MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design

Gururaj Saileshwar  
gururaj.s@gatech.edu  
Georgia Institute of Technology

Moinuddin Qureshi  
moin@gatech.edu  
Georgia Institute of Technology

Abstract

Shared caches in modern processors are vulnerable to conflict-based attacks, whereby an attacker monitors the access pattern of a victim by engineering cache-set conflicts. Recent mitigations propose a randomized mapping of addresses to cache locations to obfuscate addresses that can conflict with a target address. Unfortunately, such designs continue to select eviction candidates from a small subset of the resident cache lines, which makes such designs vulnerable to algorithms that can quickly identify the conflicting addresses.

This paper presents Mirage, a practical design for a fully associative cache, wherein eviction candidates are selected randomly from among all the lines resident in the cache, to be immune to set-conflicts. A key challenge in naïvely adopting such designs for large shared caches (containing tens of thousands of lines) is the complexity of cache-lookup, as that can require searching through all the lines resident in the cache in such designs. Mirage practically enables a fully-associative design, while maintaining the access latency similar to a traditional set-associative cache using: (1) Pointer-based indirection from the tag-store to the data-store, which allows a newly installed address to evict data of any resident line, (2) Skewed-associative tag-store with extra invalid tags, wherein incoming addresses can be installed without set-conflicts, and (3) Load-aware placement that maximizes the availability of sets with invalid tags, to eliminate set-conflicts. Our analysis shows Mirage provides the global eviction property of a fully-associative cache throughout the system lifetime (violations of full-associativity, i.e set-conflicts, occur less than once in $10^4$ to $10^{17}$ years), offering a principled defense against set-conflict based attacks. Mirage incurs negligible slowdown (0.3%) and 12–15% extra storage compared to the recently proposed Scatter-Cache.

1 Introduction

Ensuring effective data security and privacy in the context of hardware side-channels is a challenge. Performance-critical hardware components such as last-level caches (LLC) are often designed as shared resources to maximize utilization. When a sensitive application shares the LLC with a malicious application running simultaneously on a different core, cache side-channels can leak sensitive information. Such cache attacks have been shown to leak sensitive data like encryption keys [4] and user data in the cloud [32]. Set-conflict based cache attacks (e.g. Prime+Probe [24]) are particularly potent as they do not require any shared memory between the victim and the spy and exploit the set-associative design of conventional caches. Such designs map addresses to only a small group of cache locations called a set, to enable efficient cache lookup. If the addresses of both the victim and the attacker map to the same set, then they can evict each other from the cache (such an episode is called a set-conflict) – the attacker uses such evictions to monitor the access pattern of the victim.

Recent proposals for Randomized LLCs [27, 28, 38, 45] attempt to mitigate set-conflict based LLCs by randomizing the locations of cachelines, i.e. addresses resident in the cache. By making the address-to-set mapping randomized and unpredictable to an adversary, these designs attempt to obfuscate the locations of the lines that are evicted. However, such defenses continue to select cachelines for eviction from a small number of locations in the cache (equal to the cache associativity), as shown in Figure 1(a), and thus set-conflicts continue to occur although their locations are obfuscated. Subsequent attacks [26,28,39] have proposed efficient algorithms to discover a minimal eviction-set (lines mapping to the same set as a target address, that can evict the target via set-conflicts) even in the presence of such defenses, rendering them ineffective. In this paper, we target the root cause of vulnerability to eviction-set discovery in prior defenses – the limitation of selecting victims for eviction from a small subset of the cache (a few tens of lines), which allows an adversary, that observes evictions, to learn finite information about installed addresses.

Our goal is to eliminate set-conflicts and attacks that exploit them, with a cache that has the property of global evictions, i.e the victims for eviction are chosen (randomly) from among all the lines in the cache. With global evictions, any line resident in the cache can get evicted when a new address is installed into the cache; all cachelines belong to a single set as shown in Figure 1(b). Hence, an adversary observing an eviction of its address gains no information about the installed address.

A fully associative cache design, where an address can map to any location in the cache, naturally provides global evictions. However, the main challenge in adopting such a design for the LLC is ensuring practical cache lookup. As a line can reside in any cache location, a cache lookup can require searching through the entire LLC (containing tens of thousands of lines) and be much slower than even a memory access. In contrast, a set-associative design has efficient lookup,
Global Evictions (GLE) on misses that avoid set conflicts. (c) Our proposal, Mirage, enables global evictions practically with: (1) Indirection from tag-store to the data-store, (2) Skewed-Associative tag-store with extra tags, and (3) Placement of lines with load-balancing that guarantees the availability of sets with invalid tags and eliminates SAE.

To enable global evictions, Mirage uses pointer-based indirection to associate tags with data-blocks and vice-versa (inspired from cuckoo hashing [29]), as shown in Figure 1(c). Unlike a traditional cache, this design does not have an implicit mapping between the tag and the data for a cacheline. Mirage provisions extra invalid tags for each set in the tag-store at a modest storage cost, while retaining the same data-store capacity. When a new line is installed, an invalid tag can be allocated from the tag-store without requiring an eviction of a line from the same set. An eviction of a line is only required to free up a data-block, which is selected randomly from all the lines in the data-store, thus providing global eviction.

It is essential to prevent the adversary from mapping several lines at a time to a specific set, to fully deplete the available tags in that set. On an install to such a fully-occupied set, the cache is forced to perform a Set Associative Eviction (SAE), where a valid tag from the same set needs to be evicted to accommodate the incoming line. By observing such an SAE, an adversary can infer the address of the installed line causing the eviction, and eventually launch a set-conflict based attack.

To eliminate set-conflicts and SAE, and ensure all evictions are global evictions, Mirage first splits the tag store in two equal parts (skews), and uses a cryptographic hash function to randomize the line-to-set mapping within each skew, like prior skewed-associative designs for secure caches [28, 45]. This allows a line the flexibility of mapping to two possible sets (one in each skew), in a manner unpredictable to the adversary. As both skews could have invalid tag-store entries, an important consideration is the skew-selection policy on a line-install. Using a random skew-selection policy, such as in prior works [28, 45], results in an unbalanced distribution of invalid tags across sets, causing the episodes of SAE to continue to occur every few microseconds (few thousand line installs). To promote a balanced distribution of invalid tags across sets, Mirage employs a load-aware skew selection policy (inspired by load-aware hashing [3, 31]), that chooses the skew with the most invalid tag-entries in the given set. With this policy, Mirage guarantees an invalid tag is always available for an incoming line for system lifetime, thus eliminating SAE.

For an LLC with 2MB/core capacity and 16-ways in the baseline, Mirage provisions 75% extra tags, and has two skews, each containing 14-ways of tag-store entries. Our analysis shows that such a design encounters SAE once per $10^{17}$ years, providing the global eviction property and an illusion of a fully associative cache virtually throughout system lifetime.

If Mirage is implemented with fewer than 75% extra tags, the probability of an SAE increases as the likelihood that the tag entries in both skews are all valid increases. To avoid an SAE in such cases, we propose an optimization that relocates an evicted tag to its alternative set that is likely to have invalid tags with high probability (note that each address maps to two sets, one in each skew). Mirage equipped with such Cuckoo Relocation (inspired from cuckoo hashing [25]), ensures an SAE occurs once every 22,000 years, with 50% extra tags.

Overall, this paper makes the following contributions:

1. We observe that conflict based cache attacks can be mitigated by having global eviction, that considers all the lines for eviction. For practical adoption, our goal is provide such a global eviction property without incurring significant latency for cache-lookup or power overhead.

