Practical and Secure Randomized Last-Level Cache Design

Dissertation submitted in partial fulfillment of the requirements

for the degree of Master of Technology

by

Anubhav Bhatla

200070008

With the guidance of

Prof. Biswabandan Panda

Department of Electrical Engineering
Indian Institute of Technology Bombay
June 2025

Dedicated to my beloved parents.

Glossary

AES **A**dvanced **E**ncryption **S**tandard. 29

ARM **A**dvanced **R**ISC **M**achines. 44

B **B**ytes. 49

BIOS **B**asic **I**nput/**O**utput **S**ystem. 44

CAS **C**olumn **A**ddress **S**trobe. 30

CMOS **C**omplementary **M**etal-**O**xide-**S**emiconductor. 42

CPU **C**entral **P**rocessing **U**nit. 5

DDR **D**ouble **D**ata **R**ate. 30

DEMUX **D**e-**Mu**ltiple**x**er. 82

DTLB **D**ata **T**ranslation **Lookaside** **B**uffer. 30

FinFET **F**in **F**ield-**E**ffect **T**ransistor. 34

FPTR **F**orward **P**oin**t**e**r**. 16

GE **G**ate **E**quivalent. 54

GHz **G**iga **H**ert**z**. 30

ITLB **I**nstruction **T**ranslation **Lookaside** **B**uffer. 30

K **K**ilo. 22

KB **K**ilo-**B**ytes. 12

L1D **L**evel-**1** **D**ata Cache. 30

L1I **L**evel-**1** **I**nstruction Cache. 30

LLC **L**ast-**L**evel **C**ache. 1

LRU **L**east-**R**ecently-**U**sed. 4

M **M**illion. 30

MOESI **M**odified **O**wned **E**xclusive **S**hared **I**nvalid. 41

MSR **M**odel **S**pecific **R**egister. 44

MUX **Mu**ltiple**x**er. 82

nJ **n**ano-**J**oules. 35

nm **n**ano-**m**eter. 34

ns **n**ano-**s**econds. 42

OS **O**perating **S**ystem. 2, 14

PLRU **P**seudo-**L**east-**R**ecently-**U**sed. 69

RCD **R**ow-to-**C**olumn **D**elay. 30

REE **R**elative **E**viction **E**ntropy. 59

ROB **R**e**o**rder **B**uffer. 30

RP **R**ow **P**recharge. 30

RPLRU **R**andomized **P**seudo-**L**east-**R**ecently-**U**sed. 69

RPTR **R**everse **P**oin**t**e**r**. 18

RRIP **R**e-**r**eference **I**nterval **P**rediction. 41

SAE **S**et-**A**ssociative **E**viction. 1

SDID **S**ecurity **D**omain **Id**entity. 17

SMT **S**imultaneous **M**ulti-**T**hreading. 28

SPEC **S**tandard **P**erformance **E**valuation **C**orporation. 15

STLB **S**econd-level **T**ranslation **Lookaside** **B**uffer. 30

TDX **T**rusted **D**omain E**x**tensions. 44

TEE **T**rusted **E**xecution **E**nvironment. 12

Abstract

The sharing of the last-level cache (LLC) among multiple cores makes it vulnerable to cross-core conflict and occupancy-based attacks. Even after significant research in secure LLC designs, modern processors employ non-secure set-associative LLCs. In general, there are two possible secure LLC designs: (i) a randomized LLC and (ii) a partitioned LLC. The state-of-the-art randomized LLC design, Mirage, mitigates conflict-based attacks. However, it incurs significant area overhead (20% additional storage) and design complexity, with marginal performance overhead. On the other hand, LLC partitioning techniques mitigate cache occupancy-based attacks as well as conflict-based attacks. However, partitioning techniques incur significant performance overheads (on average more than 5% and as high as 49%). These factors pose major obstacles to the industrial adoption of secure LLCs. In this work, we ask the following question: can we achieve LLC security with minimal modifications to a conventional set-associative LLC, enabling morphable secure LLC design (security only when needed) while keeping performance, power, and area overheads low? Several other randomized cache designs have also been proposed recently, offering distinct security guarantees. While these designs incorporate several microarchitectural modifications (we call them *security knobs*) over the conventional set-associative cache to ensure security, the individual impact of these microarchitectural modifications has never been evaluated. This leaves a gap in the understanding of randomized LLCs – the design space has not been explored completely and systematically.

We observe that more than 80% of last-level cache's data store entries are dead on arrival, providing negligible utility in terms of performance improvement as they do not get reused in their lifetimes. Also, in general, the data store entries occupy ≈ eight times more storage than tag store entries. Based on these observations, we propose Maya, a storage efficient and yet secure last-level randomized cache that increases the tag store entries for security and reuse detection and uses fewer data store entries that only store the reused data. Our proposal provides a strong security guarantee, which is one set-associative eviction in 10^{32} line fills (10^{16} years) at the last-level cache, with 28.11% less area and 5.46% less static power when compared to a non-secure baseline of 16 MB cache.

We propose Avatar cache, a secure, morphable LLC with three modes—*non-secure* (Avatar-N), *randomized secure* (Avatar-R), and *partitioned secure* (Avatar-P)—and the ability to switch dynamically between them. Its design closely resembles a conventional set-associative LLC, facilitating industry adoption. Avatar-R provisions extra invalid entries and high associativity to provide a strong security guarantee—only one set-associative eviction per million years—while incurring just 1.5% storage overhead, a 2.7% increase in static power consumption, and a mere 0.2% slowdown compared to a non-secure 16 MB baseline LLC. Avatar can also morph into a partitioned LLC in Avatar-P to mitigate both conflict-based and occupancy-based attacks. Avatar-P has a slightly higher 3% performance overhead compared to the non-secure baseline, performing significantly better than a conventional way-based partitioned LLC. When security is not a priority, we can switch to Avatar-N, morphing into a traditional LLC to optimize performance and power efficiency.

Finally, we identify and systematically analyze the design knobs employed in state-of-the-art secure randomized cache designs that mitigate conflict-based attacks. Using conventional set-associative caches as our baseline, we study five key knobs: skewing, extra invalid tags, high associativity, replacement policy, and remapping. We also evaluate their impact on occupancy-based attacks. Our findings show that no single knob provides a comprehensive security guarantee. Instead, only specific combinations of knobs yield effective protection, while others offer little to no security benefit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Last-level Cache (LLC) hides off-chip memory access latency and improves system performance. Typically, L1 and L2 caches are dedicated to each core, while the shared LLC serves all cores. However, this shared LLC can facilitate side-channel attacks that may reveal sensitive data, such as cryptographic keys [8, 1, 9], user data in the cloud [10], and architecture of neural networks [11]. Three types of attacks are possible in set-associative LLCs: flush-based [12], conflict-based [1] and occupancy-based [13]. In general, these attacks exploit the timing difference between an LLC hit (fast) and a miss (slow). Flush-based attacks are shared memory attacks where the attacker and the victim *share* cache lines, and the attacker flushes the cache line from the LLC and checks for LLC hit and miss. A future reload hit means the victim has filled the cache line into the LLC. In contrast, conflict-based [1] and occupancy-based [13] attacks do not expect shared memory. Conflict-based attackers exploit LLC address-to-set mapping, causing set conflicts where victim lines evict attacker lines, known as a set-associative eviction (SAE). Attackers use SAEs to build an *eviction set*, a collection of addresses that map to the same LLC set. Another class of attacks, called occupancy attacks, involves occupying cache lines to estimate the proportion accessed by a victim process [13]. Occupancy attacks do not need eviction sets.

In recent years, several defense mechanisms have been proposed to counter these attacks, among which LLC randomization has emerged as a promising candidate for flush and conflict-based attacks, and recently for occupancy-based attacks [14]. There have been several proposals for a randomized LLC design. Some of the initial proposals such as RPCache [15], CEASER [16], CEASER-S [17], and Scatter-Cache [18] were quickly compromised by newer and faster attacks [17, 7, 19], and eventually improved designs were proposed. Recent secure randomized LLC designs, such as Mirage[20], introduce multiple microarchitectural modifications to a conventional set-associative cache to provide security. Mirage [20] provides a proxy for a fully associative cache by making insertion and eviction completely uncorrelated. To achieve this, Mirage implements a skewed tag store which is decoupled from the data store, and maintains a pointer-based mapping between the tag and data entries. This allows it to have a set-associative lookup and provision extra invalid tags in the tag store, keeping the data store size the same. Secondly, the eviction candidate for an incoming line is chosen randomly from the entire data store, providing global random evictions. The presence of extra invalid tag ways with a load-balancing insertion policy [20] ensures that there is always an invalid line available in the mapped set for the incoming line. This makes Mirage completely immune to conflict-based side-channel attacks.

However, most randomized LLC designs, including Mirage, fail to address the threat of occupancy-based attacks [21], leaving LLC partitioning as the only effective mitigation. Secure LLC partitioning techniques [14, 22, 23, 24, 25] mitigate occupancy-based attacks but incur a high performance overhead, as high as 49% (for `cam4` on 8-core 16 MB LLC with BCE [24]), and an average performance overhead of at least 5%. In addition, some partitioning techniques need OS support for fine-grained LLC partitions [22, 24]. A way-based partitioning technique needs minimal support from the OS. However, a 16-way LLC limits its scalability.

The problem. Mirage mimics a fully associative LLC in terms of the insertion and eviction of entries, making it secure against conflict-based eviction attacks. But this security comes at an additional cost. Mirage incurs a marginal performance overhead, an extra storage of 20% at the LLC with a static power overhead of 17.50%, which is a costly trade-off. Besides high overheads, Mirage also involves significant additional design complexity compared to a traditional non-secure set-associative cache. Its decoupled tag and data store necessitates pointers, further complicating implementation.

On the other hand, Mirage can only mitigate conflict-based attacks and fail to address the threat of occupancy-based attacks introduce substantial design complexity compared to traditional non-secure set-associative LLCs. Secure LLC partitioning techniques, which offer strong isolation against all LLC attacks, are expensive in terms of performance and scalability.

Additionally, the security guarantee of each of these designs is usually evaluated as a whole, without clarifying the role of each microarchitectural modification. Secondly, many of the designs propose similar modifications or identical security implications. Therefore, it is crucial to identify the set of security "knobs" or modifications used by these designs.
Our goal is to propose an LLC design that can provide the illusion of a fully associative cache and, hence, the security guarantee without significant storage, power, and performance overhead. We also want to design a secure LLC design that offers *security-on-demand*, i.e., one can choose security against LLC attacks only if required. If one chooses to enable security by operating in *secure* mode, it should come with low design complexity and runtime overhead, making industry adoption easier. At the same time, if security is not required, the same LLC design should seamlessly operate in *non-secure* mode, morphing into a traditional set-associative LLC. Finally, we want to answer questions such as (i) *why a security knob works and to what extent?* and (ii) *what is a minimal set of knobs to secure a conventional set-associative cache?* This modular systematization can enhance the understanding of randomized caches, helping designers and security analysts select optimal designs.

1.1 Key Insights and Contributions

(i) We propose Maya, a secure, fully associative, and randomized LLC, which is storage-efficient. The crux of our proposal is a decoupled tag and data store with additional tag entries but fewer data entries (Section 3.2).
(ii) We argue about the security guarantee of Maya in terms of the number of LLC line installs required to mount an eviction-based attack. We prove that such an attacker must perform more than 10^{32} LLC line installs (around 10^{16} years) to get one set-associative eviction, which is larger than the system lifetime (Section 3.3).
(iii) Maya provides a strong security guarantee without additional storage (storage savings of 2%). Maya saves the LLC area by 28.11%, and leakage power by 5.46% (Section 3.4).
(iv) We also present Avatar, a morphable LLC substrate that enables dynamic switching between *non-secure* (Avatar-N), *randomized secure* (Avatar-R), and *partitioned secure* (Avatar-P) modes. This allows the LLC to operate either as a secure cache or as a traditional set-associative LLC with minimal changes (Section 4.2).
(v) Avatar-R reimagines Mirage's design using high associativity without incurring its 20%

storage overhead by employing implicit tag-to-data indirection instead of explicit indirection. Avatar-P enables scalable, high-associativity way-based partitioning to defend against occupancy-based attacks (Sections 4.2 and 4.3).

(vi) We provide a framework for dynamic mode switching that ensures low runtime overhead and prevents new attack vectors. We show that it reduces the design and runtime overheads, making complete LLC security on-demand and yet practical (Section 4.2.3). We also quantify the performance, power, and area overheads (Section 4.4 and 4.5).

(vii) We iteratively analyze the impact of each knob on the attacker's achievable eviction rate, noting the conditions needed for each knob to be effective and explaining why it works. Combinations of different security knobs are also explored, which gives rise to some interesting designs not explored in prior literature. One important insight is that certain knobs are dependent on each other to provide security gains (Section 5.2).

(viii) We argue that decoupling the tag and data store does not affect the security of a cache design. The security of designs such as Mirage [20] and Maya [26] comes from the additional invalid tags, not from the decoupling itself, which primarily introduces overhead in terms of indirection pointers and design complexity (Section 5.2.3).

(ix) We identify high associativity as a powerful knob for mitigating conflict-based attacks. Our findings show that increasing cache associativity significantly improves security with minimal changes to conventional set-associative designs, making it a promising direction for secure randomized caches. While associativity up to 16 ways has been explored for security [27], we advocate for extending beyond 16 ways, up to 128 ways (Section 5.2.4).

(x) We examine how various security features impact occupancy-based attacks. First, we find that randomness in skew selection and eviction policy provides some protection, but it is not foolproof. While deterministic eviction policies like LRU are more effective against conflict-based attacks, they are less effective for occupancy-based attacks. Second, in full occupancy-based attacks, the eviction policy—global or local—has minimal impact on security. We evaluate one recent randomized cache design against occupancy-based attack, which provides soft partitioning (e.g., SassCache [14]), showing that it makes such attacks significantly harder than other randomized caches. We also analyze low-occupancy attacks (attacks that use buffers smaller than cache capacity), by extending prior evaluations [28] to high-associativity cache designs, showing that they offer similar resistance to such attacks as SassCache and ScatterCache (Section 5.3).

Chapter 2

Background and Related Works

This chapter discusses all the necessary background information required to understand the rest of the document. We discuss the various performance optimization techniques used to improve processor performance to date, as well as the security vulnerabilities they expose. Modern processors employ many techniques to get high performance from the available silicon. It is not just the increasing clock speed that has led to the high performance, but the ability to pump out multiple instructions in a single clock cycle while maintaining the program's correctness has led to performance improvement by numerous folds. Techniques like out-of-order execution, speculative execution, multi-threading, caching, and hardware prefetching are some key contributors. The traditional approach of increasing clock speeds has reached its limits. Hence, strategies to enhance the processor's overall throughput are now employed. Out-of-order and speculative execution enable processors to execute instructions concurrently, reducing idle time and enhancing efficiency. Multi-threading allows for the parallel execution of multiple threads, further leveraging the processing power of modern CPUs. Caching, a fundamental aspect of processor design, involves the use of high-speed memory to store frequently accessed data and instructions closer to the execution units. This minimizes the need for repeated access to slower main memory, accelerating processing speed. Hardware prefetching, another optimization technique, involves predicting and preloading data into the cache before it is explicitly requested, reducing latency and improving overall responsiveness. However, this singular approach of improving performance over the past few decades without critically analyzing their side effects has led to an insecure foundation. Before delving into the security implications, a thorough exploration of these performance-boosting techniques is imperative. By understanding the intricacies of how modern processors achieve their impressive speeds, we can better appreciate the trade-offs and challenges that arise in security.

2.1 Memory Hierarchy

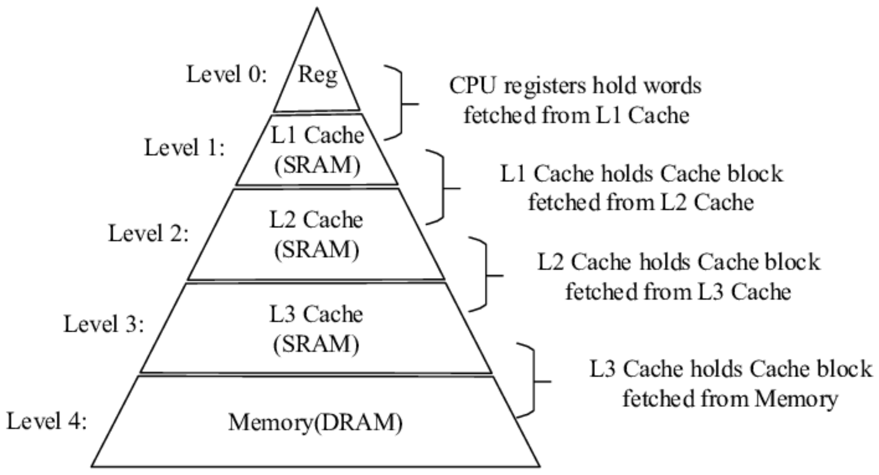
Modern computing is highly data-driven, which increases the demand for fast and large memory units. Most of the data actively used by processors resides in Dynamic Random Access Memory (DRAM). Ideally, programmers would have access to an extremely fast memory unit with unlimited storage capacity. However, DRAM technology has struggled to keep pace with the rapid advances in processing speeds. The mismatch between CPU and memory speeds leads to pipeline stalls, commonly referred to as the *memory wall problem*. Programs typically exhibit patterns where data that are close together or frequently used are accessed repeatedly. Temportal locality is the property that the same address is accessed repeatedly within a short span of time. Spatial locality, on the other hand, refers to the tendency for addresses that are physically close to one another to be accessed within a short time frame. These two properties, temporal and spatial locality, are key strategies for mitigating the memory wall problem.

2.1.1 Caches

To speed up data retrieval and keep up with modern computing demands, caches have been introduced. Caches are smaller, faster memory units that store frequently accessed data. This allows for quick access to memory that the CPU uses often. Caches are organized into blocks, each holding a set number of bytes or words. When data is retrieved from the cache, it's called a cache *hit*. However, the downside is that faster caches are typically smaller and can only store part of the program's working set—the actively used memory.

Modern processors solve memory access delays using multi-level caches, typically employing a set-associative structure [29]. The cache is split into sets, each containing ways of cache blocks. A cache set index (derived from part of the address) determines where a block is stored. When the processor accesses an address, it checks for a matching tag in the corresponding set. If it finds one (a *hit*), data is returned quickly. If it doesn't (a *miss*), the processor has to fetch the block from DRAM, which introduces a miss penalty.

The trade-off between cache size and speed is a balancing act. Larger caches reduce the miss rate but increase latency and power consumption. To bridge the gap, systems typically have multiple levels of caches. L1, the closest to the processor, matches the CPU's fast clock speeds, while L2 and LLC handle more data but with slightly higher latency. Most systems have a three-level hierarchy—L1, L2, and the LLC—where the LLC is shared by all processor cores. Figure 2.1 illustrates an overview of the memory hierarchy in moder processors.



Level 0:
Reg
CPU registers hold words fetched from L1 Cache
Level 1:
L1 Cache (SRAM)
L1 Cache holds Cache block fetched from L2 Cache
Level 2:
L2 Cache (SRAM)
L2 Cache holds Cache block fetched from L3 Cache
Level 3:
L3 Cache (SRAM)
L3 Cache holds Cache block fetched from Memory
Level 4:
Memory(DRAM)

Figure 2.1: An overview of the memory hierarchy in modern processors. As we move to higher levels, the access latency reduces but the memory capacity reduces.

2.2 LLC Side-Channel Attacks

In general, four kinds of LLC side channel attacks are prevalent that primarily exploit the latency (timing) difference between an LLC hit and a miss.

Conflict-based (or eviction-based) attacks. These attacks depend on the attacker observing the eviction of its cache lines due to addresses accessed by a victim, causing set conflicts. As a first step, the attacker identifies an eviction set (defined as a collection of addresses with a high degree of contention within a subset of cache lines). For example, such a subset of cache lines is constituted by an entire LLC set for conventional set-associative caches. Figure 2.2(a) illustrates the process of generating such an eviction set for a set-associative cache. Once the eviction set is established, the attacker can launch a conflict-based side-channel attack, such as Prime+Probe [1]. Figure 2.2(b) depicts a Prime+Probe attack, where the spy first (❶) *primes* (fills) an LLC set shared with the victim, and then

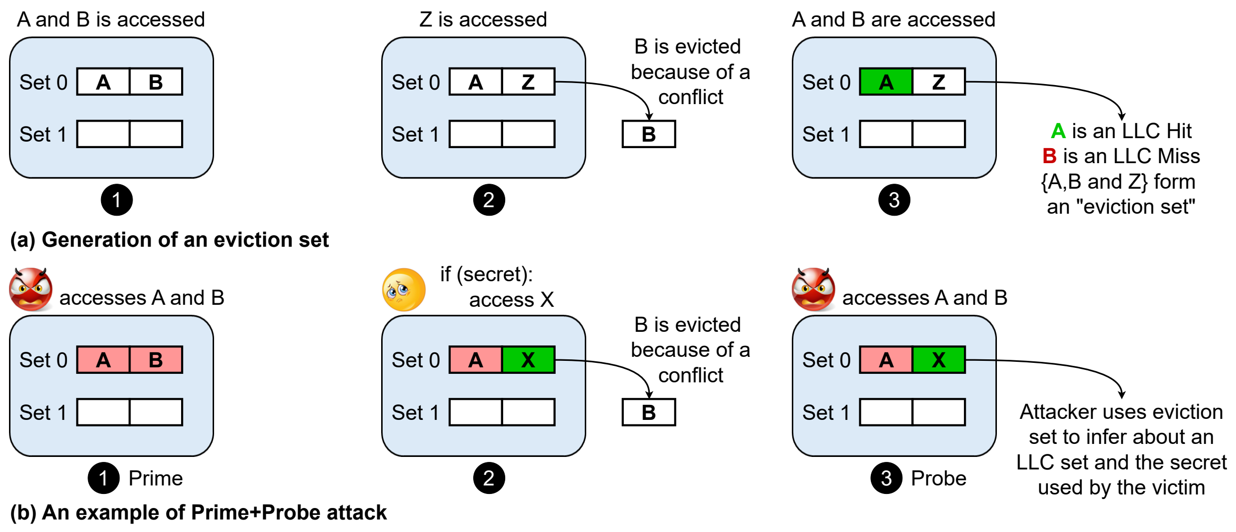

A and B is accessed
Set 0
A
B
Set 1
1
Z is accessed
Set 0
A
Z
Set 1
B is evicted because of a conflict
B
2
A and B are accessed
Set 0
A
Z
Set 1
3
A is an LLC Hit
B is an LLC Miss
{A,B and Z} form an "eviction set"
(a) Generation of an eviction set
accesses A and B
Set 0
A
B
Set 1
1 Prime
if (secret): access X
Set 0
A
X
Set 1
B is evicted because of a conflict
B
2
accesses A and B
Set 0
A
X
Set 1
3 Probe
Attacker uses eviction set to infer about an LLC set and the secret used by the victim
(b) An example of Prime+Probe attack

Figure 2.2: An overview of the steps involved in (a) generating an eviction set, and then using it to mount the (b) Prime+Probe [1] conflict-based attack which leaks information about the victim's secret-sensitive accesses.

(❷) the victim executes and potentially evicts the spy's cache lines, and finally (❸) the spy *probes* (checks) which lines were evicted to infer the victim's memory access behavior.
Occupancy-based attacks. These attacks rely on the attacker observing changes in its LLC working set due to the victim's cache usage. For example, in website fingerprinting [13], the attacker fills a significant portion of the LLC with its own cache lines and then lets the victim execute. During the *probe* step, the attacker can infer the victim's LLC occupancy based on evictions of its own lines. Unlike set-conflict-based attacks, no eviction set is required for these attacks, but they lead to more coarse-grained information leakage.
Shared memory-based (or flush-based) attacks. These attacks depend on the attacker observing hits on addresses shared with the victim. In the Flush+Reload [12] attack, the attacker *flushes* a shared line from the cache and waits for the victim to execute. In case the victim accessed the shared line, the attacker can infer this during the *reload* step. These shared lines usually consist of read-only shared memory from shared libraries or memory shared through de-duplication by the OS [30]. Such attacks are limited by the requirement of sharing of cache lines between distrusting processes.
Flush-based eviction attack. The attacker can mount an eviction-based attack by flushing her/his private data while creating an eviction set [31]. This technique accelerates the process compared to traditional eviction-based attacks like Prime+Probe. By flushing pri-

vate data and creating targeted cache contention, the attacker efficiently forces eviction of the victim's data from the cache. Note that this differs from shared memory-based flush attacks, where the attacker flushes shared cache lines from the LLC.

2.3 Eviction Set Creation Algorithms

For the rest of this paper, we abbreviate the number of sets in the cache using S, the number of ways using W, and the size of the cache using N. An eviction set is a collection of addresses built to detect the access of a particular target (victim) address or to evict the target (victim) from the cache. Such eviction sets are constructed using some heuristics as discussed below.

To create an eviction set, most existing eviction set search algorithms [1, 17, 7, 32] start from a large collection of candidate addresses (called *candidate set*). Then, they filter out addresses from this set that are *congruent* to this target address, and therefore, can evict the target address. Two addresses are said to be congruent if they map to the same subset of cache lines, and can evict each other with non-zero probability. In case this eviction probability is one, we call such addresses *fully congruent*. Otherwise, we say they are *partially congruent*. The **Single Holdout** [1] method tests one address at a time from the candidate set. It evaluates this address by testing whether the other addresses in the candidate set still evict the target. If not, this address is included in the eviction set, otherwise, it is excluded. This method requires $O(S^2 \cdot W^2)$ or $O(N^2)$ cache accesses, which is too slow to conduct any practical attacks. The **Group Elimination** [17, 33] method reduces the complexity to $O(N{\cdot}W)$ or $O(S{\cdot}W^2)$ accesses. They utilize the simple observation that if the candidate set is divided into $W+1$ groups, then one group certainly does not contain any congruent address and, therefore, can be removed. **Conflict Testing** [7], in contrast, builds the eviction set by testing each address (instead of starting from a candidate set) to check whether or not it is congruent with the target address. The time complexity of this algorithm is $O(S \cdot W^2)$.

In randomized caches, finding addresses that are always congruent to the target is highly unlikely, as the each address have multiple cache lines to map depending on the randomness. Therefore, attackers target partial congruence–addresses that are congruent with the target only with a certain probability. Algorithms like **Prime, Prune and Test** [7] (or Prime, Prune and Probe [19]) add a pruning step to remove addresses from the candidate set which are self-evicting, i.e., addresses which can evict each other with certain probability. The goal is to form a candidate set that remains in the cache, so accessing the target address and then re-accessing the candidate set reveals exactly those elements congruent to the target address. In terms of complexity, these algorithms are similar to conflict testing. Another optimization, **Prune+Plumtree** [32], salvages the pruned addresses to generate other eviction sets, ultimately reaching a complexity of $O(S \cdot W^2 \cdot \log(S))$ for caches with random replacement and $O(S \cdot W \cdot \log(S))$ with LRU replacement. However, the authors mention that "it does not apply to more advanced cache architectures such as skewed randomized caches".

2.4 Randomized LLC Designs

In a traditional set-associative cache, the address-to-set mapping is computed (deterministically) by a subset of the line address bits. Conflict-based attacks exploit this easy-to-discover determinism in address mapping to create efficient eviction sets. To counter this, randomized cache designs introduce *unpredictability* in the mapping, which makes the mapping hard to discover in the first place. However, this is not the only security-enhancing modification. In this section, we survey different generations of randomized caches.

