INTERFACE: An Indirect, Partitioned, Random, Fully-Associative Cache to Avoid Shared Last-Level Cache Attacks

by

Yonas Girma Kelemework

B.Sc., Korea Advanced Institute of Science and Technology(KAIST), 2021

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2023

Declaration of Committee

Name:	**Yonas Girma Kelemework**
Degree:	**Master of Science**
Thesis title:	**INTERFACE: An Indirect, Partitioned, Random, Fully-Associative Cache to Avoid Shared Last-Level Cache Attacks**
Committee:	**Chair:** Keval Vora Assistant Professor, Computing Science
	Alaa Alameldeen Supervisor Associate Professor, Computing Science
	Arrvindh Shriraman Committee Member Associate Professor, Computing Science
	Zhenman Fang Examiner Assistant Professor, Engineering Science

Abstract

Shared Last-level caches are increasingly facing severe security risks from occupancy attacks and set-conflict-based side-channel attacks, e.g., Prime+Probe. Attackers use unrestricted cache occupacy, or use conflicts in limited-size cache sets, to observe access patterns of a victim process which can leak a victim's secret data. To eliminate shared LLC attacks, an ideal solution is to use a partitioned fully-associative cache design with random replacement so attackers cannot observe a victim's access patterns. Prior work proposed mechanisms that approximate such design at non-trivial power, area, performance and complexity costs.

In this work, we propose a practical INdirect, parTitionEd, Random, Fully-Associative CachE (INTERFACE) design which consists of a fully-associative data store and a skewed set-associative tag store. Each set in the primary tag store is linked to two sets, one from each extra (secondary) tag store. Each entry in the fully-associative data store is indexed by a valid entry from the tag store. We use a novel architecture to manage free data blocks without modifying the data store. We isolate processes by partitioning the cache to prevent occupancy attacks. Compared to prior work, we show that INTERFACE provides strong security guarantees by eliminating occupancy and conflict-based attacks with lower area and power overheads, lower complexity, and with a similar performance overhead compared to prior work.

Keywords: Last Level Caches; Side-Channel attacks; Occupancy Attacks; Conflict-based attacks; Fully Associative Cache

Acknowledgements

I would like to thank my Supervisor Prof. Alaa Alameldeen, my lab-mates, and everyone who supported this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Modern datacenter and client computing systems share execution and memory resources across many concurrent processes. Shared resources enable cost-effective designs that maximize performance without significantly increasing cost. Unfortunately, shared resources are subject to security vulnerabilities if malicious processes share them with victim processes that have sensitive data. Timing-based side-channel attacks were shown to leak victim secret data [1, 2, 3, 4]. These attacks rely on detecting victim-induced state changes in shared structures (e.g., caches). If such changes are correlated with victim secret data, the pattern of state changes can be exploited to leak secrets. Due to these vulnerabilities, security has become a primary consideration in designing shared execution resources of computing systems.

Shared last-level caches (LLCs) are a main target of high-bandwidth side-channel attacks due to their prevalence and well-understood use. Caches are usually organized in a hierarchy of levels where the last level is shared among multiple cores. LLCs are set-associative caches, where each set is shared by addresses that have the same hashed index value. A direct-mapped cache has a low access latency at the expense of high miss rate due to conflicts since each set includes only one cache block. A fully-associative cache uses the whole cache as a single set, eliminating all conflict misses with a higher hit latency and design complexity. A set-associative LLC is a compromise between the two extremes.

LLC side-channel attacks correlate cache state changes with secret data accesses by a victim process. Among these attacks, conflict-based attacks, e.g., Prime+Probe[5] enable high-bandwidth channels to leak secret data. The main step in conflict-based attacks is detecting addresses that are mapped to the same set in a set-associative cache, i.e., an *eviction set*. An attacker can fill a cache set with its own addresses, and track miss delays for accesses that were evicted by victim accesses. This enables the attacker to detect victim access patterns and associate them with the victim's private data such as passwords and encryption keys. Extending Prime+Probe to the cache level, occupancy attacks are con-

ducted when an attacker process fills the whole cache, then measures the timing delay due to victim accesses [4, 6, 7, 8].

LLCs are vulnerable to conflict-based side-channel attacks due to a combination of two factors. First, static address-to-set mapping is used to simplify cache accesses and reduce access latency. Unfortunately, static mapping enables easy discovery of eviction sets by an attacker. Second, cache sharing (allowing co-location of victim and attacker threads in the same set) enables better overall performance and improved quality of service via dynamic resource utilization. Unfortunately, cache sharing facilitates attacks such as Prime+Probe. Combined with unrestricted cache occupancy per process, cache sharing facilitates occupancy attacks. A secure cache design needs to eliminate one or both of the above factors with minimal performance overhead.

To address static address-to-set mapping of set-associative designs, prior works proposed using a randomized design where addresses are mapped to sets using a dynamic hash function [9, 10, 11, 12]. Such designs have low performance overheads but are still limited by the small size of eviction sets which necessitates frequent changes of the address-to-set mapping function at a high cost. Furthermore, these mechanisms leave the door open to attacks as shown in [13] where an attacker can accumulate information across multiple remapping epochs. To address the cache sharing vulnerability, prior works have proposed using partitioned set-associative designs [14, 15, 16, 17]. A similar approach has been adopted by Intel's Cache Allocation Technology (CAT) [18]. Partitioning provides a strong security guarantee by eliminating resource contention among different processes, but still suffers from attacks such as Spectre V1 [19] where conflicts are caused by different functions within the same thread, or by threads in the same process that share the same virtual address space.

One approach to eliminate conflict-based attacks is using a fully-associative cache design with random replacement, e.g., NewCache [20]. By using the whole cache as a single set, a Prime+Probe attack would need to fill up the whole cache which is impractical for large caches. With random replacement, an attacker cannot determine which victim address caused an eviction. Unfortunately, such a fully-associative design is impractical due to the high power, latency, and area cost of performing an associative tag match.

A recent proposal, MIRAGE, by Saileshwar and Qureshi [21] addresses the practicality aspect of a fully-associative cache by leveraging the V-way cache design [22]. MIRAGE uses a skewed-associative tag store where each set has extra tags that store the metadata of a newly-installed blocks with a low probability of causing a set-associative eviction (SAE). Each tag store entry includes a forward pointer to a location in the data store to enable fast cache lookup. Each cache block in the data store includes a reverse pointer to the current tag store entry that maps to it. Reverse pointers are used to enable a global eviction policy where a random block can be selected for eviction among all data store blocks, and the associated tag store entry can be invalidated.

While MIRAGE presents a major improvement towards a practical and secure fully-associative cache design, it does not prevent occupancy attacks and incurs nontrivial costs. MIRAGE requires custom data store designs as it needs to store reverse pointers with each data block. MIRAGE adds a hardware free list of pointers to non-allocated data blocks. Both components increase design complexity as they require significant effort to test and validate. In addition, avoiding all SAEs requires adding 75% extra tags in the tag store, where each entry is significantly larger than a set-associative cache tag due to additional address tag and pointer bits.

In this thesis, we propose a practical **in**direct, par**t**ition**e**d, **r**andom **f**ully-**a**ssociative **c**ach**e** design (**INTERFACE**) that addresses the shortcomings of prior work. Our proposal is inspired by the indirect-index cache [23] which uses indirection and chaining in the tag store to locate blocks in the data store. INTERFACE is organized as a skewed set-partitioned set-associative tag store and a fully-associative data store. Each set in the base (primary) tag store is chained into two sets, one from each extra (secondary) tag store. Each tag store entry contains status bits, tag bits and an index pointer into the fully-associative data store. A cache lookup uses six hash functions of the block address to access two primary skews and four secondary skews in parallel. We use an encrypted set-associative tag store to reduce tag comparisons, latency and power overheads per access. To eliminate cross-process set conflict and occupancy attacks, we modify the cache indexing function to partition the LLC among processes. To eliminate intra-process set-conflict-based attacks, we use a global eviction policy to avoid exploitable SAEs. We propose a novel, lower-complexity architecture *using an unmodified data store with no reverse pointers or hardware free list.*

To eliminate MIRAGE's reverse pointers, we use the observation that each data store block is pointed to by a single valid tag store entry. Therefore, random replacement can randomly select a valid tag and evict its corresponding data block. To eliminate the extra hardware free list space, we propose a novel in-situ free list implementation in the data store that only requires two extra registers with no additional storage. We make the following main contributions:

- We propose INTERFACE, a practical fully-associative cache design that *eliminates all set-associative evictions (SAEs) and prevents occupancy attacks* while using unmodified data arrays (Chapter 3.1).
- We propose a novel free list implementation that enables data allocation in a fully-associative cache without requiring a new hardware structure.
- We show that INTERFACE eliminates all set-associative evictions (SAE) at a lower area overhead compared to prior work (Chapter 4.1).
- We incorporate static and dynamic cache partitioning in INTERFACE to prevent cross-core occupancy attacks.

- Our evaluation shows that INTERFACE has 26.2% smaller tag store area/leakage, 7% smaller data store area/leakage, and 10.4% smaller total cache area/storage in a 16MB LLC at comparable performance overhead while providing stronger security guarantees compared to the state of the art (Chapter 5.1).

Chapter 2

Background

Cache side-channel attacks [5, 3, 2, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 4, 6, 7, 8] are major sources of micro-architectural security vulnerabilities in modern computer systems. These attacks use changes in the shared cache state (typically the Last-Level Cache, i.e., LLC) to track the access pattern of a victim process and steal victim secrets. Shared LLC attacks belong to two main categories: Conflict-based and Occupancy-based attacks.