2. We propose Mirage, a practical way to get the global eviction benefits of a fully associative cache. Mirage uses indirection from tag-store to data-store, an intelligent tag store design, and a load balancing policy to en-
3. We propose **Mirage with Cuckoo Relocation**, whereby set-associative evictions in the tag store are mitigated by relocating a conflicting entry to an alternative location. This ensures set-associative evictions occur once in 22,000 years while reducing the extra tags needed.

As Mirage requires extra tags and indirection, it incurs a modest storage overhead of 12% to 15% for a cache design with 64-byte linesize (the storage overhead halves at 128-byte linesize). Our evaluations with a hardware-performance simulator show that Mirage incurs negligible slowdown (0.3%) compared to recently proposed Scatter-Cache. With these modest costs, Mirage provides the property of global evictions virtually for system lifetime, providing principled security against conflict-based attacks, and remains robust regardless of the advances in the algorithms for forming eviction sets.

## 2 Background and Motivation

### 2.1 Threat Model

We assume a threat model where the attacker and victim execute on different physical cores of a system and share a last-level cache (LLC) that is inclusive of the L1/L2 caches, which are private to each core. We primarily focus on cache side-channel attacks where the attacker causes set-conflicts to evict the target line in order to monitor the access pattern of the co-running victim. Such attacks are potent as they do not require the victim and the attacker to access lines in shared memory. For simplicity, we assume no shared memory between the victim and the attacker as there are existing solutions [45] that are effective at mitigating attacks on shared lines.\(^1\)

### 2.2 Problem: Conflict-Based Cache Attacks

Without loss of generality, we describe the Prime+Probe attack [24] as an example of conflict-based cache attack. As shown in Figure 2, the attacker first primes a set with its addresses, then allows the victim to execute and evict an attacker line due to cache-conflicts. Later, the attacker probes the addresses to check if there is a miss, to infer that the victim accessed that set. Prior attacks have monitored addresses accessed in AES T-table and RSA Square-Multiply Algorithms to leak secret keys [21], addresses accessed in DNN computations to leak DNN model parameters [48], etc.

To launch such attacks, the attacker first needs to generate an **eviction-set** for a victim address, i.e. a minimal set of addresses mapping to the same cache set as the victim address.

![Figure 2: Example of Conflict-Based Attack (Prime+Probe).](image)

### 2.3 Recent Advances in Attacks and Defenses

Given how critical eviction-set discovery is for such attacks, recent defense works have proposed randomized caches to obfuscate the address to set mapping and make it harder to learn eviction sets. At the same time, recent attacks have continued to enable faster algorithms for eviction set discovery. We describe the key related works in this spirit and discuss the pitfalls of continuing with such an approach.

**Move-1: Attack by Eviction Set Discovery in** \(O(n^2)\)

Typically, set-selection functions in caches are undocumented. A key work by Liu et al. [21] proposed an algorithm to discover eviction-sets without the knowledge of the address to set mappings – it tests and eliminates addresses one at a time, requiring \(O(n^2)\) accesses to discover an eviction-set.

**Move-2: Defense via Encryption and Remapping**

CEASER [27] (shown in Figure 3(a)) proposed randomizing the address to set mapping by accessing the cache with an encrypted line address. By enabling dynamic re-keying, it ensures that the mapping changes before an eviction-set can be discovered with an algorithm that requires \(O(n^2)\) accesses.

![Figure 3: Recent Works on Randomized Caches](image)

**Move-3: Attack by Eviction Set Discovery in** \(O(n)\)

Subsequent works [28, 39] developed a faster algorithm that could discover eviction-sets in \(O(n)\) accesses, by eliminating groups of lines from the set of potential candidates.
The goal of our paper is to enable randomized cache with global eviction, which were shown to be immune to faster eviction-set discovery algorithms. As its remapping rate needs to be increased beyond practical limits.

Move-4: Defense via Skewed Associativity

Scatter-Cache [45] and CEASER-S [28] adopt skewed associativity in addition to randomized mapping of addresses to sets, to further obfuscate the LLC evictions. As shown in Figure 3(b), such designs partition the cache across ways into multiple skews, with each skew having a different set-mapping and a new address is installed in a randomly selected skew. Such a design provides greater obfuscation as eviction sets get decided by the line to skew mapping as well. These designs were shown to be immune to faster eviction set discovery algorithms [28,39] that require \(O(n)\) steps.

Move-5: Attack by Probabilistic Eviction Set Discovery

A recent work [26] showed that faster eviction-set discovery in Scatter-Cache is possible with an intelligent choice of initial conditions, that boosts the probability of observing conflicts in Scatter-Cache. This allows discovery of partial eviction-sets (lines that evict a target in a subset of the ways) within 140K accesses in Scatter-Cache, that can enable a conflict-based attack. We believe this attack can also target CEASER-S, as conceptually it has a similar design as Scatter-Cache.

Pitfalls: There is an interplay between robustness of defenses and algorithms for eviction set discovery. The security of past defenses has hinged on obfuscation of eviction-sets, i.e. making them harder to discover with existing algorithms. However, newer algorithms enabling faster eviction-set discovery continue to break such defenses. Ideally, we seek a defense that eliminates Set-Associative Evictions (SAE), which are the root-cause of the vulnerability, as they allow the adversary to learn eviction-sets. Eliminating SAE would not only safe-guard against current algorithms for eviction set discovery but also against a hypothetical oracular algorithm that can learn an eviction-set after observing just a single conflict.

2.4 Goal: A Practical Fully-Associative LLC

As a principled defense against conflict based attacks, we seek to design a cache that provides Global Eviction (GLE), i.e. the eviction candidates are selected from among all of the addresses resident in the cache when new addresses are installed. Such a defense would eliminate SAE and be immune to eviction-set discovery and conflict based attacks, as evicted addresses are independent of the addresses installed and leak no information about installed addresses. While a fully-associative design provides global evictions, it has prohibitive overheads in terms of access latency and power when adopted for an LLC. The goal of our paper is to enable an LLC design that guarantees all evictions are global evictions, while retaining the practical lookup of a set-associative cache.

3 Full Associativity via MIRAGE

To guarantee global evictions in a practical manner, we propose Mirage (Multi-Index Randomized Cache with Global Evictions). Mirage provides the abstraction of a fully associative cache with random replacement, as shown in Figure 4(a). This design has the property that on a cache miss, a random line is evicted from among all resident lines in the cache. This ensures that the evicted victim is independent of the incoming line and no subset of lines in the cache form an eviction set.

3.1 Overview of Mirage

Mirage has three key components, as shown in Figure 4(b). The first component is a cache organization that decouples tag and data location and uses indirection to link tag and data entries, as indicated by \(\textcircled{1}\) in Figure 4(b). Provisioning extra invalid tags allows accommodating new lines in indexed
sets without tag-conflicts, and indirection between tags and data-blocks allows victim-selection from the data-store in a global manner. The second component of Mirage is an intelligent tag-store design that splits the tag entries into two structures (skews) and accesses each of them with a different hashing function, as indicated by 2 in Figure 4(b). Finally, to maximize the likelihood of getting an invalid tag on cache-install, Mirage uses an intelligent load-balancing policy for skew-selection leveraging the "power of 2 choices" [31] as indicated by 3 in Figure 4(b), which ensures that no SAE occurs in system lifetime and all evictions are global. We describe each component next.

