully associative design and efficiency of set-associative design. MIRAGE's design is to decouple the LLC into *set-associative tag-store* and *fully associative data store* and to provide extra invalid tags in the tag store. MIRAGE uses a random eviction policy for all entries in the data store, which is termed as *global eviction*. Therefore, any valid data entry from the data store can get evicted. This is in contrast to fully associative caches, where valid entries can be evicted only when all invalid and free blocks have been exhausted. Likewise, this is also different from set-associative caches, where valid entries are evicted only when there is no invalid *way* for the concerned *set*.

Also note that such global eviction happens with very high probability for each new cache-line install, since the chances of getting set associative eviction is negligible [37]. In other words, any incoming address is accompanied with a random eviction from the data store, which is again followed by a reverse-pointer (RPTR)-based invalidation of the corresponding entry in the tag store. As the overall occupancy of the LLC for a process grows, the chances that valid addresses from other processes already installed in the cache will get evicted also increase. For other caches, including traditional and randomized designs (e.g. ScatterCache, CEASER-S) that are designed based on the set-associative cache primitive, eviction occurs only when all the available ways in a particular set get occupied with valid entries.

Takeaway: MIRAGE, due to its policy of compulsory eviction from the data store for every new address installation, ensures a *higher* probability of evicting a valid cache-line than any other design under consideration. As such, it becomes more vulnerable to cache occupancy attacks than other designs and even the traditional set-associative cache.

Such *higher* probability of evicting a valid cache-line can be further exploited to improve the bandwidth of the covert channel even with low LLC occupancy like 10%. We defer to Appendix B for details.

6 Case 2: Process fingerprinting

Our results on covert channel experiments make it clear that *all* cache design variants are vulnerable to cache occupancy attacks, with MIRAGE being the most vulnerable. In this section, we extend this result to *template-based fingerprinting of processes.* The attack rationale is that different processes exhibit different number of **load**s/**store**s during their lifetime. As demonstrated in Sec. 5, such differences in the number of memory accesses exert a statistically distinguishable influence in the cache hit/miss ratio in the LLC occupancy of an attacker, and leading to process fingerprinting.

We use gem5 for this evaluation. Our gem5 setup is similar to the setup followed for benchmarking strategy **B2**:
- **Platform**: gem5
- **Workload**: Spurious occupancy followed by SPEC2017
- **L1 size**: L1d : 512 KB, L1i : 32 KB
- **LLC size**: 16 MB
- **LLC replacement policy** (wherever applicable): random replacement
- **Skews** (wherever applicable): 2
- **Encryption latency**: 3
- **CPU model**: TimingSimpleCPU

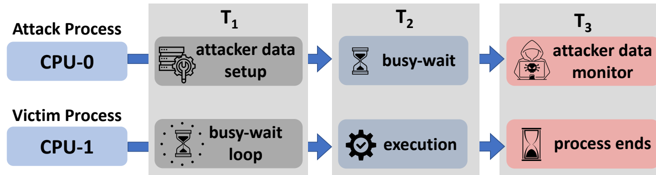


Figure 7: Schematic of experimental setup used for fingerprinting attacks on SPEC2017 workloads. Phases T_1, T_2, and T_3 ensure synchronization between (otherwise independent) victim and attacker.

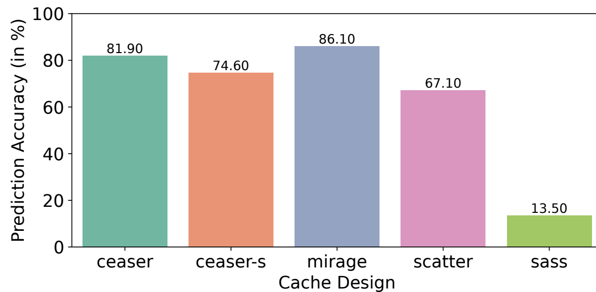


Figure 8: Comparative analysis of fingerprinting accuracy across different cache designs across 500 predictions of randomly sampled workloads from SPEC2017.

In order to launch the attack, we create two process binaries: *attacker process* and *victim workload*, as depicted in Fig. 7. The simulation is run with 2 CPUs, with each process bound to one CPU. This ensures sharing of the LLC and a realistic attack setting. Moreover, in order to ensure synchronization, we use busy-wait loops enabled by use of `rdtsc` instructions. Note that *use of busy-wait loops in the victim workload is not a violation of our threat model*. In a realistic system, the micro-architectural state of the LLC would be correctly initialized, allowing the attack to launch instantaneously (i.e. not requiring any warm-up phase, and thereby no busy-wait in the victim). In gem5, however, the synchronization between the attacker and the victim is not possible, as the simulation assigns and simulates both processes on parallel cores from the very first tick of the simulation. Hence, we use *busy-wait* on victim process to force synchronization. On a realistic system, the assumption that the *attacker can trigger the victim* is made to ensure synchronization. This assumption is in line with other works in literature on similar attack vectors.

The fingerprinting attack proceeds in two phases: offline and online phase. In the offline phase, the *attacker process* on CPU-0 creates templates for different workloads by observing LLC misses (observed through gem5 metric `system.l2.overall_misses::.cpu0.data`) for 15% LLC occupancy. In the online phase, it uses these templates to predict the workload being executed on CPU-1. Fig. 8 captures the relative accuracies of different cache designs for 500 predictions of workloads randomly sampled from SPEC2017.

Takeaway: With respect to process fingerprinting attack through cache occupancy, only SassCache exhibits a remarkably less prediction accuracy, implying no statistically significant adversarial advantage. All other cache designs exhibit vulnerability, with MIRAGE empirically being highly vulnerable, and ScatterCache being (comparatively) less vulnerable.