2.1 Conflict-based attacks

Conflict-based attacks occur when an attacker induces conflict evictions of a victim process's cache blocks to track the victim's access pattern. A key component in conflict-based cache side-channel attacks is identifying a collection of addresses that map to a particular set, called an eviction set. Once an eviction set is discovered, an attacker can orchestrate conflict-based attacks e.g., Prime+Probe. Miss delays (upon requests to the eviction set blocks) are then used to infer data accesses in the victim's address space. LLCs are mainly targeted by such attacks due to their large, shared and set-associative design, which enables the use of low-resolution timers for fast eviction set discovery algorithms to construct an eviction set, e.g., [24, 34].

2.2 Occupancy-based attacks

Occupancy-based attacks are carried out by filling the LLC with a buffer size equal to the cache capacity [4, 6, 7, 8]. The attacker then builds a trace by periodically gathering timing delay samples of re-accessing (sweeping) the buffer. The attacker lastly correlates the trace with the co-located victim process' activity, e.g., identifying visited websites. State-of-the-art occupancy attacks are cross-process. For instance in Website Fingerprinting Attacks[6], the attacker and victim are different processes running on multiple browser tabs. For the rest of the thesis, we refer to cross-process occupancy attacks as *occupancy attacks*.

2.3 Mitigation Strategies

Two main approaches have been proposed to mitigate shared LLC vulnerabilities[35]: randomized caches and cache partitioning[36]. Other replacement policy-based approaches [37, 38] detect suspicious cache accesses and alter their replacement policy to thwart attackers from completing an attack. These approaches require re-learning access patterns for each emerging attack and updating the pattern detection logic.

2.3.1 Partitioned Caches

Way-based cache partitioning[18, 39, 14, 16, 40, 41], and set-based cache partitioning[42, 43, 15, 9, 17] eliminate conflicts and occupancy attacks by creating isolation in the cache on a per-process basis. DAWG [16], defines protection domains to provide full, way based, isolation among processes with a minimal modification to the OS. Catalyst [40], modifies Intel CAT [18] to implement a hardware-software managed, way partitioned LLC. Chuncked Cache [15], introduced a set partitioning mechanism where applications that require secure contexts carve out a private section of the cache, and all other applications utilize the remaining part of the cache. However, these mechanisms are still vulnerable to intra-process conflict-based attacks, where the attacker shares the same virtual address space with the victim, as in Spectre V1 [19], require the user or OS to specify each process' cache requirements, or limit the number of active threads using the cache.

2.3.2 Randomized Caches

Randomized Caches [9, 44, 10, 11] defend against conflict-based attacks by making the address-to-set mapping principally random, thereby reducing the probability of successfully forming an eviction set that has a high eviction success rate. *However, these mechanisms do not defend against occupancy attacks.* Many implementations of randomized caches have been developed in parallel with the development of state-of-the-art eviction set formation attack algorithms.

NewCache[9] introduced randomized cache designs by using a table indexed by the set bits of the address, where each entry contains a string of bits that map an address to a specific set. CEASER[10] proposed an encryption-based principled randomized mapping of addresses to cache-sets, where the encryption key of the hash function is refreshed after a given number of cache accesses. By using hash functions, CEASER significantly reduced latency, area and power overhead of mapping tables used in NewCache. It was proven to be effective against algorithms[24] which use $O(n^2)$ accesses to form an eviction set. Unfortunately, newer algorithms [11, 45] which form an eviction set with $O(n)$ accesses were able to exploit CEASER. Song and Liu[34] further reduced the number of required accesses for $O(n)$ algorithms by a constant factor with an attack model that uses multi-threaded

execution. CEASER fails to practically defend against such attacks since it would need to significantly increase the re-randomization frequency, which greatly degrades performance.

By extending CEASER, Skewed-CEASAR[11] is based on skewed-associative caches [46], where each skew uses a distinct encryption key and is chosen randomly among other skews for each cache access. Using an analysis independent of search algorithms and replacement policies, Skewed-CEASAR reported a vulnerability of 1ms over 18 years for a cache with a remap-rate of 1% and where the attacker focuses on a 1MB cache. Even though Skewed-CEASAR provided protection against O(n) set discovery algorithms, it is still vulnerable to more efficient algorithms[47] that construct high confidence probabilistic eviction sets in less than O(n) accesses. Similarly, ScatterCache[44] used skewed caches, where each skew has its own hash function. On cache accesses, the entries in the mapping table are swapped every time one process evicts a cache line from another process. ScatterCache addressed attack algorithms with O(n) complexity. However, it incurred large performance overheads due to the frequency of randomization needed to defend against probabilistic set-conflict-based attacks [47, 48]. Chameleon Cache[49] couples random, skewed caches(RSCs) with a fully associative victim cache to immediately re-locate evicted line from one mapping to another, thereby obfuscating set associative evictions. While Chameleon Cache efficiently boosts RSCs security, its security guarantee is significantly weaker than MIRAGE and INTERFACE. Chameleon Cache evaluated the success rate of discovering an eviction set with PRIME+PRUNE+PROBE and randomly selected addresses. For a cache with 16384 cache-lines, 16 ways and 8 victim cache entries, the success rate was reported to be 0.05.

2.3.3 Randomized Cache: Issues

CaSA [13] demonstrated that set-associative-based randomization defenses make two incorrect assumptions. First, randomized caches assume successful attacks need eviction sets with high eviction rate CaSA demonstrated that attackers can exploit eviction sets with low eviction rates. Second, randomized caches assume an attack must conclude within the lifetime of a given hash function. CaSA demonstrated that cache state changes accumulated across epochs of hash function updates can increase the success rate of an attack. Brutus [50] shows that the low latency block cipher used in CEASER only consists of linear functions. Hence, dynamic re-randomization based defences [9, 44, 10, 11] fail to principally address the root cause of conflict-based attacks.

2.3.4 Pseudo-Fully Associative Caches

Several prior works have tried to address conflict-based cache side-channel attacks by introducing restricted levels of full associativity to the cache design. Phantom-Cache [12] randomly inserts a new cache block into one of eight candidate sets, hence increasing the effective associativity by 8x compared to the baseline associativity at a high dynamic power overhead due to performing 128 comparisons for each cache access. HybCache [41] provides

full associativity for a subset of the cache. Hence, processes which map their cache blocks to this subset can avoid SAEs. However, HybCache is expensive to apply to the LLC due to the large LLC size.

MIRAGE [21] recognized that conflict-based attacks stem from the set-associative design of shared LLCs, and proposed a pseudo-fully associative cache based on the V-way cache [22]. MIRAGE's set-organized tag store is detached from the fully-associative data store, and indirection is used to associate a tag store entry with a corresponding data store entry. MIRAGE uses extra tag entries per tag set, a 2-skewed tag store and an adaptive hash function selection mechanism to reduce the probability of SAEs. MIRAGE principally addresses the root cause of conflict-based attacks with modest performance overhead but a high area and complexity cost due to (1) the need to add 75% extra tags in the tag store, where each tag entry is significantly larger than a set-associative cache tag; (2) using a separate hardware linked list structure whose size grows proportionally to the number of cache data blocks to help with data allocation; (3) adding a reverse pointer in each data store block to a tag store entry to use in global evictions. Reverse pointers increase the complexity of cache line updates since it requires custom data arrays to maintain points and cannot use off-the-shelf data arrays from existing pointers. Custom hardware needs extra design, testing and validation efforts which hinder their adoption. Reverse pointers also need strong error correction since an error can invalidate the wrong tag pointing to a different data block which becomes an orphan, not pointed to by any tag, causing lower available cache capacity over time. MIRAGE uses a ball and bucket model with random accesses, and formulates an analytical model to show the probability of success of discovering an SAE is 10^{-35}.

In this work, we propose INTERFACE, a practical set-partitioned fully-associative LLC design that addresses inter-process and intra-process conflict attacks in addition to occupancy attacks. Our threat model assumes a multi-core CPU with private L1/L2 caches and a shared LLC, where attackers and victim processes can run on the same or different cores, and attackers use the shared LLC to leak victim data. INTERFACE addresses the root cause of both conflict and occupancy attacks (Chapter 3.1) at a lower area, power, and complexity cost compared to prior works. The cache is sectioned into four skews of set-partitioned set-associative tag stores and a fully-associative data store. We use global eviction in the data store, hence the attacker cannot associate miss delays with a specific group of addresses mapping to a cache set. The tag store's set-associativity enables tag comparisons with a limited number of cache lines from a skew, reducing the cost of checking tags of all data entries in a traditional fully-associative cache. The extra tag skews increase effective ways per set, avoiding SAEs. The set-partitioned tag store provides isolation for inter-core conflict and occupancy attacks. Our evaluation uses ball-and-bucket analytical modeling (Chapter 4.1), a cycle-accurate simulator and area/power analysis to model the functionality and overhead of our design (Chapter 5.1). Table 2.1 summarizes INTERFACE's better

security guarantees and lower attack success probability (in parenthesis) compared to prior works.

Table 2.1: Security Coverage and Probability of Attack Success of Different Mitigation Approaches

Mitigation	**Set-conflict attacks**		**Cross--process occupancy attacks**
	Cross-process	**Intra-process**	
Randomized Caches[10, 44]	✓	✓	
MIRAGE[21]	✓(10^{-35})	✓(10^{-35})	
CEASAR-S[11]	✓(10^{-12})	✓(10^{-12})	
Chameleon[49]	✓(0.04)	✓(0.04)	
Partitioned Caches [14, 42, 17, 16, 15, 40]	✓(0)		✓(0)
INTERFACE	✓(0)	✓(10^{-49})	✓(0)

Chapter 3

Architecture

3.1 INTERFACE Architecture

INTERFACE consists of a fully-associative data store and a set partitioned, skewed, set-associative tag store.