### 3.2 Tag-to-Data Indirection and Extra Tags

Figure 5 shows the tag and data store organization using pointer-based indirection in Mirage. This design is inspired by the V-way cache [29] that originally used it to reduce conflict-misses in LLCs and improve performance. Here, the tag-store is over-provisioned to include extra invalid tags, which can accommodate the metadata of a new line (i.e., the address, valid-bit, dirty-bit, etc.) without a set-associative eviction (SAE). Each tag-store entry has a pointer (FPTR) to allow it to map to an arbitrary data-store entry. 3 On a cache-miss, two types of evictions are possible: if the incoming line finds an invalid tag, a Global Eviction (GLE) is performed; otherwise, an SAE is performed to invalidate a tag in the set where the line is to be installed (alongwith the corresponding data-entry). For a GLE, a random data entry from the entire data-store is selected (using a hardware PRNG) and the tag associated with this line is invalidated using the reverse pointer (RPTR) stored with each data entry. In either case, the RPTR in the invalidated data-entry is reset to an invalid value. This data-entry and the invalid tag-entry in the original set are then used by the new line to be installed.

Although indirection and extra tags enable GLE, they are by themselves insufficient to eliminate SAE. For example, if an adversary has arbitrary control over the placement of new lines in specific sets, they can map a large number of lines to a certain set and deplete the extra invalid tags provisioned in that set. When a new (victim) line is to be installed to this set, the cache is then forced to evict a valid tag from the same set and incur an SAE. Thus, an adversary who can discover the address to set mapping can force an SAE on each miss, making such a design vulnerable to the same attacks present in conventional set-associative caches.

### 3.3 Skewed-Associative Tag-Store Design

To virtually eliminate SAE and ensure GLE on each line install, Mirage reshapes the tag organization. To offer flexibility for a new address to map to multiple sets in the tag store and increase the probability of obtaining an invalid tag, Mirage architects the tag-store as a skewed-associative structure [35]. Here, the tag store is split into two partitions or skews, and a different randomizing hash function is used to map addresses to sets in each skew as shown in Figure 4. The hash function to map addresses-to-sets is constructed using QARMA-64 [2] block-cipher, similar to Scatter-Cache (SCv1) [45], with each skew using a different 64-bit key. Note that, unlike prior defenses using skewed-associativity [28, 45], each tag-store skew in Mirage contains invalid tag-entries. Offering the flexibility for a new line to map to two sets (one in each skew) in the presence of invalid entries, significantly increases the chance of finding an invalid tag in which it can be installed and avoiding an SAE. Moreover, as the adversary does not know the secret key and the address-to-set mapping, they cannot arbitrarily deplete these invalid tags within a set.

### 3.4 Load-Aware Skew Selection

An adversary may attempt to exploit natural imbalance in usage of tags across sets and observe resultant SAE to learn the address-to-set mappings. On a line-install, the skew-selection policy, that decides the skew in which the line is installed, determines the distribution of invalid tags across all sets. Prior works, including Scatter-Cache [45] and CEASER-S [28], use a random skew selection policy, that randomly picks one of the two skewers on a line-install. In the presence of invalid tags, this policy is prone to having imbalanced sets, some with few tags in use and some with all the tags in use. Our analysis indicates that such a random skew-selection policy results in an SAE every few misses (every 2600 misses with 6 extra ways/skew), and provides robustness only for microseconds.

To guarantee the availability of invalid tags in every set and eliminate SAE, Mirage uses a load-aware skew selection policy inspired from "Power of 2 Choices" [3, 31], a load-balancing technique used in hash-tables. As indicated by 3 in Figure 4, this policy makes an intelligent choice between the two skews, installing the line in the skew where the indexed set has a higher number of invalid tags. In the case

![Figure 5: Overview of the cache substrate used by Mirage with indirection and extra tags (inspired by V-Way Cache).](image-url)
of a tie between the two sets, one of the two skews is randomly selected. With this policy, an SAE occurs only if the indexed sets in both skews do not have invalid tags, that is a rare occurrence as this policy actively promotes balanced usage of tags across sets. Table 1 shows the rate of SAE for Mirage with load-aware skew selection policy, as the number of extra tags per skew is increased from 0 to 6. Mirage with 14-ways per skew (75% extra tags) encounters an SAE once in $10^{10}$ cache-installs, or equivalently $10^{17}$ years, ensuring no SAE throughout the system lifetime. We derive these bounds analytically in Section 4.3.

Table 1: Frequency of Set-Associative Eviction (SAE) in Mirage as number of extra ways-per-skew is increased (assuming 16-MB LLC with 16-ways in the baseline and 1ns per install)

| Ways in each Skew (Base + Extra) | Installs per SAE | Time per SAE |
|----------------------------------|------------------|-------------|
| 8 + 0                            | 1                | 1 ns        |
| 8 + 1                            | 4                | 4 ns        |
| 8 + 2                            | 60               | 60 ns       |
| 8 + 3                            | 8000             | 8 us        |
| 8 + 4                            | $2 \times 10^6$  | 0.16 s      |
| 8 + 5                            | $7 \times 10^6$  | 2 years     |
| 8 + 6 (default Mirage)           | $10^{14}$        | $10^{17}$ years |

4 Security Analysis of Mirage

In this section, we analyze set-conflict-based attacks in a setting where the attacker and the victim do not have shared memory (shared-memory attacks are analyzed in Section 5.4). All existing set-conflict based attacks, such as Prime+Probe [24], Prime+Abort [8], Evict+Time [24], etc. exploit eviction-sets to surgically evict targeted victim-addresses, and all eviction-set discovery algorithms require the attacker to observe evictions dependent on the addresses accessed by the victim. In Mirage, two types of evictions are possible – a global eviction, where the eviction candidate is selected randomly from all the lines in the data-store, that leak no information about installed addresses; or a set-associative eviction (SAE), where the eviction candidate is selected from the same set as the installed line due to a tag-conflict, that leaks information. To justify how Mirage eliminates conflict-based attacks, in this section we estimate the rate of SAE and reason that even a single SAE is unlikely to occur in system-lifetime.

Our security analysis makes the following assumptions:

1. Set-index derivation functions are perfectly random and the keys are secret. This ensures the addresses are uniformly mapped to cache-sets, in a manner unknown to the adversary, so that they cannot directly induce SAE. Also, the mappings in different skews (generated with different keys) are assumed to be independent, as required for the power of 2-choices load-balancing.

2. Even a single SAE is sufficient to break the security. The number of accesses required to construct an eviction-set has reduced due to recent advances, with the state-of-the-art [21, 28, 39] requiring at least a few hundred set-associative evictions to construct eviction-sets. To mitigate even future advances in eviction-set discovery, we consider a powerful hypothetical adversary that can construct an eviction-set with a single SAE (the theoretical minimum), unlike previous defenses [27, 28, 45] that only consider existing eviction-set discovery algorithms.

4.1 Bucket-And-Balls Model

To estimate the rate of SAE, we model the operation of Mirage as a buckets-and-balls problem, as shown in Figure 6. Here each bucket models a cache-set and each ball throw represents a new address installed into the cache. Each ball picks from 2 randomly chosen buckets, one from each skew, and is installed in the bucket with more free capacity, modeling the skew-selection in Mirage. If both buckets have the same number of balls, one of the two buckets is randomly picked. If both buckets are full, an insertion will cause a bucket spill, equivalent to an SAE in Mirage. Otherwise, on every ball throw, we randomly remove a ball from among all the balls in buckets to model Global Eviction. The parameters of our model are shown in Table 6. We initialize the buckets by inserting as many balls as cache capacity (in number of lines), and then perform 10 trillion ball insertions and removals to measure frequency of bucket spills (equivalent to SAE). Note that having fewer lines in the cache than the capacity is detrimental to an attacker, as the probability of a spill would be lower; so we model the best-case scenario for the attacker.