We attribute the resilience of SassCache to our fingerprinting attack to its design decision of security-domain isolated subseting of LLC. Since the amount of *overlap* between the two processes in Fig. 7 is bounded, there is limited leakage of victim workload through attacker's cache occupancy channel.

7 Case 3: Key recovery on AES

We now detail our experiments wrt. using cache occupancy side-channel to recover AES-128 bit key. While prior work [12, 34] has demonstrated the vulnerability of randomized caches to occupancy attacks by distinguishing between two keys, but they do not report key recovery which would be expected for a practical attack.

7.1 Attack Description

We use gem5 for this evaluation. Our gem5 setup is similar to the setup followed for benchmarking strategy **B2**) in Sec. 3.3:
- **Platform**: gem5
- **Workload**: Spurious occupancy followed by **AES victim**
- **L1 size**: L1d : 512 KB, L1i : 32 KB
- **LLC size**: 16 MB
- **LLC replacement policy** (wherever applicable): random replacement
- **Skews** (wherever applicable): 2
- **Encryption latency**: 3
- **CPU model**: TimingSimpleCPU

Our **AES victim**, like [12, 34], is an OpenSSL styled T-Table implementation. It accepts a randomly generated plaintext, and for a fixed secret key, performs a *single* encryption and exits. Our **attacker** is a novel variant of cache timing attacks on AES T-Table implementations [3], where instead of timing victim AES execution, we rather time the accesses to an attacker's LLC occupancy. Algo. 1 (in Appendix) summarizes the sequence of operations performed by the attacker, which we now explain:

1. The attacker first lets the AES victim setup its secret key, and precompute T-Tables.
2. For the same reasons as detailed in Sec. 3.2, before the experiment begins, the attacker fills the LLC with spurious occupancy. These memory accesses are *never* re-accessed again during the course of the actual attack.

This step also ensures that the AES victim T-Tables are reliably flushed from the LLC.

3. Given a fixed occupancy $X\%$, the attacker `malloc`s about $\frac{16\times X}{100}$ MB of memory, and repeatedly accesses it to ensure occupancy of complete L1d and $X\%$ of LLC, since LLC is inclusive cache[18].
4. Given a randomly generated plaintext P, the attacker then lets AES victim run a *single* encryption of P, and obtains the ciphertext C.
5. Finally, the attacker uses `rdtsc` to time access to its previously allocated $\frac{16\times X}{100}$ MB of memory. Call it T.

A single observation point for the attacker is the tuple (P,C,T): the time T taken to access its *own* cache occupancy *after* the AES victim operates upon plaintext P to generate C, for an **unknown key** K[19]. The adversary repeats the aforementioned steps and obtains an observation trace $O = \{(P_1,C_1,T_1),(P_2,C_2,T_2),(P_3,C_3,T_3),\cdots,(P_{|O|},C_{|O|},T_{|O|})\}$. Thereafter, the attacker *repeats* the same steps as above, but rather than invoking the AES victim, it *simulates* AES T-Table encryption for a **known key** K^*. It builds another observation trace $O^* = \{(P^*_1,C^*_1,T^*_1),(P^*_2,C^*_2,T^*_2),(P^*_3,C^*_3,T^*_3),\cdots,(P^*_{|O^*|},C^*_{|O^*|},T^*_{|O^*|})\}$. Recovery of the **unknown key** K^* then proceeds as follows:

1. For each $(P^*_i,C^*_i,T^*_i) \in O^*$, the attacker first computes $X_i = \texttt{InvSBox}(C^*_i \oplus K^*)$ to derive the output of the `SBox` operation in the last round of AES. The adversary thus has $X = \{X_1,X_2,\cdots,X_{|O^*|}\}$.
2. For each key byte $\{b_j \in K^* : 0 \le j \le 15\}$, the attacker creates a dictionary $\mathcal{T}^*_j$, where labels range from $\{0,1,2,\cdots,255\}$. Against each label $l \in \{0,1,2,\cdots,255\}$, $\mathcal{T}^*_j[l]$ stores the *mean* of all timing values $\{T^*_{(1)},T^*_{(2)},T^*_{(3)},\cdots\}$ for which the corresponding entry in X equals l. At the end of this phase, the attacker has 16 dictionaries $\{\mathcal{T}^*_0,\mathcal{T}^*_1,\mathcal{T}^*_2,\mathcal{T}^*_3,\cdots,\mathcal{T}^*_{15}\}$.
3. For the observation trace O for the **unknown key** K, the attacker follows a similar procedure as in ②. The only difference is that since the key is **unknown**, the attacker makes *guesses* for *each* of the key byte. This leads to 256 templates for *each* key byte, implying the total number of templates created by the adversary is (256×16). Call this template set: $\{\mathcal{T}^i_j : 0 \le i \le 15, 0 \le j \le 255\}$.
4. Finally, for each key byte index $i \in \{0,1,2,\cdots,15\}$, the attacker correlates $\{\mathcal{T}^i_0,\mathcal{T}^i_1,\mathcal{T}^i_2,\mathcal{T}^i_3,\mathcal{T}^i_4,\cdots,\mathcal{T}^i_{255}\}$ with $\mathcal{T}^*_i$. At this phase, the attacker sorts the key guesses based on the achieved correlation, and outputs the *ranks* of the correct key bytes as $R = \{R_0,R_1,R_2,\cdots,R_{15}\}$.

[18]Most cache designs (including the ones we evaluate here) consider an inclusive cache-hierarchy.

[19]Without loss of generality, we target recovery of last round AES key. Keys of other rounds can be recovered through the AES key schedule once last round key is recovered.