Cache Partitioning. The cache management unit statically partitions the cache into N domains (equal to number of physical cores) and sets up the partition register values (masking bits and partition offset) as shown in Figure 3.2. Each scheduled process is allocated a partition of the LLC based on its Core ID, Figure 3.3a. When a process finishes or a context switch occurs, its partition is flushed and allocated to a waiting process, e.g., Process A's partition from Figure 3.3a is allocated to process D (Figure 3.3b). In addition to mitigating cross-core conflict and occupancy attacks, this cache partitioning approach avoids adding process ID bits in the tag (as in [21]) to defend against shared memory attacks [3, 2].

Dynamic Partitioning. While static partitioning provides a strong security guarantee, its strict capacity could hurt memory intensive workloads. We implemented a dynamic partitioning defense mechanism that can be turned on for better performance at an acceptable leakage rate. When the OS turns on this mechanism, we set all the masking bits to **1** and offset values to **0**. We then start monitoring the occupancy level of each core via per-core counters by repurposing partition registers (Figure 3.2). If a core's cache occupancy exceeds a random bound (generated for each cache fill request) between 40-60% of the total cache capacity, we evict a random cache-line allocated to that process. If the current occupancy is below the random bound, we randomly evict from any process. We analyze the security implications in Section 4.1.4.

Tag store. Our tag store has 4 skews: 2 primary skews organized into sets of 8 ways (base ways, similar to a set-associative cache) and 2 additional secondary skews which have the same number of sets as the primary skews. Each set in the secondary skews has ways equal to the number of extra ways (Figure 3.1). Inspired by the Indirect-Index Cache [23], each set in the base tag store is linked to two sets, one from each secondary skew. Each tag store entry contains status bits, address tag bits, and indexing bits used for indirection into the

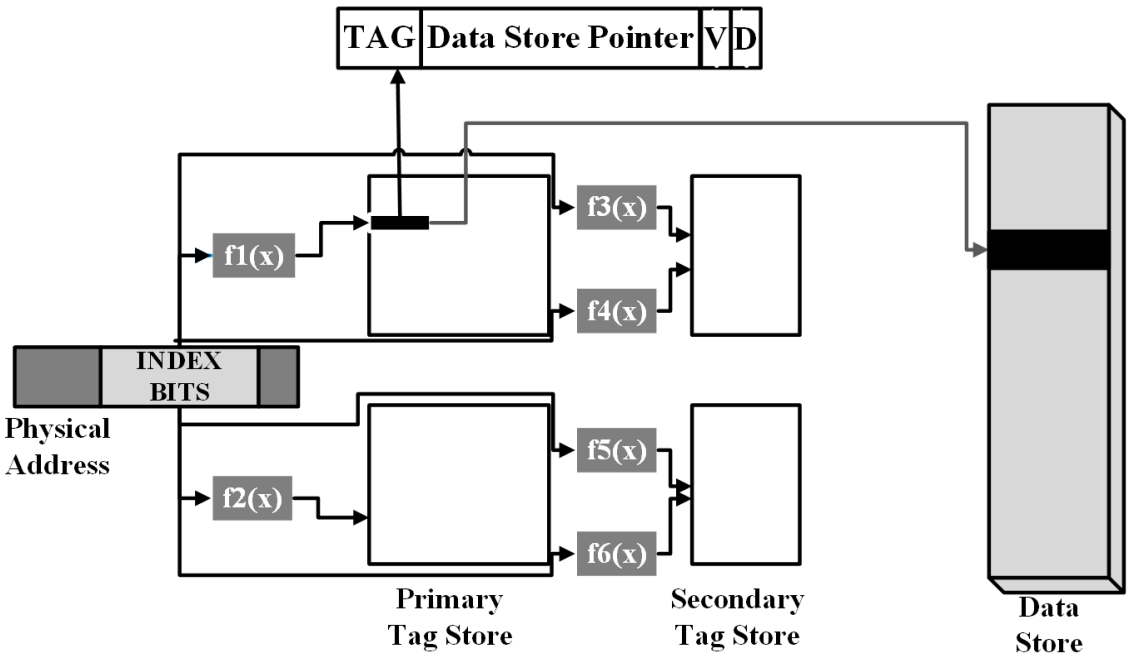


Figure 3.1: INTERFACE architecture.

data store (Figure 3.1). Each block in the fully-associative data store is indexed by an entry from the tag store. INTERFACE differs from prior skew-based approaches in 3 ways:

- Primary skews contain the base tag store if we didn't add extra ways, and secondary skews are extra ways added to ensure the availability of invalid tags for cache fill requests.
- Secondary skews are multi-ported where an input address is mapped to 2 sets per skew to facilitate sharing of the extra ways between 2 sets, while primary skews are single-ported.
- For cache fill requests, we first check the availability of an invalid tag in the primary skews before accessing the secondary skews, which reduces the dynamic power overhead of accessing secondary skew tags. Non-constant cache access latency for fill requests doesn't create an extra side channel because the overall fill latency is hidden by DRAM latency.

In addition to the tag store and data store, we have 6 cache indexing hash-functions [51] to index into the tag store, and 4 pseudo-random number generators(PRNGs) to support global random eviction in the data store (Figure 3.1). We describe the INTERFACE cache operations next.

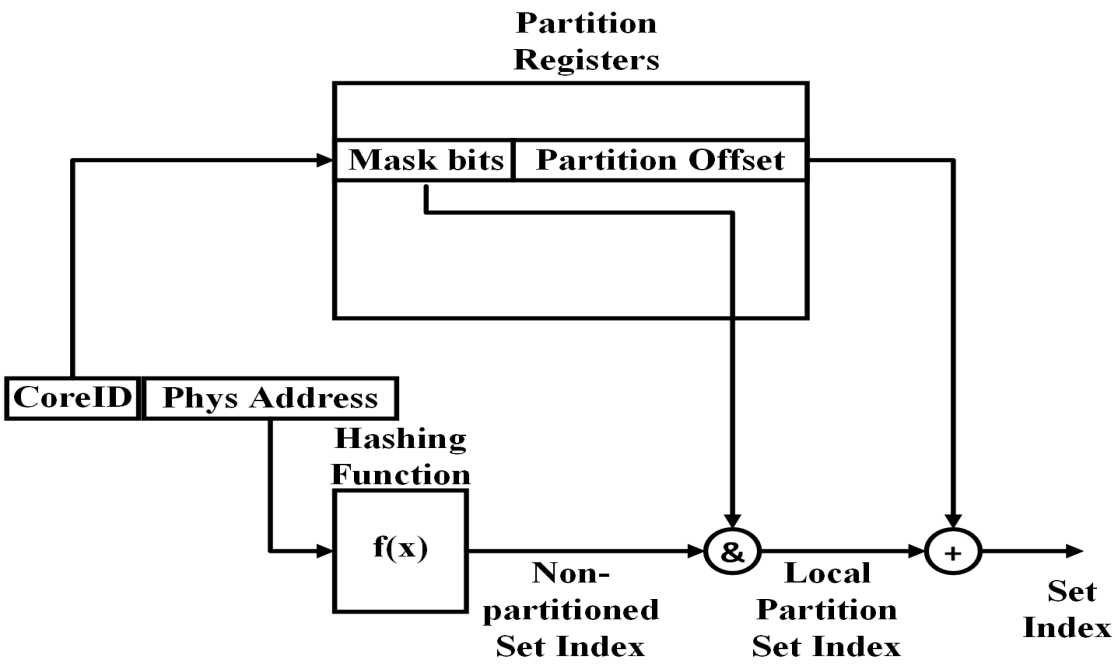


Figure 3.2: Indexing Logic for INTERFACE's Partitioned LLC.

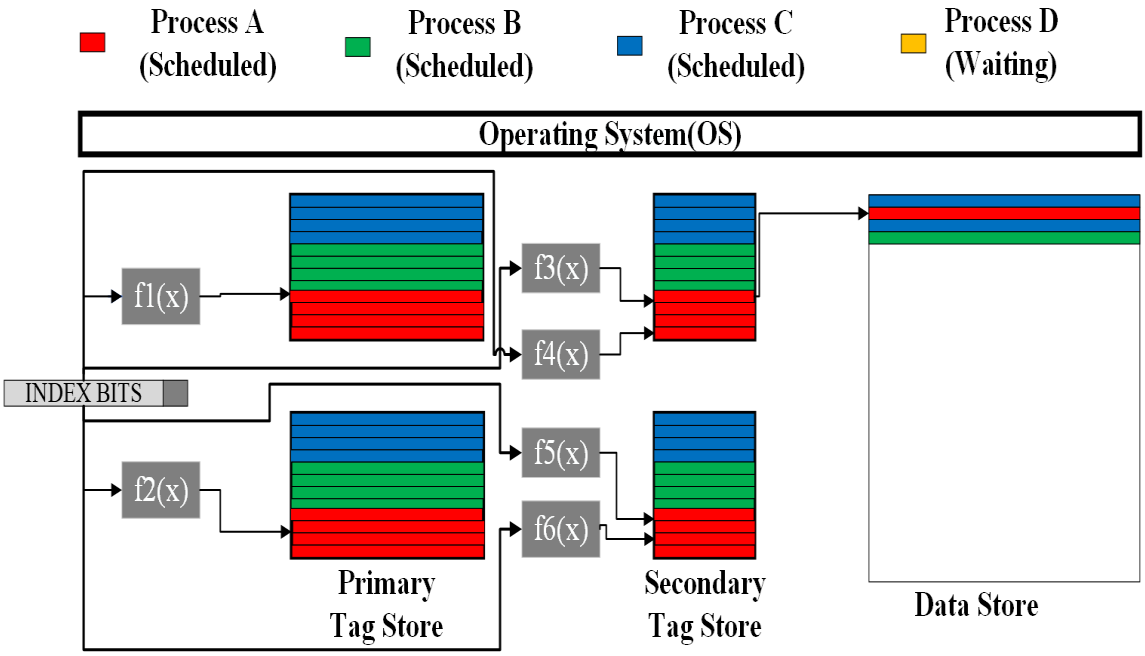
3.1.1 Cache Indexing

LLC cache access requests contain the physical address and physical Core ID of the requesting process (Figure 3.2). We concatenate Core ID with the address utilizing the unused most-significant bits of the physical address. We concurrently fetch a process' partition information (masking bits, partition offset), and compute the hash function on the physical address to get the non-partitioned set index. Finally, we apply the mask on the non-partitioned set index and add the starting index (partition offset) to get the partitioned set index.