The first generation of secure randomized caches aimed to randomize cache interference between an attacker and a victim process to prevent useful information leakage. One approach was **RPCache** [15], which used a permutation table to randomize the address-to-set mapping. This was followed by **CEASER** [16], which employed a block cipher to randomize the address-to-set-mapping. However, since set conflicts could still occur with block-cipher-based set indexing, CEASER introduced *dynamic remapping* to update the address-to-set mapping (by re-keying the block cipher) before an attacker could create an eviction set, making the mapping *updatable*. **PhantomCache** [34] uses localized randomization within a limited number of cache sets, effectively increasing cache associativity to make the discovery of eviction sets harder. Another secure randomized cache, **H^{2}Cache** [35], used a table-based randomization for the L1 cache and block-cipher-based randomization for the LLC.

The first generation of secure randomized caches was compromised by newer and faster eviction set discovery algorithms [17, 33]. In response, **CEASER-S** [17] and **Scatter-Cache** [18] introduced a skewed associative design on top of randomized address-to-set mapping, providing a probabilistic defense. In this design, the cache ways are partitioned into multiple skews, each with a unique set mapping, and the incoming addresses are randomly assigned to one of the skews. This obfuscates cache accesses and makes eviction set construction more difficult. These caches represent the second generation of secure randomized caches. A recent design, called **ClepsydraCache** [36], uses a similar randomized design to ScatterCache but uses time-based evictions instead of remapping of the set mapping used in ScatterCache.

In the third generation, **Mirage** [20] builds on the V-Way cache design [37], incorporating load-aware insertion and global random eviction. This approach significantly reduces the probability of set-associative evictions (SAEs), making it highly unlikely for attackers to construct eviction sets. **Maya** [26] is a variant of Mirage that optimizes space efficiency with a reuse-based insertion policy. Interestingly, a relatively older design, called **NewCache** [38], uses a register-based dynamic mapping along with randomization to eventually achieve a fully associative mapping. The abstraction of a fully associative cache was also achieved by the **Chameleon Cache** [39], a modification of CEASER-S featuring a fully associative victim cache with a random replacement policy to decouple set conflicts from evictions. **RECAST** [40] is a set-associative cache that uses core-private caches to protect against cross-core attackers. Each cache line is tagged with a secret bit, which determines the line-to-set mapping, thereby creating the abstraction of a fully associative cache. **Song et al.** [41] modified CEASER by introducing a mechanism that detects ongoing attacks and triggers re-keying.

The fourth generation of secure randomized caches was designed to defend against both occupancy-based and conflict-based attacks. *Cache partitioning* is the most straightforward approach as it provides LLC space isolation. **INTERFACE** [42] is a cache design which combines indirect partitioning and a randomized fully associative cache (like Mirage) to address both attack classes. A more interesting recent design is **SassCache** [14], which only relies on randomization of the address-to-set mapping to achieve security against occupancy attacks. In particular, SassCache enhances the address-to-set mapping of a skewed associative randomized cache like ScatterCache and enables LLC space isolation. As evident from the above survey, each generation of randomized caches introduces new modifications to address existing and potential future attacks. However, a notable gap in the current literature is the lack of an ablation study on the individual capabilities of these security modifications. Such a study would not only enhance our understanding of existing designs but could also inspire new, optimized designs that are better suited to varying resource constraints.

2.5 Secure LLC Partitioning Techniques

Page coloring [22] based LLC partitioning reduces cache conflicts by creating isolated regions at the LLC set level. However, this technique has limitations, as it cannot independently manage LLC and DRAM space. **Dynamically Allocated Way Guard (DAWG)** [23] employs a software-configurable mask to manage way allocations among multiple security domains on a multi-core system. However, the available LLC ways limit DAWG's maximum number of isolated domains. **Bespoke Cache Enclave (BCE)** [24] provides flexible cache partitioning to enhance data security, creating isolated cache partitions as small as 64 KB to protect against cache-based side-channel attacks. However, the partitions need to be re-sized and remapped after every allocation. **SassCache** [14] is a secure skewed associative cache with randomization that offers a soft-partitioning solution using cryptographic functions. However, it suffers from a significant increase in miss rate for memory-intensive GAP homogeneous workloads on an 8-core system, resulting in a performance slowdown of over 23%. **Ceviche** [43] achieves fine-grained partitioning of the cache using a capability-based cache lookup. However, Ceviche has a performance slowdown of more than 20% for memory-bound workloads on an 8-core system. This places both SassCache and Ceviche on the same level as LLC partitioning techniques, which have similar performance overheads. **TEE-SHirT** [25] provides cache security within Trusted Execution Environments (TEEs). TEE-SHirT mitigates cache side-channel leakage by restructuring the cache hierarchy to block information leaks while ensuring scalability. It enhances security through a combination of set and way partitioning. **SCALE** [44] employs dynamic partitioning by implementing bank-level way partitioning, where the number of partitions is constrained by the cache associativity multiplied by the number of banks. Dynamic cache-partitioning techniques improve cache utilization but at the cost of an occupancy attack. In general, static partitioning techniques completely mitigate occupancy-based attacks. Additionally, way-based partitioning is easy to implement and requires minimal OS support.

2.6 High Associativity Caches

Z-Cache [45] provides higher associativity than traditional set-associative caches with the same number of ways by employing a tag-store walk and a sequence of cache line reallocations, increasing the number of replacement candidates. However, these candidates are limited to

Table 2.1: Utility of randomized and partitioned LLCs.

Design	**Randomized**	**Partitioned**
Security	Conflict (✓) Occupancy (×)	Conflict (✓) Occupancy (✓)
Scalability	High	Low
OS support	No	Yes
Design time overhead	High	Low
Runtime overhead	Low	High

a subset of cache lines, leaving them susceptible to future, faster eviction-set-discovery algorithms. In contrast, Avatar-R eliminates set-associative evictions over a system's lifetime, preventing eviction-set discovery even by advanced future algorithms. **HybCache** [46] is a hybrid cache design combining various caching strategies to enhance performance and security. It uses adaptive techniques to dynamically balance between different cache policies, optimizing for workload demands and minimizing cache contention. However, the authors mention that "applying HybCache to the LLC or larger caches in general would be expensive (in terms of hardware)". Additionally, HybCache provides security against side-channel attacks at a performance overhead of up to 5%, significantly higher than the overhead guaranteed by Avatar-R. Note that cache designs such as Z-Cache and Phantom Cache achieve high associativity through re-allocations and remapping, but this alone does not fully prevent set-associative evictions. **Futility Scaling** [47] dynamically adjusts cache associativity within a partition based on performance. Unlike static partitioning or utility-driven approaches, it reduces associativity when the performance gains from additional ways diminish, ensuring more efficient cache utilization by adapting to workload behavior. This allows for increased performance, but unlike Avatar-P, it cannot protect against occupancy-based attacks.

2.7 Threat Model

We assume the following capabilities in our attacker:

- The attacker and the victim are running on two different cores on a multi-core system, sharing the LLC. Private caches are spatially and temporally partitioned among processes and flushed on context switches, providing isolation.

- The attacker can access the LLC by sending memory accesses to her data, but she cannot access any cache line that is not part of her own address space. However, she can accurately differentiate between an LLC hit and an LLC miss for her own data access.
- She can flush a cache block from the cache hierarchy as long as it is her own data. This way, she can expedite the process of creating an eviction set. She can make use of various eviction strategies to create an eviction set.
- She possesses all the design details of the randomized cache, but the encryption algorithm and the block cipher key used for set mapping remain concealed from the attacker.
- Her goal is to obtain the complete address or the cache index bits of the victim's address for eviction-based attacks.
- We exclude attacks on other micro-architecture units like on-chip interconnect, cache coherence directories, and other port contention-based attacks as we focus primarily on LLC attacks. We also exclude cache occupancy attacks as they are outside the scope of this study.
- Similar to other LLC partitioning techniques [22, 23, 48], we assume secure system software support (OS or security monitor [22]) is available.

Chapter 3

Maya Cache Design

3.1 Motivation

Mirage provides the illusion of a fully associative LLC, incurs a marginal performance overhead, and hence provides a sweet spot in terms of security and performance. However, it incurs an additional storage of 20% at the LLC with a static power overhead of 18.16%, which is a costly tradeoff. For example, for an 8-core system with 16 MB baseline LLC, the combined storage of tag store and data store is 16.91 MB, whereas Mirage has a storage requirement of 20.31 MB. This additional storage requirement leads to an increase in static power consumption, from 622mW with the baseline 16MB LLC to 735mW with the Mirage cache. This storage overhead increases linearly with higher number of cores, making it impractical for large-core systems.

Mirage enhances security by increasing the number of tag entries and introducing additional invalid tags. However, it doesn't alter the entries in the data store, which results in greater storage requirements. In Figure 3.1, we see the percentage of data store entries that remain dead (meaning they aren't reused after being added to the LLC) for both the baseline cache and Mirage when tested with the SPEC CPU2017 [2] and GAP [3] benchmarks. On average, over 80% of the data entries end up being dead, which takes up space in the LLC data store. This observation isn't new; it's a well-known finding within the micro-architecture community.

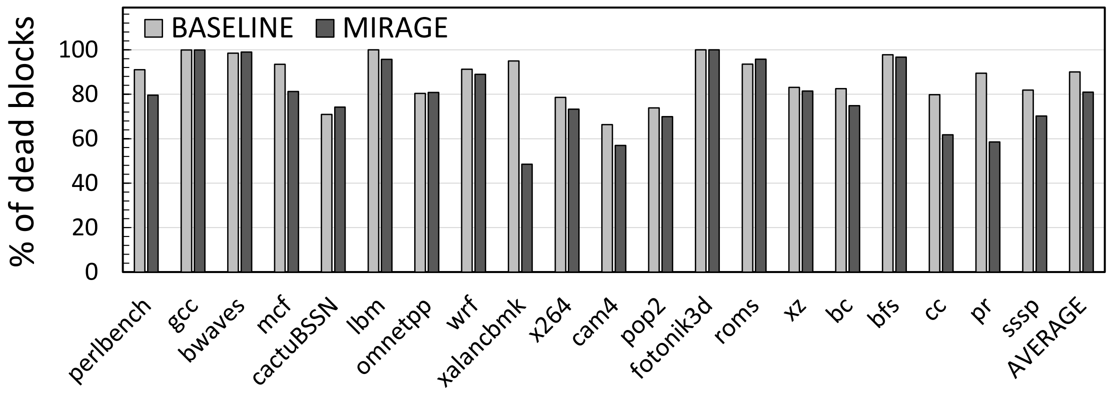


Figure 3.1: Percentage of dead blocks inserted into the LLC for SPEC CPU2017 [2] and GAP [3] benchmarks on a single core system for a 2 MB Mirage cache and a 2 MB non-secure baseline.

3.2 The Maya Cache

The Maya cache design has four key components:
(i) It uses a skewed-associative decoupled tag store with additional *invalid* tag entries for security. It also uses extra tag entries for *reuse* detection.
(ii) The tag store of Maya uses a new *priority* bit for each tag entry. Maya stores two kinds of tag entries: priority-0 and priority-1. Priority-0 entries have no associated data entry in the data store until they get a future tag hit and are promoted to priority-1 entries. Priority-1 entries are the entries in the tag store that have corresponding data-store entries.
(iii) For the tag store, Maya uses an insertion policy that keeps tag priorities in mind and is also equipped with *load-awareness* similar to Mirage [20]. Maya uses global random tag eviction in the tag store for only priority-0 entries. This ensures a fixed number of invalid tags are available in the tag store to avoid set-associative-evictions (SAEs).
(iv) Motivated by the Reuse Cache [49], Maya uses a smaller data store that stores entries, which will be *reused*. Maya uses a global random data eviction policy that evicts a data entry randomly *downgrading* its corresponding priority-1 tag entry to priority-0.

3.2.1 Tag and Data Store Design

Maya uses a skewed associative and decoupled cache design where each tag entry stores a forward pointer (FPTR), which allows it to point to an arbitrary data entry. A security

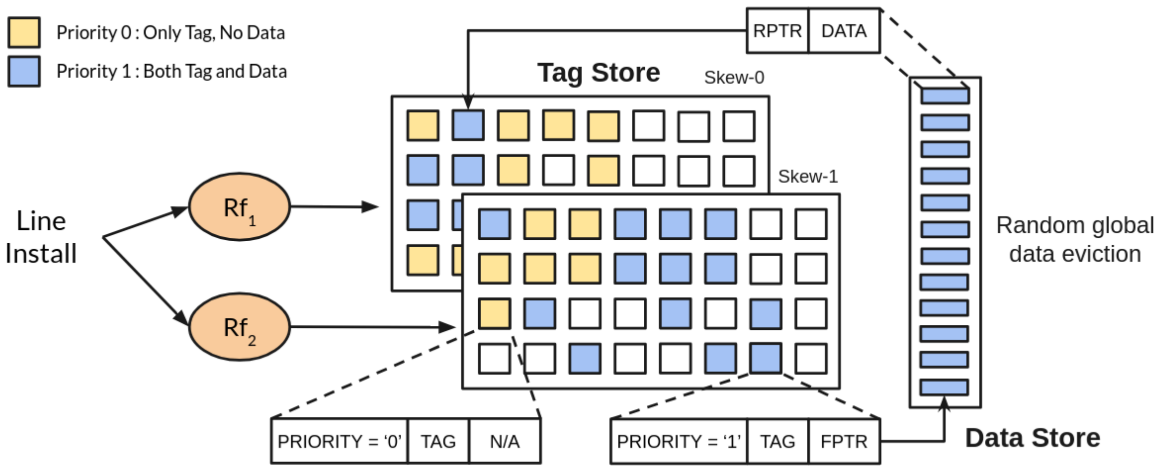


Figure 3.2: An overview of the Maya cache design.

domain ID (SDID) is also stored for each tag entry to help distinguish between copies brought in by different domains. This helps in mitigating shared memory attacks like Flush+Reload [12].

Skewed-associative design. The tag store is split into two skews [50], each with its independent hash function used to determine the set mapping for a cache line. Each incoming cache line now maps to a set in each of the two skews and can appropriately choose between the two sets.

Priority bit. We introduce an additional bit for each tag store entry, called the *priority* bit. This bit indicates if a valid tag entry has a corresponding valid data entry in the data store. If the priority bit for a valid tag entry is '0', it indicates no valid data entry exists corresponding to this tag. In this case, the forward pointer is invalid. On the contrary, if the priority bit is '1', a valid data entry exists in the data store, along with valid forward and reverse pointers linking these tag store and data store entries.

Extra tag store ways. In the Maya cache design, we provide extra invalid tag ways, similar to Mirage. We also provision additional ways, termed as *reuse ways*, to keep track of valid entries with priority-bit set to zero. These entries do not have a corresponding valid entry in the data store and, thus, an invalid forward pointer. Once such an entry gets a hit in the tag store, its priority is set to '1', and a valid data entry is assigned, along with appropriate forward and reverse pointers. The number of priority-1 entries in the tag store is the same as the total number of data store entries. The cache also holds a fixed number of priority-0 entries to ensure that the data store is only used to store "useful" entries. Additional invalid tags are reserved such that on every line install, there is at least one invalid tag available, and therefore, no SAE occurs. This helps provide security against eviction-based attacks. Note that the sets in the tag store are not statically partitioned for storing priority-0, priority-1, and invalid tag entries. Rather, the total number of entries of each type is kept constant in the tag store once the cache is operating at its maximum capacity.

Decoupled data store. As the tag store is decoupled from the data store, and the data store has fewer entries than the tag store, Maya needs to store the reverse pointer (RPTR), which points to the corresponding tag entry, for each entry in the data store. Figure 3.2 shows an overview of the Maya cache design.

3.2.2 Insertion and Eviction Policies

Insertion policy. Because of the skewed associative design of the Maya tag store, each new cache line gets mapped to two different sets (one in each skew). The decision of the skew chosen directly affects the distribution of valid tag entries (both priority-0 and priority-1) across the sets. This, in turn, affects the distribution of invalid tag entries available in each set, which is crucial for preventing SAEs and thus maintaining security. Previous works [18, 17] use random skew-selection, which randomly picks one of the skews for the incoming line to be installed in. However, such a policy could lead to an imbalance in the number of available invalid tags, where some sets may end up with no invalid tag and thus become prone to an SAE. Inspired by Mirage, we use a load-aware skew-selection policy, which fills the incoming line into the set with more invalid tags. This promotes balanced use of tags across sets, and an SAE can occur only if both the mapped sets do not have any invalid tag available, which is a rare occurrence. Experimental results show that with a load-aware skew selection and six extra invalid tag entries per skew, an SAE occurs once in 10^{16} years, well beyond the system lifetime.

Note that when a cache line gets filled into the LLC, the corresponding tag is stored in the tag store, with its priority bit set to 0. However, the associated data is not yet stored in the data store, which results in an LLC miss. Subsequently, when a request arrives again for the same tag, the priority is set to 1, the corresponding data is brought into the data store, and the data is available in the LLC for subsequent accesses.

Eviction policy. Maya uses a random global eviction policy, which chooses a random eviction candidate from the entire data store to ensure no information is leaked to an attacker. We term this as *global random data eviction.* When a priority-0 entry gets a hit in the tag store and is upgraded to a priority-1 entry, a random priority-1 entry is chosen globally for eviction from the data store, and its priority bit is reset to '0', thus downgrading it to a priority-0 entry. Maya also introduces global eviction of priority-0 entries from the tag store, called *global random tag eviction.* Such an eviction occurs every time a new priority-0 entry is brought into the tag store and a random priority-0 entry is invalidated from the tag store. These two eviction policies ensure that the tag store has a fixed number of invalid tag entries, which is crucial for security.

There can be a case where the tag store has not yet been filled up with the appropriate number of priority-0 entries (until all the reuse ways are filled). In such a case, when a new priority-0 entry gets filled into the LLC, we do not perform global random tag eviction. Similarly, if the data store is not full and a priority-0 entry needs to be upgraded to a priority-1 entry, then we do not perform global random data eviction.

States in the tag store. With Maya, a tag entry can be in one of three possible states: Invalid represents tag entries with their valid bit set to '0'. Priority-0 entries are valid, but their priority bit is set to '0', i.e. they have tag only and no data. Priority-1 entries are valid and with a priority set to '1', i.e. both tag and data are stored in the LLC. Dirty and clean denote if the corresponding data entry has been modified or is up-to-date with the main memory, respectively. Figure 3.3 shows all possible transitions between these states.

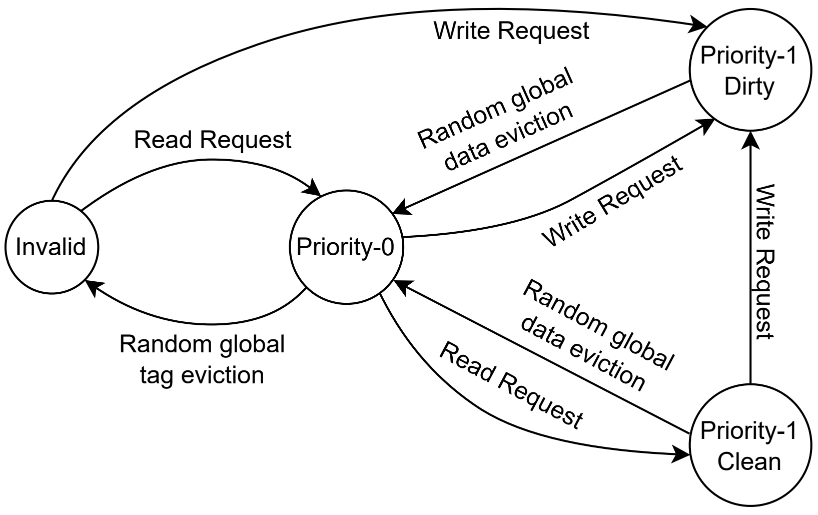


Figure 3.3: State transition diagram for tag-store entries in Maya.

A tag entry starts in the Invalid state when the LLC is initialized. When a demand read comes in for an invalid tag, it transitions to the priority-0 state, and a tag entry is assigned to this tag. If a write request comes for an invalid tag, it is automatically assigned both tag and data entries (priority-1) and marked as dirty. Once a priority-0 entry gets accessed, it is upgraded to a priority-1 entry, and its corresponding data is brought into the cache. It is marked as dirty or clean based on whether it was a write or a read request. When a clean priority-1 entry gets a write request, it is marked as dirty since its data is no longer up-to-date with the main memory. Priority-1 entries can transition to the priority-0 state if selected for *random global data eviction*, where a random priority-1 entry is selected and downgraded to a priority-0 entry. Similarly, a priority-0 entry can go to the invalid state if it gets chosen by *global random tag eviction*.

3.2.3 Implementation

Our goal with the Maya cache design is to find the sweet spot between performance, security, and storage overhead. According to the security simulations in Section 3.3, we require 6 extra invalid tag entries per skew such that no set-associative eviction occurs in the system lifetime. To compensate for the storage overhead of the larger tag store, we reduce the size of the data store to only 6 ways per skew (12 ways in total) instead of the 16 ways per set for the baseline. This, along with the 3 reuse ways per skew, leads to storage savings of around 2% compared to the baseline. If we reduce the data store size further, it will save more storage but also lead to performance loss.

Table 3.1 shows the security guarantees with different reuse and invalid tag ways per skew. We observe a reduction in the security guarantee as we increase the number of reuse ways because security is affected by the associativity of the tag store (as shown in Section 3.3). With six extra invalid ways and one reuse way, we get the storage-efficient Maya that provides the best security guarantee. However, with one reuse way, there is a marginal performance overhead (Figure 3.4). Therefore, we use Maya with three reuse ways per skew as it offers a sweet spot between performance, security, and storage.

Figure 3.4 shows that when we move from one reuse way to three reuse ways, it facilitates reuse prediction. Applications like `fotonik3d` see a normalized performance improvement from 0.97 to 1.04 when we move from one way to three ways. For five and seven reuse ways, there is a slight increase in tag lookup latency, which causes a minor performance drop. Note

Table 3.1: Cache line installs per SAE as the reuse ways vary from 1 to 7 ways with for 5 and 6 invalid ways per skew.

Reuse ways per skew	5 invalid ways per skew	6 invalid ways per skew
1-way	10^{18} (30 yrs)	$2{\cdot}10^{36}$ (10^{19} yrs)
3-ways	10^{16} (180 days)	$4{\cdot}10^{32}$ (10^{16} yrs)
5-ways	$6{\cdot}10^{15}$ (80 days)	$7{\cdot}10^{31}$ (10^{15} yrs)
7-ways	10^{15} (12 days)	$2{\cdot}10^{30}$ (10^{13} yrs)

that the ratio of number of ways for priority-0 to priority-1 entries for one, three, five, and seven reuse ways are $\frac{1}{6}$, $\frac{3}{6}$, $\frac{5}{6}$, and $\frac{7}{6}$, respectively. We do not change the data store entries for the sensitivity study. This is because each data store entry carries almost eight times the number of bits as compared to a tag store entry, which leaves little room for changing the data store size while keeping the storage same with different reuse ways.

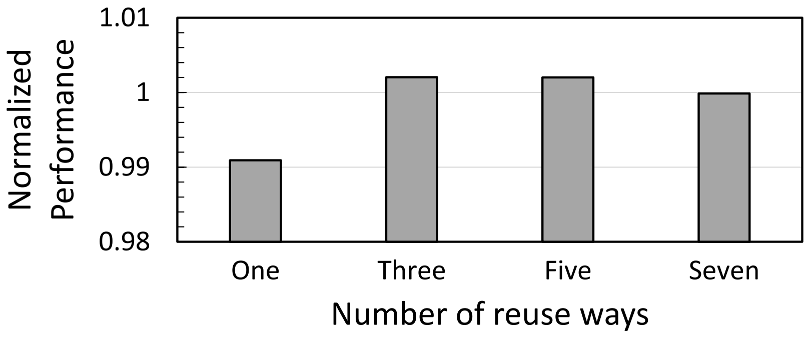


Figure 3.4: Effect of the number of reuse ways per skew on the performance of Maya cache, normalized to the non-secure baseline LLC for SPEC CPU2017 homogeneous mixes.

Each tag entry holds 40 tag bits for a 46-bit line address. Three coherence bits for the MOESI coherence protocol and one priority bit are also stored for each tag entry. To map to an arbitrary data entry, an 18-bit FPTR is used. The SDID helps keep track of the domain responsible for bringing in a cache line to allow duplication of shared cache lines. This ensures that the LLC fills of one domain do not affect the fills of another. Maya uses an 8-bit SDID for supporting up to 256 domains. In total, we use a total of 70 bits for each tag entry.

The tag store in Maya is split up into two skews, each with 16K sets (same as a non-secure baseline). Each set consists of six base ways per skew (total 12 ways, corresponding to the number of priority-1 entries), three reuse ways per skew (corresponding to the number of priority-0 entries), and an additional six invalid ways per skew to help maintain system security. With this, we get 192K (16K $\times$ 6 ways) priority-1 entries, 96K (16K $\times$ 3 ways) priority-0 entries, and 192K (16K $\times$ 6 ways) invalid tag entries in the tag store, resulting in 480K total tag store entries. This, multiplied by the total tag bits (70), leads to a tag store of size 4.1 MB.

The data store has 192K entries, each storing 512 bits of data (64 B cache lines). A data store entry can map to any arbitrary tag store entry, requiring a 19-bit RPTR. A total of 531 bits are stored for each data store entry, resulting in a total data store size of 12.44 MB.

For the randomizing function, we use a 12-round PRINCE cipher [51], which is a 64-bit block cipher using 128-bit keys. It is optimized for latency and has been used in previous works such as [18, 20]. This adds latency of three cycles for every LLC lookup. We also assume one additional cycle for tag and data lookup because of the tag-to-data indirection. In total, the Maya cache design requires four additional cycles per lookup.

3.3 Security Analysis of Maya

Recent advances in eviction-based attacks show that only a few SAEs are required to construct an eviction set and break security. Mirage argued that if an LLC design ensures that no SAEs occur in the system's lifetime, it potentially mitigates future attacks that could break the security of an LLC even with a single SAE. To guarantee security even against such strong attacks, we show that even a single SAE is highly unlikely to occur in the system's lifetime with Maya, and thus, it guarantees security. We consider the worst-case scenario, where every LLC access is a miss as it increases the chances of getting SAEs. An LLC miss can be classified into three categories: demand tag miss, demand or writeback tag hit with priority-0 entry, and writeback tag miss. All these cases cause a change in the tag store state, either by changing the number of entries in a set or changing the composition of a set. Note that a tag hit to priority-1 entry does not lead to any fills in the tag store or data store. so we skipped it for the worst-case scenario. Our security evaluation accommodates all these categories of LLC miss.

3.3.1 Bucket-and-Balls Model

To estimate the probability of an SAE for the Maya cache, we use a variation of the bucket-and-balls model as used in [20]. The buckets represent cache sets, the balls denote tag entries, and a ball throw represents a fill. With Maya tag store, we can have two types of balls: priority-0 and priority-1. Priority-0 balls represent tag entries with the priority bit set to '0' (only tag and no data), whereas priority-1 balls represent tag entries with the priority bit set to '1' (both tag and data). The buckets are initialized with a fixed number of priority-0 and priority-1 entries to model the Maya tag store design. This ensures that we model the best-case scenario for the attacker. Table 3.2 provides the parameters used for the bucket-and-balls model for a 12 MB Maya cache. For the experiment, each iteration consists of three types of LLC accesses that affect the distribution of balls in the tag store.