A successful key recovery is achieved when all key byte ranks are 1. However, a better indicator of whether a side-channel leaks is the guessing entropy [20]:

$$\text{GE} = \Sigma_{i=0}^{15} log_2(R_i)$$

Lower GE implies the key bytes are lower ranked post correlation. In practice, achieving a GE lesser than a certain pre-defined threshold is sufficient to declare an implementation vulnerable to successful key recovery with reasonable computational complexity [20, 41]. Usually a GE of about 32 is considered to be the threshold [20]; it implies the correct key bytes (on average) are among the top 4 guessed key bytes by the adversary. A side-channel achieving a GE of 32 can reliably converge to GE = 0 by increasing the number of observations [20]. Furthermore, even at GE of about 32, the brute force complexity of the key-space has reduced from 2^{128} to 2^{32}, which is tractable on modern systems (refer Sec. 7.4).

We now define parameters for estimating *adversarial complexity* for mounting this attack. This complexity can be influenced by adjusting either of the following parameters:

- **% of LLC that the adversary is allowed to occupy**: A higher occupancy makes it easier to observe activity of the victim program (cf. Sec. 6), but is easier to detect by defence mechanisms.
- **Number of observations that the adversary is allowed to measure**: Note that the success of the attack higher depends upon $\mid O \mid$ and $\mid O^* \mid$. Higher $\mid O \mid$ and $\mid O^* \mid$ imply better chances at convergence, but increases the required time to attack.

7.2 Key Recovery Under Varying Occupancy

We first validate our attack against a varying number of LLC occupancy rates. Since our objective is to just choose an optimal occupancy rate against which we will evaluate *all* designs, we only consider three designs: ScatterCache, MIRAGE, SassCache. We choose ScatterCache as it is the most resilient design among all the set-associative based randomized designs we consider in this work. Likewise, we consider MIRAGE as it allows us to also compare the pseudo-fully associative design rationale against set-associative based designs wrt. our attack. Finally, we consider SassCache in order to validate the extent of its resilience to occupancy attacks.

Fig. 9 captures our findings. First, we note that the guessing entropy of SassCache shows limited improvement with increasing attacker occupancy. This can be attributed to its security-domain isolation of LLC occupancy. Next, ScatterCache does show improvement as the cache occupancy increases, but the Guessing Entropy is not sufficiently low to perform any key recovery. Finally, MIRAGE's Guessing Entropy falls as the attacker occupancy increases, eventually reaching

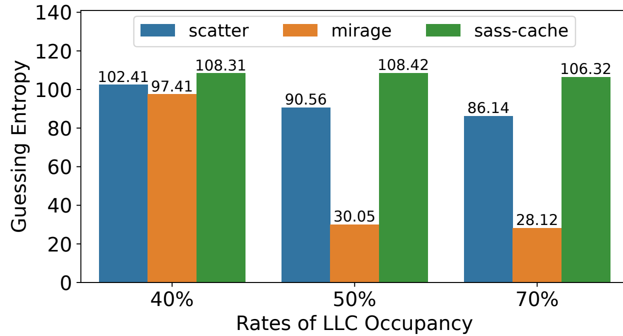


Figure 9: Key Recovery experiment across three occupancy rates: 40%, 50%, 70% for fixed 12000 observations.

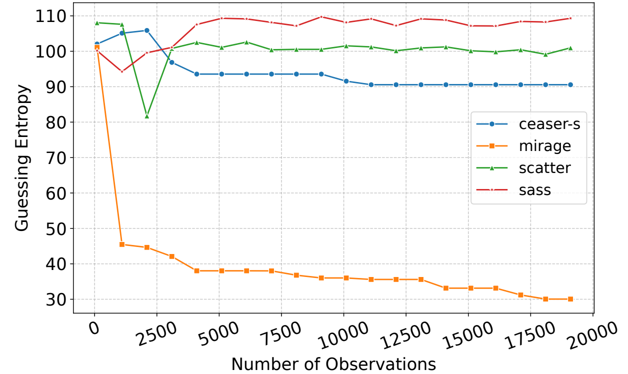


Figure 10: Key Recovery experiment across a fixed occupancy rate: 50%, and for varying number of observations.

the chosen threshold of 32. Note however that we were able to observe leakage only for occupancy rates > 50%[20].

7.3 Key Recovery Under Varying Number of Observations

We now fix the attacker LLC occupancy at a conservative 50%, and evaluate the *trends* in Guessing Entropy against *all* designs considered in this work: CEASER-S, ScatterCache, MIRAGE, and SassCache. We vary the number of observations ($|O|$ and $|O^*|$) the adversary is allowed to make from 100 to 20000[21] and repeat the attack on all cache designs.

Our results are summarized in Fig. 10. First, SassCache never improves even upon increasing the number of observations. Then between CEASER-S and ScatterCache, both improve from their initial value, depicting *some* leakage, but not enough to mount any key recovery attack. Finally, MIRAGE improves significantly, and eventually crosses the GE threshold of 32, implying all secret key bytes are on average ranked within the top 4 key guesses by the adversary.

Takeaway: Comparing between designs based on set-associative caches and designs based on pseudo-fully associative caches, our evaluations demonstrate that the former is more resilient to our AES key recovery attack than the latter. [a] Between the different designs within the class of set-associative based randomized caches, SassCache is the most secure (against all evaluated levels of occupancy), validating its design rationale as the best possible choice to protect against *both* contention and occupancy attacks.

[a] This takeaway is consistent with the takeaways in Sec. 5 and Sec. 6.

[20] We need a higher LLC occupancy than the attacks in Sec. 5 and Sec. 6 needed because victim AES has relatively low memory footprint than the workloads considered in Sec. 5 and Sec. 6.

[21] We scheduled about 350 single-threaded gem5 simulations in parallel, allowing us to collect observations at a rate of 500 observations per hour. Such inhibitory rate is because we work with gem5; on an actual LLC, the rate of collection of observations would be much higher.