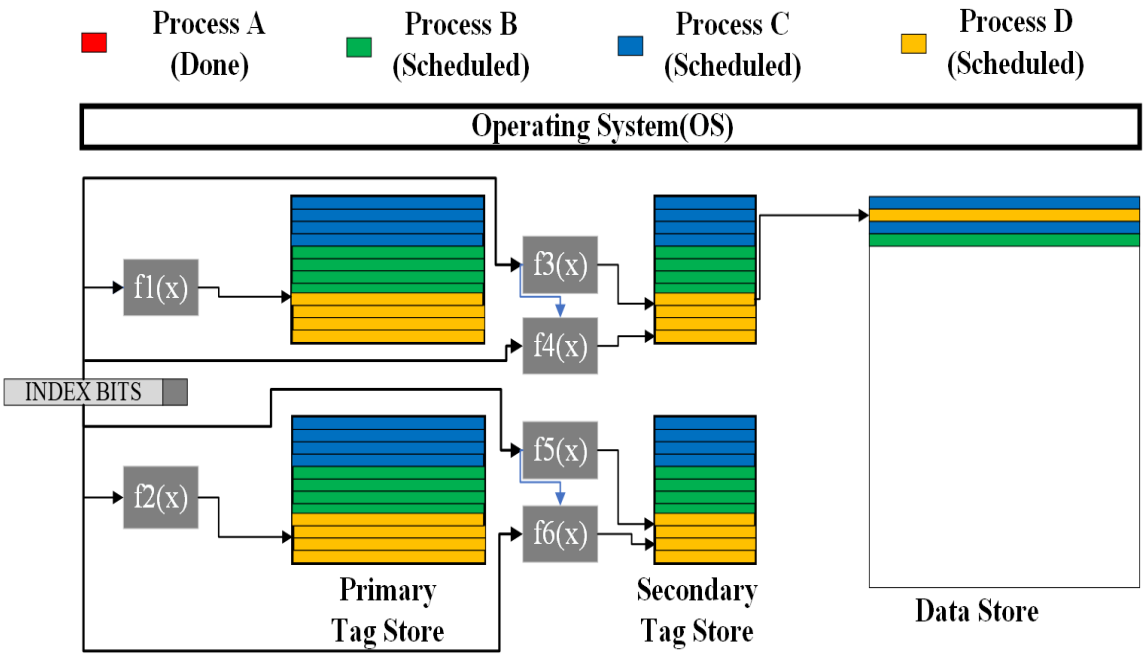
Static partitioning achieves complete process isolation for processes running on different cores, providing a strong guarantee against cross-core occupancy and conflict-based attacks. Within each partition, our pseudo-fully associative design prevents SAEs, which mitigates conflict-based LLC attacks such as Spectre V1 [19] where conflicts are caused by different functions within the same thread, or by threads in the same process that share the same virtual address space. Dynamic partitioning can improve performance with added vulnerability to occupancy attacks. We are not aware of occupancy-based attacks where the attacker and victim are running on the same process, so we leave such mitigations for future work.

3.1.2 Cache Access

When an LLC request arrives from higher cache levels, we use the indexing hash functions to access two sets (one from each primary skew), and compare the tag bits. On a tag match



(a)



(b)

Figure 3.3: Static set partitioning based on Core ID.

(i.e, hit), we use the data store pointer obtained in parallel with the tag store line's tag bits to serially index into the data store. If we cannot find a match in the primary skews, we index into two sets in each of the secondary skews and compare the tags in parallel. Following a tag hit in the secondary skews, we access the data store by using the corresponding data store indexing bits. A tag miss in both primary and secondary skews signals an LLC miss initiating a main memory request, and allocate a tag store entry and a data store block to write the data block loaded from memory.

This approach (inspired by the serial chaining method used in [23]) can lead to non-uniform hit latency where primary skew hits are faster than secondary skew hits. This could be a point of vulnerability that can be exploited by a timing attack [52, 53, 54, 55, 56]. To address variable hit latency, we access all primary and secondary skews in parallel, which leads to a modest increase in dynamic power for tag accesses.

3.1.3 Cache Fill

On an LLC miss, we need to allocate a tag store entry and a corresponding data store block. To allocate a tag store entry, we hash the cache fill request address and index into two primary skew sets, one from each skew. If at least one of the two sets is not full, i.e., has invalid tags, we use power-of-2 load balancing [57] to pick a candidate set for the fill line. If both sets have equal loads, but are not full, we randomly choose one of the two sets and allocate an invalid tag store entry to the fill line. If both primary skew sets are full, we use the secondary skew hash functions to access the secondary skews, and pick the set with the lowest occupancy, or pick a set randomly if all candidate sets have the same occupancy. After exhausting all the set choices for a line fill, if all possible sets are full, we have to carry out a SAE in the tag store. For a SAE, we randomly select a skew (primary + secondary), apply random eviction policy in the set/sets within the selected skew, and invalidate the data store block pointed to by the evicted tag store entry. However, our goal is to eliminate all SAEs to avoid all potential set-conflict-based attacks. Hence, we over-provision the number of extra ways in the secondary skews to ensure an extremely low probability of SAEs.

Once we have obtained an invalid tag store entry, we set the candidate tag store entry's valid bit and check if the data store is full. If not full, we allocate an empty data store block (explained in Chapter 3.1.4) and set the tag store entry's data pointer to the newly allocated data store block. However, if the data store is full, we need to carry out a random global eviction in the data store, invalidate the tag store entry associated with the evicted data store block (Chapter 3.1.5), and associate the evicted data store block with the allocated tag store entry by setting the tag entry's data pointer.

3.1.4 Data Store Allocation

A major challenge in managing a fully-associative data store is keeping track of free data blocks. A brute-force approach would maintain a free list of data store blocks using dedicated hardware storage of pointers, which incurs a high complexity and cost for managing a hardware free list. While not discussed in the MIRAGE paper [21], the gem5 implementation artifact uses a hardware pointer-based queue where pointers to all data store blocks are inserted initially, then removed one by one as data blocks are allocated. In addition to the hardware complexity, this hardware free list uses an extra 256K (blocks) x 18b (pointer bits per block) = 576KB extra storage in a 16MB LLC. INTERFACE introduces a novel hardware mechanism that utilizes an unmodified data store with two additional hardware registers and simple logic to maintain a free list of data store blocks. We use a LIFO stack of free blocks instead of a FIFO list. This avoids the complexity and overhead of adding a dedicated storage component in the cache system. In addition, since a data block's capacity is much larger than the size of the data store indexing (i.e., pointer) bits, we can use the extra space to store three redundant indexes, and use a simple majority vote circuit to eliminate errors in free list pointers.

We classify free data store blocks into two groups. Group 1 consists of data store blocks which were previously allocated but are currently free due to invalidations (potentially triggered by the cache coherence protocol). Group 2 consists of data store blocks which are free because they are yet to be allocated by a cache fill request. When the system boots up, all data store blocks are in Group 2. Our mechanism uses two registers: A *Free-Index Register* that points to the tail of the free list and a *Counter Register* to track the number of allocated data store blocks. The Counter Register is useful after system boot to keep track of Group 2 blocks. After all Group 2 blocks are allocated at least once, the Counter Register remains the same until the next system boot.

Initially, both the Counter and Free-Index registers point to the first block (block 0) of the data store (Figure 3.4a). On a data block allocation request, if Counter and Free-Index registers point to the same block, we allocate that block and increment (by one) both the Free-Index register and Counter register to point to the next consecutive free block (Figure 3.4b). Successive allocation requests will allocate consecutive blocks when Free-Index and Counter registers point to the same free block as in Figure 3.5a.

When a data block is invalidated (e.g., block 6 in Figure 3.5a), it becomes a Group 1 free block. To add an invalidated block to the free list: (1) the current Free-Index register's value (9), i.e., current tail, is written to the invalidated data block (Figure 3.5b), which could be done redundantly by writing multiple copies of the value for pointer error correction); and (2) the invalidated data block's index (6), is written to the Free-Index register and becomes the new tail of the free list (Figure 3.5b). Subsequent invalidations result in more insertions

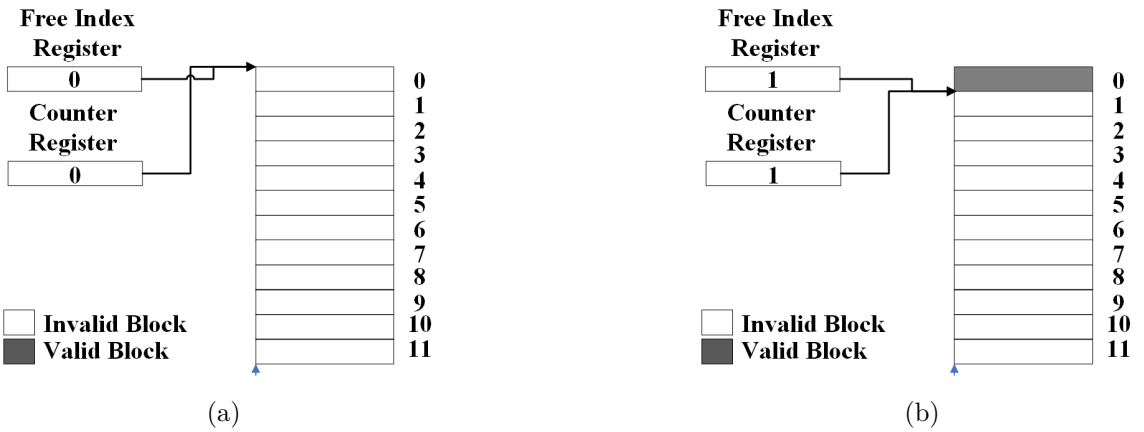


Figure 3.4: Allocation in a Data store (size=12 Blocks) when Free-Index and Counter registers point to the same block.

at the tail of the free list as shown in Figure 3.6a. The Counter register remains constant while the Free-Index register is updated to the latest invalidated block.