Table 3.2: Parameters used for the Bucket-and-Balls Model.

Bucket-and-Balls Model	**Maya Cache Design**
Total priority-0 balls - 96K	Total priority-0 entries - 96K
Total priority-1 balls - 192K	Total priority-1 entries - 192K
Skews - 2	Skews - 2
Buckets/skew - 16K	Sets/skew - 16K
Average priority-0 balls/bucket - 3	Average priority-0 entries/set - 3
Average priority-1 balls/bucket - 6	Average priority-1 entries/set - 6
Bucket capacity - 9 to 15	Ways per skew - 9 to 15

Empirical results. Figure 3.5 shows the expected number of iterations required to get a bucket spill with the given bucket capacity. As we increase bucket capacity from 9 to 15, the frequency of spills drastically reduces. We observe no spills for bucket capacities 14 and 15, and it is impractical to compute the spill frequency for these configurations in a reasonable amount of time (an experiment with one trillion iterations already takes a few days to simulate). We now demonstrate an analytical model to estimate the probability of a spill for 14 and 15 ways/skew.

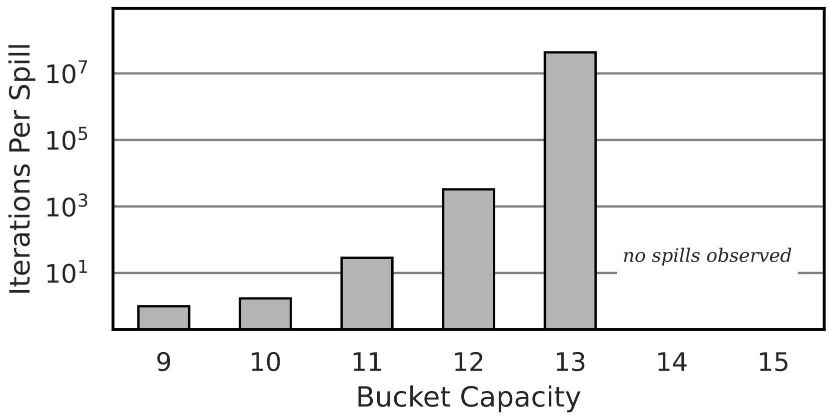


Figure 3.5: Number of iterations required to cause a bucket spill. As the bucket capacity increases from 9 to 13, the frequency of bucket spills reduces.

3.3.2 Analytical Model

Using bucket-and-ball simulations for one trillion iterations (three trillion different accesses), we observe no bucket spills for bucket capacities 14 and 15. We propose an analytical approach based on the bucket-and-balls model to estimate these cases' spill frequency. To analytically calculate the probability of a bucket spill, we create a model of our buckets-and-balls system in a spill-free scenario, where the buckets have unlimited capacity. The number of balls in a bucket is modeled as a Birth-Death Markov chain [52], where the birth event corresponds to a ball insertion and the death event to a ball removal. Refer to Table 3.3 for the terminology used in the model. A classic result for Birth-Death chains says that the net conversion rate between any two states (here, state refers to the number of balls in a bucket) becomes zero in the steady state. Using this result, we obtain Equation 3.1, which equates the transition probability from N to $N+1$ balls to the transition probability from $N+1$ to N balls for a bucket.

$$Pr(N \rightarrow N+1) = Pr(N+1 \rightarrow N) \tag{3.1}$$

A bucket transitions from N to $N+1$ balls on a ball throw in one of three cases: (i) both buckets randomly chosen from skew-1 and skew-2 have N balls; (ii) the random bucket chosen from skew-1 has N balls and the random bucket from skew-2 has more than N balls; or (iii) the random bucket chosen from skew-2 has N balls and the one from skew-1 has more

Table 3.3: Terminology used in the analytical model.

Symbol	**Interpretation**
$Pr(X \rightarrow Y)$	Probability that a bucket with X balls transitions to Y balls
$Pr(n=N)$	Probability that a bucket contains N balls
$Pr(n_0=X, n_1=Y)$	Probability that a bucket contains X priority-0 balls and Y priority-1 balls
$Pr(n_0=X \mid n=Y)$	Probability that a bucket contains X priority-0 balls and Y total balls
$\mathbb{E}_X[n_0=X \mid n=Y]$	Expected number of priority-0 balls in a bucket with Y total balls
B_{tot}	Total number of Buckets (32K)
b^0_{tot}	Total number of priority-0 balls (96K)

than N balls. The transition probability from N to $N+1$ balls is given by Equation 3.2.

$$Pr(N \rightarrow N+1) = Pr(n=N)^2 + 2 \times Pr(n=N) \times Pr(n>N) \tag{3.2}$$

For a bucket, the transition from N+1 balls to N balls can occur only on a global random tag eviction wherein a random priority-0 ball is globally selected for removal from all the balls. The probability of choosing a ball in a bucket with $N+1$ balls is given by the following equation:

$$Pr(N+1 \rightarrow N) = \frac{B_{tot} \times \sum_{r=1}^{r=N+1} \big(r \times Pr(n_0=r, n_1=N+1-r)\big)}{b^0_{tot}}$$

Here, r represents the number of priority-0 balls in a bucket with a total of N+1 balls. r varies from 1 to N+1 since a bucket with no Priority-0 balls will never be selected for Global random tag eviction.

Using B_{tot}/b^0_{tot} = (1/3) (number of buckets/priority-0 balls) and splitting $Pr(n_0=r, n_1=N+1-r)$ into the conditional probability, $Pr(n_0=r \mid n=N+1) \times Pr(n=N+1)$, we obtain the following equation:

$$Pr(N+1 \rightarrow N) = \frac{\sum\limits_{r=1}^{r=N+1} \big(r \times Pr(n_0=r \mid n=N+1) \times Pr(n=N+1)\big)}{3}$$

The expression $\sum_{r=1}^{r=N+1}(r \times Pr(n_0\!=\!r|n\!=\!N\!+\!1)$ simplifies to $\mathbb{E}_r[n_0\!=\!r|n\!=\!N\!+\!1]$, which provides the Equation 3.3.

$$Pr(N\!+\!1 \!\rightarrow\! N) = \frac{\Big(\mathbb{E}_r[n_0\!=\!r|n\!=\!N\!+\!1] \times Pr(n\!=\!N\!+\!1)\Big)}{3} \tag{3.3}$$

Since priority-0 balls constitute a (3/9) fraction of the total balls in the LLC (refer Table 3.2), $\mathbb{E}_r[n_0\!=\!r|n\!=\!N\!+\!1] = (3/9)(N\!+\!1)$, and Equation 3.3 simplifies to Equation 3.4.

$$Pr(N\!+\!1 \!\rightarrow\! N) = \frac{(N\!+\!1) \times Pr(n\!=\!N\!+\!1)}{9} \tag{3.4}$$

Using the earlier results from Equation 3.1, 3.2, and 3.4, we get a recursive relation for $Pr(n\!=\!N)$ as given in Equation 3.5

$$Pr(n\!=\!N\!+\!1) = \frac{9}{N\!+\!1} \times \Big(Pr(n\!=\!N)^2 + 2 \times Pr(n\!=\!N) \times Pr(n\!>\!N)\Big) \tag{3.5}$$

As we increase n, $Pr(n = N) \rightarrow 0$ and therefore $Pr(n > N) \ll Pr(n = N)$. Using this approximation, Equation 3.5 can be simplified to Equation 3.6 for larger values of n (we use this approximate equation once $Pr(n\!=\!N)$ becomes smaller than 0.01).

$$Pr(n\!=\!N\!+\!1) = \frac{9}{N\!+\!1} \times Pr(n\!=\!N)^2 \tag{3.6}$$

We simulate the bucket-and-ball model for one trillion iterations and obtain the probability of a bucket with no balls as $Pr_{exp}(n = 0) \approx 7.7 \times 10^{-7}$. Using this value in Equation 3.5, we recursively calculate $Pr_{est}(n = N\!+\!1)$ for $N \in [1, 12]$, and then use Equation 3.6 for $N \in [13, 15]$, when the probabilities become less than 0.01. Figure 3.6 shows that the estimated values closely match the experimental values. Using the above-described analytical model, we obtain the spill probabilities for 14 and 15 ways.

Frequency of spills. Using the analytical model described above, we calculated a probability estimate for $N = 14, 15$. If we consider a cache design with W ways per skew, the probability of an SAE (or a bucket spill) will be given by $Pr(n\!=\!W\!+\!1)$. The spill probability follows a double-exponential reduction, as seen in Figure 3.6. For $W = 13, 14, 15$, an SAE occurs every 10^8, 10^{16}, and 10^{32} line installs, respectively. Thus, the Maya cache design with 15 ways per skew has a frequency of one SAE in $4 \cdot 10^{32}$ line installs or once in around 10^{16} years, effectively providing complete security against eviction-based attacks.

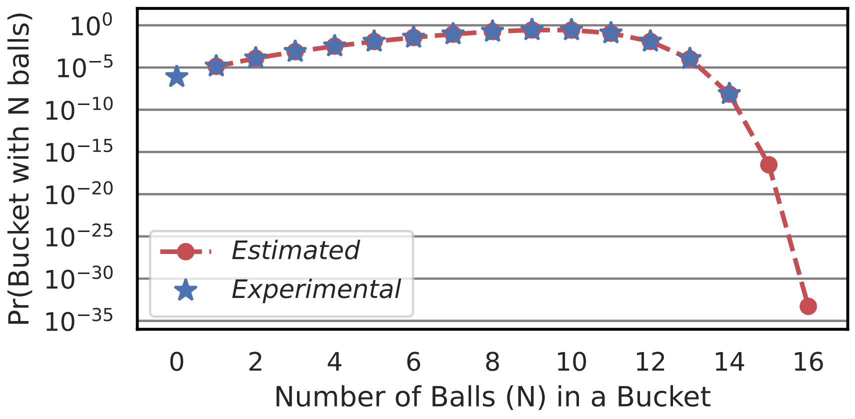


Figure 3.6: Probability of a bucket having N balls (Pr(n = N)) - experimental and estimated using the analytical model.

Key management. The key used in Maya is set during the system boot. Although the probability of an SAE is significantly low, in the event of an SAE, the key used in the cipher for set mapping is refreshed followed by a cache flush.

Sensitivity to associativity. We now vary the associativity of the Maya tag store, keeping the data store size at 12 MB. The base associativity varies from 8 to 36 ways, with the default configuration having 18 total ways (6 base and 3 reuse ways per skew). Table 3.4 shows the rate of SAE for these configurations. We can observe that for the same extra invalid ways per skew, the 8-way configuration is the most secure (one SAE in 10^{23} years), and security reduces as the base associativity increases. However, even for the 36-way configuration, the rate of SAE is once in 10^{10} years, which is beyond the system lifetime.

Table 3.4: Cache line installs per SAE as the base associativity of the tag-store varies from 8 ways to 36 ways. 18-ways (6+3): On average, it consists of 6 base and 3 reuse ways/skew.

	Associativity		
Invalid Ways	**8-ways (3+1)**	**18-ways (6+3)**	**36-ways (12+6)**
4 extra ways/skew	10^{10} (7 s)	10^{8} (0.1 s)	10^{7} (9 ms)
5 extra ways/skew	10^{20} (10^{3} yrs)	10^{16} (180 days)	10^{14} (1 day)
6 extra ways/skew	10^{40} (10^{23} yrs)	10^{32} (10^{16} yrs)	10^{28} (10^{11} yrs)

3.3.3 Need for Domain IDs

In situations where attacker and victim do share cache lines, various attacks like Flush+Reload [12], Flush+Flush [53], Flush+Prefetch [54], and Evict+Reload [55], could potentially leak victim data.

Mirage stores the Security-Domain-ID (SDID), which is 8 bits, denoting the domain installing the line along with the tag of the line in the tag store. This guarantees the duplication of shared lines. Similarly, Maya includes an 8-bit SDID for each tag entry, accommodating a maximum of 256 domains. This ensures that the LLC fills of one domain do not affect the fills of another domain, and thus, the system is secure against shared-memory attacks. The length of the SDID can be adjusted to support more or fewer domains, depending on the requirement. Maya also mitigates Reload+Refresh [56] attack as it guarantees global evictions with random replacement.

Maya and the private caches. In the case of a private L1 or L2 cache, which is mostly shared by a 2-way simultaneous multi-threading (SMT) processor, it is relativity an easy design choice to partition the private L1 or L2 among two threads as done in the prior works [57]. Usage of randomization, global random replacement, and additional reuse ways at L1 can lead to performance degradation as high as more than 10%. Similarly, the cache coherence directory can be partitioned [58]. So overall, the cache hierarchy will have heterogeneous solutions for security.

3.3.4 The Cat and Mouse Game

Sophisticated attacker. One can argue that the timing difference between accessing a priority-1 entry and a priority-0/invalid tag entry can be exploited to mount a new timing-based side-channel attack. However, in the Maya cache design, the entries owned by the victim and the attacker have different SDIDs and are filled in isolation. The only way the attacker can exploit the reuse-based fills is by mounting an eviction-based attack, which we have already shown to be impossible since there are no set-associative evictions for 10^{16} years.

Cache occupancy attack and Maya. The Maya cache, by design, does not mitigate cache occupancy attacks and even a fully associative cache is prone to cache occupancy attacks. However, the Maya cache, while mitigating conflict-based attacks, should not make it easier to mount an occupancy-based attack. To evaluate the same, we mount an LLC occupancy-based attack. We attack AES (OpenSSL implementation with T tables) and modular exponentiation and compare the number of encryptions required to break the keys using cacheFX [27]. To make it a strong attack, we simulate AES and modular exponentiation with two different keys, each having different reuse profiles at the LLC so that an attacker can exploit the Maya cache. The goal of the attacker is to distinguish these keys based on the reuse profiles. We run the attack 1,000,000 times and report the median of number of encryptions required for distinguishing the keys.

Normalized to a fully associative cache that uses a random replacement policy, the Maya cache behaves almost similarly (not the same) to a fully associative cache, with normalized values of 0.996 and 0.992 for AES and modular exponentiation, respectively (Figure 3.7). Note that we normalize the number of encryptions to the number of encryptions required for a fully associative cache (10590 for AES and 94 for modular exponentiation). As expected, a 16-way associative cache makes it easier to mount an attack, with normalized encryptions of 0.85 (15% easier) and 0.63 (37% easier) for AES and modular exponentiation, respectively.

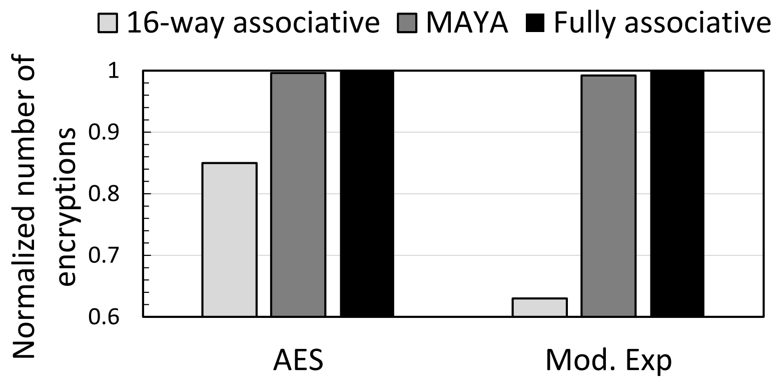


Figure 3.7: LLC occupancy attack: number of encryptions required to break AES and modular exponentiation with a 16-way associative cache, Maya cache, and a fully associative cache. The number of encryptions is normalized to a fully associative cache with a random replacement policy.

3.4 Performance Analysis

3.4.1 Methodology

We use the ChampSim [59] micro-architecture simulator to evaluate different cache designs. We use a non-secure 8-core 16 MB, 16-way set-associative last-level cache with 64-byte cache lines as the baseline. Table 4.4 provides the simulated parameters of the baseline non-secure system configuration. We evaluate 42 homogeneous workloads created from 42 different sim-points from the SPEC CPU2017 benchmark suite [2] and 20 homogeneous workloads from 20 different sim-points from the GAP benchmark suite [3], with more than one LLC miss per kilo instruction (MPKI) for the baseline configuration. Note that we select benchmarks based on their LLC MPKI for a single core 2 MB LLC. We also use a set of 21 heterogeneous mixes with randomly chosen benchmarks from the SPEC CPU2017 and GAP suites. Table 3.6 shows the heterogeneous mixes representing the behavior of more than 1000 heterogeneous mixes. Table 3.7 shows the average LLC MPKI for a 16 MB LLC with eight cores of SPEC CPU2017 and GAP homogeneous mixes and heterogeneous mixes, respectively. Note that with Maya, there are tag-only misses at the LLC on top of tag+data misses.

Table 3.5: Simulation parameters of the baseline system.

Core	8 cores, Out-of-order, bimodal [60], 4 GHz with 6-issue width, 4-retire width, 512-entry ROB
TLBs	L1 ITLB/DTLB: 64 entries, 4-way, 1 cycle, STLB: 2048 entries, 16-way, 8 cycles
L1I	32 KB, 8-way, 1 cycle, LRU
L1D	48 KB, 12-way, 5 cycles, LRU, IPCP prefetcher [61]
L2	512 KB 8-way associative, 10 cycles, LRU, non-inclusive
LLC	2 MB/core, 16-way, 24 cycles, Hawkeye [4], non-inclusive
MSHRs	8/16/32 at L1I/L1D/L2, 64/core at the LLC
DRAM controller	DDR4-3200, two channels/8-cores 4 KB row-buffer per bank, open page, burst length 16, $t_{\text{RP, RCD, CAS}}$: 12.5 ns

We simulate 1.6B instructions for eight cores (200M instructions per core in the region of interest after a warmup of 200M instructions per core). We use the weighted speedup [62] performance metric to compare the performance of different cache designs for an 8-core multi-core system. We compare the performance of the Maya cache design with a non-secure baseline and the Mirage cache design. We also perform a sensitivity study on the LLC size per core and later analyze the performance of Maya for higher-core systems.

Table 3.6: Heterogeneous mixes as per Table 3.7.

Mix	Composition	Bin
M1	cactuBSSN(2)-wrf(1)-xalancbmk(1)-pop2(1)-roms(1)-xz(1)-sssp(1)	L
M2	bwaves(1)-mcf(1)-cactuBSSN(1)-wrf(1)-xalancbmk(1)-xz(1)-bfs(1)-sssp(1)	L
M3	mcf(1)-cactuBSSN(1)-omnetpp(1)-xalancbmk(1)-roms(1)-bfs(1)-cc(1)-sssp(1)	L
M4	perlbench(1)-bwaves(1)-mcf(3)-cam4(1)-xz(1)-bc(1)	L
M5	perlbench(1)-mcf(2)-cactuBSSN(1)-roms(1)-xz(1)-bc(1)-pr(1)	L
M6	gcc(1)-mcf(2)-cactuBSSN(1)-lbm(2)-fotonik3d(1)-roms(1)	L
M7	bwaves(1)-mcf(1)-cactuBSSN(1)-pop2(1)-xz(1)-bc(2)-sssp(1)	L
M8	gcc(2)-bwaves(1)-x264(1)-bc(1)-cc(1)-pr(1)-sssp(1)	M
M9	gcc(1)-cactuBSSN(1)-lbm(1)-xalancbmk(1)-x264(1)-cam4(1)-pr(1)-sssp(1)	M
M10	mcf(3)-lbm(1)-wrf(1)-fotonik3d(2)-sssp(1)	M
M11	mcf(3)-lbm(1)-omnetpp(1)-pop2(1)-roms(1)-cc(1)	M
M12	mcf(2)-cactuBSSN(1)-fotonik3d(1)-roms(2)-cc(1)-pr(1)	M
M13	bwaves(1)-mcf(1)-xalancbmk(1)-fotonik3d(1)-roms(2)-bc(1)-sssp(1)	M
M14	mcf(1)-lbm(1)-xalancbmk(1)-roms(1)-bc(1)-cc(1)-sssp(2)	M
M15	bwaves(1)-cactuBSSN(1)-lbm(1)-roms(2)-bfs(1)-pr(1)-sssp(1)	H
M16	mcf(3)-cactuBSSN(1)-lbm(1)-bfs(2)-cc(1)	H
M17	mcf(1)-cactuBSSN(1)-wrf(1)-xalancbmk(1)-x264(1)-bc(1)-pr(2)	H
M18	omnetpp(1)-wrf(1)-fotonik3d(1)-roms(1)-bc(2)-cc(1)-sssp(1)	H
M19	bwaves(1)-mcf(2)-cactuBSSN(1)-xalancbmk(1)-bfs(1)-pr(1)-sssp(1)	H
M20	perlbench(1)-mcf(2)-omnetpp(1)-fotonik3d(1)-pr(1)-sssp(2)	H
M21	gcc(1)-bwaves(1)-mcf(2)-lbm(1)-bc(1)-pr(2)	H

3.4.2 Performance

Homogeneous mixes. Figure 3.8 shows the performance for Maya and Mirage normalized to the baseline for various homogeneous SPEC and GAP workloads. For the SPEC benchmarks, on average, Maya outperforms Mirage marginally, with a marginal performance improvement over the baseline. For benchmarks such as `mcf`, `wrf`, `fotonik3d`, we observe a substantial increase in performance for Maya despite the higher LLC latency. We observe that for these benchmarks, Maya reduces the inter-core interference in the data store by more than 70% compared to the baseline due to the notion of storing only "useful" entries in the data store. This compensates for the higher access latency and results in a performance gain

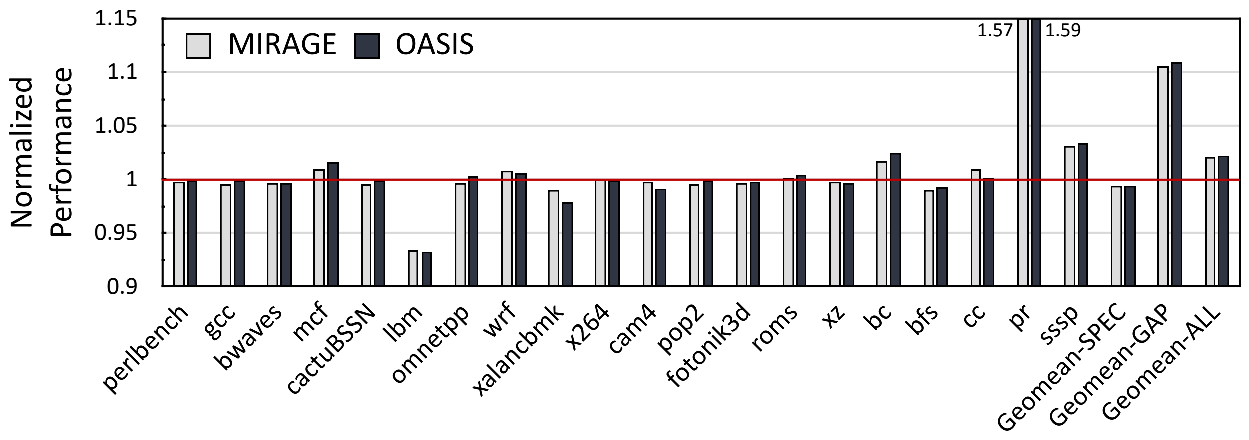


Figure 3.8: Performance of Maya for 8-core homogeneous mixes.

for Maya. For benchmarks such as `lbm`, `cactuBSSN`, and `cam4`, Maya incurs a performance slowdown compared to the baseline. The `cam4` and `cactusBSSN` workloads have a relatively low dead block percentage in the LLC and low inter-core interference, even for the baseline. Therefore, it benefits from the larger data store of the baseline and Mirage, and we observe a performance slowdown of with Maya. For `lbm`, which is a streaming workload with almost zero load hit rate in the LLC, Mirage incurs a slowdown of around 8% compared to the baseline because of the extra 4-cycle access latency.

Table 3.7: Average LLC MPKIs.

Workloads		Baseline	Mirage	Maya
SPEC and GAP-RATE		13.9	12.5	12.5
HETERO	**LOW**	8.01	8.05	8.53
	MEDIUM	14.72	14.73	15.31
	HIGH	21.51	21.48	21.04

On average, Maya performs 5% better than the baseline for GAP workloads. The average improvement is influenced by 50% performance improvement with `pr`. For the `pr` workload, Mirage and Maya deliver 57% and 50% better performance than the baseline, respectively. This trend is contributed by a weak baseline for `pr`, where the IPCP prefetcher impacts the baseline performance as it behaves worse than no prefetching and LRU policy. The `cc` workload incurs a 5% performance slowdown compared to the baseline due to an increase in the inter-core interference in the data store.

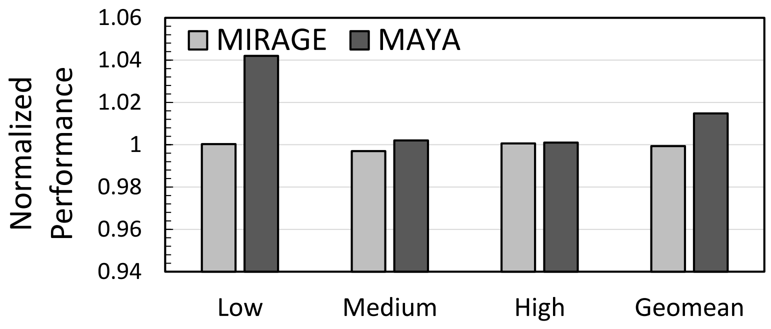


Figure 3.9: Performance of Maya for 8-core heterogeneous mixes.

Heterogeneous mixes. For the heterogeneous workloads (shown in Figure 3.9), Maya shows a 1.5% average performance improvement compared to the baseline, whereas Mirage incurs a marginal performance slowdown. Maya shows an improvement of more than 4% in performance for low-MPKI mixes because of the reduction in inter-core interference. Whereas medium-MPKI and high-MPKI mixes get a marginal performance slowdown because of their large working sets. In general, Maya helps improve performance for workloads with high inter-core interference and a high dead block percentage in the LLC. Note that in many mixes, Maya increases the miss rate at the LLC as it does not fill the cache line into the LLC on its first miss, providing tag-only hits. However, overall, it improves the performance as the useful entries are retained.

Performance of LLC fitting benchmarks. As Maya reduces the data store sizes, benchmarks that fit into the LLC may result in performance slowdowns. We simulate LLC fitting benchmarks from SPEC CPU2017 (LLC MPKI less than 0.5) and observe an average performance loss of 0.63% compared to a non-secure baseline.

Impact of random global tag eviction on performance. Random global tag evictions enhance the security provided by Maya. However, it can impact performance when a priority-0 entry that is yet to get reused (to be promoted to priority-1) gets invalidated by random global tag eviction. We quantify this event across all homogeneous mixes, and on average, less than 0.022% of the random global tag evictions to priority-0 entries would have gotten reused if we had used a non-random policy. We use the SRRIP [63] policy as the non-random policy.

Sensitivity to LLC size. For the Maya cache, we used an LLC data store size of 12 MB (1.5 MB per core). We now evaluate the performance of Maya with 6 MB to 96 MB data stores (baseline LLC size varying from 8 MB to 128 MB). Note that we also scale the tag store proportionately to the data store. We observe that the 6 MB Maya configuration shows the best performance compared to its counterpart baseline configuration. Performance decreases marginally as we increase the LLC size beyond 24 MB as a large fraction of the working set starts getting LLC hits.

Sensitivity to number of cores. Compared to an 8-core system, with 16 and 32-core systems, we observe marginal performance improvements over their respective baselines. We observe that the performance degradation for 32 cores compared to 16 cores is smaller than that of 16 cores compared to 8 cores, which signifies that the performance loss saturates as the number of cores increases. This shows that the Maya cache design can be extended to many-core systems.