7.4 Full Key Recovery

In MIRAGE each key byte is on average among the top 4 ranks, making the brute-force complexity of AES for MIRAGE to be essentially 2^{32}. We can simply brute force the secret key at this point. We deploy four Intel(R) Xeon(R) Gold 6226*R* CPU servers (running Ubuntu 20.04.6 LTS, 32 GB RAM), with a cumulative logical core count of about 350. We parallelize the brute force search across all logical cores, reducing the workload complexity of each logical core to $\frac{2^{32}}{2^8} \approx 2^{24}$. In about 6 hours, we were able to leak the entire secret key.

Note from Fig. 10 that a similar attack is infeasible for CEASER-S and ScatterCache, because although their attack complexity has reduced significantly from 2^{128} (thereby establishing leakage), it is still computationally infeasible to recover the key. However, as with increasing number of observations, the GE shows a downward trend, it is an interesting future problem to estimate the number of adversarial observations required to achieve the same Guessing Entropy in CEASER-S and ScatterCache as we observe with MIRAGE.

8 Discussion and Future Directions

Considerations of cache occupancy in designs of randomized caches have largely been overlooked in contemporary literature. However, the security evaluations we perform in this work (especially related to occupancy-based full key recovery attack on AES) reliably reinforce the need to consider *both* occupancy attacks and contention-based attacks in randomized cache designs.

Our results have a few recurring and novel takeaways. First, ① some set-associative randomized caches (like ScatterCache and CEASER-S) can *outperform* baseline set-associative caches when paired with the correct replacement policies. Prior works only report slowdown in their comparative analysis, since they usually don't consider replacement policies as a parameter in their evaluation. Secondly,

② pseudo-fully associative cache designs like MIRAGE are worse off than designs based on set-associativity wrt. resilience against occupancy attacks. Finally, ③ SassCache resists all our attack vectors. This is expected; SassCache implements security-domain based non-overlapping partitions. As such, each security-domain has a fixed subset of LLC it can occupy for operation; this design decision gives good security against *both* contention-based and occupancy-based attacks. However, our performance evaluation on realistic LLC configurations also establishes that this very design decision is prohibitive in terms of SassCache's performance across all considered replacement policies.

These findings thereby establish an interesting open problem: Design of a randomized cache of comparable efficiency with modern set-associative LLCs, while still resisting *both* contention-based and occupancy-based attacks. We believe the design rationale of dynamic partitioning of the LLC may provide solutions to the questions we raise in this work. Randomized caches designed around dynamically changing partitions would be capable of restricting LLC occupancy for critical workloads (thereby providing security), while also generously allowing LLC occupancy for non-critical workloads (thereby providing better performance in general). More research however is required to concretely establish how such a cache design shall function, how process requests for dynamic changes in partitioning will be handled, and how to implement this design in an uncomplicated manner.

9 Conclusion

The design motivation of state-of-the-art cache randomization schemes is majorly directed towards protecting against contention-based attacks while essentially ignoring other cache attack variants that are equally practical and powerful. In this work, we walk a different route by evaluating the relative resilience of contemporary secured cache schemes against *cache occupancy attacks*. We do so in two verticals: performance and security. For our performance evaluation, we first establish that existing benchmarking strategies provide an unfair perspective on comparative performance (due to varying LLC configurations, choice of benchmarks, choice of simulation platform, implementation-specific assumptions etc). We thus propose a uniform benchmarking strategy for fair evaluation, and evaluate five cache designs: CEASER, CEASER-S, MIRAGE, ScatterCache, and SassCache across five replacement policies. For our security evaluation, we evaluate the resistance of these five cache designs against different threat models, and comparatively analyze the resilience of different cache designs against our attacks. Moreover, for the first time, we show full AES key recovery attack through a novel cache-occupancy side-channel, thereby establishing the potency of occupancy-based attacks on randomized caches.

Our results therefore highlight the need for a holistic performance and security evaluation as we demonstrate that design decisions to protect against contention-based attacks alleviate vulnerability towards cache occupancy attacks as well as slowdown in performance in general.

Acknowledgments

We would like to thank the authors of SassCache who very generously provided us with their custom simulator for SassCache upon request, which allowed for a fair performance and security evaluation across all designs. We would also like to thank the anonymous reviewers for their constructive comments that helped in improving the overall message of our work. We further acknowledge Kuheli Pratihar for helpful discussions and assistance with the preparation of figures. Nimish Mishra, Sarani Bhattacharya, and Debdeep Mukhopadhyay would additionally like to acknowledge Centre on Hardware-Security Entrepreneurship Research and Development (C-HERD), MeitY, Govt. of India, and Information Security Education and Awareness (ISEA), MeitY, Govt. of India, for partially funding this research.

Ethical Consideration and Open Science

Ethical Consideration

The authors acknowledge and uphold the importance of maintaining high ethical standards in conducting and evaluating computer security research. This work analyses multiple secured cache designs proposed in literature over the past decade. None of the designs-under-test are part of commercially available product and thus require no vulnerability disclosure. The designs implemented using in-house simulator are based on the design descriptions from individual papers. The gem5 implementations are adapted from the respective open-source repositories associated with original papers and have been appropriately attributed, wherever applicable.

Open Science Compliance

As part of open science initiative, we have open-sourced all the tools, scripts, and datasets developed in this work and needed for meaningful reproduction of the results shown in the paper. The permanent link for the source codes used in this paper can be found at: https://doi.org/10.5281/zenodo.14737392. Note that this permanent repository includes the codes and scripts *as-is*, as used by the authors for developing the paper. A more detailed version of the artifact will be made available later, as 'Artifact Appendix'. We now elucidate the components provided in the above-mentioned artifact repository.