Figure 3.5: Free-list insertion due to coherence invalidation.

When an invalidation occurs after boot while some blocks haven't been allocated for the first time, the Free-Index and Counter registers will not point to the same free data block (Figure 3.6a). In such case, data block allocation requests are serviced by only using the Free-Index register. Upon an allocation request, we fetch the data block pointed to by the Free-Index register (block 3), write its content, i.e., the next free block (block 6) pointer, to the Free-Index register as the new free list tail, and release the data block for a cache fill, as shown in Figures 3.6a and 3.6b.

For an incoming allocation request:

- If the Free Index register and Counter register contain different values, we read the value stored in the data block pointed to by the Free Index register and write it into the Free Index register

- If the Free Index register and Counter register contain the same value, each is incremented by 1 to indicate the block ID of the next free data store entry.

Hence, we prioritize allocating invalidated free blocks (Group 1) followed by blocks that are yet to be allocated (Group 2) by using either compare & increment or compare & store operations without linear search (Figure 3.4b).

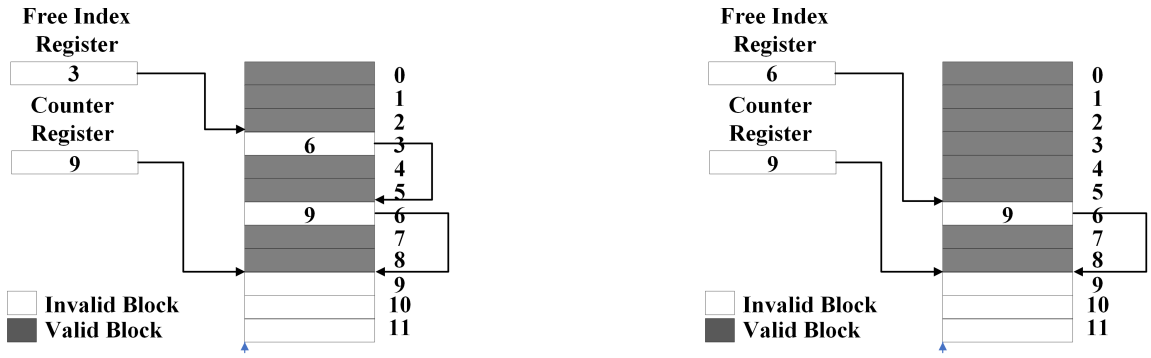


Figure 3.6: Data store allocation when Free-Index and Counter registers point to different free blocks.

If the Counter register points beyond the data store limit, then no more Group 2 blocks exist since all data blocks have been previously allocated. If both Free-Index and Counter registers point beyond the data store limit, we infer that the data store is full, and initiate a Global Random Eviction.

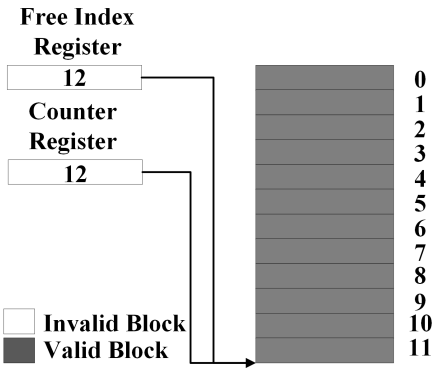


Figure 3.7: Data store allocation with Global Random Eviction when Data Store is full.

3.1.5 Global Random Eviction

Our condition for invoking Global Eviction is when the occupancy level of a process exceeds its allocated capacity limit and/or when Free-Index and Counter registers point beyond the data store limit (our cache is at full capacity). When one or both conditions are true (Figure

3.7), we need to evict a valid data block. MIRAGE uses a hardware PRNG to randomly pick a victim block from the data store, and utilizes the reverse tag store pointer to invalidate the tag store entry associated with the evicted data block. INTERFACE, however, does not use reverse tag store pointers to simplify design and avoid problems discussed in Chapter 2. We rely on the observation that each data store block is pointed to by *one and only one* valid tag in the tag store. To evict a random data store block, we can just randomly select a valid tag from the tag store. In our secure design discussed in Chapter 4.1, we use 50% extra tags so a third of all tags are invalid when the data store is full. This implies that the probability of finding a valid tag after one randomly generated index is 2/3. If the selected tag is invalid, we need to try again by generating an additional random tag until we hit a valid tag. The probability of not finding a valid tag after N lookups is $(1/3)^N$. To avoid incurring a long latency to identify a valid tag sequentially, we use 4 hardware PRNGs in parallel to generate indices of four victim tag store entries. If at least one of the victim tag entries is valid, we invalidate the tag store entry and evict the data store block pointed to by the invalidated tag store entry. If more than one victim tag entries are valid, we randomly choose an entry. Due to the extra tags in INTERFACE, the generated random indices might all reference invalid tag entries.

With 4 hardware PRNGs the probability of not finding a valid tag is 1/81. We handle this rare case by generating four more indices until we find a valid victim tag entry. For dynamic partitioning, If the occupancy level of a process is above the randomly generated bound, we need to evict a valid block with the same process ID. On average the occupancy bound of a process is 50%, hence the probability of not finding a valid tag with the given process ID is 1/40.5. Because we only need to access the tag store, the access latency for the search algorithm is smaller than the hit latency. The global random eviction tag store accesses also occur in the background while memory is servicing the LLC miss, thus additional access latency and variability are hidden from an attacker. In the rare case an attacker observes the delay caused by our random search approach; this delay is associated neither with the inserted address nor the evicted cache-line, hence the attacker cannot infer any secrets.

3.1.6 Comparison with MIRAGE

INTERFACE architecturally differs from MIRAGE in numerous ways. Most notably, INTERFACE is set-partitioned with an optional dynamic partitioning mechanism while MIRAGE does not have any partitioning feature. Secondly, we add the extra-ways in separate skews from the base(primary skews), use different keys to access the extra-ways, and multi-port the secondary skew, hence adding more randomness in the address-to-set mapping, while MIRAGE adds the extra-ways in the primary skews and uses the same keys to access the extra-ways. Thirdly, INTERFACE manages cache evictions from the tag store by randomly selecting an entry in the tag store and invalidating the associated data store entry.

This approach enables us use an un-modified data array and to easily implement set partitioning because we can limit our random search algorithm to only the target partition. However, MIRAGE requires a modified data array because the eviction policy randomly selects a data store entry to evict and uses the reverse pointer to invalidate the associated tag store entry. If we want to directly apply the our set partitioning approach to MIRAGE, we would suffer from longer latency to locate an appropriate data store entry to evict from a given partition because MIRAGE's search algorithm is not limited to the target partition. Lastly, INTERFACE utilized 2 registers and the data store to implement a LIFO free-list to manage the allocated and free data blocks. However, MIRAGE uses a dedicated hardware free-list which introduces additional power and storage overheads. In general, by using more skews(encryption keys), multi-porting our extra-tags stores and, removing the reverse pointers and dedicated hardware free-list, we provide a less complex and more secure pseudo-fully associative cache.

Table 3.1: INTERFACE vs MIRAGE Comparison

MIRAGE	**INTERFACE**
Merged extra-ways	Separated extra-ways
No multi-ported secondary skews	Multi-ported secondary skews
Reverse Data store pointer	No Reverse Data store pointer
Dedicated Hardware Free-list	2 registers and data store Free-list
Data-store managed global eviction	Tag-store managed global eviction

Chapter 4

Security Evaluation

4.1 Security Analysis

INTERFACE has a set-associative tag store, which leaves the door open for security vulnerabilities caused by set-associative evictions (SAEs). To eliminate SAEs, INTERFACE uses two primary skews and over-provisions two secondary skews to ensure the availability of invalid tags for cache fill requests. Additionally, INTERFACE is secure without domain-specific keys and dynamic remapping because the cache lines filled by the attacker can be randomly evicted anytime, hence providing dynamic randomness through the global eviction policy. The next sub-chapter explains the model we used to evaluate the number of additional tags needed to eliminate all SAEs.