3.5 Storage, Area, and Power overheads

Storage. A self-contained Table 3.8 shows the storage requirements of Maya, Mirage, and the baseline.

Power consumption and energy. To estimate the static power and the dynamic access energy, along with the area required, we use the 7nm FinFET technology simulated using P-CACTI [64]. Table 3.9 summarizes the observed dynamic energy and static power results for all three cache designs. We observe a reduction of 15.55% in dynamic read energy and 11.40% in dynamic write energy for the Maya cache compared to the baseline. On the other hand, Mirage shows a 3.81% increase in dynamic read energy and 4.52% increase in dynamic write energy compared to the baseline. We observe that the dynamic read/write energy is largely dominated by the energy required by the data store. Since Maya uses a smaller data store, we observe savings in dynamic energy for both reads and writes. Regarding the static power, Maya incurs 5.46% less power compared to the baseline. In contrast, Mirage incurs a power overhead of 18.16%, owing to the larger tag store and same-sized data store compared to the baseline.

Area. The data store largely takes up the area of the LLC. Because of this, the small data store design of the Maya cache can show savings of over 28.11% compared to the baseline, as seen in Table 3.9, whereas Mirage suffers a 6.86% area overhead due to the larger tag store. Note that Maya with ISO area budget consumes more static power as the ISO area implementation incurs a slight increase in area.

Table 3.8: Storage overheads.

Configurations		**Baseline**	**Mirage**	**Maya**
Tag Entry	Tag Bits	26	40	40
	Coherence	3	3	3
	Priority	-	-	1
	FPTR	-	18	18
	SDID	-	8	8
	Total bits	29	69	70
Tag Entries		262144	458752	491520
Tag Store Size		928 KB	3864 KB	4200 KB
Data Entry	Data Bits	512	512	512
	RPTR	-	19	19
	Total Bits	512	531	531
Data Entries		262144	262144	196608
Data Store Size		16384 KB	16992 KB	12744 KB
Total Storage		17312 KB	20856 KB (+20%)	16994 KB (-2%)

Table 3.9: Energy, power, and area overheads. Maya ISO area is Maya with a similar area (16.085 mm^2) as Mirage.

Design	Read Energy / Access (nJ)	Write Energy / Access (nJ)	Static Power (mW)	Area (mm^2)
Baseline	3.153	4.652	622	14.868
Mirage	3.274	4.857	735	15.887
Maya	2.661	4.116	588	10.686
Maya ISO	3.276	4.862	760	16.085

Table 3.10: Storage and performance overheads. Performance is evaluated on SPEC CPU2017 homogeneous mixes.

Cache Design	**Security (Installs per SAE)**	**Storage**	**Performance**
Maya	10^{32} (10^{16} yrs)	-2%	$+0.20\%$
Mirage	10^{34} (10^{17} yrs)	$+20\%$	-0.55%
Mirage-Lite*	10^{21} ($22,000$ yrs)	$+17\%$	-0.55%
Maya ISO	10^{30} (10^{14} yrs)	$+26\%$	$+1.84\%$

Chapter 4

Avatar Cache Design

4.1 Motivation

Mirage and Maya [20, 26], two state-of-the-art randomized LLC designs, mimic a fully associative LLC by decoupling insertion and eviction, making it immune to conflict-based side-channel attacks. These designs do not need OS support and incur marginal performance overheads. However, a new class of LLC attack, called occupancy-based attack [13], exploits cache occupancy to leak information and most randomized LLC designs, including Mirage and Maya, fail to address this threat [21], leaving LLC partitioning as the only effective mitigation. Secure LLC partitioning techniques [14, 22, 23, 24, 25] mitigate occupancy-based attacks but incur a high performance overhead, as high as 49% (for `cam4` on 8-core 16MB LLC with BCE [24]), and an average performance overhead of at least 5%. In addition, some partitioning techniques need OS support for fine-grained LLC partitions [22, 24]. A way-based partitioning technique needs minimal support from the OS. However, a 16-way LLC limits its scalability.

Pertinent question. Why incur such high overheads and design complexity for applications that do not require security? Moreover, the high overheads and design complexity make secure LLC designs impractical for industry adoption as they diverge significantly from conventional set-associative LLCs.

What we need? First, from a commercial deployment perspective, the industry demands a simple LLC design that is not significantly different from a traditional non-secure set-associative LLC in terms of design overhead and runtime overhead while providing complete immunity against LLC attacks. Recent proposals fail to meet this practical industry requirement. Second, from the user's perspective, a secure LLC is not required for all applications; thus, incurring significant performance and power overheads for non-sensitive workloads is inefficient.

Our goal is to propose a secure LLC design that offers *security-on-demand*, i.e., one can choose security against LLC attacks only if required. If one chooses to enable security by operating in *secure* mode, it should come with low design complexity and runtime overhead, making industry adoption easier. At the same time, if security is not required, the same LLC design should seamlessly operate in *non-secure* mode, morphing into a traditional set-associative LLC.

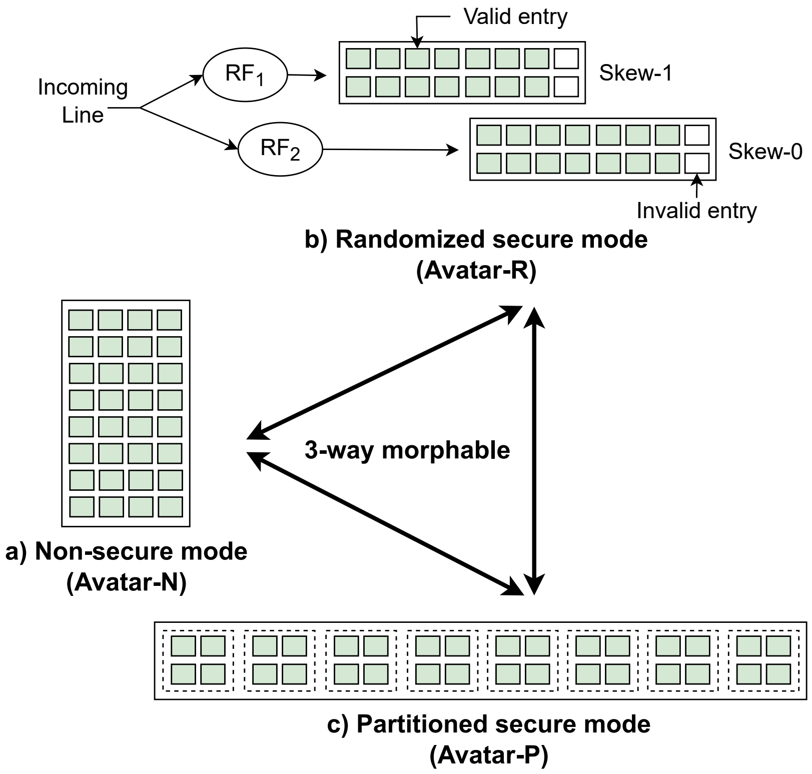

Valid entry
Incoming Line
RF1
RF2
Skew-1
Skew-0
Invalid entry
b) Randomized secure mode (Avatar-R)
3-way morphable
a) Non-secure mode (Avatar-N)
c) Partitioned secure mode (Avatar-P)

Figure 4.1: Overview of the Avatar LLC design with all operating modes. RF is the randomizing function implemented using a block cipher.

4.2 Design and implementation

The Avatar cache can operate in one of three modes: *non-secure* (Avatar-N), *randomized secure* (Avatar-R), and *partitioned secure* (Avatar-P). In Avatar-N, the cache operates similarly to a traditional non-secure set-associative LLC. To protect the system against conflict-based attacks, Avatar can switch to Avatar-R, where the cache operates as a skewed randomized cache with high associativity and global random eviction as the replacement policy. Additionally, the Avatar cache can also be used as a partitioned cache with high associativity to provide much stronger security. In this Avatar-P mode, Avatar utilizes the existing high-associativity hardware and bypasses the cipher, as there is no need for randomization; instead, it provides a statically partitioned LLC. Figure 4.2 shows how the LLC organization changes with each Avatar mode.

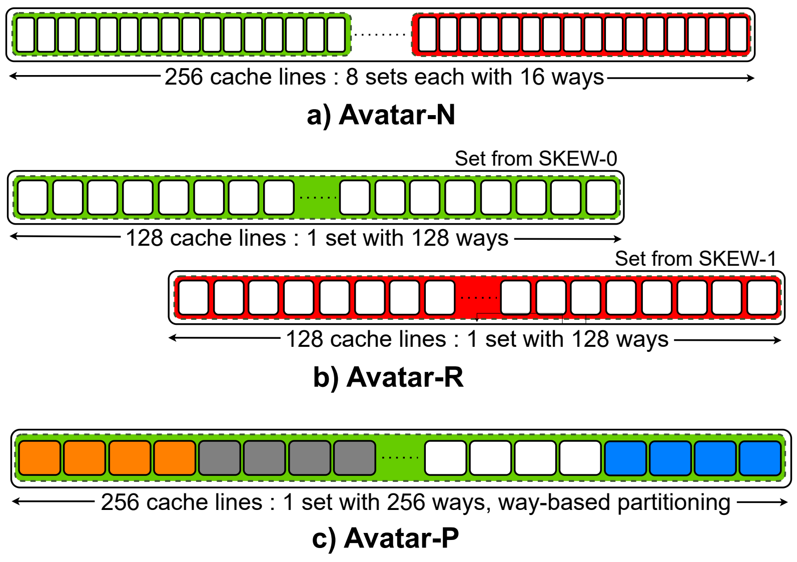

256 cache lines : 8 sets each with 16 ways
a) Avatar-N
Set from SKEW-0
128 cache lines : 1 set with 128 ways
Set from SKEW-1
128 cache lines : 1 set with 128 ways
b) Avatar-R
256 cache lines : 1 set with 256 ways, way-based partitioning
c) Avatar-P

Figure 4.2: The morphing of sets and ways between the different modes in Avatar.

4.2.1 Modes of Operation

Avatar-N

Avatar-N is intended for scenarios where security is not required and operates like a conventional set-associative LLC without skews. It employs smart replacement policies like Hawkeye [4] to optimize performance and uses lower associativity than the secure modes to reduce power consumption.

Avatar-R

Avatar-R mitigates set-associative evictions by using a Mirage-like design and high associativity to provide security with minimal runtime overheads.

Mirage with implicit mapping. Avatar-R adopts a skewed associative randomized LLC architecture inspired by Mirage [20], utilizing ciphers for randomized address-to-set mapping and a load-aware skew-selection policy [20] to evenly distribute cache lines across skews. Unlike Mirage, which uses explicit indirection via pointers, Avatar-R employs implicit indirection between tag and data entries, significantly reducing overhead and enabling a lightweight, secure design. To preserve security, Avatar-R reserves a portion of the cache as invalid entries and operates at partial capacity. It also employs a *global random eviction* policy, where a candidate is randomly selected from all valid LLC entries to prevent information leakage. A global counter tracks the number of valid LLC entries. Once it reaches a predefined threshold—corresponding to the maximum number of valid entries allowed to maintain the security guarantee—each subsequent fill triggers a global random eviction, where a random entry is evicted. If the counter is below the threshold, new lines are inserted directly without eviction.

High associativity. For Avatar-R, invalidating useful entries in a low-associativity cache can result in a significant performance loss. For example, in a 16-way associative cache (eight ways per skew), invalidating six ways per skew (this is important for providing complete security in the system's lifetime) leaves only 25% of the cache capacity usable, drastically affecting performance. In contrast, a 256-way associative cache (128 ways per skew) would only require less than 5% of its cache lines to be invalidated, making it a more practical solution. Thus, to maintain usable cache capacity while providing invalid ways, the Avatar LLC operates at high associativity, minimizing performance degradation.

Avatar-P

Avatar-P mitigates both conflict- and occupancy-based side-channel attacks by employing way-based partitioning, wherein a fixed subset of ways is allocated to each security domain. It operates at high associativity, which not only reduces the performance overhead typically associated with LLC partitioning but also enables support for a significantly larger number of concurrent security domains compared to conventional partitioned LLCs. Like Avatar-N, it bypasses ciphers and employs a deterministic address-to-set mapping and leverages a smart replacement policy to enhance performance.

4.2.2 Morphable Implementation

Avatar-N

For a 16 MB LLC, Avatar-N utilizes 16K sets with 16 ways per set and does not employ skews, requiring only 26 bits of the 46-bit line address for tag bits. Additionally, three coherence bits for the MOESI protocol and three re-reference interval prediction (RRIP) bits for the Hawkeye replacement policy [4] are necessary. Figure 4.3 illustrates the organization of a tag entry in different opertating modes of Avatar. The Avatar design allows the same hardware to be adapted for both modes. For a 16 MB LLC, Avatar-P uses all 40 bits to store useful information, while Avatar-N only utilizes 32 bits, leaving 8 bits unused per tag entry.

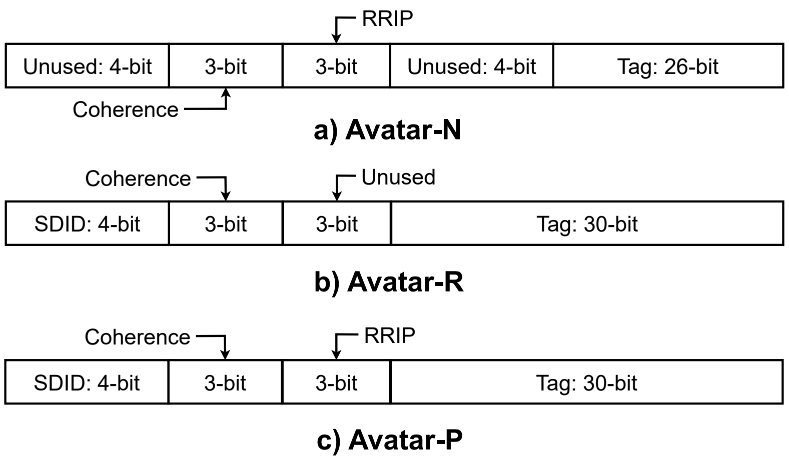


Figure 4.3: Tag bits in each of the possible modes of operation for Avatar. SDID: secure domain ID and RRIP is re-reference interval prediction [4] replacement policy priority bits.

Avatar-R

Avatar-R utilizes two skews and leaves some cache entries invalid to prevent set-associative evictions. Table 4.1 presents the per-access energy and performance overheads (based on SPEC CPU2017 workloads [5]) for an 8-core system using a 16 MB Avatar cache, with associativity varying from 128 ways (64 ways per skew) to 512 ways (256 ways per skew), while keeping the invalid ways constant at six ways per skew. As associativity increases, a larger percentage of the cache becomes usable; for instance, with 128 ways, only 90% of the cache is usable, increasing to 95% with 256 ways and 97% with 512 ways. Consequently, the 512-way configuration exhibits the least performance overhead. However, higher associativity also increases energy consumption, as more lines must be accessed for tag matching with each cache access. The 256-way configuration strikes a balance, offering performance nearly equivalent to the 512-way setup but with significantly lower energy requirements.

Table 4.1: Overheads for the energy required per LLC access and performance as the associativity of Avatar-R varies from 128 ways to 512 ways.

Associativity	**128-ways**	**256-ways**	**512-ways**
Access Energy	+3.5%	+10%	+25%
Performance	-0.9%	-0.6%	-0.5%

Table 4.2 quantifies the security guarantee of Avatar-R in terms of the number of cache lines needed to trigger a set-associative eviction (SAE) with varying invalid ways per skew (refer Section 4.3 for more details). With six invalid ways per skew, Avatar-R provides complete protection against eviction-based attacks throughout the system's lifetime, resulting in only a marginal performance overhead. For those seeking stronger security, Avatar-R can be configured with seven invalid ways per skew (we name it Avatar-R$^+$) to provide an even stronger security guarantee of no set-associative evictions in 10^{30} years, albeit at the cost of losing slightly more useful cache capacity. On average, Avatar-R uses 122 ways per skew and 6 invalid ways per skew for security. In a 16 MB Avatar LLC, this results in a total of 244K (244×1024) valid cache entries and 12K (12×1024) invalid cache entries, leading to a minimal capacity loss of $<5\%$.

For randomization, we use the Speck Cipher [65], which is a 32-bit, 22-round lightweight block cipher using 64-bit keys. We use its sister algorithm, Simon [65], for a hardware implementation of Avatar. The cipher, when turned ON, adds an access latency of three cycles for every LLC lookup. To determine the additional latency for accessing a 256-way skewed tag store in Avatar-R compared to a traditional set-associative cache, we used P-CACTI [64] with 7nm FinFET technology and CACTI 6.0 [66] with 22nm CMOS technology to model the highly associative design. Both tools show an increase of approximately 0.1ns

Table 4.2: LLC line installs per SAE as the invalid ways vary from 5 to 7 ways per skew for Avatar-R with 256 ways (128 ways per skew).

Invalid ways per skew	**Installs**
5 invalid ways per skew	$2{\cdot}10^{12}$ (40 mins)
6 invalid ways per skew	10^{23} ($3{\cdot}10^{6}$ yrs)
7 invalid ways per skew	$3{\cdot}10^{46}$ (10^{30} yrs)

in access latency, which remains within the 0.25ns cycle time. It is important to note that even with high associativity, the lookup latency remains essentially unchanged since the tag ways are accessed in parallel. However, this comes at the cost of higher energy consumption, as shown in Table 4.1. In total, Avatar-R requires four additional cycles per lookup due to the use of ciphers.

Avatar-P

To mitigate any kind of covert channel, including occupancy-based attacks, one can choose to operate in Avatar-P, with static way partitioning. This prevents the attacker from gaining any knowledge of a secure domain other than its own. In this mode, the cache skips the cipher thus eliminating extra 4 cycles of latency and deploying a smart replacement policy restricted to a domain. Avatar-P operates with high associativity with 256 ways, making it more scalable for servers to deploy. This mode can allow the system to partition the cache among a large number of domains (256 domains compared to the 16 domains with a conventional 16-way partitioned LLC), but this comes at the cost of reduced LLC space per domain.

Common details for both secure modes

In Avatar-R and Avatar-P, a 16 MB LLC employs 1024 sets and 256 ways (128 ways per skew, in case of Avatar-R) for an 8-core system, requiring 30 tag bits from the 46-bit line address. Each tag entry includes three additional MOESI coherence protocol coherence bits and three RRIP (unused in Avatar-R) bits. The secure domain ID (SDID) tracks the domain responsible for bringing in a cache line, enabling the duplication of shared cache lines to enhance security against shared-memory attacks. Avatar-R and Avatar-P use four

SDID bits to accommodate up to 16 domains, and the number of secure domains supported can be increased by adding more SDID bits to each tag entry. For example, increasing the number of domains to 256 (same as Mirage [20]) will result in an additional storage overhead of 0.8%. The necessity for SDIDs in Avatar-R and Avatar-P is discussed in Section 4.3.5. Note that the RRIP bits used for replacement are isolated per domain. We use 40 bits for each tag, resulting in a tag store size of 1.25 MB. The data store contains 256K (256×1024) entries, each storing 512 bits of data (64 B cache lines). Due to the one-to-one mapping with the tag store, there is no need for a reverse pointer. Thus, 512 bits are stored for each data store entry, yielding a total data store size of 16 MB.

4.2.3 On-demand Mode Switching

The mode switching in Avatar is performed by assigning two MSR (Model Specific Register) bits, with ring zero and exception level-1 privileges for x86 and ARM, respectively, so that only the OS, privileged user or security monitor [22, 67] can toggle them. The LLC can be operated in Avatar-N, Avatar-R or Avatar-P mode. The steps for the switching process are shown in Figure 4.4. Note that while the cache is switching modes, LLC accesses are not allowed; otherwise, it can lead to new channels due to mode switching. As the LLC is sliced, for each slice, the cache controller invalidates all LLC lines on a mode switch concurrently. Once the invalidation is complete, the cache controller updates the `stable` bit, allowing the user to access the LLC. During the mode switch, the `stable` bit is reset, which prevents any LLC accesses. The user, OS, or security monitor needs to *poll* the `stable` bit at a fixed interval based on the minimum time required for switching modes. A detailed circuit-level overview of the hardware modifications required to support Avatar's morphability is provided in Appendix I. It shows the specific LLC components activated in each operating mode (unused hardware can be power-gated to save power).

This runtime mode switching may be more practical in cloud environments. Other dynamic switching methods in Avatar are also viable, with implementation decisions left to the manufacturer based on specific requirements and constraints. In a static switching setup, where the user selects the mode at boot time via BIOS settings, end-user access to this register may also be provided.

Switching support for Trusted Execution Environments (TEEs). Switching to Avatar-R or Avatar-P can be done in conjunction with Intel TDX [68] or ARM Trust-

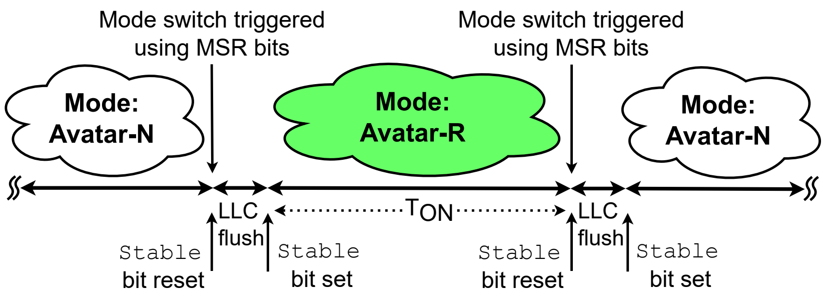


Figure 4.4: An overview of the steps involved in switching modes in Avatar. Note that this is just one possible example and the system can switch from any mode to any other mode.

Zone [69]. In this scenario, when an application enters a secure enclave, Avatar transitions to one of these secure modes and invalidates the LLC.

DoS attacks. We mandate a minimum operation time T_{ON} of one-second for any mode in Avatar. This is to prevent denial-of-service attacks in which the attacker continuously switches between modes and keeps the LLC occupied.

Switching overhead. The overhead for switching depends on how often we switch between modes in a fixed time interval while running our application of interest. We analyze the impact on the system's performance based on the operation time of a secure mode. During a switch, the LLC must be flushed so that no information about the secure execution is leaked to an adversary. Figure 4.4 shows a typical switching cycle where the system changes mode twice with an operation time of T_{ON}.

We quantify the switching overhead for a range of operation times T_{ON}, as shown in Figure 4.5. We analyze two approaches for performing LLC flushing during mode switching. In the LLC-STALL approach, all LLC accesses are halted during the transition. This causes the L1 and L2 caches to stall on misses, resulting in temporary performance degradation. Alternatively, the LLC-BYPASS approach allows continued LLC access during the switch. As flushing progresses, the LLC hit rate gradually declines, eventually reaching a 100% miss rate—effectively mimicking a system with no LLC. In this case, we approximate it to a system with a completely bypassed LLC to get an upper bound of degradation. Figure 4.5 illustrates the performance overhead of both approaches as the operation time (T_{ON}) varies from one-eighth of a second to eight seconds.

We suggest using the LLC-STALL approach, as it is much simpler to manage in hardware, and the differences in these methods vanish beyond T_{ON} of one second. The LLC flushing

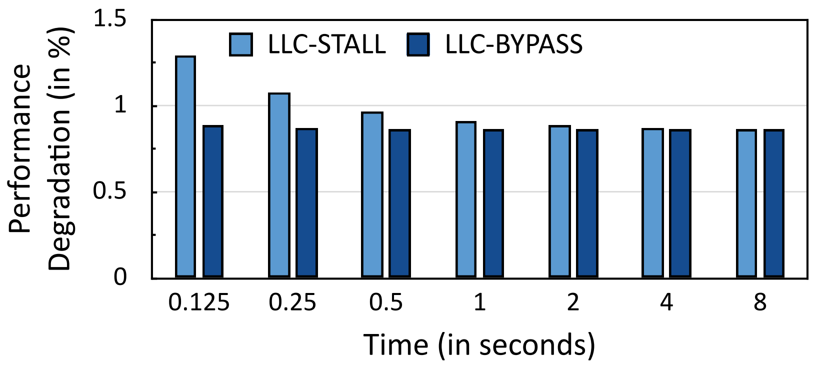


Figure 4.5: Performance degradation with varying T_{ON} between two consecutive mode switches (for SPEC CPU2017).

latency is primarily constrained by the number of read ports per slice and the DRAM write-back bandwidth. As shown in Figure 4.6, the average flush latency is approximately 1.25 million cycles—significantly smaller than the one-second (four billion cycle) cooldown period. Furthermore, as illustrated in Figure 4.7, the flush time is dictated by the proportion of lines that must be written to DRAM.

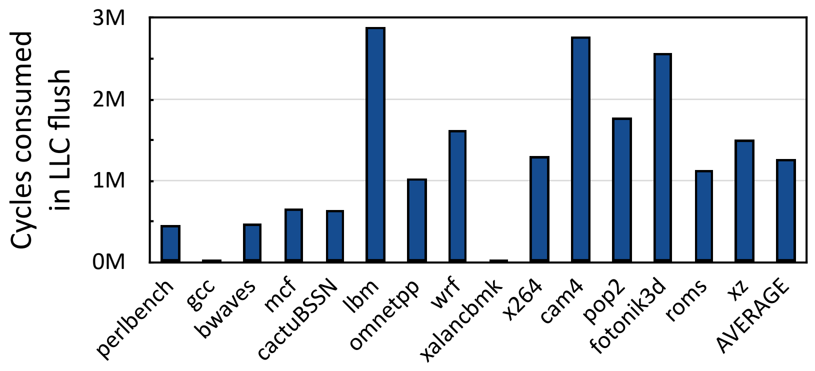


Figure 4.6: CPU cycles consumed in LLC flush during a mode switch. The average cycle count is around 1.25 million cycles.

4.2.4 Avatar and Private Caches

In modern processors, private L1 and L2 caches are often shared by 2-way simultaneous multi-threading (SMT) cores. Partitioning these caches between threads is a common and straightforward design choice, as shown in prior work [57]. The 4-cycle access latency of Avatar-R, combined with high associativity and global random replacement at L1, can intro-

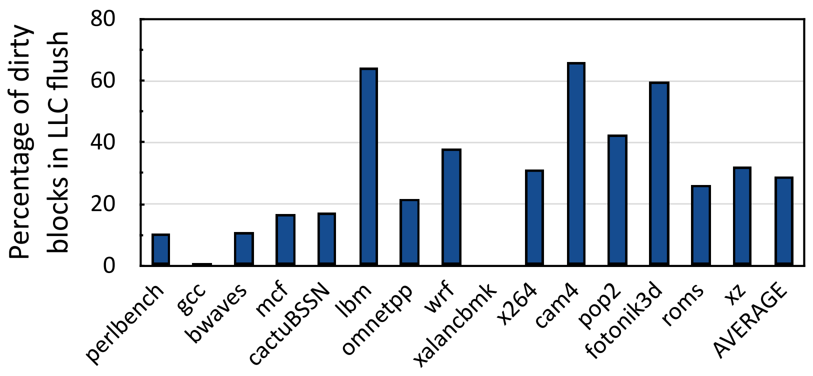


Figure 4.7: Percentage of blocks that cause writebacks during an LLC flush. On average, around 29% of blocks are dirty.

duce significant performance overhead. Thus, partitioning these private caches is advisable. Similarly, the cache coherence directory can also be partitioned [58].

4.3 Security Analysis of Avatar-R

To ensure security against conflict attacks, we demonstrate, with Avatar-R, that even a single SAE is exceedingly improbable over a system's lifetime, thereby ensuring security (Appendix II).