Table 2: Parameters for Buckets and Balls Modeling

| Buckets and Balls Model | Mirage Design |
|------------------------|---------------|
| Balls - 256K           | Cache Size - 16 MB |
| Buckets/Skew - 16K    | Sets/Skew - 16K |
| Skews - 2              | Skews - 2      |
| Avg Balls/Bucket - 8   | Avg Data-Lines Per Set - 8 |
| Bucket Capacity - 8 to 14 | Ways Per Skew - 8 to 14 |

4 A biased tie-breaking policy [40] that always picks Skew-1 on ties further reduces the frequency of bucket spills by few orders of magnitude compared to random tie-breaks. However, to keep our analysis simple, we use a random tie-breaking policy.
Figure 6: Buckets-and-balls model for Mirage with 32K buckets (divided into 2 skews), holding 256K balls in total to model a 16MB cache. The bucket capacity is varied from 8-to-14 to model 8-to-14 ways per skew in Mirage.

4.2 Empirical Results for Frequency of Spills

Figure 7 shows the average number of balls thrown per bucket spill, analogous to the number of line installs required to cause an SAE on average. As bucket capacity increases from 8 to 14, there is a considerable reduction in the frequency of spills. When the bucket capacity is 8, there is a spill on every throw as each bucket has 8 balls on average. As bucket capacity increases to 9 / 10 / 11 / 12, the spill frequency decreases to once every 4 / 60 / 8000 / 160Mn balls. For bucket capacities of 13 and 14, we observe no bucket spills even after 10 trillion ball throws. These results show that as the number of extra tags increases, the probability of an SAE in Mirage decreases super-exponentially (better than squaring on every extra way). With 12 ways/skew (50% extra tags), Mirage has an SAE every 160 million installs (equivalent to every 0.16 seconds).

Figure 7: Frequency of bucket spills, as bucket capacity is varied. As bucket-capacity increases from 8 to 14 (i.e. extra-tags per set increase from 0% to 75%), bucket spills (equivalent to SAE) become more infrequent.

While this empirical analysis is useful for estimating the probability of an SAE with up to 12 ways/skew, increasing the ways/skew further makes the frequency of SAE super-exponentially less. Hence, it is impractical to empirically compute the probability of SAE in a reasonable amount of time beyond 12 ways/skew (an experiment with 10 trillion ball throws already takes a few days to simulate). To estimate the probability of SAE for a Mirage design with 14 ways/skew, we develop an analytical model, as described in the next section.

Table 3: Terminology used in the analytical model

| Symbol | Meaning |
|--------|---------|
| Pr(n = N) | Probability that a Bucket contains N balls |
| Pr(n ≤ N) | Probability that a Bucket contains ≤ N balls |
| Pr(X → Y) | Probability that a Bucket with X balls transitions to Y balls |
| W | Capacity of a Bucket (beyond which there is a spill) |
| Btot | Total number of Buckets (32K) |
| Btot | Total number of Balls (256K) |

4.3 Analytical Model for Bucket Spills

To estimate the probability of bucket spills analytically, we start by modeling the behavior of our buckets and balls system in a spill-free scenario (assuming unlimited capacity buckets). We model the bucket-state, i.e. the number of balls in a bucket, as a Birth-Death chain [19], a type of Markov chain where the state-variable (number of balls in a bucket) increases or decreases by one at a time, due to a birth (insertion) or death (deletion) of a ball.

Figure 8: Bucket state modeled as a Birth-Death chain, a Markov Chain where the state variable N (number of balls in a bucket) increases or decreases by 1 at a time due to birth or death events (ball insertion or deletions), as shown in Figure 8.

\[ \Pr(N \rightarrow N + 1) = \Pr(N + 1 \rightarrow N) \] (1)
To calculate \( \Pr(N \rightarrow N+1) \), we note that a bucket with \( N \) balls transitions to \( N+1 \) balls on a ball insertion if: (1) the buckets chosen from both Skew-1 and Skew-2 have \( N \) balls; or (2) bucket chosen from Skew-1 has \( N \) balls and from Skew-2 has more than \( N \) balls; or (3) bucket chosen from Skew-2 has \( N \) balls and from Skew-1 has more than \( N \) balls. Thus, if the probability of a bucket with \( N \) balls is \( \Pr(n = N) \), probability that it transitions to \( N+1 \) balls is given by Equation 2.

\[
\Pr(N \rightarrow N+1) = \Pr(n = N)^2 + 2 \times \Pr(n = N) \times \Pr(n > N) \quad (2)
\]

To calculate \( \Pr(N + 1 \rightarrow N) \), we note that a bucket with \( N+1 \) balls transitions to \( N \) balls only on a ball removal. As a random ball is selected for removal from all the balls, the probability that a ball in a bucket with \( N+1 \) balls is selected for removal equals the fraction of balls in such buckets. If the number of buckets equals \( B_{tot} \) and the number of balls is \( b_{tot} \), the probability of a bucket with \( N+1 \) balls losing a ball (i.e. the fraction of balls in such buckets), is given by Equation 3.

\[
\Pr(N + 1 \rightarrow N) = \frac{\Pr(n = N+1) \times B_{tot} \times (N + 1)}{b_{tot}} \quad (3)
\]

Combining Equation 1, 2 and 3, and placing \( B_{tot}/b_{tot} = 1/8 \), (the number of buckets/balls) we get the probability of a bucket with \( N+1 \) balls, as given by Equation 4 and 5.

\[
\Pr(n = N+1) = \left( \frac{8}{N+1} \right) \left( \Pr(n = N)^2 + 2 \times \Pr(n = N) \times \Pr(n > N) \right) \quad (4)
\]

\[
= \left( \frac{8}{N+1} \right) \left( \Pr(n = N)^2 + 2 \times \Pr(n = N) \right)
\quad -2 \times \Pr(n = N) \times \Pr(n \leq N) \quad (5)
\]

As \( n \) grows, \( \Pr(n = N) \rightarrow 0 \) and \( \Pr(n > N) \ll \Pr(n = N) \) given our empirical observation that these probabilities reduce super-exponentially. Using these conditions Equation 4 can be simplified to Equation 6 for larger \( n \).

\[
\Pr(n = N + 1) = \left( \frac{8}{N+1} \right) \times \Pr(n = N)^2 \quad (6)
\]

From our simulation of 10 trillion balls, we obtain probability of a bucket with no balls as \( \Pr_{obs}(n = 0) = 4 \times 10^{-6} \). Using this value in Equation 5, we recursively calculate \( \Pr_{est}(n = N + 1) \) for \( N \in [1, 10] \) and then use Equation 6 for \( N \in [11, 14] \), when the probabilities become less than 0.01. Figure 9 shows the empirically observed (\( \Pr_{obs} \)) and analytically estimated (\( \Pr_{est} \)) probability of a bucket having \( N \) balls. \( \Pr_{est} \) matches \( \Pr_{obs} \) for all available data-points.

Figure 9: Probability of a Bucket having \( N \) balls – Estimated analytically (\( \Pr_{est} \)) and Observed (\( \Pr_{obs} \)).

Figure 10 shows the probability of a set having \( N \) lines decreases double-exponentially beyond 8 lines per set (the average number of data-lines per set). For \( N = 13 / 14 / 15 \), the probability reaches \( 10^{-8} / 10^{-17} / 10^{-34} \). This behavior is due to two reasons – (a) for a set to get to \( N+1 \) lines, a new line must map to two sets with at least \( N \) lines; (b) a set with a higher number of lines is more likely to lose a line due to random global eviction. Using these probabilities, we estimate the frequency of SAE in the next section.