4.1.1 Ball and Bucket Model

Our ball and bucket model corresponds to a 16MB cache. A bucket represents a set in the tag store which requires protection against SAEs. We have 4 sets of buckets, each representing a skew. The first two set of buckets, *primary skew buckets*, represent the primary skews. Each primary skew is organized into 16K buckets with a capacity of 8 to represent the number of ways. The other two sets, *secondary skew buckets*, represent the secondary skews, and each skew is organized into 16K buckets with capacity that varies in the range [0,8], equivalent to the extra ways. To model data store capacity, the total number of balls in all buckets cannot exceed 256K (since a 16MB cache has 256K 64B lines). We throw balls to represent cache line fill requests. We also represent our global eviction policy by randomly removing balls from buckets once the total number of balls in all buckets exceeds the data store capacity. We run Monte Carlo simulations with billions of ball insertion/eviction events. The model is initialized by throwing 256K balls. After initialization, newly inserted balls trigger an eviction of a randomly-selected ball to keep the total number of balls the same as the data store capacity. In each iteration, our model first randomly picks a ball from the buckets and evicts it. Next, we generate a random index for each primary skew and access the buckets with the respective index from each skew to make an observation about

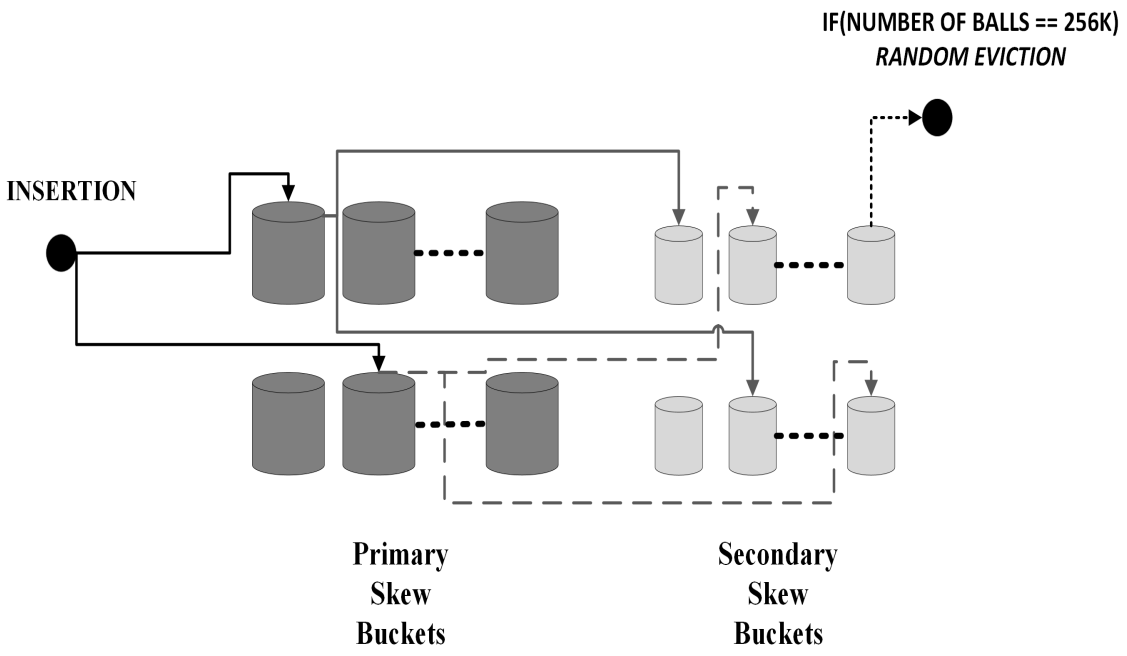


Figure 4.1: Ball and bucket model.

their current occupancy. If either or both of buckets are not full, we insert the ball in the bucket with the lower occupancy; selecting randomly between the skews in case of a tie. However, if both primary buckets are full, we generate 4 random indices (two per skew) for the secondary skew buckets. If at least one of the indexed secondary skew buckets is not full, we insert the ball in the bucket with the lowest current occupancy. However, if all indexed buckets are full, we increment the spill counter to indicate the occurrence of a SAE. In our architecture, we access primary and secondary skews in parallel (to reduce the latency of secondary skew hits) but give preference to primary skews if they are not full. Sequential accesses in our Monte Carlo model (primary then secondary) provides the same functionality.

To estimate the number of additional tags we need in INTERFACE, we varied the number of extra tag store entries per bucket in the secondary skews from 1 to 6 and threw 10 billion balls per experiment. We computed the average number of ball insertions (i.e., cache accesses) per spill (i.e., SAE), and observed the probability distribution of having a certain number of balls in a bucket to validate our probabilistic model presented in the next Chapter. We also reproduced the results from MIRAGE [21] (which represents the only practical fully-associative design we are aware of) to compare the two mechanisms.

Figure 4.2 shows the average number of insertions per spill (i.e., SAE) for INTERFACE and MIRAGE. Using INTERFACE with one and two extra ways per secondary skew, we observed 8 and 5000 average insertions per spill, but didn't observe any spills for extra

ways greater than two across five Monte Carlo simulations with 10 billion insertions each. For MIRAGE, we observed spills for up to 4 extra ways. The lower probability of SAEs in INTERFACE is explained by its more efficient use of extra ways. For each insertion in INTERFACE, the total number of candidate locations is the sum of all locations in two primary skews and 4 secondary skews. For MIRAGE, however, the total number of candidate locations is the sum of all locations in the two primary skews and two secondary skews. Therefore, INTERFACE provides more potential candidates to locate a free bucket space (i.e., an invalid tag) which reduces the need for additional tags. To provide a robust,

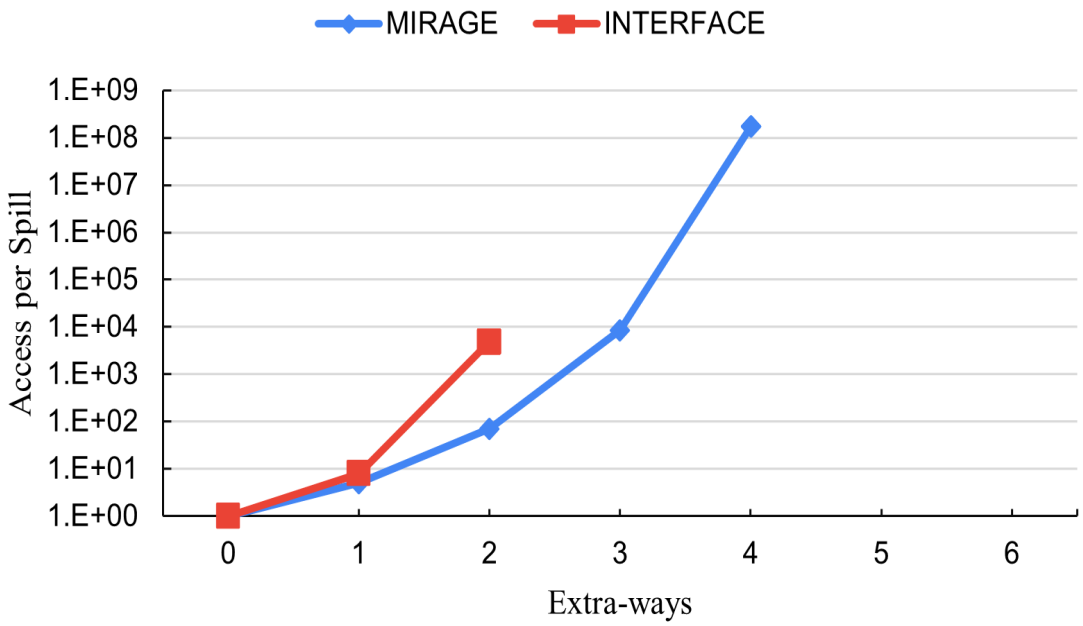


Figure 4.2: The observed average number of balls inserted per spill vs. #extra ways for MIRAGE and INTERFACE.

quantitative security guarantee that extends beyond 10 billion accesses, we will next develop an analytical model to estimate the average number of cache access per spill.

4.1.2 Analytical Model

The ball and bucket model's ability to provide quantitative security guarantee beyond three extra ways is limited by the number of balls we can throw within a reasonable simulation time. To scale our results, we define an analytical model based on the observed probability distribution of the number of balls in a bucket. Our analytical model is based on the insertion and eviction policies of the INTERFACE cache, which depend on whether the primary skews are full. We consider two cases: One where there are available space in the primary skews, and another where the primary skews are full.

Case 1: Either primary skew buckets below full occupancy. When a ball is thrown, we randomly index 2 buckets, one from each primary skew. In INTERFACE, this corresponds to hashing into primary skews. If one (or both) buckets is not full, we consider scenarios where a bucket in a primary skew transforms from containing N balls to N + 1 balls: (1) If both buckets contain N balls; or (2) if one bucket contains N balls while the other contains more than N balls. The probability of an N to N+1 transition is therefore computed as:

$$P(N \rightarrow N+1) = P(n = N)^2 + 2 * P(n = N) * P(n > N) \quad (1)$$

Where P($n = N$), is the probability that a bucket contains N balls and P($n > N$) is the probability that a bucket contains more than N balls.

When a ball is evicted, we pick a random ball from one of the two primary or two secondary skews. Given a ball is evicted from a primary skew, the probability that a bucket in the primary skew transforms from containing N + 1 balls to N balls is given as:

$$P(N+1 \rightarrow N) = \frac{P(n = N+1) * B_{total} * (N+1)}{b_{total}} \quad (2)$$

Where B_{total} is the total number of buckets, and b_{total} is the total number of balls that can exist in our model at any time.

Our insertion and eviction policies are random in this model are random The INTERFACE architecture uses hash functions (not random) to select an insertion location. But this is equivalent to a random insertion in the probabilistic model, hence the probability of a bucket transitioning from N to N+1 balls is the same as the probability of transitioning from N+1 to N balls:

$$P(N+1 \rightarrow N) = P(N \rightarrow N+1) \quad (3)$$

Substituting (1) and (2) in (3) and taking B_{total}=32K, b_{total}=256K to model our 16MB cache, we get:

$$P(n = N+1) = \frac{8}{(N+1)} * (P(n = N)^2 + 2 * P(n = N) * P(n > N)) \quad (4)$$

To compute the occupancy of primary skews, we use P(n = 0) that we observed from our Monte Carlo simulations, and recursively calculate the probabilities of P(n=N) (i.e., the probability that a primary skew has N valid tags) for N=[1,8].