1. **llc_simulator**: In-house simulator to produce results of Sec.5 (Fig. 6) related to covert channels. Each subdirectory within this directory corresponds to a specific

randomized cache design. The results can be produced by running `python3 main.py` in each sub-directory, followed by executing the `plot.py` script.

2. **scripts**: contains the scripts for producing plots for the following:
   (a) **perf_eval**: scripts to produce benchmarking results from Sec. 3 (Fig. 5).
   (b) **fingerprinting**: `accuracy.py` produces results from Sec. 6 (Fig. 8).
   (c) **aes**: scripts to produce Fig. 9 and Fig. 10 from Sec. 7.
3. **ceaser**: gem5 implementation of CEASER.
4. **ceaser-s**: gem5 implementation of CEASER-S.
5. **mirage**: gem5 implementations of MIRAGE, Scatter-Cache and baseline cache.
6. **sasscache**: gem5 implementation of SassCache.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *ACM workshop on Computer security architecture*, pages 11–18, 2007.

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[3] Hassan Aly and Mohammed ElGayyar. Attacking aes using bernstein's attack on modern processors. In *Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*, pages 127–139. Springer, 2013.

[4] Anubhav Bhatla, Biswabandan Panda, et al. The maya cache: A storage-efficient and secure fully-associative last-level cache. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 32–44. IEEE, 2024.

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

[6] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 450–466. Springer, 2007.

[7] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1110–1123. IEEE, 2020.

[8] Anirban Chakraborty, Sarani Bhattacharya, Sayandeep Saha, and Debdeep Mukhopadhyay. Are randomized caches truly random? formal analysis of randomized-partitioned caches. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 233–246. IEEE, 2023.

[9] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 570–581, 2014.

[10] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 451–468, 2020.

[11] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ Abort: A timer-free high-precision l3 cache attack using Intel TSX. In *USENIX Security Symposium 2017)*, pages 51–67, 2017.

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

[13] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 845–858. Association for Computing Machinery, 2017.

[14] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. Scatter and split securely: Defeating cache contention and occupancy attacks. In *2023 IEEE Symposium on Security and Privacy (S&P)*, pages 1101–1115. IEEE Computer Society, 2022.

[15] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[16] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[17] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy (S&P)*, pages 191–205. IEEE, 2013.

[18] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ a: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 591–604. IEEE, 2015.

[19] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.

[20] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1977–1991, 2022.

[21] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215. IEEE, 2014.

[22] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[24] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, pages 48–65. Springer, 2015.

[25] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12*, pages 46–64. Springer, 2015.

[26] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *NDSS*, volume 17, pages 8–11, 2017.

[27] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[29] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *42th IEEE Symposium on Security and Privacy*, volume 5, 2021.

[30] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+ probe attacks and covert channels in scattercache. *arXiv preprint arXiv:1908.03383*, 2019.

[31] Moinuddin K Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787. IEEE, 2018.

[32] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 360–371. IEEE, 2019.

[33] Moinuddin K Qureshi, David Thompson, and Yale N Patt. The v-way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 544–555. IEEE, 2005.

[34] Kartik Ramkrishnan, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. Non-fusion based coherent cache randomization using cross-domain accesses. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 186–202, 2024.

[35] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[36] Gururaj Saileshwar. Battle for secure caches: Attacks and defenses on randomized caches, Sep 2021.

[37] Gururaj Saileshwar and Moinuddin Qureshi. {MIRAGE}: Mitigating conflict-based cache attacks with a practical fully-associative design. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[38] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Website fingerprinting through the cache occupancy channel and its real world practicality. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2042–2060, 2020.

[39] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, 2019.

[40] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *2021 IEEE Symposium on Security and Privacy (S&P)*, pages 955–969. IEEE, 2021.

[41] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*, pages 443–461. Springer, 2009.

[42] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[43] Thomas Unterluggauer, Austin Harris, Scott Constable, Fangfei Liu, and Carlos Rozas. Chameleon cache: Approximating fully associative caches with random replacement to prevent contention-based cache attacks. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 13–24. IEEE, 2022.

[44] Tarunesh Verma, Achilleas Anastasopoulos, and Todd Austin. These aren't the caches you're looking for: Stochastic channels on randomized caches. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 37–48. IEEE, 2022.

[45] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 39–54. IEEE, 2019.

[46] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[47] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE, 2008.

[48] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium USENIX Security 19)*, pages 675–692, 2019.

[49] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[50] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on Security and Privacy (S&P)*, pages 313–328. IEEE, 2011.

A Investigating Additional Assumptions in MIRAGE's Implementation

We experimentally validate the extent to which MIRAGE's implementation specific assumptions are valid. Our experiment aims to establish how many of these memory accesses MIRAGE fails to consider in the measurement of its performance statistics. The first statistic- `system.l2.tags.data_accesses`- denotes the total number of accesses made by the benchmark in LLC[22]. Likewise, the second statistic- `system.l2.overall_hits::cpu.data`- denotes the hits in LLC cache for data accesses. Finally, the remaining two statistics- `system.l2.tags.repl_valid_data` and `system.l2.tags.repl_valid_tag`- effectively denote "`Scenario-C`" (GLE) and "`Scenario-B`" (SAE) respectively. We note from [37] that "`Scenario-B`" is statistically improbable, implying (with arbitrarily high probability), all cache misses result in a global eviction (i.e. "`Scenario-C`"). This implies the following relation holds with high probability: "LLC accesses = LLC hits + `Scenario-C` misses".

[22] We consider L2 cache and LLC as analogous in line with the cache configuration in [37].