4.3.1 Requirements on Randomizing Function

The randomizing function used to determine the set mapping in Avatar-R is crucial for achieving a balanced distribution of invalid entries across sets, thus eliminating SAEs. For our analysis, we assume perfectly random set-mapping functions, ensuring uniform cache line distribution. However, recent shortcut attacks [70] have demonstrated the potential to exploit vulnerabilities in these algorithms to create deterministic collisions. To counter such attacks, we employ a cryptographic function computed in hardware with a secret key that remains unknown to attackers. Additionally, each skew utilizes an independent mapping function with a unique secure key, ensuring mutual independence of the mapping functions. We use the bucket and balls model to estimate the probability of an SAE. Avatar-R with 128 ways per skew has a frequency of one SAE in 10^{23} line installs or once in around $3 \cdot 10^{6}$ years, effectively providing complete security against conflict-based attacks. Please refer to

Appendix II for detailed calculations.

4.3.2 Sensitivity to Associativity

We now vary the associativity of Avatar-R, keeping the LLC size at 16MB. The associativity varies from 128 to 512 ways, with the default configuration having 256 total ways (128 ways per skew). Six ways per skew are kept invalid for all these Avatar-R configurations. Table 4.3 shows the rate of SAE for these configurations. We can observe that the 128-way configuration is the most secure (one SAE in 10^6 years), and security reduces as the LLC associativity increases. However, even for the 512-way configuration, the rate of SAE is once in 10^5 years.

Table 4.3: LLC line installs per SAE as the associativity of Avatar-R varies from 128 ways to 512 ways. Note that the LLC size is kept constant at 16 MB.

Associativity	**128-ways**	**256-ways**	**512-ways**
Installs	10^{25} (10^8 yrs)	10^{23} (10^6 yrs)	10^{22} (10^5 yrs)

4.3.3 Effect of No Decoupling on Security

Mirage uses a decoupled tag and data store design with a pointer-based indirection between the tag and data entries. Mirage's use of a decoupled tag-data design arises from additional invalid tag entries, which creates a mismatch between the number of tags and data entries, necessitating tag-data indirection. However, our security simulations demonstrate that removing this decoupling while keeping the extra invalid tags as unused entries has no impact on the security of Avatar-R or Mirage, assuming the tag and data stores contain an equal number of entries.

4.3.4 Key Management

The key used in Avatar-R is established during system boot. The keys are stored in hardware and are not visible to any software, including the OS. Avatar-R does not require a continuous refresh of the keys, as no information about the set mapping can be leaked without SAEs. SAEs are easily detected by monitoring invalid entries in each skew as part of the load-aware insertion policy. If both skews are full, an entry is randomly evicted from one skew, causing an SAE. Upon detecting an SAE, the cache can be flushed and re-keyed. Our analysis shows that the probability of an SAE in Avatar-R is extremely low, with re-keying likely only if an attacker brute-forces the key (one in 2^{64} probability).

4.3.5 Need for Secure Domain IDs

In situations where the attacker and victim do share LLC lines, various attacks like Flush+Reload [12], Flush+Flush [53], Flush+Prefetch [54], and Evict+Reload [55], could potentially leak victim data. Avatar-R includes a 4-bit SDID for each tag entry, allowing up to 16 domains. This design ensures that LLC fills from a secure domain don't impact those from another, thus providing security against shared-memory attacks. Avatar maintains the existing protocol for cache coherence but includes the SDID in all coherence packets. The SDID length can be adjusted to support different numbers of domains as needed. Avatar-R also mitigates the Reload+Refresh [56] attack as it guarantees global

4.4 Performance Evaluation

4.4.1 Methodology

We evaluate different LLC designs using the ChampSim [59] microarchitecture simulator. Building on the artifact provided by Maya [26], we modify it to implement the Avatar LLC. Our baseline is a non-secure 8-core system with a 16 MB, 16-way set-associative LLC, and 64-byte cache lines (Table 4.4). We test 15 homogeneous workloads (using 42 sim-points) from SPEC CPU2017 [2] and five homogeneous workloads (using 20 sim-points) from GAP [3], selecting benchmarks with more than one LLC miss per kilo instruction (MPKI) for the baseline configuration. Benchmarks are chosen based on their LLC MPKI for a single-core 2MB LLC. Additionally, we evaluate 21 heterogeneous mixes with randomly selected benchmarks from the SPEC CPU2017 and GAP suites. Simulations run 1.6B instructions across eight cores (200M per core) in the region of interest, following a 50M instruction warmup per core. To assess performance, we use the weighted speedup [62] metric for an 8-core system. First, we compare Avatar-R against the non-secure baseline (equivalent performance to Avatar-N). Then, we evaluate the performance gains of Avatar-P over conventional partitioned LLCs.

Table 4.4: Simulation parameters of the baseline system.

Core	8 cores, Out-of-order, bimodal [60], 4 GHz with 6-issue width, 4-retire width, 512-entry ROB
TLBs	L1 ITLB/DTLB: 64 entries, 4-way, 1 cycle, STLB: 2048 entries, 16-way, 8 cycles
L1I	32 KB, 8-way, 1 cycle, LRU
L1D	48 KB, 12-way, 5 cycles, LRU, IPCP prefetcher [61]
L2	512 KB 8-way, 10 cycles, LRU, non-inclusive, IPCP pref.
LLC	2 MB/core, 16-way, 24 cycles, Hawkeye [4], non-inclusive
MSHRs	8/16/32 at L1I/L1D/L2, 64/core at the LLC
DRAM controller	DDR4-3200, two channels/8-cores 4 KB row-buffer per bank, open page, burst length 16, $t_{RP, RCD, CAS}$: 12.5 ns

4.4.2 Avatar-R Performance

Homogeneous mixes. Figure 4.8 illustrates the performance of the Avatar-R normalized to the non-secure baseline for various homogeneous SPEC CPU2017 and GAP workloads. On average, for the SPEC CPU2017 benchmarks, the Avatar-R experiences a marginal performance loss of 0.6% compared to the non-secure baseline. However, Avatar-R incurs large performance slowdowns for specific benchmarks, such as `fotonik3d`. `fotonik3d` is a prefetcher-sensitive application, which means the prefetcher significantly aids in reducing MPKI for `fotonik3d`, but the global random replacement policy of Avatar-R degrades the prefetcher's performance.

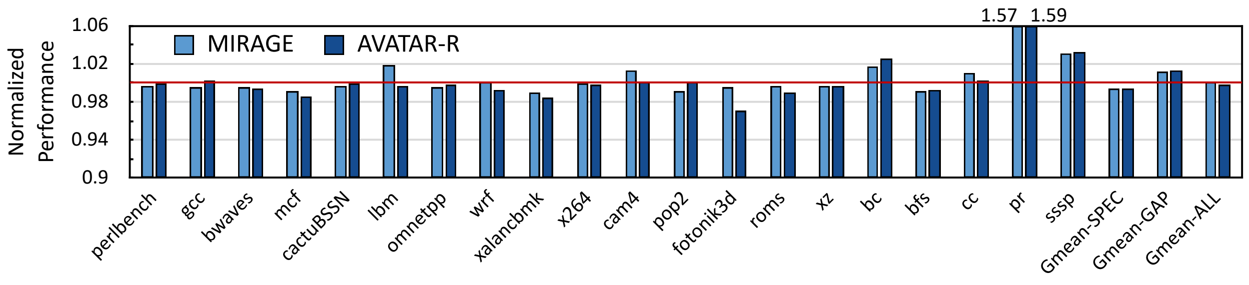


Figure 4.8: Performance of Avatar-R for 8-core homogeneous mixes (from SPEC CPU2017 [5] and GAP [6] suites), normalized to the non-secure baseline. The geomeans exclude `pr`. Both Avatar-R and Mirage have a marginal performance overhead of 0.2%.

For the `pr` workload, Avatar-R and Mirage outperform the non-secure baseline by 59% and 57%, respectively. This is due to a weak baseline, where the IPCP prefetcher negatively affects Hawkeye's performance, making it worse than using no prefetching with the LRU policy. For the remaining benchmarks, Avatar-R performs quite similarly to Mirage. Since `pr` is an outlier, we exclude it from geomean performance calculations. Without `pr`, Avatar-R performs 1.3% better than the non-secure baseline for GAP workloads. The geomean across SPEC CPU2017 and GAP workloads (excluding `pr`) shows that Avatar-R incurs a minimal 0.2% overhead. Avatar-R$^+$, which uses seven invalid ways per skew (reducing usable capacity to 94%), experiences a slightly higher 0.25% performance overhead compared to the non-secure baseline.

Heterogeneous mixes. For heterogeneous workloads, Avatar-R incurs a 1% average performance degradation compared to the non-secure baseline. This is due to the 4-cycle additional access latency and the global random replacement policy, same as Mirage. However, it performs slightly worse than Mirage due to its smaller usable data store (95% of total capacity).

Sensitivity to cipher latency. For Avatar-R, we use a 32-bit, 22-round lightweight Simon cipher [65] with 64-bit keys and a 3-cycle latency. We evaluate Avatar-R's performance for SPEC CPU2017 workloads across cipher latencies ranging from one to five cycles. We observe that with a 1-cycle cipher the slowdown for Avatar-R is 0.4%. In contrast, a 5-cycle cipher increases the slowdown to 0.8%.

Impact of global random eviction on performance. Global random eviction is key to ensuring strong security in Avatar-R but can affect performance. However, state-of-the-art replacement policies like Hawkeye are not effective for GAP workloads [71], and the usage of a global random eviction policy does not affect LLC performance. It is important to note that this does not undermine the effectiveness of smart replacement policies. Given that evictions are performed globally across the entire cache, the expanded pool of candidates diminishes the likelihood of selecting the worst candidate for eviction, in contrast to a random replacement policy limited to a set where the likelihood of selecting the worst possible candidate is much higher. Avatar-R uses skews that reduce conflict misses and improve cache performance [50], amortizing the negative effect of the global random eviction policy.

4.4.3 Avatar-P Performance

Traditional partitioning techniques face high-performance overheads and scalability challenges. We show that Avatar-P significantly reduces these overheads, making it a practical defense against both occupancy-based and eviction-based attacks.

Homogeneous mixes. Figure 4.9 demonstrates that Avatar-P achieves notable performance improvements over conventional 16-way partitioned LLCs when used with way-based partitioning. We note that memory-intensive workloads such as `mcf` and `cam4`, and graph workloads such as `cc` and `pr` benefit greatly from higher available LLC ways/set, reducing conflict misses. On average, Avatar-P has a 3% performance overhead compared to the non-secure baseline.

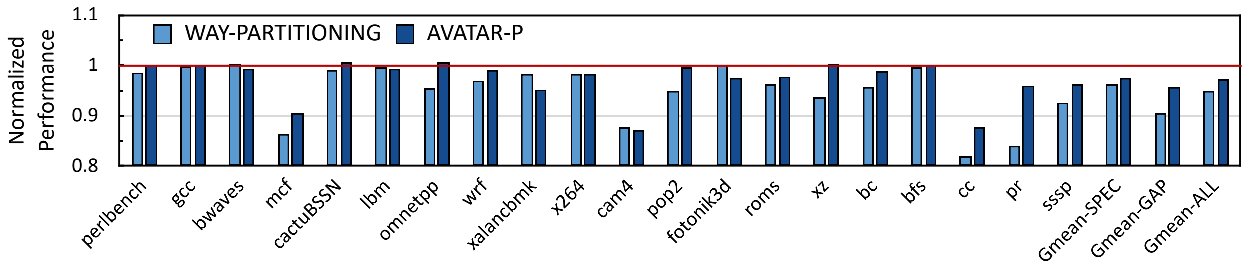


Figure 4.9: Performance of Avatar-P (256-way static way-based partitioning) for 8-core homogeneous mixes (from SPEC CPU2017 [5] and GAP [6] suites), normalized to the non-secure baseline. Avatar-P outperforms the 16-way 16 MB static way-based partitioning, with an overhead of less than 3% compared to the non-secure baseline.

Heterogeneous mixes. We also evaluate Avatar-R on heterogeneous mixes comprising SPEC CPU2017 and GAP workloads. On average, Avatar-R incurs a 4% performance degradation compared to the non-secure baseline. This overhead is primarily due to the static partitioning used in Avatar-P, which can lead to increased conflict misses, particularly for memory-intensive workloads.

4.4.4 Sensitivity to LLC Size

We evaluated Avatar-R and Avatar-P using SPEC CPU2017 workloads across varying LLC sizes, ranging from 8 MB to 128 MB (corresponding to 1-16 MB per core), extending beyond the default 16 MB (2 MB per core) Avatar LLC configuration. Figure 4.10 shows the resulting performance trends. At 8 MB, Avatar-R slightly outperforms its non-secure baseline by approximately 0.3%. As LLC size increases beyond 32 MB, the performance advantage diminishes, stabilizing at about a 1% overhead due to the higher LLC hit rate and reduced opportunity for further gains. This demonstrates that Avatar-R remains practical even for systems with large LLCs. Avatar-P consistently outperforms a conventional way-partitioned LLC across all cache sizes, with the performance gap increasing at larger sizes. Notably, Avatar-P begins to surpass Avatar-R at 64 MB (8 MB per core). This suggests that Avatar-P may be preferable for securing workloads on systems with very large LLC/core, which is typically not the case with commercial processors as 2-4 MB/core is the current trend.

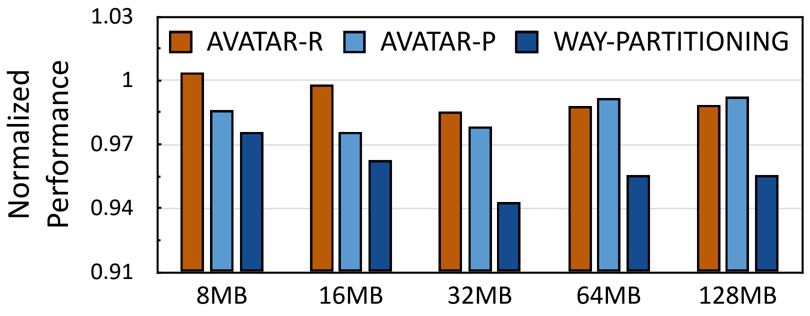


Figure 4.10: Performance of Avatar-R, Avatar-P, and 16-way 16 MB 8-core static way-based partitioned LLC with LLC size varying from 8 MB to 128 MB.

4.5 Storage and Power Overheads

Storage. As detailed in Section 4.2.2, Avatar requires 40 tag bits for its morphable implementation, compared to 32 tag bits in the non-secure baseline. This increases Avatar's total tag store size to 1280 KB, up from 1024 KB in the baseline. Avatar and the baseline have a data store of 512 bits per entry, resulting in a total storage requirement of 16,384 KB. Consequently, Avatar's overall storage requirement is 17,664 KB, while the baseline requires 17,408 KB, leading to a 1.5% storage overhead for Avatar.

Power consumption and energy. We use 7nm FinFET technology, simulated using P-CACTI [64] in sequential access mode, to estimate the static power and the dynamic access energy. Table 4.5 summarizes the observed dynamic energy and static power results for an 8-core 16 MB Avatar LLC design in Avatar-N and Avatar-R/Avatar-P modes and compares these results to those of Mirage and the non-secure baseline. Avatar's static power marginally increases by 2.7% compared to the non-secure baseline. This increase is due to the additional eight tag bits Avatar uses for each tag entry. In contrast, Mirage incurs a static power overhead of 18.16%, owing to the larger tag store and same-sized data store compared to the non-secure baseline.

Table 4.5: Energy and power overheads for the non-secure baseline, Mirage, Avatar-N, and Avatar-R/Avatar-P.

Design	Read Energy / Access (nJ)	Write Energy / Access (nJ)	Static Power (mW)
Baseline	0.171	0.192	520
Mirage	0.175	0.222	611
Avatar-N	0.171	0.192	534
Avatar-R / Avatar-P	0.190	0.275	534

We observe an increase of 11.11% in dynamic read energy and 43.23% in dynamic write energy for the Avatar LLC design in Avatar-R and Avatar-P modes as compared to Avatar-N and the non-secure baseline. The static power remains the same for all three operating modes because the same hardware is used. This significant increase in dynamic access energy is due to increased associativity (a more complex tag lookup). Figure 4.11 shows the read and write power overhead of Avatar-R/Avatar-P over the non-secure baseline across various homogeneous SPEC CPU2017 workloads. We observe that, on average, the read power overhead is around 0.60mW, and the write power overhead is around 0.68mW. These are negligible ($\approx$0.2%) compared to the static LLC power.

Logic. The Avatar LLC design requires two modules of the Simon block cipher [65]. Recent works [72, 73] have shown efficient hardware implementations of the Simon cipher and evaluated their hardware costs. For 7nm FinFET technology, the two modules of the Simon cipher have an estimated hardware cost of 6,160 Gate Equivalents (GEs, number of equivalent 2-input NAND gates) and an area requirement of around 61.6μm^2. Avatar also requires extra logic for highly associative cache lookups and load-aware skew selection in the randomized secure mode. The additional logic gates that facilitate the morphing ability of Avatar between its operating modes (Figure I.1) amount to around 100,000 GEs or 1,000 μm^2. The load-aware skew-selection circuit (counting 1s among valid bits of 128 tags from the indexed set in each skew, followed by an 8-bit comparison) requires around 1,660 GE

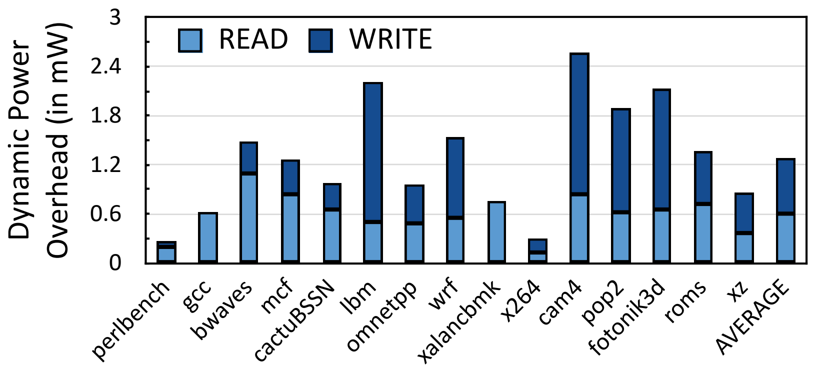


Figure 4.11: Dynamic read and write power overheads of Avatar-R/Avatar-P as compared to the non-secure baseline. The average dynamic power overhead is 1.28mW, which is only around 0.2% of the LLC static power.

(16.6 μm^2). All the extra logic required for the secure operation modes and morphability of Avatar can fit in less than 110,000 GE, which is negligible compared to the several hundred million gates required for a 16 MB LLC.

Chapter 5

Systematization of Secure Randomized Caches

5.1 Motivation

In recent years, several defense mechanisms have been proposed to counter these attacks, among which LLC randomization has emerged as a promising candidate for flush and conflict-based attacks, and recently for occupancy-based attacks [14]. There have been several proposals for a randomized LLC design. Some of the initial proposals such as RPCache [15], CEASER [16], CEASER-S [17], and Scatter-Cache [18] were quickly compromised by newer and faster attacks [17, 7, 19], and eventually improved designs were proposed. Recent secure randomized LLC designs, such as Mirage [20], introduce multiple microarchitectural modifications to a conventional set-associative cache to provide security. However, the security guarantee of each of these designs is usually evaluated as a whole, without clarifying the role of each microarchitectural modification. Secondly, many of the designs propose similar modifications or identical security implications. Therefore, it is crucial to identify the set of security "knobs" or modifications used by these designs. We identify, evaluate, and provide a taxonomy of five knobs, both in isolation and in combination. The knobs of interest are: (i) skewing, (ii) extra invalid tags, (iii) high associativity, (iv) replacement policy, and (v) remapping. We introduce high associativity as a knob in this study and argue its effectiveness in enhancing security. Even though cache associativity has previously been evaluated [27] in the context of security, no prior work has evaluated high associativity as a security knob.

We quantify security using the two metrics – i) *eviction rate*, defined as the probability that an *eviction set* evicts the target address; and, ii) number of evictions to generate an eviction set with real-world eviction set finding algorithms. We iteratively analyze the impact of each knob on the attacker's achievable eviction rate, noting the conditions under which each knob is effective and explaining why it works. For the most promising knob combinations, we use the second metric to make the evaluation stronger.

Through our evaluation, we answer questions such as (i) *why a security knob works and to what extent?* and (ii) *what is a minimal set of knobs to secure a conventional set-associative cache?* This modular systematization enhances the understanding of randomized caches, helping designers and security analysts select optimal designs. Several works have evaluated randomized caches from different perspectives. **CaSA** [74] developed a framework to quantitatively analyze the security of randomized caches against covert channel attacks. **Song et al.** [7] analyzed remapping periods for certain cache designs exploiting design flaws. **CacheFX** [27] proposed a generic framework for assessing cache resilience to side-channel attacks. Our focus differs: we study the extra microarchitectural features added for security, both individually and collectively.

Finally, occupancy-based attacks pose a significant threat to cache security [28], and even complex designs like Mirage [20] cannot fully mitigate them. Partitioning [25, 75] is an effective defense but incurs a substantial performance penalty. For randomized caches, only designs that use partitioning or soft partitioning-based solutions [42, 14] can defend against occupancy-based attacks. Moreover, the security of these designs depends entirely on their partitioning properties. This raises the question: *Can the knobs proposed by secure randomized cache designs mitigate occupancy-based attacks?*

5.2 Systematization of Randomized Caches

What are security knobs? We define microarchitectural design modifications that are specifically implemented to enhance security in secure randomized cache designs as *security knobs*. Ideally, the block cipher would be the first security knob to study, as it is a critical component in nearly every randomized cache design, providing the unpredictability essential for security. It is advisable to use a block cipher that is robust to *shortcut cryptanalytic attacks* [19, 76], and provides a uniform distribution of the encrypted set indices. Latency

Table 5.1: A comparison of secure randomized cache designs. A register stores the set permutation per word line in table-based indexing, unlike block cipher-based indexing, where the same hardware derives the set for all addresses. Reported performance and storage overheads are based on claims from the respective papers.

Design	Indexing Policy	Remapping?	Skews?	Other Security Knobs	Mitigates Conflict based Attacks?	Mitigates Occupancy based Attacks?	Mitigates Flush based Attacks?	Performance Overhead	Storage Overhead
RPCache [15]	Table-based	✘	✘	N/A	✘	✘	✘	0.15%	1%
NewCache [38]	Table-based	✔	✘	N/A	✔	✘	✔	< 1%	10%
CEASER [16]	Block Cipher-based	✔	✘	N/A	✘	✘	✘	1.1%	0%
CEASER-S [17]	Block Cipher-based	✔	✔	N/A	✘	✘	✔	0.7%	0%
ScatterCache [18]	Block Cipher-based	✔	✔	N/A	✘	✘	✔	2%	5%
PhantomCache [34]	Block Cipher-based	✘	✘	Localized Randomization	✘	✘	✘	1.2%	0.5%
Mirage [20]	Block Cipher-based	✘	✔	Decoupling, Load Aware, Global Eviction	✔	✘	✔	1.7%	20%
H^2Cache [35]	Block Cipher-based	✘	✔	N/A	✘	✘	✘	10.7%	0%
Chameleon [39]	Block Cipher-based	✔	✔	Fully Associative Victim Cache	✔	✘	✘	< 1%	< 0.1%
SassCache [14]	Block Cipher-based	✘	✔	Soft Partitioning	✔	✔	✔	N/A[1]	N/A[2]
ClepsydraCache [36]	Block Cipher-based	✘	✘	Time-based Evictions	✔	✘	✔	1.38%	< 8%
Song et al. [41]	Block Cipher-based	✔	✘	Attack Detection, Multi-step Reallocation	✘	✘	✘	0.89%	1.9%
RECAST [40]	Block Cipher-based	✘	✘	Address-Sensitive Secret Generation	✔	✘	✘	2.29%	1.1%
Maya [26]	Block Cipher-based	✘	✔	Decoupling, Load Aware, Global Eviction	✔	✘	✔	-0.2%	-2%
INTERFACE [42]	Block Cipher-based	✘	✔	Decoupling, Load Aware, Global Eviction, Partitioning	✔	✔	✔	3.4%	14%

of the state-of-the-art ciphers [77, 51] can be an issue here. Additionally, the correct implementation of the block cipher is also crucial to the security properties of a secure randomized cache, as highlighted in [78]. However, since the block cipher cannot be considered optional, we exclude it from further analysis. Instead, we focus on other microarchitectural modifications that serve as security knobs. Identifying security knobs is not always straightforward, as there are often dependencies between different knobs. Some of these dependencies are explicit, while others only become apparent through empirical analysis. Additionally, some knobs are merely attributes or components of another knob and are referred to as *sub-knobs*. A taxonomy of the knobs and sub-knobs employed in the randomized cache designs is presented in Figure 5.1.

[1]The authors have used cache hit rate to quantify the performance of their design instead of misses per kilo instructions. Thus, the impact of this design on performance remains unknown.

[2]No storage overhead analysis has been performed by the authors

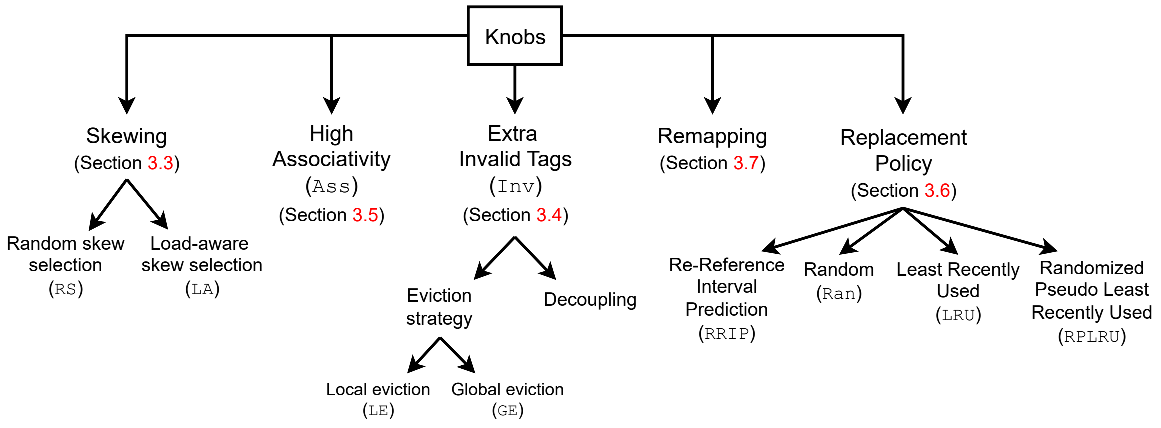


Figure 5.1: An overview of the knobs and sub-knobs identified in modern secure randomized cache designs, along with their abbreviations used throughout the paper.

5.2.1 Evaluation Strategy and Simulation Setup

Metrics

Evaluating the full taxonomy of security knobs is challenging, starting with the selection of appropriate evaluation metrics. Prior work, notably [27], advocates using multiple metrics, including: (i) Relative Eviction Entropy (REE), which quantifies information leakage from attacker-victim conflicts; (ii) the difficulty of constructing eviction sets using state-of-the-art algorithms; and (iii) the effectiveness of attacks on cryptographic targets. Our work also adopts a multi-metric approach. We first use a well-established metric—eviction rate—to identify cache designs that are promising from a security standpoint, then apply additional metrics for deeper analysis.