4.4 Analytical Results for Frequency of Spills

For a bucket of capacity \( W \), the spill-probability (without relocation) is the probability that a bucket with \( W \) balls gets to \( W+1 \) balls. By setting \( N = W \) in Equation 2 and \( \Pr(n > W) = 0 \), we get the spill-probability as Equation 7.

\[
\Pr_{spill} = \Pr(W \rightarrow W + 1) = \Pr(n = W)^2 \quad (7)
\]

Figure 10: Frequency of bucket-spill, as bucket-capacity varies – both analytically estimated (\( \Pr_{spill,est} \)) and empirically observed (\( \Pr_{spill,obs} \)) results are shown.

Figure 10 shows the frequency of bucket-spills (SAE) estimated by using \( \Pr_{est}(n = W) \), from Figure 9, in Equation 7. The estimated values (\( \Pr_{spill,est} \)) closely matches the empirically observed values (\( \Pr_{spill,obs} \)) from Section 4.2. As the number of tags per set, i.e. bucket-capacity \( W \) increases, the rate of SAE, i.e. the frequency of bucket-spills shows a double-exponential reduction (which means the exponent itself is increasing exponentially). The probability of
a spill with \(x\) extra ways, is of the form \(P(2^x)\); therefore with 5–6 extra ways, we get an extremely small probability of spill as the exponent term reaches 32 – 64. For \(W = 12 / 13 / 14\), an SAE occurs every \(10^8 / 10^{10} / 10^{14}\) line installs. Thus, the default Mirage design with 14-ways per set, with a rate of one SAE in \(10^{14}\) line installs (i.e. once in \(10^{17}\) years), effectively provides the security of a fully associative cache.

5 Discussion

5.1 Key Management in Mirage

Mirage uses secret keys for the set-index derivation function, that are stored in hardware and not visible to any software including the OS. Mirage does not require keys to be provisioned per-process/domain (unlike [45]) and does not require key-refreshes (unlike [27, 28]). While device-specific keys could be fused into the hardware, we recommend that the keys are generated at boot-time locally within the cache controller (using a pseudorandom number generator in hardware) to guard against inadvertent key leakage. For example, if the keys leak or the adversary guesses the key (1 in \(2^{64}\) chance), that breaks all prior secure cache designs, as SAE are required to orchestrate conflicts, which are not expected during regular operation. If multiple SAE are encountered, indicating that the mapping is no longer secret, then the keys in Mirage can be refreshed (followed by cache flush) to ensure continued security.

5.2 Security for Sliced LLC Designs

Recent Intel CPUs provision LLCs that are made up of smaller physically distinct entities called slices, that are each a few MBs in size, with separate tag-store and data-store structures per slice. In such designs, Mirage can be implemented at the granularity of a slice (with per-slice keys) and can guarantee global evictions within each slice. We analyzed the rate of SAE for an implementation of Mirage per 2MB slice (2048 sets, as used in Intel CPUs), with the tag-store per slice having 2 skews and 14-ways per skew and observed it to be one SAE in \(2 \times 10^{17}\) years, whereas a monolithic 16MB Mirage provides a rate of once in \(5 \times 10^{17}\) years. Thus, both designs (monolithic and per-slice) provide protection for a similar order of magnitude (and well beyond the system lifetime).

5.3 Security as Baseline Associativity Varies

The rate of SAE strongly depends on the number of ways provisioned in the tag-store. Table 4 shows the rate of SAE for a 16MB LLC, as the baseline associativity varies from 8 ways – 32 ways. As the baseline associativity varies, with just 1 extra way per skew, the different configurations have an SAE every 13 – 14 installs. However, adding each extra way squares the rate successively as per Equation 7. Following the double-exponential curve of Figure 10, the rate of an SAE goes beyond once in \(10^{13}\) years (well beyond system lifetime) for all three configurations within 5–6 extra ways.

| LLC Associativity | 8-ways | 16-ways (default) | 32-ways |
|-------------------|--------|------------------|---------|
| 1 extra way/skew  | 13 (<20ns) | 14 (<20ns) | 14 (<20ns) |
| 5 extra ways/skew | \(10^{13}\) (10^4 yrs) | \(10^{16}\) (2 yrs) | \(10^{14}\) (8 days) |
| 6 extra ways/skew | \(10^{14}\) (10^6 yrs) | \(10^{14}\) (10^7 yrs) | \(10^{10}\) (10^13 yrs) |

5.4 Mitigating Shared-Memory Attacks

Thus far, we have focused primarily on attacks that cause eviction via set conflicts and without any shared data between the victim and the attacker. If there is shared-memory between the victim and the attacker, attacks such as Flush+Reload [51], Flush+Flush [13], Invalidate+Transfer [16], Flush+Prefetch [12], Thrash+Reload [34], Evict+Reload [14] etc. are possible, where an attacker evicts the shared line from the cache using \texttt{clflush} instruction or cache-thrashing [34] or eviction-sets [14], and measures the latency of subsequent loads or flushes [13] of that line.

Shared Read-only Memory: These attacks are easily mitigated for read-only memory by placing distrusting programs (i.e. victim and attacker) in different security domains and storing duplicate copies of the shared line in the cache for each security domains, as adopted in several prior works [7, 18, 45]. For example, Scatter-Cache (SCv1) [45] uses the Domain-ID as an input to the hash-functions used for set-index-derivation, to ensure a shared address maps to different cache locations for different domains. Mirage uses an identical hash-function construction as Scatter-Cache SCv1, and hence inherits this feature that mitigates these attacks. We refer the reader to Scatter-Cache [45] for a detailed analysis of this mitigation.

Shared Writable Memory: It is infeasible to store duplicate copies in the cache for writable memory shared across domains, as such a design is incompatible with cache-coherence protocols. For security in this scenario, we require that writable shared-memory is only used for data-transfer between the victim and the attacker and not for any sensitive computations, a similar assumption as Scatter-Cache [45].

5.5 Implications for Other Cache Attacks

Replacement Policy Attacks: Reload+Refresh [6] attack exploited the LLC replacement policy to influence eviction-
decisions within a set, and enable a side-channel stealthier than Prime+Probe or Flush+Reload. Mirage guarantees global evictions with random replacement, that has no access-dependent state. This ensures that an adversary cannot influence the replacement decisions via its accesses, making Mirage immune to any such replacement policy attacks.

**Cache-Occupancy Attacks:** Mirage prevents an adversary that observes an eviction, from gaining any information about the address of an installed line. However, the fact that an eviction occurred continues to be observable, similar to prior works such as Scatter-Cache [45] and Hyb-Cache [7]. Consequently, Mirage and these prior works, are vulnerable to attacks that monitor the cache-occupancy of a victim by measuring the number of evictions, like a recent attack [36] that used cache-occupancy as a signature for website-fingerprinting. The only known way to effectively mitigate such attacks is static partitioning of the cache space. In fact, Mirage can potentially provide a substrate for global partitioning of the data-store that is more efficient than current way/set partitioning solutions to mitigate such attacks. We leave the study extending Mirage to support global partitions for future work.

6 Mirage with Cuckoo-Relocation

The default design for Mirage consists of 6 extra ways/skew (75% extra tags) that avoids SAE for well beyond the system lifetime. If Mirage is implemented with fewer extra tags (e.g. 4 extra ways/skew or 50% extra tags), it can encounter SAE as frequently as once in 0.16 seconds. To avoid an SAE even if only 50% extra tags are provisioned in Mirage, we propose an extension of Mirage that relocates conflicting lines to alternative sets in the other skew, much like Cuckoo Hashing [25]. We call this extension **Cuckoo-Relocation**.