Table 3: The difference in the global evictions reported by MIRAGE's gem5 implementation as opposed to the ideal number of misses (computed by *Overall access - Cache hits*), against SPEC2017 benchmarks. The subset of SPEC2017 benchmarks considered is carefully curated considering several parameters: ① code size (measured in kilo lines of code), ② programming language (C / C++ / mixture of C and C++), and ③ application domain.

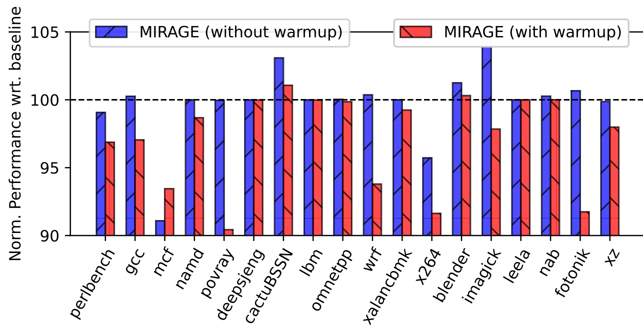
Benchmark	Overall data accesses	Cache hits	Reported global evictions	Actual misses
998.specrand_is	1749227552	31235387	0	1717992165
500.perlbench_r	3938592619	57603967	149011	3880988652
502.gcc_r	4984210067	65119091	64907	4919090976
505.mcf_r	4083326439	72091393	548823	4011235046
508.namd_r	4651478331	82965267	0	4568513064
511.povray_r	24833452	431524	0	24401928
531.deepsjeng_r	3448850191	34224185	22494607	3414626006

We capture experimental results in Tab. 3. Against each benchmark, Tab. 3 details the overall data accesses (reported by the gem5 statistic `system.l2.tags.data_accesses`) as well as the number of cache hits (reported by the gem5 statistic `system.l2.overall_hits::cpu.data`). Moreover, the number of *actual misses* (computed by subtracting `system.l2.overall_hits::cpu.data` from `system.l2.tags.data_accesses`) are also reported. Finally, the "`Scenario-C`" misses, reported by the statistic `system.l2.tags.repl_valid_data` is shown as 'Reported Global Evictions'. As evident, the equation "LLC accesses = LLC hits + `Scenario-C` misses" is not satisfied by MIRAGE's current gem5 implementation.

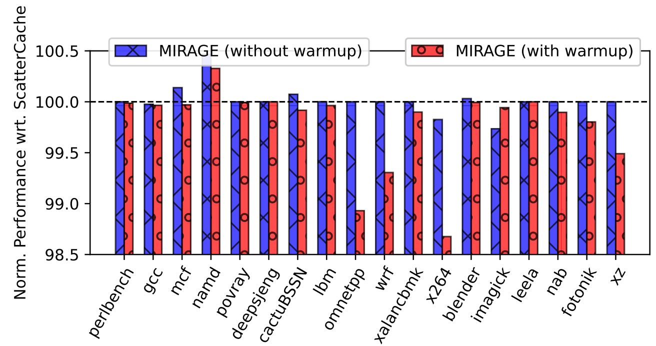
To investigate the performance degradation because of this issue and provide a comparative evaluation, we run SPEC2017 benchmarks on gem5 implementations of MIRAGE, ScatterCache and baseline set-associative cache (with random replacement policy). We use the phrase *cache warmup* to refer to some spurious occupancy done *prior* to benchmark invocation, that helps in exhausing "Scenario-A" occurrences in MIRAGE. We note that, on average, the number of global evictions performed by MIRAGE, once the queue is exhausted by the *cache warmup* phase, are significantly higher than reported evictions by original MIRAGE gem5 implementation. Fig. 11 shows the relative performance for MIRAGE normalized to baseline (Fig. 11a) and ScatterCache (Fig. 11b) where the blue bars denote the performance without warmup and red with warmup. It is clear that for most workloads, MIRAGE with warmup shows relative degradation in performance (both with respect to ScatterCache and baseline). Therefore, as Fig. 11 depicts, the *original* gem5 MIRAGE implementation reports an overly optimistic performance wrt. ScatterCache and baseline cache across several SPEC benchmarks. Note that MIRAGE with warmup phase enabled performs worse than its original implementation as well as set associative and ScatterCache with warmup. This is because a majority of cache accesses in the original MIRAGE gem5 implementation are serviced by fully-associative design rationale (corresponding to `Scenario-A` in MIRAGE implementation) instead of being serviced by the global-eviction policy of original MIRAGE design.

B Transmitting Byte-sized data in MIRAGE

To further evaluate the amplification of cache occupancy channel on MIRAGE, we extend the bit-wise covert channel into a byte-wise channel to increase the bandwidth. In this case, the receiver first prepares a *pre-attack step* where it simulates different number of accesses to form templates for cache misses. The receiver allocates a memory space and accesses $\ell = 10,000$ addresses to install them in the cache in distinct cache lines. The sender keeps an array of 8 elements $\{1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000\}$, each representing a byte value as their index position in the array and the number of accesses the sender needs to make to transmit that value. For example, to transmit integer value 2, the sender makes 3000 accesses, whereas to transmit 7, it makes 8000 accesses. The rest of the process follows similar to the bit-wise communication channel. Fig. 14 shows the number of cache misses observed (x-axis) in MIRAGE by the receiver for different number of accesses made by the sender. While $\ell = 10,000$ ensures an error-free communication channel, the channel can also perform with lesser number of receiver accesses. Fig. 12 depicts the trend of cache misses observed by the receiver with varying number of receiver accesses. As the number of accesses from the receiver increase, the reliability of the channel increases accordingly. Further, to simulate noise from other processes in real-world setting, we added random number of spurious loads. Fig. 13 shows the trend of cache misses observed by the receiver. The blue lineplot shows the number of spurious accesses which we term as noise. Note that we vary the noise level in accordance with the increasing number of accesses from the receiver. Note that due to introduction of noise, the channel becomes error-free at $\ell = 15,000$, instead of 10,000 (as in Fig. 12). The robustness of the channel against noise is due to the fact that we use the relative difference of cache misses, not their absolute values. Figs. 15, 16, 17, 18 show the statistics of the covert channel in MIRAGE for 1000, 5000, 10000, and 15000 accesses by the receiver. The boxplots show the range of cache misses observed by the receiver over 10 iterations, with the median value depicted as an orange line inside the box. Further, the mean (μ) and standard deviation (σ) corresponding to each boxplot is annotated with green and blue, respectively. One can observe that for 10000 and 15000 receiver accesses (Fig. 17, 18 resp.) offer almost non-overlapping statistic for the covert channel across different number of sender accesses. Whereas, for low occupancy, such as 1000 and 5000, receiver accesses (Fig. 15, 16 resp.) show overlapping statistic, making the channel erroneous.