Metric-I: Eviction rate. The *eviction rate* is defined as the fraction of times a target address is evicted by an eviction set of a given size, measured over n iterations. In each iteration: (i) a target address x is randomly selected; (ii) its corresponding eviction set is identified; (iii) the target is accessed; (iv) the eviction set is accessed; and (v) it is checked whether the target has been evicted. Prior works [18, 7, 36] also refer to this metric as eviction probability in the context of randomized caches. For strong security, a design should exhibit a low eviction rate even with large eviction sets. This *eviction rate experiment* follows the methodology described in [7]. The rationale behind choosing the eviction rate is as follows:

- The experiment for measuring this metric closely resembles what happens in a real conflict-based attack. For example, in the Prime+Probe attack [1], the eviction rate represents the probability of detecting a secret-dependent victim access.

- Eviction rate is measured for a given eviction set size. If generating an eviction set of that size is already difficult using existing algorithms, it provides a clear, intuitive explanation for a design's security. Notably, the difficulty of eviction set generation is a recognized metric in several works [27], including ours.

Generating an eviction set for the eviction rate experiment is a challenging task due to the existence of various eviction set search algorithms. To avoid algorithmic bias, we assume an oracle provides the eviction set, independent of any specific search algorithm. These sets are identified by sampling a large collection of random addresses, determining their set indices using full knowledge of the block cipher key, and selecting addresses partially congruent to the target address. Note that selecting perfect eviction sets doesn't make our evaluation unrealistic; we later assess how real-world eviction set generation algorithms [7][19] can replicate such sets.

Metric-II: Number of evictions to generate an eviction set. After filtering secure and promising designs based on eviction rate, we evaluate them using state-of-the-art eviction set finding algorithms [7, 70]. The goal is to assess the "difficulty" of constructing an eviction set of a given size, measured by the number of LLC evictions required. This forms the second key metric in our evaluation and directly impacts the critical design knob—the *remapping period*.

We do not explicitly evaluate specific cryptographic targets for conflict-based attacks. While such targets could serve as an additional metric, we argue that the difficulty of evicting a single victim address—as measured by our eviction rate experiments—naturally generalizes to evicting multiple addresses. Prior work [70] demonstrated this by combining eviction sets for multiple distinct addresses to attack AES. Thus, our single-address eviction rate and eviction set construction results reflect the broader difficulty of targeting cryptographic implementations. In contrast, for occupancy-based attacks, we rely primarily on cryptographic benchmarks, as no alternative metrics are yet established in the literature.

Simulation setup. To analyze the impact of various design knobs on security against conflict-based attacks, we extend the open-source behavioral cache simulation model from [79] to incorporate all knobs outlined in Figure 5.1. Unless otherwise specified, all experiments use a 2 MB LLC with 16-way associativity. For eviction set experiment, we set the number of iterations $n = 1000$, based on empirical observations, as most experiments converge reliably with this value.

5.2.2 Knob 1: Skewing

Traditional cache designs have one set that a particular address can map to; however, under a `Skew-k` cache, an address could go to one of the `k` possible sets, each belonging to one skew. Skew is one of the most important security knobs used in various secure randomized cache designs starting from CEASER-S [17]. We note two possible sub-knobs based on the skew selection (choosing one of `k` possible sets) strategy for an incoming cache line – i) Random skew selection (`RS`); ii) Load-aware skew selection (`LA`).

Random Skew Selection (`RS`)

In random skew selection [17, 18, 35], a new cache line is inserted into a randomly chosen skew. As shown in Figure 5.2, the eviction rate decreases with the number of skews for a given eviction set size. However, for a fixed number of skews, the eviction rate increases with the eviction set size. The key observation is that there is a non-zero eviction rate even for a small eviction set size, suggesting that even a small eviction set can lead to information leakage.

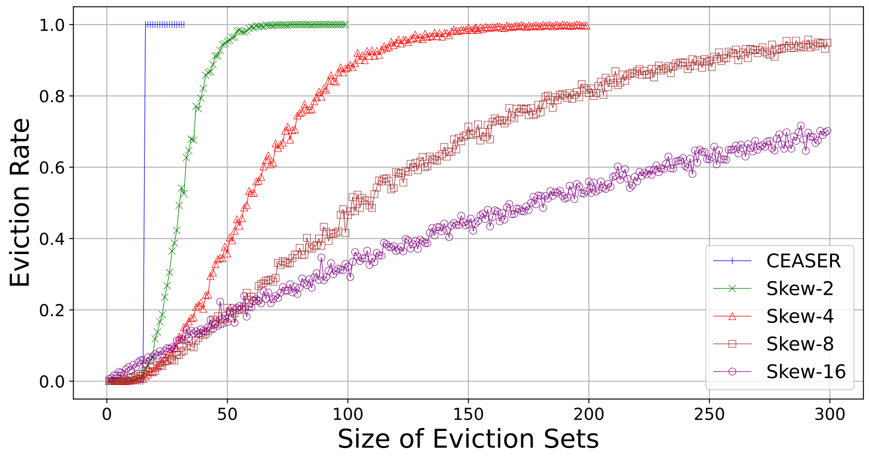


Figure 5.2: Eviction rate for a varying number of skews (`Skew-k` has `k` skews) with random skew selection and LRU eviction. CEASER does not use any skews and `Skew-2` is a representative for skewed designs such as CEASER-S.

Explanation. To explain the trend in Figure 5.2, we refer to [19], which provides the expression for the *eviction probability* (p_e) of an eviction set E of size $|E|$. For a random replacement policy, $p_e = 1 - (1 - \frac{1}{n_w})^{\frac{|E|}{k}}$, where n_w is the total number of ways across skews and k is the number of skews. For LRU, $p_e = 1 - \text{binom}(\frac{|E|}{k}, \frac{n_w}{k} - 1, \frac{1}{k})$, where binom is the cumulative binomial distribution function. In both cases, p_e decreases as k increases, assuming a fixed $|E|$. The eviction rate in our plots reflects this probability, excluding cache warm-up effects. Notably, even low eviction rates can lead to successful attacks, as shown in prior work [7, 19].

Load-aware Skew Selection (`LA`)

The load-aware skew selection policy compares the remaining capacity of the k-sets that an address can map into, and inserts the address into the set having the largest remaining capacity. In case the remaining capacity is the same for all the skews, the tie is broken with a random skew selection. As shown in Figure 5.3, load-aware provides slightly better security than random skew selection. However, the gain gradually diminishes with an increase in the skew count.

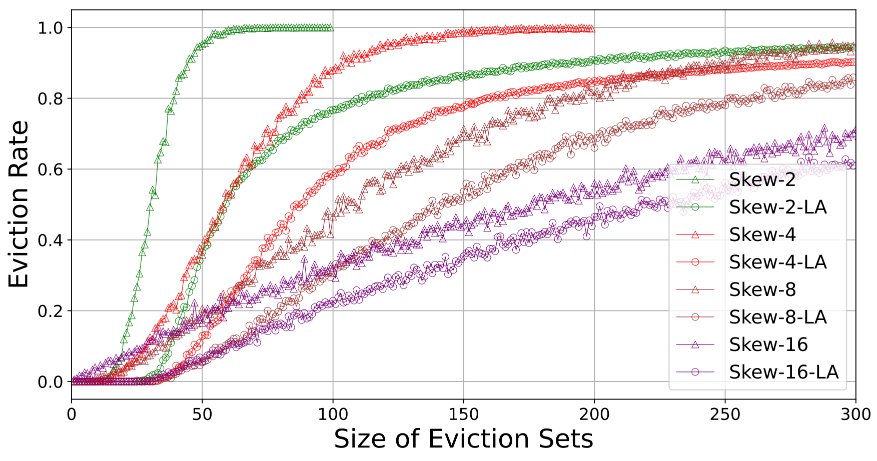


Figure 5.3: Eviction rate for varying number of skews (`Skew-k` has `k` skews) with random vs load-aware skew selection (`LA`) with LRU eviction.

Explanation. The key difference between random and load-aware skew selection is that the latter depends on the cache state. Let us consider an address e mapping to sets $\{s_1, s_2, \cdots s_k\}$ in skews $\{1, 2, \cdots k\}$. Without loss of generality, we consider the probability that e gets mapped to s_1, which depends on the occupancy of the other skews, which in turn reflects the overall cache state. As a result, e can only evict a target address x when the cache is nearly full—eviction occurs only if all skews are close to capacity. Since eviction set creation algorithms ignore cache state and select e based solely on observed evictions during construction, these sets tend to be ineffective unless the cache is at least $\approx$90% full. However, the eviction rate experiments shown in Figure 5.3 indicate a slight improvement with load-aware skew selection compared to random skew selection. This counter-intuitive trend, given that higher eviction rates are generally expected as caches fill quickly, was investigated and found to be an artifact of the cache's warm-up state and the method used to compute the eviction rate. A detailed explanation of this behavior is provided in Appendix IV. In summary, a minor variation in the eviction rate calculation, which accounts for the size of the warm-up state, reconciles this observation with our overall understanding. We recommend using this alternative calculation, even though it requires significantly more simulation effort, only for a subset of designs with promising security characteristics. Overall, we conclude that load-aware skew selection, even when combined with multiple skews, offers no clear security benefit over random skew selection.

Takeaway

Skewing is useful as it reduces the eviction rate of an eviction set of a particular size. The skew selection policy only has a limited impact on security–load-aware provides a security benefit only when the cache is not full.

5.2.3 Knob 2: Extra Invalid Tags (`Inv`)

Mirage [20] uses extra invalid tags to ensure that an incoming cache line can find space in its mapped set and not cause an SAE. To maintain such invalid tags in the cache, we keep a threshold on the total number of valid entries that can be kept in the cache and trigger the eviction mechanism once this threshold is exceeded. Seeing invalid tags as a knob is motivated by the fact that it involves multiple choices (as sub knobs) to be made regarding cache structure and cache management. The first possible sub-knob is whether to decouple the tag and data store. Another important sub-knob is the eviction policy. Eviction policy is tightly coupled with invalid tags, as the sole reason behind declaring a tag invalid is to control when the evictions are supposed to be made. Therefore, we consider it as a sub-knob for invalid tags.

Decoupled tag store has no security impact

There are two ways in which extra invalid tags can be provisioned in the cache. The first is to use a decoupled tag and data store with extra invalid tag entries in the tag store, which has been used in Mirage. The data store size remains the same as a traditional non-secure cache and the tag store is expanded to store the extra invalid tag ways. An alternate approach to remove the decoupling and instead operate the cache at a lower capacity. In other words, whenever the total valid entries inside the cache exceed a threshold number, we trigger the eviction mechanism so that the cache never fills up. This implies having less number of valid entries in the cache compared to a non-secure cache of the same size.

While these choices indeed qualify as sub-knobs, one observation is that functionality-wise both are the same. We observe this throughout our experiments as the results with and without decoupling are the same. Without loss of generality, we go with no decoupling. Also, we would like to point out that decoupling the tag and data store and adding extra invalid tags leads to a hefty overhead regarding storage, area, and power requirements that Maya [26] takes care of.

Extra invalid tags is not a standalone knob

One crucial observation regarding invalid tags is that they are not useful without skews. A non-skewed cache with invalid tags is nothing but CEASER operating with a lower capacity. This is depicted in Figure 5.4, where we depict the combination of invalid tags both with and without skews. As expected, skewing has a positive impact on security. Therefore, the rest of our analyses in this subsection will assume a minimum of two skews. However, deciding the proper skew selection policy is complex, as we discuss in the following paragraphs.

Local Eviction (`LE`) vs. Global Eviction (`GE`)

Interplay between skew selection and eviction policy: Given that skews are essential, the next question is which skew-selection policy fits well with extra invalid tags. Our prior observation is that both policies perform almost equally with skews. However, in the context of extra invalid tags, we observe a complex interplay between the eviction policies and the skew-selection policies. We explain this by considering the eviction and skew-selection policies in pairs. The eviction policy in the presence of extra invalid tags can be *local* (`LE`), that is, from the same set as the one in which an insertion occurred; or *global* (`GE`), that is, a

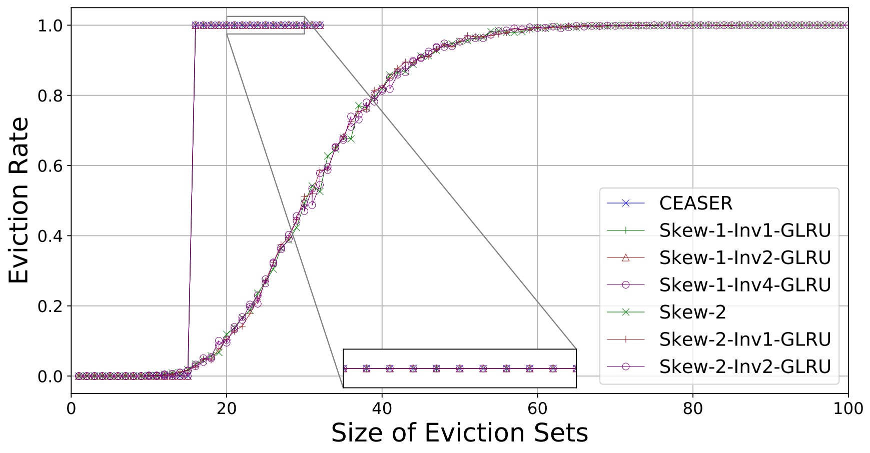


Figure 5.4: Eviction rate for CEASER and two skews (`Skew-2`) with random skew selection, extra invalid tags (`Inv`), and global LRU eviction (`GLRU`).

valid entry chosen randomly from the entire cache is evicted upon each insertion. Therefore, there are four combinations to evaluate: i) `GE` + `RS`; ii) `GE` + `LA`; iii) `LE` + `RS`; iv) `LE` + `LA`.

Without loss of generality, we begin with the global LRU eviction and random skew selection (`GE`+`RS`), which is also illustrated in Figure 5.4. *The observation here is that the combination of skews, invalid tags, global eviction, and random skew selection does not have any advantage over standalone skewing with random skew selection.* Next, we modify the skew selection policy to load-aware skew selection (`GE`+`LA`). As depicted in Figure 5.5, invalid tags now become effective, and the eviction rate steadily decreases with an increase in invalid tags. *This establishes invalid tags with global LRU eviction and load-aware skew selection as a useful knob-subknob combination.* The same combination used by Mirage.

Next, we consider local eviction policies. The results with load-aware skew selection (`LE`+`LA`) are depicted in Figure 5.6. As observed, *there is no impact of extra invalid tags, and the results are the same as skewing with load-aware.* We note that the results do not change even when local eviction is combined with random skew selection (`LE`+`RS`). Therefore, we have omitted the plot for this combination.

So far in this evaluation, we have only evaluated the locality of the eviction policies. However, the replacement policy, e.g., LRU or Random, should also be evaluated. We note that one of the most successful design with extra invalid tags, namely Mirage, uses global random replacement. We defer this discussion till Section 5.2.5. We conclude the current subsection explaining why load-aware skew selection becomes effective in the presence of

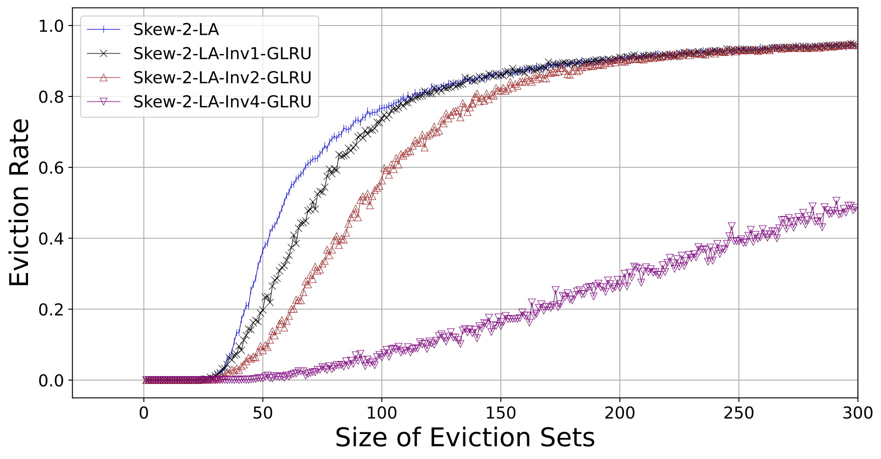


Figure 5.5: Eviction rate for two skews (`Skew-2`) with load-aware skew selection (`LA`), extra invalid tags (`Inv`), and global LRU eviction (`GLRU`).

invalid tags and global eviction.

Why is `LA` important? The positive impact of load-aware skew selection warrants further clarification, as it initially appeared ineffective as a sub-knob of skews (see Section 5.2.2). However, this new observation does not contradict the earlier finding. Load-aware skew selection is effective when the cache is not full, and keeping the cache from becoming full improves both the eviction rate and overall security. This is achieved through the combination of extra invalid tags and global eviction: the cache (including invalid tags) never becomes full, since consuming an invalid tag creates another. However, this mechanism alone is insufficient. An eviction set can still contain enough addresses mapping to a single set in a skew, depleting all its invalid tags and triggering set-associative evictions, compromising security. In such cases, random skew selection becomes ineffective. Load-aware skew selection mitigates this by lowering the likelihood of depleting all invalid tags in a skew, as it tends to distribute insertions across skews more evenly. It effectively requires most sets in other skews to be nearly full, which is unlikely under global eviction. Thus, load-aware skew selection and global eviction work in tandem to keep the probability of any skew becoming full very low, preserving low eviction rates. However, this dynamic is sensitive to the number of extra invalid tags per skew, which influences the global eviction threshold.

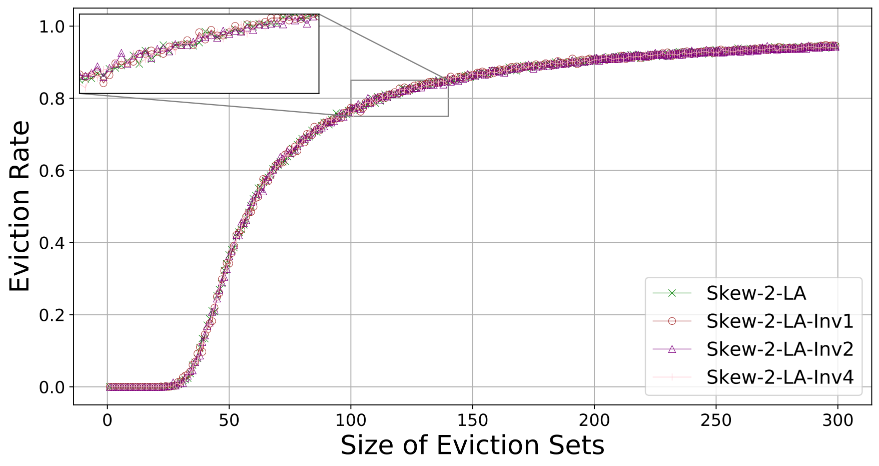


Figure 5.6: Eviction rate for two skews (`Skew-2`) with load-aware skew selection (`LA`), extra invalid tags (`Inv`), and local LRU eviction.

Takeaway

Extra invalid tags can enhance resilience against conflict-based attacks. But it has to be paired with skews, global eviction, and load-aware skew selection. No other knob-subknob combinations work. Decoupling the tag and data store has no measurable impact on security.

5.2.4 Knob 3: High Associativity (`Ass`)

Prior work [27] explored the effect of cache associativity as a security knob. We have also observed this effect in Section 5.2.2, where the eviction probability (p_e) is inversely proportional to the cache associativity. However, [27] limited their evaluation to up to 16 ways. In this section, we extend this analysis to high-associativity designs and demonstrate that they offer significantly stronger security guarantees, making high associativity a powerful design knob against conflict-based attacks attacks.

We begin with the simplest possible configuration of knobs. Once again we note that associativity without skews is nothing but a traditional set-associative cache. As the set of knobs related to extra invalid tags also does not make much sense without skews, starting with skews and associativity seems to be the most obvious choice. Figure 5.7 shows the trend of eviction rate as we vary the cache associativity. Here we only consider two skews and increasing associativity with random skew selection. We observe that as we increase the cache associativity, the eviction rate decreases for a given eviction set size. Most interestingly, the eviction rate is almost zero up to a certain eviction set size, and this size increases with associativity. As we can observe from Figure 5.7, `Skew-2-Ass128` performs really well till an eviction set size of ≈ 270.

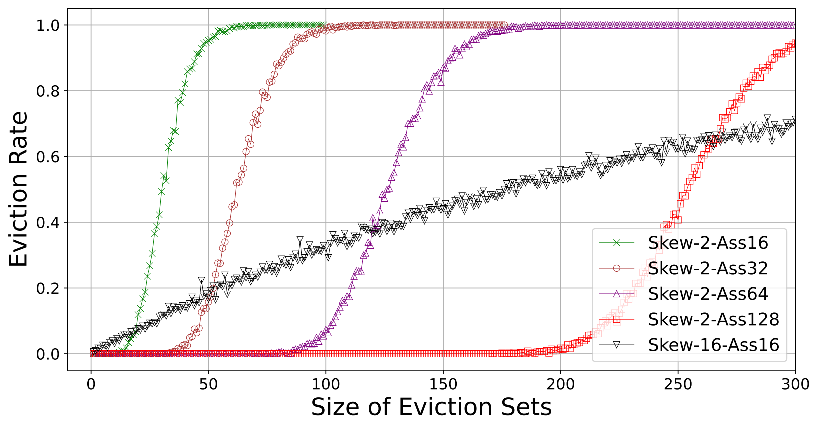


Figure 5.7: Eviction rate for two skews (`Skew-2`) with random skew selection, local LRU eviction, and associativity (`Ass`).

Explanation. Intuitively, high associativity requires a very large eviction set to evict the target address. This is true even without any skews. Skews, on the other hand, make such eviction sets probabilistic, thereby decreasing the eviction rate. However, as we can observe from Figure 5.7, the higher associativity works as a dominating factor, keeping the eviction rate close to zero up to a certain size of the eviction set. The eviction rate increases rapidly after this threshold, but due to skews, the increase in eviction rate is tapered. We have also tested the effect of using load-aware skew selection rather than random skew selection. However, since our setup has neither extra invalid tags nor global eviction, when we consider the cache state to be full, there is no difference in the security provided by the two skew-selection methods.

We also show that skews and high associativity become more effective with load-aware, extra invalid tags, and global eviction (refer Figure 5.8). The combination of all these knobs can provide a zero eviction probability, even for large eviction sets. However, such a combination of knobs requires various modifications to conventional set-associative caches.

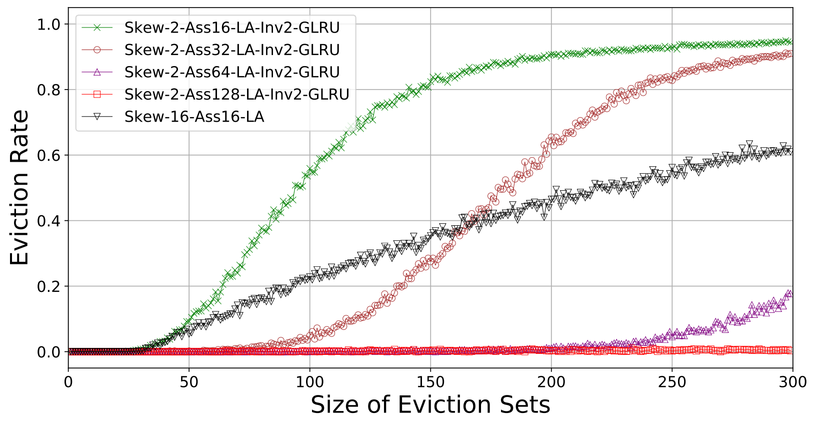


Figure 5.8: Eviction rate for two skews (`Skew-2`) with load-aware skew selection (`LA`), extra invalid tags (`Inv`), global LRU eviction (`GLRU`), and varying associativity (`Ass`).

Takeaway

High associativity even with two skews rapidly reduces eviction rate, without requiring any other knobs. Moreover, the eviction rate remains close to zero up to a certain eviction set size.

5.2.5 Knob 4: Replacement Policy

The replacement policy consists of the cache line insertion policy, state update policy, and cache line eviction policy. So far in this paper (Section 5.2.3), we have considered the LRU replacement policy (both global and local). Prior works [80, 81] have shown the effect of replacement policy on security. In this subsection, we explore five popular policies: *Random replacement* (`Ran`), *least recently used* (`LRU`), *pseudo-least recently used* (`PLRU`), *randomized pseudo-least recently used* (`RPLRU`) [81], and *Re-Reference Interval Prediction* (`RRIP`) [82]. LRU is rarely used in practice due to its implementation complexity; instead, approximations like PLRU are commonly used. Prior work [81] adapted PLRU to skewed randomized caches via RPLRU. Since we focus exclusively on skewed randomized designs in this section, we evaluate only RPLRU and omit PLRU.

We evaluate the high-associativity configurations as well as the successful configurations we have found with invalid tags. Figure 5.9 shows the effect of the replacement policy on the eviction rate of highly associative skewed caches. An interesting observation is that the random replacement policy significantly reduces the gain obtained due to high associativity, and the trend is almost identical to having 16 skews each with associativity 16. However, we can provide this security with two skews only, making it more practical to implement. Another important point here is that only local replacement makes sense for such designs as these caches are skewed set-associative caches. There is no mechanism (such as invalid tags) to detect if a set is "almost full" and trigger a global eviction. The only way an eviction can be triggered is by detecting if a set is full, and this leaks the same information about the sets as a local eviction. Therefore, we limit our evaluation for this configuration with local eviction policies only. We also observe that RPLRU has a similar behavior to LRU, suggesting that LRU approximations can achieve security comparable to true LRU.

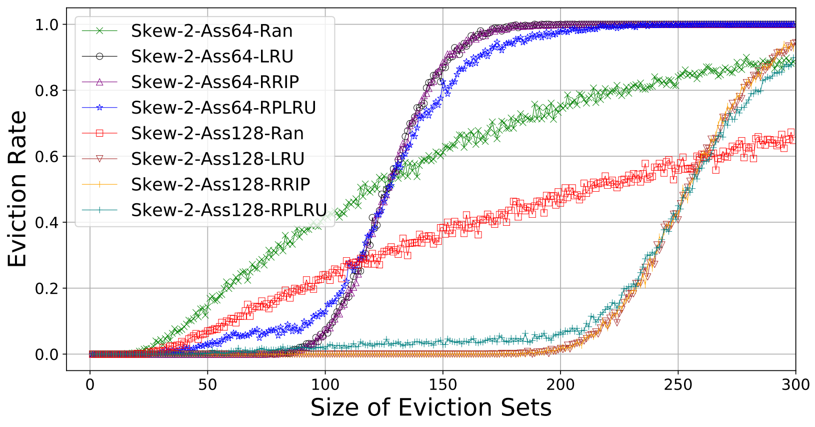


Figure 5.9: Eviction rate for two skews (`Skew-2`) with random skew selection, varying associativity (`Ass`), and different local replacement policies (`LRU`, `Ran`, `RRIP`, and `RPLRU`).