6.1 Design of Cuckoo-Relocation

We explain the design of Cuckoo-Relocation using an example shown in Figure 11. An SAE is required when an incoming line (Line Z) gets mapped in both skews to sets that have no invalid tags (Figure 11(a)). To avoid an SAE, we need an invalid tag in either of these sets. To create such an invalid tag, we randomly select a candidate line (Figure 11(b)) from either of these sets and relocate it to its alternative location in the other skew. If this candidate maps to a set with an invalid tag in the other skew, the relocation leaves behind an invalid tag in the original set, in which the line to be installed can be accommodated without an SAE, as shown in Figure 11(c). If the relocation fails as the alternative set is full, it can be attempted again with successive candidates till a certain number of maximum tries, after which an SAE is incurred. For Mirage with 50% extra tags, an SAE is infrequent even without relocation (less than once in 100 million installs). So in the scenario where an SAE is required, it is likely that other sets have invalid tags and relocation succeeds.

Figure 11: Cuckoo Relocation, a technique to avoid an SAE if Mirage is implemented with 50% extra tags.

6.2 Results: Impact of Relocation on SAE

For Mirage with 50% extra tags, the chance that a relocation fails is approximately \( p = 1/\text{sets per skew} \). This is because at the time of an SAE (happens once in 100 million installs), it is likely that the only full sets, are the ones that are currently indexed (i.e. only 1 set per skew is full). For relocation to fail for a candidate, the chance that its alternative set is full is hence approximately \( p = 1/\text{sets per skew} \). After \( n \) relocation attempts, the chance that all relocation attempts fail and an SAE is incurred, is approximately \( p^n \).

Table 5 shows the rate of SAE for Mirage with 50% extra tags and Cuckoo-Relocation, as the maximum number of relocation attempts is varied. Attempting relocation for up to 3 lines is sufficient to ensure that an SAE does not occur in system-lifetime (SAE occurs once in 22000 years). We note that attempting relocation for up to 3 lines can be done in the shadow of a memory access on a cache-miss.

| Max Relocations | 0   | 1   | 2   | 3   |
|-----------------|-----|-----|-----|-----|
| Installs per SAE| \( 2 \times 10^8 \) | \( 3 \times 10^{12} \) | \( 4 \times 10^{16} \) | \( 7 \times 10^{20} \) |
| Time per SAE    | 0.16 seconds | 45 minutes | 1.3 years | 22,000 years |

6.3 Security Implications of Relocation

For Mirage with 50% extra tags, upto 3 cuckoo relocation are done in the shadow of memory access on a cache-miss. A typical adversary, capable of only monitoring load latency or execution time, gains no information about when or where relocations occur as – (1) Relocations do not stall the processor or cause memory traffic, they only rearrange cache entries within the tag-store; (2) A relocation occurs infrequently (once in 100 million installs) and any resultant change in occupancy of a set has a negligible effect on the probability of
an SAE. If a future adversary develops the ability to precisely monitor cache queues and learn when a relocation occurs to perceive a potential conflict, we recommend implementing Mirage with a sufficient number of extra tags (e.g. 75% extra tags) such that no relocations are needed in system lifetime.

7 Performance Analysis

In this section, we analyze the impact of Mirage on cache misses and system performance. As relocations are rare, we observe that performance is virtually identical for both with and without relocations. So, we discuss the key results only for the default Mirage design (75% extra tags).

7.1 Methodology

Adopting a methodology similar to Scatter-Cache [45], we use a simulator based on Intel Pintool [22] to model our hardware modifications. We generate an execution trace of 1 billion instructions for each of the SPEC CPU2006 benchmarks using a Pintool and feed it to a C++ based simulator that models in-order cores, inclusive L1/L2 caches, a shared LLC and DRAM. The parameters of our baseline are shown in Table 6. As our baseline, we use a 16-way, 16MB shared LLC, arranged in the Scatter-Cache organization (two skews of 8 ways each). The LLC lookup latency in Scatter-Cache is increased by 5 cycles to account for the QARMA-64 encryption. For Mirage, we increase the latency by an additional 1 cycle to account for the larger tag-store and FPTR lookup. We use a total of 58 workloads, including all the 29 SPEC CPU2006 benchmarks (each has 8 duplicate copies running on 8 cores) and 29 mixed workloads (each has 8 randomly chosen SPEC benchmarks).

Table 6: Baseline System Configuration

| Processor and Last-level Cache | Non-Secure | Scatter-Cache | Mirage |
|-------------------------------|------------|---------------|--------|
| Core                          | 8-cores, In-order Execution, 3.2GHz | 8-cores, In-order Execution, 3.2GHz | 8-cores, In-order Execution, 3.2GHz |
| L1 and L2 Cache Per Core       | 1.1-32KB, 2.56KB, 8-way, 64B line-size | 1.1-32KB, 2.56KB, 8-way, 64B line-size | 1.1-32KB, 2.56KB, 8-way, 64B line-size |
| LLC (shared across all cores)  | 16MB, 16-way, 64B line-size, Random Repl. | 16MB, 16-way, 64B line-size, Random Repl. | 16MB, 16-way, 64B line-size, Random Repl. |
| DRAM Memory-System            | 24 cycle lookup + 5 cycles for QARMA-64 | 24 cycle lookup + 5 cycles for QARMA-64 | 24 cycle lookup + 5 cycles for QARMA-64 |

| Workloads | SpecInt-12 | SpecFp-17 | Mix-29 | All-58 |
|-----------|------------|-----------|--------|--------|
| SpecInt-12| 10.79      | 11.23     | 9.52   | 9.58   |
| SpecFp-17 | 8.82       | 8.51      | 9.97   | 9.80   |
| Mix-29    | 9.52       | 9.97      | 9.98   |        |
| All-58    | 9.58       | 9.80      | 9.98   |        |

7.2 Impact on Cache Misses

Table 7 shows the LLC Misses Per 1000 Instructions (MPKI) for Mirage, Scatter-Cache, and a Non-Secure set-associative design (using LRU replacement), averaged over each workload suite SpecInt, SpecFp and Mix. Mirage and Scatter-Cache have near-identical number of misses, as both designs randomize evictions. Although Scatter-Cache selects replacement victims from a smaller subset of the cache compared to Mirage, both designs adopt a similar randomized address set mapping, which dominates to ensure that both designs have similar MPKI. Compared to a Non-Secure LLC, both Mirage and Scatter-cache incur 2% extra misses on average (0.2 MPKI extra on average) due to the randomized evictions. We observe that randomization can both increase or decrease conflict misses in different workloads – for example, Mirage and Scatter-Cache adds up to 7% extra misses for mcf and xalane, while reducing misses by up to 30% for sphinx.

7.3 Impact on Performance

Figure 12 shows the performance (measured as weighted speedup in Instructions Per Cycle for a multi-program workload [37]) of Mirage normalized to Scatter-Cache. Mirage has similar performance to Scatter-Cache across most workloads and suffers a slowdown of 0.3% on average. This slowdown is mainly due to Mirage requiring one extra cycle for LLC-accesses to lookup a larger tag-store (with 14 ways/skew) compared to Scatter-Cache (with 8-ways/skew). Both designs have further slowdown compared to a non-secure LLC – Mirage has a 2.5% slowdown, while Scatter-Cache has 2.2% slowdown on average. This is because both designs require 5 extra cycles to compute the QARMA-64 encryptor based set-mapping function (as studied in [45]), that is not required in the non-secure LLC. This gap can be reduced by faster encryption algorithms. For workloads such as mcf or omnet, Mirage increases both the LLC misses and access latency compared to a non-secure LLC and hence causes from 6% to 8% slowdown. On the other hand, for workloads such as sphinx, dealIII and gcc, Mirage reduces LLC-misses and improves performance by 4 to 18% versus the non-secure LLC.