Case 2: Both primary buckets are full. When the two randomly picked primary buckets are full. we need to insert a ball into a secondary bucket. To model the INTERFACE chaining feature, we randomly pick 4 buckets, 2 from each secondary skew. We then compare the occupancy of all candidate buckets, and choose the bucket with the lowest occupancy

for insertion. Hence, given both primary skew buckets are full, a bucket in the secondary skews transitions from containing N balls to N+1 balls in one of the following scenarios:

- All four buckets contain N balls.
- Three buckets contain N balls and one contains more than N balls.
- Two buckets contain N balls and the other two contain more than N balls.
- One bucket contains N balls and the other three contain more than N balls.

Where B is the event where both primary buckets are full (i.e, include 8 balls each).

$$P(B) = P(n = 8) * P(n = 8) \quad (5)$$

Using Bayes' Theorem:

$$P(N \to N + 1|B) = \frac{P(B|N \to N + 1) * P(N \to N + 1)}{P(B)} \quad (6)$$

Since insertion in a secondary bucket occurs only if B has occurred, the prob. of B given an N to N+1 transition is 1:

$$P(B|N \to N + 1) = 1 \quad (7)$$

Substituting (7) into (6):

$$P(N \to N + 1|B) = \frac{P(N \to N + 1)}{P(B)} \quad (8)$$

Where the Probability of transitioning from N to N+1 is computed from all four scenarios above:

$$P(N \to N + 1) = P(n = N)^4 + 4 * P(n = N)^3 * P(n > N)$$

$$+6 * P(n = N)^2 * P(n > N)^2 + 4 * P(n = N) * P(n > N)^3 \quad (9)$$

Substituting (9) into (8):

$$P(N \to N + 1|B) = \frac{1}{P(B)} * (P(n = N)^4 +$$

$$4 * P(n = N)^3 * P(n > N) + 6 * P(n = N)^2 * P(n > N)^2 +$$

$$4 * P(n = N) * P(n > N)^3) \quad (10)$$

The probability that a secondary skew bucket transitions from N to N+1 balls (given B has occurred) is equal to the probability that a secondary skew bucket transitions from N+1 to N balls:

$$P(N \to N + 1|B) = P(N + 1 \to N|B) \quad (11)$$

Since the events $N+1->N$ and B are independent:

$$P(N \to N+1|B) = P(N+1 \to N) \quad (12)$$

During global eviction, a secondary skew bucket transitions from N+1 to N balls if the evicted ball belongs to a bucket in the secondary skew AND the bucket contains N+1 balls:

$$P(N+1 \to N) =$$

$$\frac{P(B) * P(n = N+1) * B_{total} * (N+1)}{b_{total}} \quad (13)$$

Substituting (13) and (10) into (12) and rearranging we get:

$$P(n = N+1) = \frac{8}{(N+1) * P(B)^2} * (P(n = N)^4 +$$

$$4 * P(n = N)^3 * P(n > N) + 6 * P(n = N)^2 * P(n > N)^2$$

$$+4 * P(n = N) * P(n > N)^3) \quad (14)$$

Based on our observed Monte Carlo results for P(n=0) in the secondary skews, we recursively calculated P(n=N) for N=[1, 6]. As shown in Figure 4.3, our analytical model accurately estimates the probability of observing a primary or a secondary bucket with N balls. W similarly modeled MIRAGE by taking P(n=N) in the secondary skews as P (n=N+8), where 8 is the maximum capacity of primary skew buckets. Results from our model matches the results from MIRAGE [21]. Finally, to compute the probability of a SAE given W extra ways, we compute the probability of a W to W+1 transition in (9):

$$P[W \to W+1] = \frac{1}{P(B)} * (P(n = W)^4 +$$

$$4 * P(n = W)^3 * P(n > W) + 6 * P(n = W)^2 * P(n > W)^2 +$$

$$4 * P(n = W) * P(n > W)^3) \quad (15)$$

4.1.3 Security Analysis

From our model in 4.1.1, Figure 4.2 shows the average number of cache accesses per set-associative eviction (SAE) for INTERFACE and MIRAGE. With INTERFACE, we didn't observe any SAEs for extra ways greater than two across five Monte Carlo simulations with 10 billion accesses each. For MIRAGE, we observed SAEs for up to 4 extra ways.

We used our analytical model in 4.1.2 to estimate the average number of ball insertions per spill. Figure 4.4 shows that with 4 extra ways (50%) in INTERFACE an attacker needs

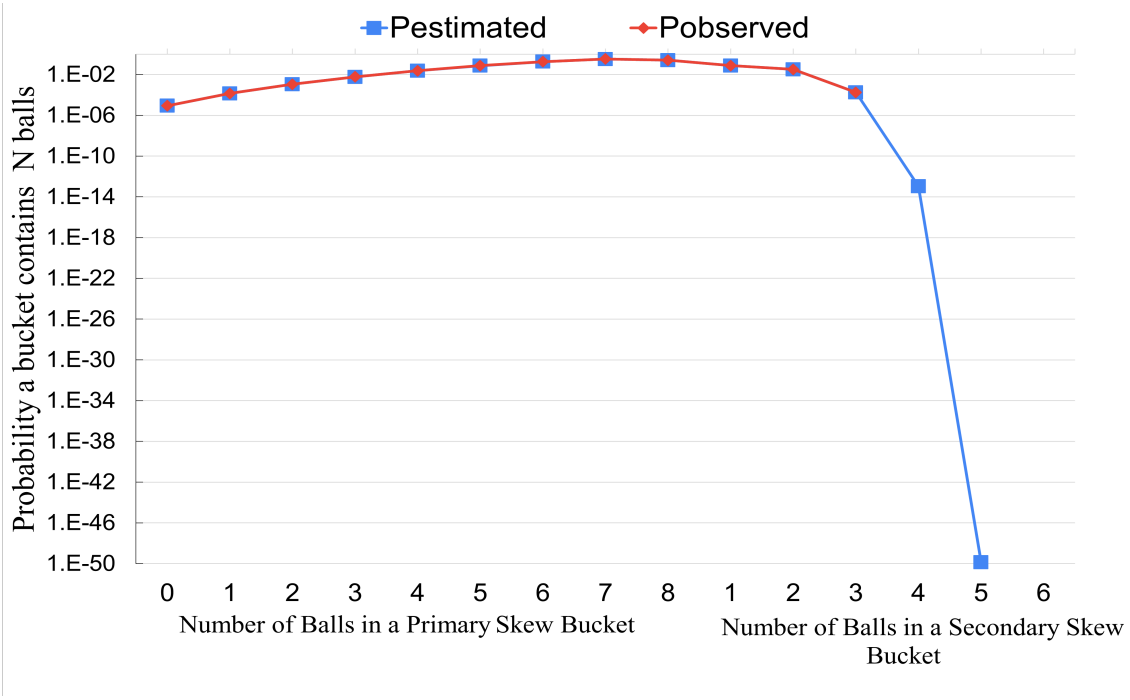


Figure 4.3: Probability of observing a Primary or Secondary skew bucket with N balls.

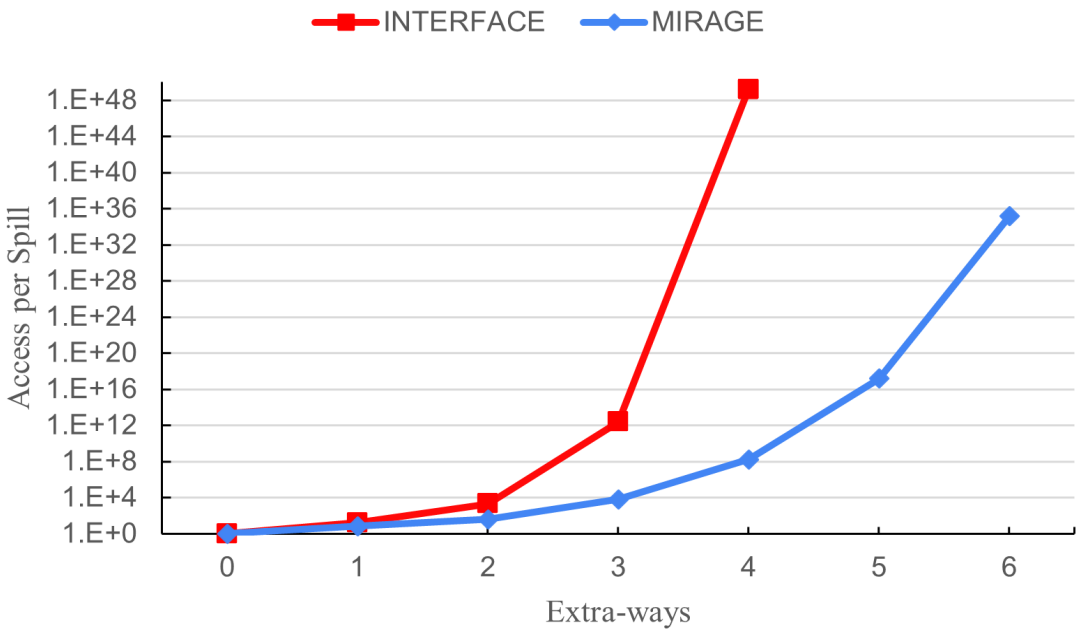


Figure 4.4: The estimated average number of balls inserted per spill vs. #extra ways for MIRAGE and INTERFACE.

10^{49} insertions (equivalent to 10^{32} years assuming 1 billion cache accesses per second) before a spill occurs. MIRAGE requires 6 extra ways (75%) to provide 10^{17} years worth of security. INTERFACE achieves a better security guarantee with 33% less extra-ways.

By using extra tags efficiently, INTERFACE provides stronger security with lower tag overhead compared to MIRAGE. For each insertion in INTERFACE, the total number of candidate locations is the sum of all locations in two primary skews and 4 secondary skews. For MIRAGE, however, the total number of candidate locations is the sum of all locations in the two primary skews and two secondary skews. Therefore, INTERFACE has a higher probability of locating an invalid tag and avoiding an SAE compared to MIRAGE, as shown in Figure 4.5. In addition to preventing set-conflict-based attacks from the same process (e.g., Spectre V1 [19]), INTERFACE also protects against inter-process occupancy-based attacks by partitioning the LLC among processes.