Explanation. We explain why random eviction performs worse than RRIP, LRU, and RPLRU, all of which use reuse information to decide eviction candidates. Consider a set in one of the skews where the target address x resides, containing warm-up entries, the target address, and eviction set entries. To evict the target address, we need n_w eviction set entries. Once these are present, the target can be evicted with certainty. With random skew selection, the eviction rate gradually increases, reaching one after a certain threshold set size. In random eviction, we do not need to completely fill a set, as each entry in the eviction set has a probability of $1/n_w$ of evicting the target. There is an equal probability of self-eviction among eviction set entries, preventing the eviction rate from reaching one. Since each address has a constant probability of evicting the target, the exact number of eviction set entries becomes less important. Even with fewer entries, eviction still occurs, leading to a non-zero eviction probability. This makes random eviction less efficient than

LRU. Figure 5.10 shows the effect of using extra invalid tags, load-aware skew selection with global random, global LRU, global RRIP, and global RPLRU eviction. We observe that the global LRU and global RRIP eviction policies perform much better than the global random eviction policy, and this gap widens with an increasing number of extra invalid tags. We also observe that the global RPLRU performs similarly to global LRU while being more practical to implement.

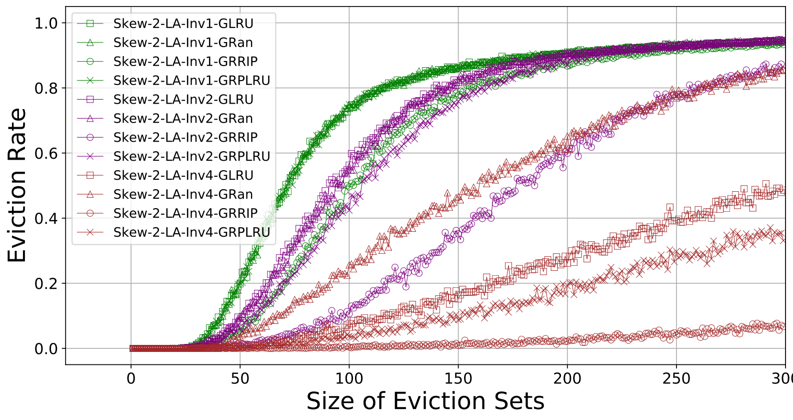


Figure 5.10: Eviction rate for two skews (`Skew-2`) with load-aware skew selection (`LA`), extra invalid tags (`Inv`), and global replacement policies (`GLRU`, `GRan`, `GRRIP`, and `GRPLRU`).

Explanation. The improved security of global LRU over global random eviction stems from a reasoning similar to why local LRU outperforms local random eviction. With global random eviction, every accessed address has a finite probability of evicting the target address. In contrast, under global LRU, the eviction set is accessed immediately after the target, making the target the most recently used cache entry. Therefore, the eviction set can only evict the target address after filling the entire cache, making successful target eviction significantly harder.

Takeaway

Random replacement policies perform worse than LRU and RRIP for conflict-based attacks, as they result in higher eviction rates.

5.2.6 Knob 5: Remapping

Remapping is a critical knob in many secure randomized cache designs. It is necessary when eviction sets (with a reasonable non-zero eviction rate) can be constructed within a finite time. Remapping changes the address-to-set mapping by updating the block cipher key and make any eviction set constructed for the existing mapping useless. Most of the designs discussed so far do require remapping, except a few leveraging several extra invalid tags (e.g., Mirage), and thereby making eviction rate construction difficult. In this section, we only evaluate configurations requiring remapping (`Skew-2-Ass64` and `Skew-2-Ass128`, and designs with a few invalid tags.).

Using Metric II: The evaluation of remapping differs from the other knobs discussed in Section 5.2. Instead of assessing its usefulness (which is evident), we evaluate the remapping period. A high remapping period (that is, remapping at a slower rate) for a randomized cache improves performance, but also provides an attacker a larger window to attack. Setting the remapping period is, therefore, a performance security tradeoff. The remapping period cannot be evaluated with the eviction rate metric used so far. Therefore, we resort to our second metric – the *number of evictions to generate an eviction set.* Recall that so far, we have calculated the eviction rate metric based on ideal eviction sets. For the remapping period, however, we need to practically construct the eviction sets using the state-of-the-art eviction set generation algorithms. Given an eviction set size ($|E|$), and a remapping period (R), the expected number of evictions ($\mathcal{E}$) needed to formulate the eviction set for a skewed set-associative cache with S sets, n_w ways, and k skews can be written as $\mathcal{E} = \frac{R\cdot S\cdot n_w}{P'}$ [7]. Here $P' = 1 - \sum_{i=0}^{\frac{|E|\cdot n_w}{k}-1} \binom{R\cdot S\cdot n_w}{i} P^i (1-P)^{R\cdot S\cdot n_w - i}$ and $P = \frac{1}{S\cdot k}$. We use this formula to estimate the remapping period R based on our obtained values of $|E|$ and $\mathcal{E}$ from Metric-II.

Figure 5.11 shows the number of LLC evictions required to construct eviction sets that achieve a 30% eviction rate for various cache configurations. The 30% threshold is chosen following [7] to enable fair comparison. For `Skew-2-Ass64`, achieving this rate requires an eviction set of 116 entries, which takes an average of 3.7 million LLC evictions using the conflict testing algorithm. `Skew-2-Ass128` requires 241 entries, needing approximately 7.8 million LLC evictions. In contrast, `Skew-16` achieves the same eviction rate with 87 entries and 2.6 million LLC evictions. More precisely, for `Skew-2-Ass128`, $|E| = 241$, and $\mathcal{E} = 7.8 \times 10^6$, which requires a remapping period $R \approx 228$. For `Skew-2-Ass64`, we obtain $R \approx 103$. Notably, for `Skew-16` (with associativity 16) $R \approx 39$, which establishes the efficacy of high-associativity.

Comparing eviction set finding algorithms: In Figure 5.11, the results presented are with respect to the Conflict Testing [7] eviction set generation algorithm. However, choosing Conflict Testing was not ad-hoc, but based on a comparison between the state-of-the-art eviction set generation algorithms, among which Conflict Testing is found to perform the best. More precisely, we compare Conflict Testing [7] and Prime, Prune and Probe [19] algorithms for eviction set generation, which are widely used for attacking randomized caches. Table 5.2 shows a comparison of the number of LLC evictions required to generate eviction sets of various sizes for `Skew-2-Ass64` and `Skew-16` using the two algorithms. We can clearly observe that Conflict Testing requires lesser number of LLC evictions. Additionally, Prime, Prune and Probe requires $\approx 10\times$ the number of LLC accesses as Conflict Testing for the same eviction set size [7]. This makes it infeasible to simulate for large eviction set sizes and therefore we only show results for smaller eviction set sizes. Such results justify our choice of eviction set finding algorithm.

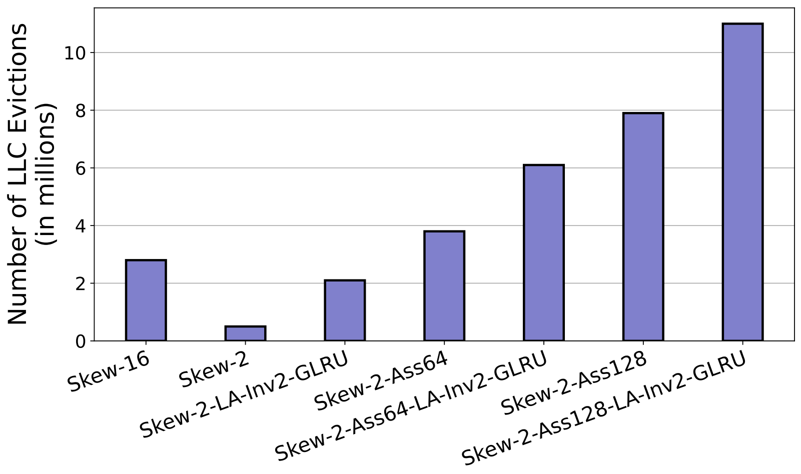


Figure 5.11: The number of LLC evictions required to create eviction sets that achieve a 30% eviction rate.

Takeaway

Randomized cache designs with high associativity can have higher remapping periods compared to designs such as CEASER-S and `Skew-16`.

Table 5.2: A comparison of the number of LLC evictions needed to create eviction sets of different sizes using Conflict Testing and Prime, Prune and Probe.

Eviction Set Size	**Conflict Testing**		**Prime, Prune and Probe**	
	Skew-2-Ass64	Skew-16	Skew-2-Ass64	Skew-16
10	0.3 million	0.3 million	0.8 million	0.8 million
20	0.7 million	0.6 million	1.6 million	1.5 million
30	1.0 million	1.0 million	2.0 million	2.1 million
40	1.3 million	1.3 million	2.6 million	2.7 million

5.2.7 Sensitivity to Cache Size

We further explore highly associative configurations by varying the cache size from 1 MB to 4 MB, and also considering a 96 MB cache, while observing the eviction set sizes required to reach a 30% eviction rate. Once again we use Metric-II. Figure 5.12 shows that the eviction rate for `Skew-2-Ass64` and `Skew-2-Ass128` remains unaffected by cache size. This is expected, as the likelihood of evicting a target address depends only on the cache associativity, not on the number of sets. However, as shown in Figure 5.13, the number of evictions required to construct such eviction sets increases with cache size. This is because current state-of-the-art eviction set search algorithms scale linearly with cache size. Extrapolating to typical LLC sizes, for a 96 MB cache, approximately 187 million LLC evictions are required for `Skew-2-Ass64`, and about 381 million for `Skew-2-Ass128`.

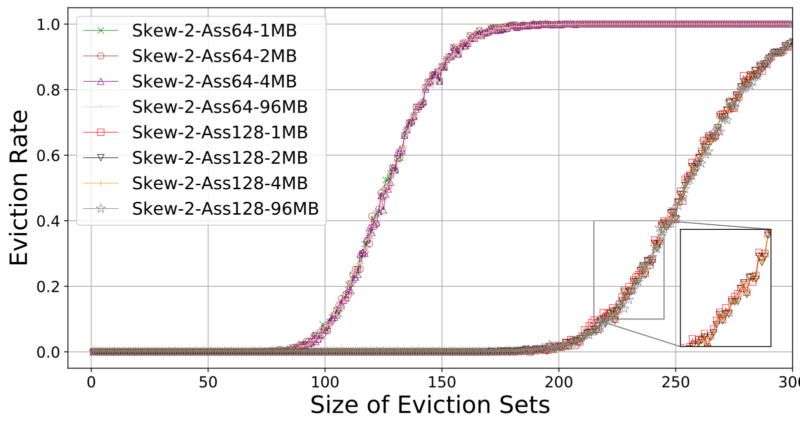


Figure 5.12: Eviction rate for two skews (`Skew-2`) with random skew selection, high associativity (`Ass64` and `Ass128`), and varying cache size (associativity remains the same).

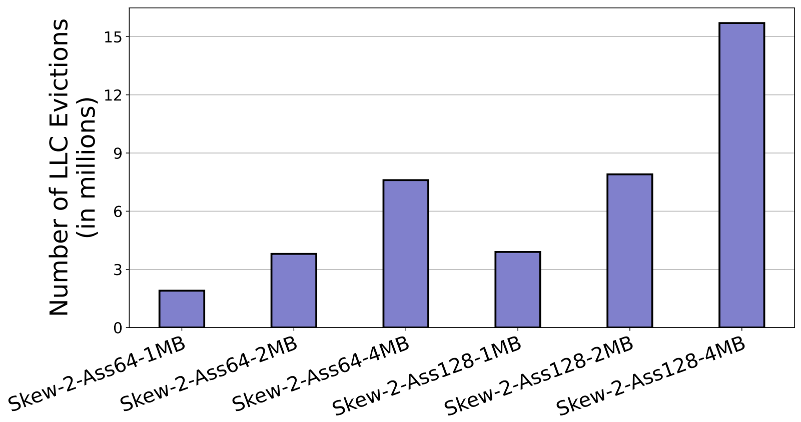


Figure 5.13: The number of LLC evictions required to create eviction sets achieving 30% eviction rate.

5.3 The Knobs and Occupancy-based Attacks

Occupancy-based attacks do not leak information at the set level and do not rely on eviction sets. Randomized cache knobs are primarily designed to prevent eviction set creation and they provide no guarantee against occupancy-based attacks. While it is essential for cache designs to defend against conflict-based attacks, they must also avoid unintentionally making occupancy-based attacks easier. This underscores the importance of evaluating cache knobs in the context of occupancy-based threats.

Most defenses against occupancy-based attacks focus on hardware partitioning [42, 75, 25, 44], with SassCache [14] being a notable exception. SassCache employs soft partitioning using block cipher-assisted domain isolation, reducing the likelihood of cross-domain evictions by limiting the bit-width of the cryptographic mapping. To date, there has been limited exploration of cache design knobs specifically targeting occupancy-based attacks. SassCache's soft partitioning remains the only prominent strategy. We begin by analyzing the effectiveness of knobs originally intended for conflict-based attacks in mitigating occupancy-based threats. We then evaluate soft-partitioned designs like SassCache. For comparison, we also consider an ideal static partitioned design that ensures complete domain isolation. We do not explore other partitioning schemes, as our focus is on evaluating knobs within randomized cache designs.

Evaluated designs. To streamline our analysis, we evaluate a selected set of representative designs from Section 5.2: a skewed design (CEASER-S), an extra invalid tags-based design (Mirage), a highly skewed design (`Skew-16`), and a highly associative design (`Skew-2-Ass128`). For soft-partitioned randomized designs, we evaluate SassCache with a coverage of 39% ($t = -1$). We also evaluate a static way-based partitioned design as the ideal mitigation against occupancy-based attacks. All designs are compared to a fully associative design with random replacement (`FA-RR`).

Benchmarking strategy. For the occupancy-based attack evaluation, we use AES (OpenSSL implementation with T-tables) and modular exponentiation, measuring the number of encryptions required to distinguish between two secret keys. The attacker begins by filling the cache with a randomly chosen *occupancy set*—a collection of addresses capable of occupying the cache. The attack proceeds by priming the cache with this set, allowing the victim to execute, and then probing the same set to count the number of cache misses. Based on these miss statistics, the attacker attempts to differentiate between encryptions using different keys. This evaluation follows the methodology of CacheFX [27], and we use the same simulator. However, we explore additional design variants, many introduced through our knob-based design exploration. We collect 100,000 encryption traces per key and report the median number of encryptions required to distinguish between the keys.

Observations. Figure 5.14 shows the number of encryptions required to distinguish keys in AES and Modular Exponentiation, normalized to `FA-RR`. We observe that all the cache designs that use knobs designed for mitigating conflict-based attacks perform quite similarly to `FA-RR`, with the exception of the set-associative design. However, SassCache (soft partitioning) provides a much better security compared to other randomized cache designs that are secure to conflict-based attacks. Moreover, a way-based static partitioned design provides complete protection.

Explanation. Let us first explain why the set-associative cache performs relatively worse than other designs. This is because, in randomized cache designs, the randomness (due to skew selection or eviction policy) enhances security. Specifically, there are always addresses in the occupancy set that map to the same cache set, and the same can happen for victim accesses. These colliding addresses create self-evictions. For highly deterministic schemes, such as a set-associative cache, the self-evictions for both the victim and attacker are also deterministic, as each address can only map to one set in the cache. In contrast, in designs with more randomness, self-eviction noise is random. Since occupancy-based attacks focus on the total number of misses, any deterministic self-eviction for the victim will be an

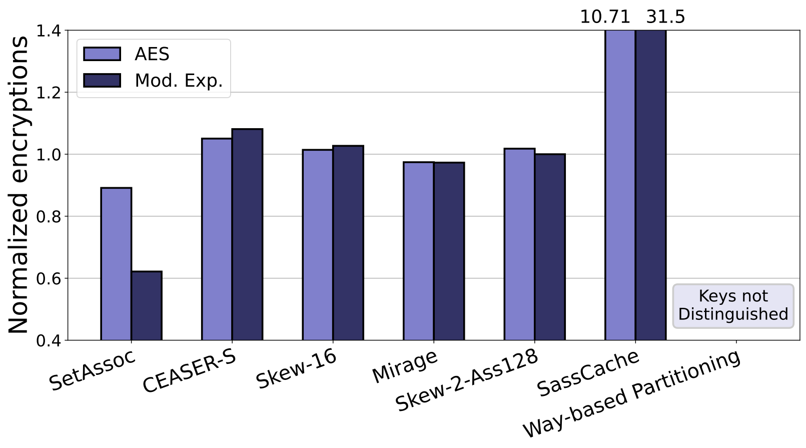


Figure 5.14: Normalized number of encryptions to distinguish the keys in AES and Modular Exponentiation for various cache designs. Normalization is done with respect to `FA-RR`.

artifact of the victim's access pattern and, therefore, more of a signal than noise for the attack. On the other hand, random self-evictions are imposed by the cache design and do not (entirely) depend on the victim's access pattern. This randomness adds noise and significantly increases the number of encryptions needed for an attack. To better understand the impact of deterministic vs. randomized eviction policies, we further examine the LRU replacement policy (see Figure 5.15). As shown, the lack of randomness significantly reduces the efficacy of schemes using deterministic policies compared to those with random eviction. The trend is similar for other deterministic replacement policies, such as RRIP, and thus the results for RRIP were omitted.

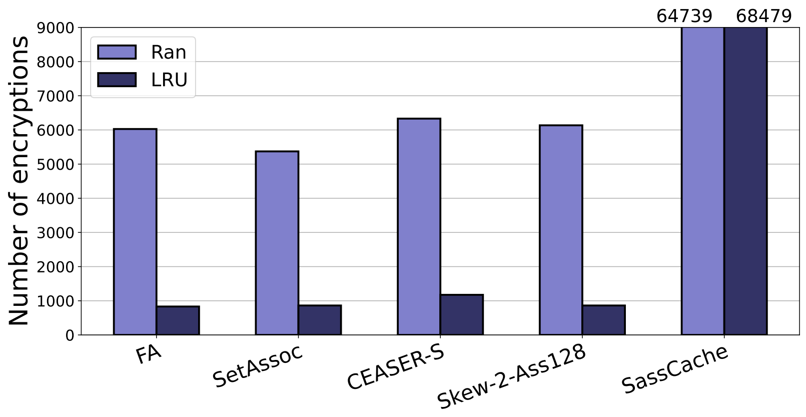


Figure 5.15: Number of encryptions to distinguish the keys in AES for different replacement policies (`Ran` and `LRU`).

Next, we explain why all the randomized designs perform almost the same with respect to occupancy-based attacks in Figure 5.14. We attribute this to the evaluation strategy where the entire cache has been occupied. *Due to full occupancy, the adversary can observe all misses caused by the victim's access irrespective of whether the replacement policy is local or global.* This is the main reason behind all representative designs having almost the same performance with respect to the occupancy-based attacker.

Finally, we examine SassCache and static way-based partitioning. SassCache guarantees a certain degree of isolation between the cache entries controlled by different security domains, while static partitioning ensures that there is no leakage between the security domains. While both designs provide significantly better security against occupancy-based attacks compared to other randomized cache designs, SassCache still cannot fully block leakage of information between domains, and thus is still vulnerable to occupancy-based attacks compared to a static way-based partitioned design, which cannot be broken by any conflict or occupancy-based attack.

Low-occupancy-based attacks. So far, we have focused on attacks that fully occupy the cache to extract sensitive data such as AES keys. Recent work [28] extends this threat model to *low-occupancy-based attacks*, which can be used for covert channels, process fingerprinting, and key recovery. [28] uses *guessing entropy*, defined as the expected number of guesses an attacker needs to make to correctly guess the secret key, as their metric to measure the threat. The lower the guessing entropy, the easier for the attacker to obtain the secret key. Their findings show that such attacks are feasible on Mirage even with only 10% cache occupancy. Moreover, at low occupancy levels, Mirage performs worse than ScatterCache and CEASER-S for covert channel attacks, indicating that *the locality of replacement impacts security under low occupancy.* This trend is attributed to Mirage's global random eviction policy [28]. We extend their analysis to our high-associativity designs (`Skew-2-Ass64` and `Skew-2-Ass128`) as shown in Figure 5.16, and find that these designs offer similar resilience to SassCache and ScatterCache, and do not exhibit the same vulnerability as Mirage.

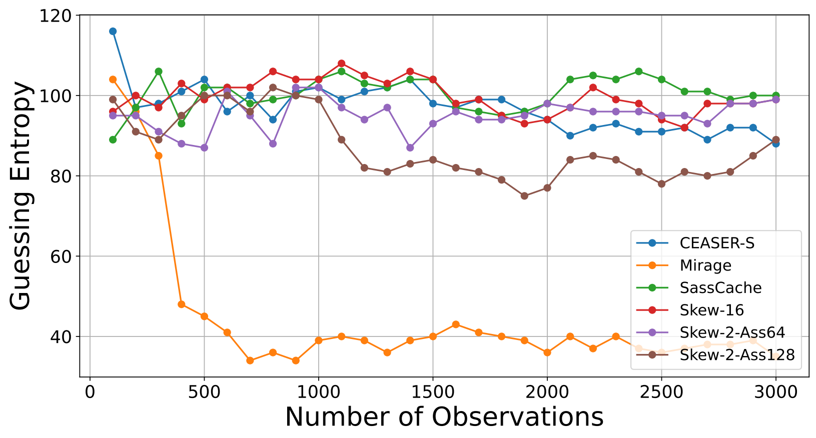


Figure 5.16: Guessing entropy for AES key recovery across a 50% occupancy rate for varying number of observations.

Takeaway

Deterministic replacement policies offer worse security than random ones against occupancy-based attacks. Local eviction policies provide better security than global ones for low-occupancy-based attacks, while performing similarly for full-occupancy-based attacks.

Chapter 6

Conclusion

We presented Maya, a randomized fully associative last-level cache that uses additional tag entries and fewer data entries. Overall, Maya guarantees that it will take 10^{16} years for one set associative eviction to initiate a conflict-based attack, which is more than the system's lifetime. Maya is energy-efficient (5.46% less static power) and area-efficient (28.11% savings) thanks to a smaller data store. Maya provides a strong security guarantee with storage savings (and not overhead) compared to a non-secure baseline cache. Overall, Maya provides the sweet spot in terms of security, performance, area, and energy overhead.

We presented Avatar, a secure, morphable last-level cache (LLC) with three dynamic operation modes—*non-secure* (Avatar-N), *randomized secure* (Avatar-R), and *partitioned secure* (Avatar-P). Designed like a conventional set-associative LLC with no decoupling, Avatar ensures easy industry adoption. Avatar-R leverages high associativity with a lightweight Mirage-like design to maintain security against conflict-based attacks while preserving cache capacity. Additionally, it morphs into a partitioned LLC to defend against occupancy-based attacks in Avatar-P. When security is unnecessary, the cache can switch to Avatar-N to operate as a traditional LLC, optimizing performance and power. Avatar-R provides a strong security guarantee against conflict-based attacks–one SAE in a million years–and Avatar-P mitigates both conflict-based and occupancy-based attacks while outperforming conventional partitioned LLCs.

The key findings from our knob-based systematization can be summarized as follows: (i) Skewing is the most versatile knob and should be included in all randomized cache designs. (ii) Extra invalid tags result in designs closest to fully associative caches, offering strong security against conflict-based attacks, but require complex knob combinations, increasing design complexity. Moreover, even with this increased design complexity, we do not get any advantage against occupancy-based attacks. Additionally, designs with global eviction, such as Mirage, have been shown to be vulnerable to low-occupancy-based attacks. (iii) High associativity, even with just two skews, provides strong security against conflict-based attacks. However, they do not provide any benefit against occupancy-based attacks. In general, they require relatively less number of knobs to achieve good security. However, remapping is required which invalid tag-based designs can avoid. Overall, while there is no clear winner in this study, from a viewpoint of design complexity versus security tradeoff, high-associativity designs should be considered as viable options.

6.1 Open Problems

Unified security metric for occupancy-based attacks. In this work, we used eviction rate and the difficulty of eviction set generation as metrics to evaluate cache resilience against conflict-based attacks. For occupancy-based attacks, we relied on cryptographic workloads such as AES and modular exponentiation, due to the lack of established, targeted metrics in the literature. With the growing threat of occupancy-based attacks, there is a pressing need for a unified metric to quantify cache resilience in such scenarios.

SassCache security against multiple adversary processes. SassCache provides a thorough security evaluation for scenarios with a single attacker domain. However, multi-domain attacks are also realistic. In such cases, the authors suggest that cloud vendors select the parameter t based on the expected co-location probability, using the formula $C_t = 1–(1–C)^{n_d}$. However, accurately predicting the number of attacker domains in advance is often impractical and may lead to security vulnerabilities if the actual number exceeds the expectation. Although t is reconfigurable in hardware, a comprehensive analysis of the multi-domain attacker threat and the effectiveness of the proposed mitigation remains essential. Finally, more alternatives should be explored for occupancy-based attack mitigation, which are agile and scalable, and yet provide the same security as hard partitioning.

Appendix I

Avatar LLC in action

In Avatar-R, the line address feeds into two independent ciphers (Cipher-1 and Cipher-2) that generate hashed addresses (❶ in Figure I.1). For Avatar-N and Avatar-P, the ciphers are power-gated, and 11 index bits and seven index bits, respectively, from the original address are used instead of hashed addresses. A 2x1 MUX (❷ in Figure I.1) selects the 11 index bits or the first hash address 7 bits (zero-extended by 4), depending on the mode. Another pair of MUXs chooses between the original and hashed addresses depending on the secure mode. A 7-bit decoder decodes the seven msb bits to enable the appropriate set for tag matching. To differentiate between the address tags for Avatar-R and Avatar-P, we add two 2x1 MUXs before tag-matching. In Avatar-R and Avatar-P, 256 tag ways are matched across two skews, while only 16 ways are matched in Avatar-N. A 1x2 DEMUX (❸ in Figure I.1) enables a 4-bit decoder in Avatar-N to select one of 16 blocks. In Avatar-R and Avatar-P, all 128 ways in Skew-1 are enabled, with another 7-bit decoder handling Skew-2 using the second hash address's seven bits. To ensure correct functioning in all operation modes, we add OR gates (❹ in Figure I.1), which enable the correct 16-way blocks for tag-matching based on the mode of operation. The tag matches with one of the 16-way blocks and outputs the data from the corresponding cache line. Figure I.1 highlights the hardware components activated in each Avatar operation mode. During mode transitions, unused components are power-gated to reduce energy consumption.