(a) MIRAGE vs Baseline



(b) MIRAGE vs ScatterCache

Figure 11: Normalized performance of MIRAGE against (a) Baseline set-associative cache and (b) ScatterCache for SPEC2017.

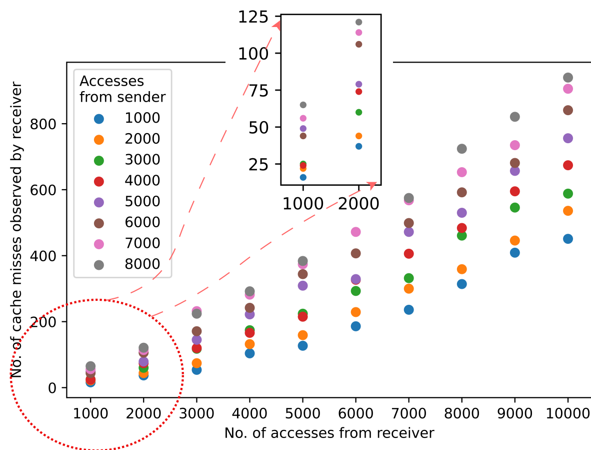


Figure 12: Covert channel in MIRAGE depicting the capability to transmit byte level information with varying number of sender and receiver accesses.

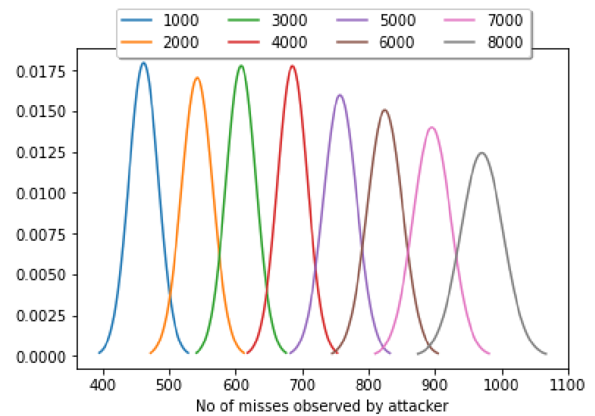


Figure 14: Templates of victim accesses with total number of accesses ranging from 1000 to 8000 and interval of 1000.

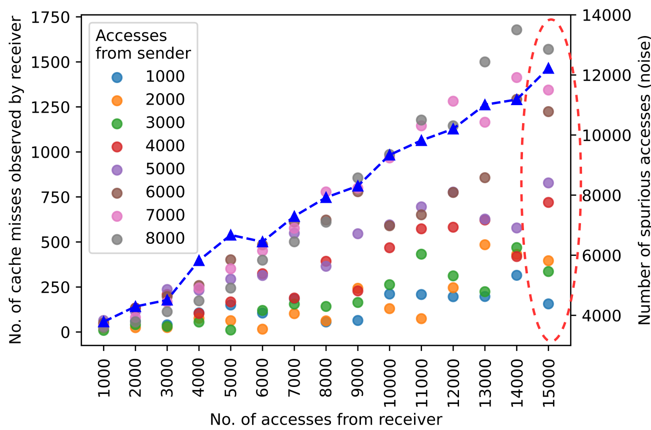


Figure 13: Covert channel in MIRAGE in noisy setting. The blue line plot (▲ marker) shows the number of spurious cache accesses.

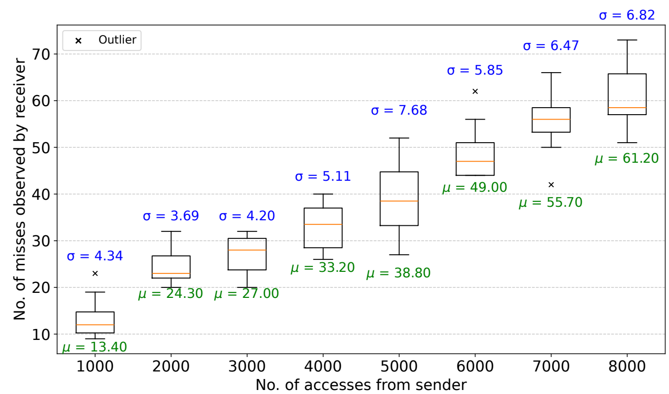


Figure 15: Covert channel statistics on MIRAGE for 1000 accesses.

Algorithm 1 Algorithmic sketch of the AES attack. GE is abbreviation for guessing entropy.