7.4 Sensitivity to Cache Size

Figure 13 shows the performance of Mirage and a non-secure design for cache sizes of 2MB to 64MB, each normalized to Scatter-Cache of the same size. As cache size increases, Mirage remains within 0.4% of Scatter-Cache. However, slowdown for Mirage and Scatter-Cache versus the non-secure LLC, increases from 1% for a 2MB cache to 3.5% for a
Figure 12: Performance of Mirage normalized to Scatter-Cache. Across 58 workloads, Mirage incurs negligible slowdown (0.3%) compared to Scatter-Cache. Scatter-Cache itself is 2.2% slower than a Non-Secure LLC.

64MB cache. This is because larger caches have higher hit-rates, which causes an increase in the latency of a cache-hit (due to QARMA-64) to have a higher impact on the performance.

Figure 13: Sensitivity of Performance to Cache-Size.

8 Cost Analysis

We analyze the latency, area, and power overheads of Mirage. For this section, we distinguish the two versions of our design as, Mirage (default design with 75% extra tags) and Mirage-Lite (with 50% extra tags and relocation).

8.1 Impact on Lookup Latency

The lookup latency in Mirage is affected by a larger tag-store and an extra mux to select the FPTR of a hitting way. Using Cacti-6.0 [23] (that reports access time, energy, etc. for different cache organizations), with 32nm technology, we estimate that adding up to 16 extra ways adds 0.2 ns delay and an extra mux adds ~0.05ns. These delays can be accounted by increasing the latency by 1 cycle compared to Scatter-Cache.

8.2 Impact on Area

The area overheads in Mirage are proportional to its storage overheads due to (1) extra tag-entries, and (2) FPTR and RPTR, the pointers between tag/data entries. This causes a storage overhead of 15% for Mirage and 12% for Mirage-Lite compared to Scatter-Cache as shown in Table 8. An area-neutral performance comparison in Table 9 shows that Mirage is 1.5% slower compared to Scatter-Cache with 18% more capacity that has a similar area. Scatter-cache (SCv1) itself needs 3% more storage compared to a non-secure LLC, as it needs 40-bit tags versus 26-bit tags in a non-secure LLC.

Table 8: Storage Overheads in Mirage for 64B linesize

| Cache Size | Scatter-Cache | Mirage | Mirage-Lite |
|------------|---------------|--------|-------------|
| 16MB (16,384 Sets) | 2 skews x 8 ways/skew | 2 skews x 14 ways/skew | 2 skews x 12 ways/skew |
| Tag Entry | Tag-Bits Status(V,D) FPTR | Bits/Entry | Tag Entries |
| | 40 | 40 | 40 |
| | 2 | 2 | 2 |
| | – | 18 | 18 |
| | 42 | 60 | 60 |
| | 262,144 | 458,752 | 393,216 |
| Tag-Store Size | 1344 KB | 3360 KB | 2880 KB |
| Data Entry | Data-Bits RPTR | Bits/Entry | Data Entries |
| | 512 | 512 | 512 |
| | – | 19 | 19 |
| | 512 | 531 | 531 |
| | 262,144 | 262,144 | 262,144 |
| Data-Store Size | 16,384 KB | 16,992 KB | 16,992 KB |
| Total Storage | 17,728 KB (100%) | 20,352 KB (115%) | 19,872 KB (112%) |

The storage overhead of Mirage depends on cache linesize, as the relative size of tag-store reduces with larger linesize. While we use 64B linesize, Table 10 shows that 128B linesize like IBM’s Power9 [46] or 256B reduces overheads to 3-7%.
Table 9: Area-Neutral Slowdown for Mirage

| Design      | Data-store Size | Slowdown |
|-------------|-----------------|----------|
| Scatter-Cache | 19MB            | 0%       |
| Mirage      | 16MB            | 1.5%     |
| Mirage-Lite | 16MB            | 1.5%     |

Table 10: Storage overhead with different linesizes

| Design      | 64 Bytes | 128 Bytes | 256 Bytes |
|-------------|----------|-----------|-----------|
| Mirage      | 14.8%    | 7.4%      | 3.7%      |
| Mirage-Lite | 12.1%    | 6.1%      | 3.0%      |

8.3 Impact on Energy per Access

The larger tag-store in Mirage requires more energy to access than Scatter-Cache. To estimate energy per access, we use Cactii-6.0 with 32nm technology to calculate the energy for accessing a 16-way and a 32-way tag-store, and interpolate the values for 28-way (Mirage) and 24-way tag stores (Mirage-Lite). While the tag-store access has a modest increase in energy versus Scatter-Cache, the data-store access is unchanged as the design is similar. Overall, Mirage incurs 0.17 nJ and Mirage-Lite incurs 0.12 nJ more energy per access compared to Scatter-Cache, as shown in Table 11. These increases in energy are insignificant compared to the energy of a DRAM access, that takes up to 3nJ/access [41].

Table 11: Energy per access (in nJ) for Mirage

| Design      | Tag-Access | Data-Access | Total-Access |
|-------------|------------|-------------|--------------|
| Scatter-Cache | 0.11       | 0.50        | 0.61         |
| Mirage      | 0.27       | 0.51        | 0.78         |
| Mirage-Lite | 0.22       | 0.51        | 0.73         |

9 Related Work

Cache design for reducing conflicts (for performance or security) has been an active area of research. In this section, we briefly describe the closely related proposals, comparing and contrasting where appropriate.

9.1 Secure Caches with High Associativity

The concept of cache location randomization for guarding against cache attacks was pioneered almost a decade ago, with RPCache [42] and NewCache [43], for protecting L1 caches. Conceptually, such designs have an indirection-table that is consulted on each cache-access, that allows mapping an address to any cache location. While such designs can be implemented for L1-Caches, there are practical challenges when they are extended to large shared LLCs. For instance, the indirection-tables themselves need to be protected from conflicts if they are shared among different processes. While RPCache prevents this by maintaining per-process tables for the L1 cache, such an approach does not scale to the LLC that may be used by several hundred processes at a time. NewCache avoids conflicts among table-entries by using a Content-Addressable-Memory (CAM) to enable a fully-associative design for the table. However, such a design is not practical for LLCs, which have tens of thousands of lines, as it would impose impractically high power overheads. While Mirage also relies on indirection for randomization, it eliminates conflicts algorithmically using load-balancing techniques, rather than relying on per-process isolation that requires OS-intervention, or impractical fully-associative lookups and CAMs.

Phantom-Cache [38] is a recent design that installs an incoming line in 1 of 8 randomly chosen sets in the cache, each with 16-ways, conceptually increasing the associativity to 128. However, this design requires accessing 128 locations on each cache access to check if an address is in the cache or not, resulting in an impractically high power overhead of 67% [38]. Moreover, this design is potentially vulnerable to future evition set discovery algorithms as it selects a victim line from only a subset of the cache lines. In comparison, Mirage provides the security of a fully-associative cache where any eviction-set discovery is futile, with practical overheads.

HybCache [7] is a recent design providing fully-associative mapping for a subset of the cache (1–3 ways), to make a subset of the processes that map their data to this cache region immune to eviction-set discovery. However, the authors state that “applying such a design to an LLC or a large cache in general is expensive” [7]. For example, implementing a fully-associative mapping in 1 way of the LLC would require parallel access to >2000 locations per cache-lookup (that could increase cache power by >2500% extrapolating results from Phantom-Cache). In contrast, Mirage provides security of a fully-associative design for the LLC with practical overheads, while accessing only 24–28 locations per lookup.