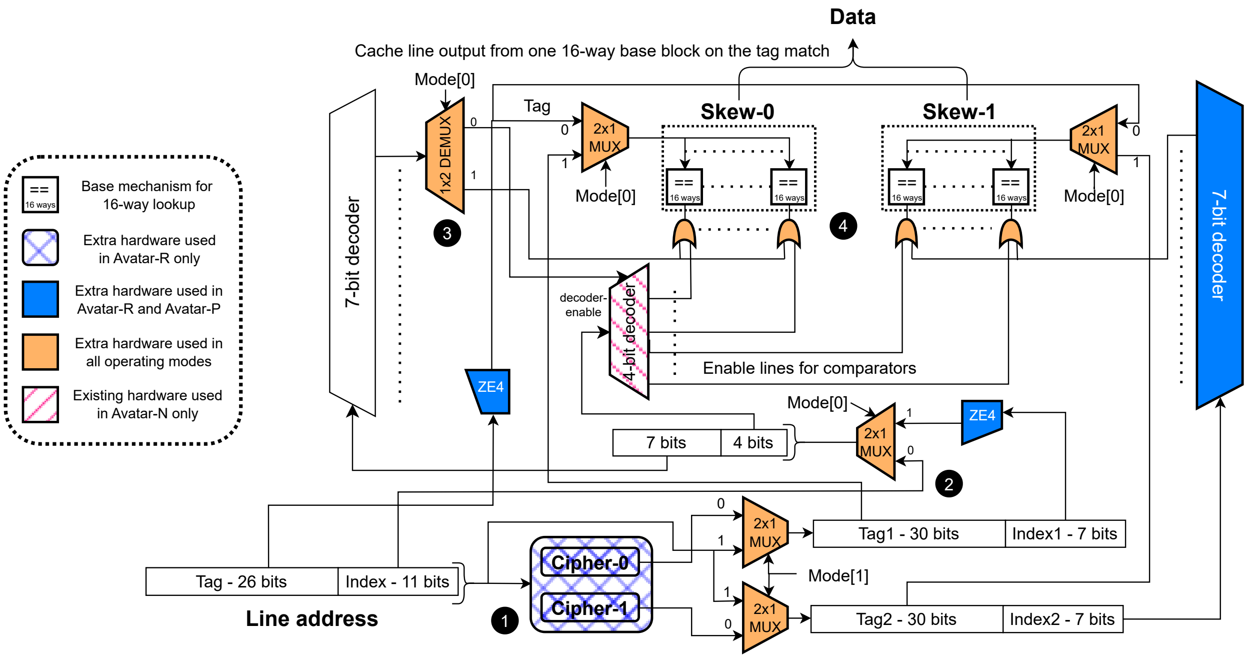

Data
Cache line output from one 16-way base block on the tag match
Mode[0]
Tag
Skew-0
Skew-1
1x2 DEMUX
2x1 MUX
7-bit decoder
4-bit decoder
decoder-enable
Enable lines for comparators
ZE4
7 bits
4 bits
Tag - 26 bits
Index - 11 bits
Line address
Cipher-0
Cipher-1
Mode[1]
Tag1 - 30 bits
Index1 - 7 bits
Tag2 - 30 bits
Index2 - 7 bits
16 ways
Base mechanism for 16-way lookup
Extra hardware used in Avatar-R only
Extra hardware used in Avatar-R and Avatar-P
Extra hardware used in all operating modes
Existing hardware used in Avatar-N only

Figure I.1: Avatar in action. `Mode` is a 2-bit vector. `Mode="00"` refers to the Avatar-N, `Mode="01"` refers to Avatar-R, and `Mode="11"` refers to Avatar-P. The extra hardware required for morphability has been marked in the figure, along with how the different components are power-gated as per their use.

Appendix II

Probability of an SAE with Avatar

To estimate the probability of an SAE for Avatar, we use the bucket-and-balls model as described in Mirage. The buckets represent cache sets, the balls denote tag entries, and a ball throw represents a fill. The buckets are initialized with as many balls as the cache capacity. This ensures that we model the best-case scenario for the attacker. On a ball throw, two random buckets are chosen, and the ball is inserted into the bucket that is less occupied. If both buckets are at full capacity, we will get a bucket spill representing an SAE. For our analysis, a spill-free scenario is modeled, where the buckets have unlimited capacity. Following the methodology outlined by Mirage, we end up with Equation II.1. Note that the numerator has 122 ways per skew because 6 out of the 128 ways per skew are invalidated in the Avatar secure mode.

$$Pr(n{=}N{+}1) = \frac{122}{N+1} \times \Big(Pr(n{=}N)^2 + 2 \times Pr(n{=}N) \times Pr(n{>}N) \Big) \tag{II.1}$$

where $Pr(n = N+1)$ represents the probability of a bucket having $N+1$ balls. As we increase n, $Pr(n{=}N) \rightarrow 0$ and therefore $Pr(n{>}N) \ll Pr(n{=}N)$. Using this approximation, Equation II.1 can be simplified to Equation II.2 for larger values of n. Similar to the security analysis for Mirage, we only use this approximation once $Pr(n{=}N)$ becomes smaller than 0.01.

$$Pr(n{=}N{+}1) = \frac{122}{N+1} \times Pr(n{=}N)^2 \tag{II.2}$$

Frequency of spills. We simulate the bucket-and-ball model for one trillion iterations and obtain the probability $Pr_{exp}(n = 92) \approx 7.5 \times 10^{-12}$. Per our simulations, there is no

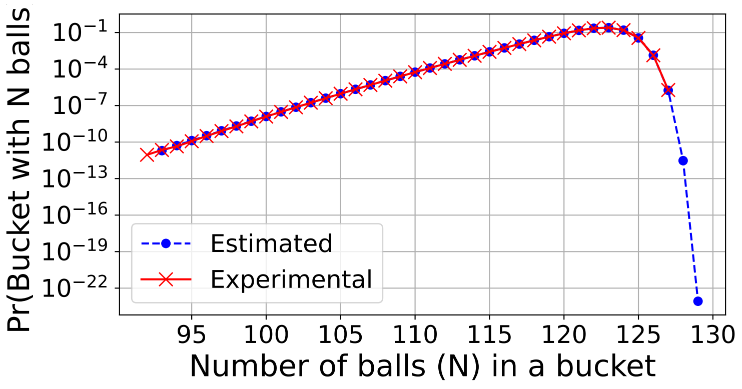


Figure II.1: Probability of a bucket having N balls (Pr(n = N)) - experimental and estimated using the analytical model.

bucket with a ball count of less than 92. Using this value in Equation II.1, we recursively calculate $Pr_{est}(n = N+1)$ for $N \in [93, 125]$. Once the probability becomes less than 0.01 ($N \in [126, 128]$), we use Equation II.2. If we consider a cache design with W ways per skew, then the probability of an SAE (i.e. a bucket spill) will be given by $Pr(n = W+1)$. The spill probability follows a double-exponential reduction. For $W = 126, 127, 128$, an SAE occurs every 10^5, 10^{11}, and 10^{23} line installs, respectively. Thus, the Avatar LLC design with 128 ways per skew in randomized secure mode has a frequency of one SAE in 10^{23} line installs or once in around $3 \cdot 10^6$ years, effectively providing complete security against eviction-based attacks. We also propose Avatar$^+$, which uses seven invalid ways per skew in the randomized secure mode and provides a security guarantee of no SAE in over $3.7 \cdot 10^{46}$ line installs or 10^{30} years (more robust than Mirage).

Appendix III

Evaluating Design Trade-offs

Security. We evaluate the security of the shortlisted designs based on the number of LLC evictions required to construct an eviction set. A higher number of required evictions indicates stronger resistance to conflict-based attacks. We use the Conflict Testing algorithm for eviction set generation, as it outperforms alternatives such as Prime, Prune and Probe. High-associativity designs, such as `Skew-2-Ass128`, and invalid-tag-based designs, like `Skew-2-Ass128-LA-Inv2-GLRU`, offer strong protection by making eviction set construction significantly more difficult and maintaining short remapping periods. Other designs, such as Mirage and SassCache, achieve an extremely low probability of set-associative evictions, rendering eviction set discovery nearly impossible.

Performance. We evaluate a range of secure randomized cache designs using the ChampSim [59] microarchitecture simulator. Our baseline is a non-secure 8-core system with a 16MB, 16-way set-associative LLC, employing the LRU replacement policy and 64-byte cache lines. We select 15 homogeneous workloads (42 sim-points) from SPEC CPU2017 [2], focusing on benchmarks with more than one LLC miss per kilo-instruction (MPKI) in a single-core 2MB cache configuration. Each simulation includes a 50-million-instruction warm-up phase, followed by the execution of 1.6 billion instructions across eight cores (200 million per core) within the region of interest. To compare performance across cache designs, we use the weighted speedup [62] metric for 8-core systems, normalizing all results to the non-secure baseline. Most designs incur only marginal performance overheads, with the largest slowdowns observed for `Skew-2-LA-Inv2-GLRU`, due to only 75% valid cache entries, and SassCache, due to soft partitioning between cores.

Logic. We analyze the total storage requirements of each design by calculating the number of bits per tag and data entry. Most designs incur a modest 2–3% storage overhead, primarily due to the addition of secure domain ID (SDID) bits in each tag. A notable exception is Mirage, which has significantly higher overhead due to a 75% larger tag store and pointer-based indirection between tag and data entries. Additionally, several designs require extra logic, such as circuitry for load-aware skew selection and block ciphers for address randomization. We compare this logic overhead to the 5.5 million gate equivalents (GEs) used in the 1MB L2 cache of the BROOM chip [83], scaled to a 2MB cache.

Power. We estimate dynamic and static power overheads using P-CACTI [64], configured in sequential access mode and modeled for 7nm FinFET technology. For LLC static power, most designs exhibit minimal overheads, with high-associativity designs incurring slightly higher overheads (up to 6%) due to more complex lookup circuitry. Mirage incurs a 19.6% static power overhead, attributed to a 75% increase in tag entries and the use of indirection pointers between tag and data arrays. The baseline has a static LLC power of $\approx$520mW.

For dynamic power, we focus on the LLC hit and miss power. Since each LLC miss goes directly to the DRAM, we also need to account for the DRAM access power. We estimate tag and data read/write energy using P-CACTI, and assume a DRAM read/write energy of approximately 60nJ. As DRAM access is much more expensive than LLC access in terms of energy, designs with lower LLC miss rates achieve better overall dynamic power efficiency compared to standard 16-way associative designs. For example, high-associativity caches, despite their higher dynamic access energy, have smaller dynamic power overheads than their 16-way associative counterparts. The baseline dynamic power (both LLC and DRAM access) is around 1800mW.

Table III.1: A comparison of secure randomized cache designs, the number of LLC evictions to create an eviction set with 30% eviction rate, and their performance, storage, power overheads.

Design	**LLC Evictions Needed to Create Eviction Set**	**Knobs Used**	**Performance Overhead**	**Logic Overhead**	**Dynamic Power Overhead**	**Static Power Overhead**
Skew-2 (CEASER-S)	0.5 million	Skews, Remapping	-1.3%	1.9%	2.5%	2%
Skew-16 (ScatterCache)	2.8 million	Large number of skews, Remapping	0.1%	1.7%	0.5%	1.4%
Skew-2-LA-Inv2-GLRU	2.3 million	Skews, Load-aware, Invalid Tags, Global Eviction, Remapping	1.3%	1.9%	5.2%	2%
Mirage	Not Possible	Skews, Load-aware, Invalid Tags, Global Eviction	0.2%	18.6%	-0.2%	19.6%
Skew-2-Ass64	3.8 million	Skews, High Associativity, Remapping	-2.1%	2.3%	1.8%	3.3%
Skew-2-Ass128	7.9 million	Skews, High Associativity, Remapping	-2.2%	2.4%	1.7%	6.4%
Skew-2-Ass64-LA-Inv2-GLRU	6.1 million	Skews, High Associativity, Load-aware, Invalid Tags, Global Eviction, Remapping	-2.2%	2.3%	1.8%	3.3%
Skew-2-Ass128-LA-Inv2-GLRU	11.0 million	Skews, High Associativity, Load-aware, Invalid Tags, Global Eviction, Remapping	-2.5%	2.5%	1.7%	6.4%
SassCache (coverage = 39%)	Not Possible	Skews, Soft Partitioning	2.3%	2.4%	7.4%	1.2%

Appendix IV

Impact of Warm-up States

In Section 5.2.2, we observed some non-intuitive trends in eviction rate results. We argue that these trends stem from the specifics of the eviction rate experiment rather than limitations of the metric itself. Following the method in [7], we gradually increase the size of the warm-up set in iterations by retaining cache entries from one iteration to the next, starting from an empty cache. This approach averages over varying cache states, smoothing out the influence of any particular initial state. It also significantly reduces simulation time by avoiding per-iteration warm-up. *However, this averaging over warm-up states is what leads to the confusing trends noted in Section 5.2.2.*

Warm-up state and load-aware skew selection. To elaborate, we refer to the eviction rate experiments in Figure 5.3, where we compare random and load-aware skew selection. Surprisingly, load-aware skew selection appears to offer some improvement over random selection, which seems counterintuitive. As discussed in Section 5.2.2, *any eviction set should be ineffective in the presence of load-aware skew selection, unless the cache is nearly full. However, since the cache being full is a common case, the overall improvement for non-full states should be negligible.* However, the eviction rate experiment averages results across all warm-up sizes, giving equal weight to states with low cache occupancy. This skews results and artificially lowers the eviction rate, creating a misleading impression. *While the trends in Figure 5.3 may seem promising from a design perspective, they do not reflect the true security behavior.* To validate this, we modified the experiment to compute eviction rate separately for each warm-up level. As shown in Figure IV.1, eviction sets become effective only at high warm-up percentages. At that point, *the benefit of load-aware skew selection vanishes*, since with a full cache, tie-breaking effectively makes it behave like random skew selection. This supports our interpretation. While averaging over cache states provides useful design-space insights and faster simulation (as done in [7] and throughout this paper) it does not always capture edge-case behaviors. Therefore, we use the more detailed, per-state eviction rate analysis selectively for promising designs.

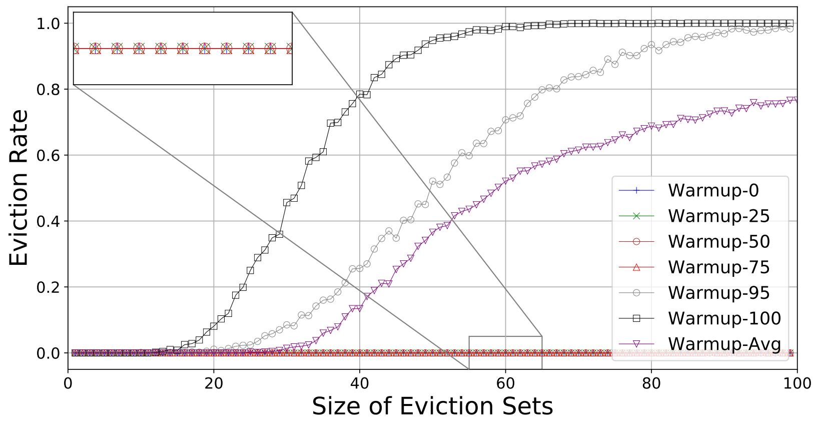


Figure IV.1: Eviction rate for two skews with load-aware skew selection, LRU eviction and different warm-up states. For example, `Warmup-75` refers to a `Skew-2-LA` cache with a 75%-filled warm-up state. `Warmup-Avg` refers to a `Skew-2-LA` cache with an average warm-up state as per [7].

Load-aware skew selection with invalid ways. Building on the previous discussion, we performed the enhanced eviction rate analysis for all promising designs but highlight only the most notable findings here. One such case involves combining load-aware skew selection with invalid ways. Figure 5.5 in Section 5.2.3 shows the standard eviction rate results for this combination. For comparison, Figure IV.2 in this section presents the enhanced eviction rate, calculated per warm-up state size. Specifically, Figure IV.2 captures the effect of starting from a full cache (excluding extra invalid tags) on configurations using load-aware skew selection, extra invalid tags, and global eviction. We observe a significant discrepancy between the standard and enhanced eviction rates when the number of invalid tags is small. This gap arises because the original experiment does not emphasize the more probable full-cache scenarios. In a fully occupied cache, eviction set addresses need only displace invalid tags in the target set, which is easier when there are fewer such tags. As the number of extra invalid tags increases, this warm-up state effect diminishes. Notably, Mirage—the most effective design leveraging invalid tags—employs a sufficiently large number of them.

However, while Mirage focuses on blocking set-associative evictions, our analysis uses a unified eviction rate-based metric, providing a broader perspective.

As a final takeaway, we argue that the issue arises not from the eviction rate metric itself, but from how it is computed to speed up evaluations. Both evaluation speed and accuracy are important. While a nearly full cache is generally common, it is not guaranteed. For example, if a target system is underutilized for a while, the attacker can choose this as an attack window. It can then craft a warm-up scenario to their advantage by exploiting such an underutilized system. Therefore, we recommend not overlooking warm-up effects, but performing enhanced eviction rate evaluation only for a selected subset of designs.

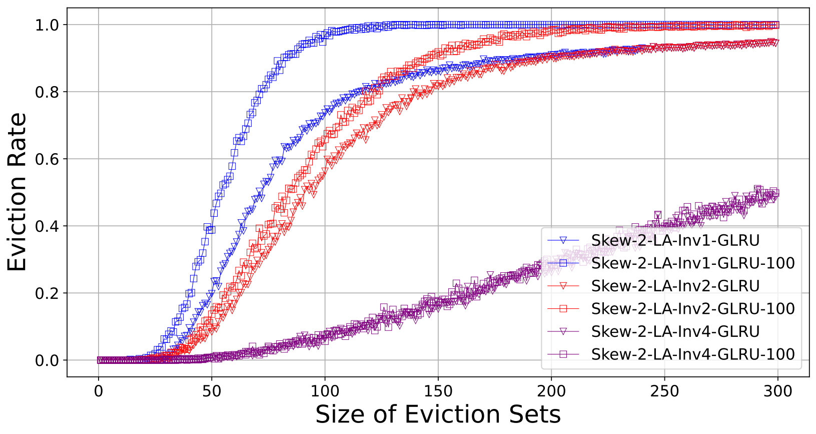


Figure IV.2: Eviction rate for two skews (`Skew-2`) with load-aware skew selection (`LA`), extra invalid tags (`Inv`), and global LRU eviction (`GLRU`) with different warm-up states. For example, `Skew-2-LA-Inv2-GLRU-100` refers to a `Skew-2-LA-Inv2-GLRU` cache with a 100%-filled (excluding the invalid entries) warm-up state.

Bibliography

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

[2] "SPEC CPU 2017 traces for champsim." https://hpca23.cse.tamu.edu/champsim-traces/speccpu/index.html, Feb. 2019.

[3] "GAP traces for champsim." https://utexas.app.box.com/s/2k54kp8zvrqdfaa8cdhfquvcxwh7yn85/folder/132804598561, Mar. 2021.

[4] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 78–89, 2016.

[5] B. et al., "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[6] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[7] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *Proceedings - 2021 IEEE Symposium on Security and Privacy, SP 2021*, pp. 955–969, 2021.

[8] B. Brumley and R. Hakala, "Cache-timing template attacks," in *Advances in Cryptology - ASIACRYPT 2009*, pp. 667–684, 12 2009.

[9] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[10] R. et al., “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, (New York, NY, USA), p. 199–212, Association for Computing Machinery, 2009.

[11] M. Yan, C. W. Fletcher, and J. Torrellas, “Cache telepathy: Leveraging shared resource attacks to learn DNN architectures,” in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2003–2020, 2020.

[12] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, L3 cache side-channel attack,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[13] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, “Robust website fingerprinting through the cache occupancy channel,” in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 639–656, USENIX Association, Aug. 2019.

[14] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, “Scatter and split securely: Defeating cache contention and occupancy attacks,” in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2273–2287, 2023.

[15] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 494–505, 2007.

[16] M. K. Qureshi, “Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 775–787, IEEE, 2018.

[17] M. K. Qureshi, “New attacks and defense for encrypted-address cache,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[18] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, “ScatterCache: Thwarting cache attacks via cache set randomization,” in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 675–692, USENIX Association, Aug. 2019.

[19] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 987–1002, 2021.

[20] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[21] A. Chakraborty, N. Mishra, S. Saha, S. Bhattacharya, and D. Mukhopadhyay, "Systematic evaluation of randomized cache designs against cache occupancy," 2025.

[22] T. Bourgeat, I. A. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pp. 42–56, ACM, 2019.

[23] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 974–987, 2018.

[24] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 37–49, 2021.

[25] K. Arikan, A. Farrell, W. Z. Cen, J. McMahon, B. Williams, Y. D. Liu, N. B. Abu-Ghazaleh, and D. Ponomarev, "Tee-shirt: Scalable leakage-free cache hierarchies for tees," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*, pp. 1–18, 2024.

[26] A. Bhatla, Navneet, and B. Panda, "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)*, pp. 32–44, 2024.

[27] D. Genkin, W. Kosasih, F. Liu, A. Trikalinou, T. Unterluggauer, and Y. Yarom, "Cachefx: A framework for evaluating cache security," in *Proceedings of the 2023 ACM*

Asia Conference on Computer and Communications Security, ASIA CCS '23, (New York, NY, USA), p. 163–176, Association for Computing Machinery, 2023.

[28] A. Chakraborty, N. Mishra, S. Saha, S. Bhattacharya, and D. Mukhopadhyay, "Systematic evaluation of randomized cache designs against cache occupancy," *To appear In 35th USENIX Security Symposium (USENIX Security 25)*, 2025.

[29] M. Cekleov and M. Dubois, "Virtual-address caches part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, pp. 64–71, sep 1997.

[30] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 168–179, 2018.

[31] W. et al., "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *Proceedings - 2021 IEEE Symposium on Security and Privacy, SP 2021*, Proceedings - IEEE Symposium on Security and Privacy, (United States), pp. 955–969, Institute of Electrical and Electronics Engineers Inc., May 2021.

[32] T. Kessous and N. Gilboa, "Prune+plumtree - finding eviction sets at scale," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 3754–3772, 2024.

[33] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 39–54, 2018.

[34] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization," *Proceedings 2020 Network and Distributed System Security Symposium*, 2020.

[35] X. Zhang, Z. Yuan, R. Chang, and Y. Zhou, " Seeds of SEED: H2Cache: Building a Hybrid Randomized Cache Hierarchy for Mitigating Cache Side-Channel Attacks ," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 29–36, 2021.

[36] J. P. Thoma, C. Niesler, D. Funke, G. Leander, P. Mayr, N. Pohl, L. Davi, and T. Güneysu, "ClepsydraCache – preventing cache attacks with Time-Based evictions," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 1991–2008, Aug. 2023.

[37] M. Qureshi, D. Thompson, and Y. Patt, "The v-way cache: demand-based associativity via global replacement," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 544–555, 2005.

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

[39] T. Unterluggauer, A. Harris, S. Constable, F. Liu, and C. 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)*, pp. 13–24, 2022.

[40] X. Zhang, H. Gong, R. Chang, and Y. Zhou, "Recast: Mitigating conflict-based cache attacks through fine-grained dynamic mapping," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 3758–3771, 2024.

[41] W. Song, Z. Xue, J. Han, Z. Li, and P. Liu, "Randomizing set-associative caches against conflict-based cache side-channel attacks," *IEEE Transactions on Computers*, vol. 73, no. 4, pp. 1019–1033, 2024.

[42] Y. Kelemework and A. R. Alameldeen, "INTERFACE: An Indirect, Partitioned, Random, Fully-Associative Cache to Avoid Shared Last-Level Cache Attacks," in *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 37–49, 2024.

[43] A. Kalita, Y. Yang, A. K. Murali, and A. Venkat, "Ceviche: Capability-enhanced secure virtualization of caches,"

[44] N. R. Holtryd, M. Manivannan, and P. Stenström, "Scale: Secure and scalable cache partitioning," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 68–79, 2023.

[45] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 187–198, 2010.

[46] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, “HybCache: Hybrid Side-Channel-Resilient caches for trusted execution environments,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[47] R. Wang and L. Chen, “Futility scaling: High-associativity cache partitioning,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 356–367, 2014.

[48] G. Saileshwar, S. Kariyappa, and M. Qureshi, “Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning,” in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 37–49, 2021.

[49] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, “The reuse cache: Downsizing the shared last-level cache,” in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 310–321, 2013.

[50] A. Seznec, “A case for two-way skewed-associative caches,” in *20st Int'l Symp. on Computer Architecture (ISCA)*, pp. 169–178, May 1993.

[51] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin, “PRINCE - A low-latency block cipher for pervasive computing applications (full version),” *IACR Cryptol. ePrint Arch.*, p. 529, 2012.

[52] D. J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[53] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+flush: A fast and stealthy cache attack,” in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[54] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch side-channel attacks: Bypassing smap and kernel aslr,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[55] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive Last-Level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[56] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1967–1984, USENIX Association, Aug. 2020.

[57] D. et al., "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," vol. 8, jan 2012.

[58] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 332–345, Association for Computing Machinery, 2019.

[59] "ChampSim simulator." http://github.com/ChampSim/ChampSim, May 2020.

[60] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, pp. 135–148, 1981.

[61] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–131, 2020.

[62] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," *SIGOPS Oper. Syst. Rev.*, vol. 34, p. 234–244, nov 2000.

[63] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *37th Int'l Symp. on Computer Architecture (ISCA)*, pp. 60–71, June 2010.

[64] Pcacti tool, Online. Available: `https://sportlab.usc.edu/downloads/`.

[65] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, "The simon and speck lightweight block ciphers," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015.

[66] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0," Tech. Rep. HPL-2009-85, HP Labs, Apr. 2009.

[67] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016* (T. Holz and S. Savage, eds.), pp. 857–874, USENIX Association, 2016.

[68] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel tdx demystified: A top-down approach," *ACM Comput. Surv.*, vol. 56, Apr. 2024.

[69] "Intel sgx." Available at `https://developer.arm.com/ip-products/security-ip/trustzone`.

[70] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 987–1002, 2021.

[71] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-opt: Practical optimal cache replacement for graph analytics," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 668–681, 2021.

[72] S. Feizi, A. Ahmadi, and A. Nemati, "A hardware implementation of simon cryptography algorithm," in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, pp. 245–250, 2014.

[73] E. Gulcan, A. Aysu, and P. Schaumont, "A flexible and compact hardware architecture for the simon block cipher," in *Lightweight Cryptography for Security and Privacy* (T. Eisenbarth and E. Öztürk, eds.), (Cham), pp. 34–50, Springer International Publishing, 2015.

[74] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1110–1123, 2020.

[75] G. Dessouky, E. Stapf, P. Mahmoody, A. Gruler, and A. Sadeghi, "Chunked-cache: On-demand and scalable cache isolation for security architectures," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28*, pp. 1–18, 2022.

[76] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 9–12, 2020.

[77] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, pp. 450–466, 2007.

[78] G. Saileshwar and M. Qureshi, "The Mirage of breaking MIRAGE: Analyzing the modeling pitfalls in emerging "attacks" on MIRAGE," *IEEE Computer Architecture Letters*, pp. 121–124, 2023.

[79] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for Side-Channel attacks against the LLC," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pp. 427–442, 2019.

[80] M. Peters, N. Gaudin, J. P. Thoma, V. Lapôtre, P. Cotret, G. Gogniat, and T. Güneysu, "On the effect of replacement policies on the security of randomized cache architectures," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, (New York, NY, USA), p. 483–497, Association for Computing Machinery, 2024.

[81] F. Stolz, J. P. Thoma, P. Sasdrich, and T. Güneysu, "Risky translations: Securing tlbs against timing side channels," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2023, no. 1, pp. 1–31, 2023.

[82] J. et al., "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), p. 60–71, Association for Computing Machinery, 2010.

[83] C. Celio, P.-F. Chiu, K. Asanović, B. Nikolić, and D. Patterson, “Broom: An open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos,” *IEEE Micro*, vol. 39, no. 2, pp. 52–60, 2019.

Publications

1. A. Bhatla, Navneet, B. Panda, "The Maya Cache: A Storage-efficient and Secure Fully-associative Last-level Cache," in *Proceedings of the ACM/IEEE 51st International Symposium on Computer Architecture (ISCA)*, pp. 32-44, 2024

2. A. Bhatla, H. Bhavsar, S. Saha, B. Panda, "SoK: So, You Think You Know All About Secure Randomized Caches?," in *Proceedings of the 34th USENIX Conference on Security Symposium*, 2025

Acknowledgment

I express my gratitude to my guide Prof. Biswabandan Panda for providing me the opportunity to work on this topic. I would also like to thank my parents for their unwavering support and my friends for always motivating me to do my best.

Anubhav Bhatla
Roll No.: 200070008
Date: 23rd June 2025
Place: Indian Institute of Technology Bombay