1: **procedure** CREATEKNOWNKEYPROFILE
2: Initialize dictionaries $\{\mathcal{T}_i^* : 0 \leq i \leq 15\}$
3: **for** `observation` **in** $\{1, 2, \cdots, |O^*|\}$ **do**
4: $P \leftarrow$ randomly generated plaintext
5: /* **Adversary State Preparation** */
6: malloc `data` of size $X\%$ of LLC
7: **for** `index` **in** $\{1, 2, \cdots, |data|\}$ **do**
8: load `data[index]`
9: /* **AES Encryption with known key** K^* */
10: $C =$ `AESEncrypt`(P, K^*)
11: /* **Adversarial Post-Processing**/*
12: `start_time` $\leftarrow$ `rdtsc()`
13: **for** `index` **in** $\{1, 2, \cdots, |data|\}$ **do**
14: load `data[index]`
15: $T \leftarrow$ `rdtsc() - start_time`
16: `state` = `InvSBox`$(C \oplus K^*)$
17: `dict_index` $\leftarrow 0$
18: **for** `byte` **in** `state` **do**
19: $\mathcal{T}^*_{\texttt{dict_index}}[\texttt{byte}] = \mathcal{T}^*_{\texttt{dict_index}}[\texttt{byte}] \cup T$
20: `dict_index` $\leftarrow$ `dict_index` + 1
21: **return** $\{\mathcal{T}_i^*\}$
22: **procedure** CREATEUNKNOWNKEYPROFILE
23: Initialize dictionaries $\{\mathcal{T}_j^i : 0 \leq i \leq 15, 0 \leq j \leq 255\}$
24: **for** `observation` **in** $\{1, 2, \cdots, |O|\}$ **do**
25: $P \leftarrow$ randomly generated plaintext
26: /* **Adversary State Preparation** */
27: malloc `data` of size $X\%$ of LLC
28: **for** `index` **in** $\{1, 2, \cdots, |data|\}$ **do**
29: load `data[index]`
30: /* **AES Encryption with unknown key** K */
31: $C =$ `AESEncrypt`(P, K)
32: /* **Adversarial Post-Processing**/*
33: `start_time` $\leftarrow$ `rdtsc()`
34: **for** `index` **in** $\{1, 2, \cdots, |data|\}$ **do**
35: load `data[index]`
36: $T \leftarrow$ `rdtsc() - start_time`
37: **for** `guess` **in** $\{0, 1, \cdots, 255\}$ **do**
38: `state` = `InvSBox`$(C \oplus$ `guess`$)$
39: `dict_index` $\leftarrow 0$
40: **for** `byte` **in** `state` **do**
41: Add T to $\mathcal{T}^{\texttt{dict_index}}_{guess}[\texttt{byte}]$
42: `dict_index` $\leftarrow$ `dict_index` + 1
43: **return** $\{\mathcal{T}_j^i : 0 \leq i \leq 15, 0 \leq j \leq 255\}$
44: **procedure** ATTACK
45: `InitializeAESTables()`
46: malloc `data` of LLC size
47: **for** `index` **in** $\{1, 2, \cdots, |data|\}$ **do**
48: load `data[index]`
49: Compute GE from `createKnownKeyProfile()` and `createUnknownKeyProfile()`

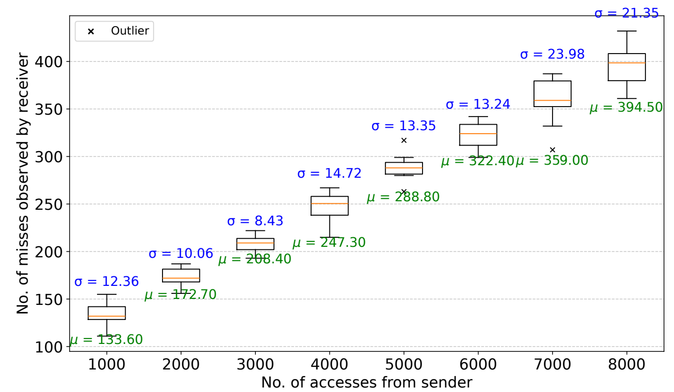


Figure 16: Covert channel statistics on MIRAGE for 5000 accesses.

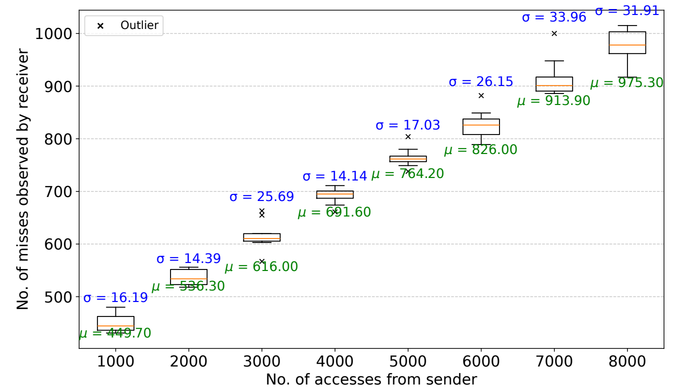


Figure 17: Covert channel statistics on MIRAGE for 10000 accesses.

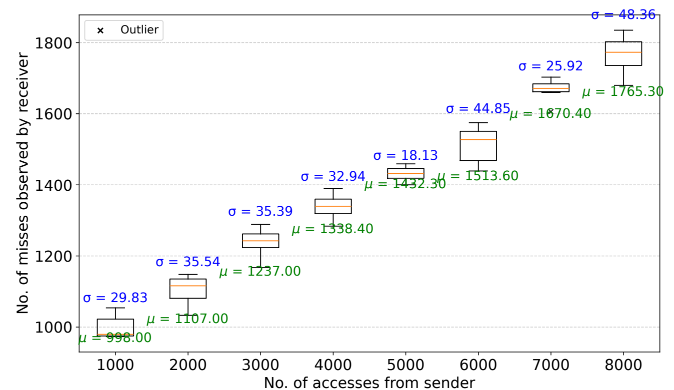


Figure 18: Covert channel statistics on MIRAGE for 15000 accesses.