9.2 Alternative Designs for High Associativity

V-Way Cache [29], which is the inspiration for our design, also uses pointer-based indirection and extra tags to reduce set-conflicts – but it does not eliminate them. V-Way Cache uses a set-associative tag-store, which means it is still vulnerable to set-conflict based attacks, identical to a traditional set-associative cache. Mirage builds on this design and incorporates skewed associativity and load-balancing skew-selection to ensure set-conflicts do not occur in system-lifetime.

Z-Cache [33] increases associativity by generating a larger pool of replacement candidates using a tag-store walk, and performing a sequence of line-relocations to evict the best victim. However, this design still selects replacement candidates from a small number of resident lines (up to 64), limited by the number of relocations it can perform at a time. As a result, a few lines can still form an eviction set, which
could potentially be learned by attacks. Whereas, Mirage selects victims globally from all lines in the cache, eliminating eviction-sets.

Indirect Index Cache [15] is a fully-associative design that uses indirection to decouple the tag-store from data-blocks, and has a tag-store designed as a hash-table with chaining to avoid tag-conflicts. However, such a design introduces variable latency for cache-hits and hence is not secure. While Mirage also uses indirection, it leverages extra tags and power of 2 choices based load-balancing, to provide security by eliminating tag-conflicts and retaining constant hit latency.

Cuckoo Directory [10] enables high associativity for cache-directories by over-provisioning entries similar to our work and using cuckoo-hashing to reduce set-conflicts. SecDir [50] also applies cuckoo-hashing to protect directories from conflict-based attacks [49]. However, cuckoo-hashing alone is insufficient for conflict-elimination. Such designs impose a limit on the maximum number of cuckoo relocations they attempt (e.g. 32), beyond which they still incur an SAE. In comparison, load-balancing skew selection, the primary mechanism for conflict-elimination in Mirage, is more robust at eliminating conflicts as it can ensure no SAE is likely to occur in system-lifetime with 75% extra tags.

9.3 Isolation-based Defenses for Set-Conflicts

Isolation based defenses attempt to preserve the victim lines in the cache and prevent conflicts with the attacker lines. Prior approaches have partitioned the cache by sets [5, 30] or ways [18, 20, 42, 52] to isolate security-critical processes from potential adversaries. However, such approaches result in sub-optimal usage of cache space and are unlikely to scale as the number of cores on a system grows (for example, 16-way cache for a 64-core system). Other mechanisms explicitly lock security-critical lines in the cache [17, 42], or leverage hardware transactional memory [11] or replacement policy [47] to preserve security-critical lines in the cache. However, such approaches require the classification of security-critical processes to be performed by the user or by the Operating-System. In contrast to all these approaches, Mirage provides robust and low-overhead security through randomization and global evictions, without relying on partitioning or OS-intervention.

10 Conclusion

Shared LLCs are vulnerable to conflict-based attacks. Existing defenses that attempt to randomize the LLC, continue to be broken with advances in eviction-set discovery algorithms. We propose Mirage a principled defense against both current and future attacks. Providing the illusion of a fully-associative cache with random-replacement, Mirage has the guarantee of always evicting a random line on a cache-fill that leaks zero information, for $10^4 - 10^{17}$ years. Mirage achieves this strong security with negligible slowdown (0.3%) and modest area overhead (12-15%), compared to recent work Scatter-Cache. More importantly, Mirage provides safeguard against future advances in eviction-set discovery algorithms.

References

[1] David H Albonesi. An architectural and circuit-level approach to improving the energy efficiency of microprocessor memory structures. In VLSI: Systems on a Chip, pages 192–205. Springer, 2000.

[2] Roberto Avanzi. The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involuntary central rounds, and search heuristics for low-latency s-boxes. IACR Transactions on Symmetric Cryptology, pages 4–44, 2017.

[3] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. SIAM journal on computing, 29(1):180–200, 1999.

[4] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[5] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019.

[6] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks. In 29th USENIX Security Symposium (USENIX Security 20), 2020.

[7] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel resilient caches for trusted execution environments. In 29th USENIX Security Symposium (USENIX Security 20), 2020.

[8] Craig D Stsselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel {TSX}. In 26th USENIX Security Symposium (USENIX Security 17), pages 51–67, 2017.

[9] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschiede, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, et al. Internal
organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. Digital Technical Journal, 7(1), 1995.

[10] Michael Ferdman, Pejman Lottif-Kamran, Ken Balet, and Babak Falsafi. Cuckoo directory: A scalable directory for many-core systems. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture, pages 169–180. IEEE, 2011.

[11] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 217–233, 2017.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel ashr. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pages 368–379, 2016.

[13] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In DIMVA, 2016.

[14] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In 24th {USENIX} Security Symposium ({USENIX} Security 15), pages 897–912, 2015.

[15] Erik G Hallnor and Steven K Reinhardt. A fully associative software-managed cache design. In Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201), pages 107–116. IEEE, 2000.

[16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In Proceedings of the 11th ACM on Asia conference on computer and communications security, pages 353–364, 2016.

[17] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In 21st {USENIX} Security Symposium ({USENIX} Security 12), 2012.

[18] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In MICRO, 2018.

[19] David Lilja. Measuring Computer Performance, pages 228–229. Cambridge University Press, 2000.

[20] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In 2016 IEEE international symposium on high performance computer architecture (HPCA), pages 406–418. IEEE, 2016.

[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In Security and Privacy (SP), 2015 IEEE Symposium on, pages 605–622. IEEE, 2015.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 40(6):190–200, 2005.

[23] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. HP laboratories, 27:28, 2009.

[24] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In Proceedings of the 2006 The Cryptographers’ Track at the RSA Conference on Topics in Cryptology, CT-RSA’06, 2006.

[25] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In European Symposium on Algorithms, pages 121–133. Springer, 2001.

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

[27] Moinuddin K. Qureshi. CEASER: Mitigating conflict-based cache attacks via dynamically encrypted address. In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, 2018.

[28] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In Proceedings of the 46th International Symposium on Computer Architecture (ISCA), 2019.

[29] 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.

[30] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In Proceedings of the 2009 ACM workshop on Cloud computing security, 2009.
[31] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[32] 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*, CCS ’09, 2009.

[33] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198. IEEE, 2010.

[34] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299, 2019.

[35] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA ’93, 1993.

[36] 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.

[37] Allan Snavely and Dean M Tullsen. Symbiotic job-scheduling for a simultaneous multithreaded processor. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 234–244, 2000.

[38] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *NDSS*, 2020.

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

[40] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[41] Thomas Vogelsang. Understanding the energy consumption of dynamic random access memories. In 2010 *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 363–374. IEEE, 2010.

[42] 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.

[43] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *International Symposium on Microarchitecture*, MICRO, 2008.

[44] Don Weiss, John J Wu, and Victor Chin. The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1523–1529, 2002.

[45] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, 2019.

[46] WikiChip. Power9 - microarchitectures - ibm. https://en.wikichip.org/wiki/ibm/microarchitectures/power9.

[47] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *ISCA*, 2017.

[48] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In 29th *USENIX Security Symposium* (*USENIX Security 20*), 2020.

[49] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904. IEEE, 2019.

[50] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. Secdir: a secure directory to defeat directory side-channel attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 332–345, 2019.

[51] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[52] Ziqiao Zhou, Michael K Reiter, and Yingjian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882, 2016.