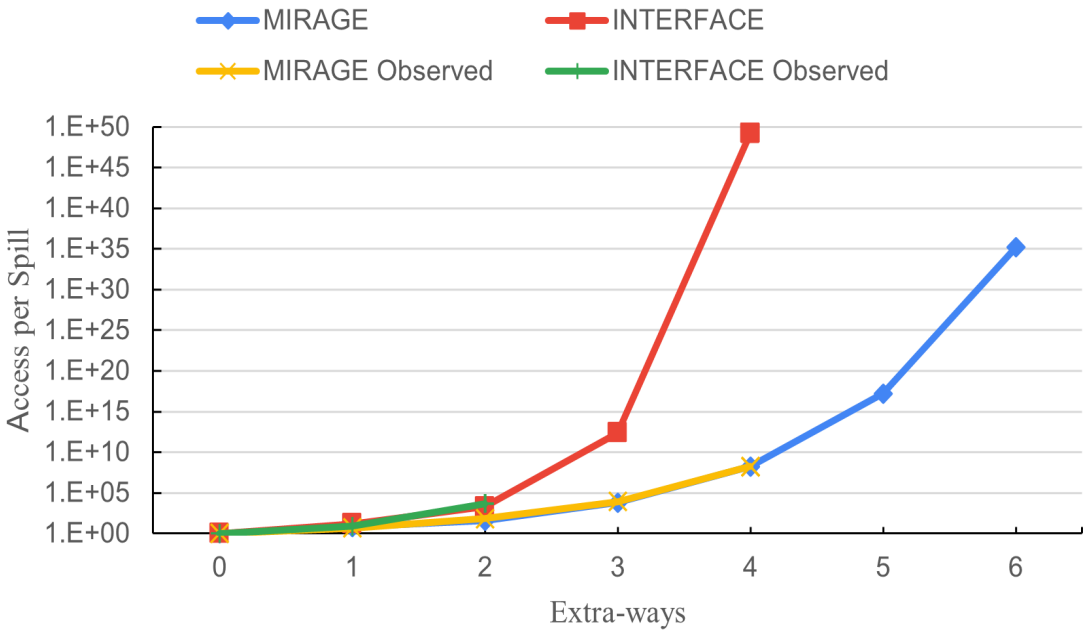


Figure 4.5: The estimated and the observed average number of balls inserted per spill vs. #extra ways for MIRAGE and INTERFACE.

4.1.4 Dynamic Partitioning Security Analysis

To evaluate security against occupancy attacks, we used Metior[58], and modeled a 128KB. We use a small cache due simulation time limit. A larger (e.g., 16MB) cache will have the same leakage but its priming requires a much longer time. Metior is a comprehensive model to quantitatively study the security guarantees of different side-channel obfuscating approaches. Metior defines leakage bits not as direct data leakage but a relative metric

to quantify the distinguishability of a victim's modulation pattern relative to randomly guessing a pattern. We modeled a dynamically-partitioned INTERFACE with a floating bound of 40-60% of total capacity. The attacker primed the entire cache multiple times before probing. After 8 iterations, we measured a maximal leakage of 0.32 bits or $2^{0.32}$, i.e., 1.25x more successful than a blind guess. A non-partitioned cache has a maximal leakage of 0.64 bits (1.56x higher success than blind guessing). Setting a moving bound on each process' occupancy adds noise into the priming step by randomly evicting priming cache lines more often than a non-partitioned cache.

Chapter 5

Performance and Power Evaluation

5.1 Evaluation Methodology

We evaluate INTERFACE using gem5[59], a cycle-accurate execution driven simulator that provides detailed CPU, cache and memory models. We modeled INTERFACE as both a partitioned and non-partitioned skewed set-associative tag store and a fully-associative data store cache with a tag-store-based global eviction, no reverse pointers and a free list implemented using the data store (Chapter 3.1).

To compare against non-partitioned INTERFACE, we use the gem5 implementation of MIRAGE [21] (provided as an artifact by the authors). We simulated representative benchmarks from SPEC CPU2017 [60]. We ran 16-core multi-program simulations for each benchmark (i.e, 16 copies) where we fast-forwarded 20 billion instructions on each core and simulated 1 billion instructions per core.

To evaluate the partitioned INTERFACE, we implemented static way-partitioning on the baseline 16-way cache with the configuration in Table 5.1. Due to the complexity of identifying trusted processes like prior work, we assume that none of the processes are trusted, so static way-partitioning is the only mechanism to guarantee preventing occupancy attacks. We used twelve 4-benchmark mixes of SPEC CPU2017 benchmarks shown in Table 5.2, and ran 4-core simulations for each mix where we fast forwarded 20 billion instructions and simulated 1 billion instructions. We use normalized instructions-per-cycle (IPC) to estimate performance. We evaluated the baseline and INTERFACE configurations shown in Table 5.1. We estimated baseline, INTERFACE and MIRAGE latencies using CACTI (Chapter 5.3).

5.2 Performance Analysis

Figure 5.1 shows the normalized IPC of non-partitioned INTERFACE and MIRAGE versus the baseline. Both mechanisms use global random replacement and similar extra latency per access, so both have similar performance. The performance drop varies depending on

Table 5.1: Baseline and INTERFACE Configurations

Structure	**Configuration**
CPU	16 cores, Timing simple
L1 Split I/D Caches	Private, 32KB, 8-way, 64B lines
L2 Cache	Private, 256KB, 8 way, 64B lines
LLC (Baseline)	Shared, 16 MB, 16-way, set-associative, 64B lines, LRU replacement
LLC (INTERFACE)	Shared, 16MB data store, Tag: 2 primary skews each 8 ways, 2 secondary-skews each 4 ways, global random replacement

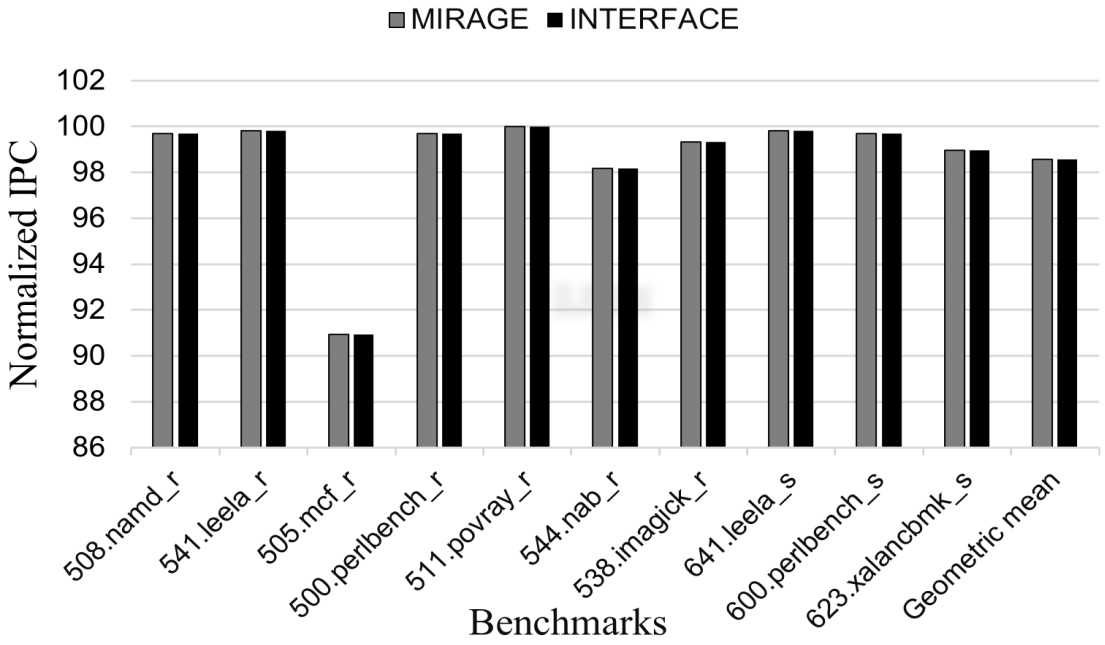


Figure 5.1: Normalized IPC for MIRAGE and INTERFACE (normalized to Baseline LLC).

the benchmark. The maximum performance drop was 9% for the 505.mcf_r benchmark, and the geometric mean drop was 1.4% across all benchmarks. Both INTERFACE and MIRAGE use random replacement instead of the baseline LRU, which affects benchmarks that are sensitive to cache replacement (e.g., mcf). Most performance losses are attributed to early random evictions of frequently used lines, while the small increase in cache access latency has a lower performance impact.

Figure 5.2 shows the normalized IPC of non-partitioned and partitioned INTERFACE, the way-partitioned baseline, and dynamically partitioned INTERFACE versus the (non-secure) non-partitioned baseline. Set-partitioned INTERFACE shows a 2.7% mean performance drop while way-partitioned baseline has a 3.1% mean performance drop. INTER-

Table 5.2: 4 Core Benchmark Mixes

Mix	Benchamarks
1	cactuBSSN, gcc, bwaves, nab
2	lbm, xalancbmk, deepsjeng,leela
3	lbm, imagick, blender, gcc
4	lbm, bwaves, deepsjeng, leela
5	lbm, bwaves, deepsjeng, cactuBSSN
6	xalancbmk, leela, bwaves, blender
7	gcc, lbm, deepsjeng, nab
8	nab, imagick, deepsjeng, blender
9	perlbench, gcc, mcf, bwaves
10	gcc, xalancbmk, imagick, cactuBSSN
11	cactuBSSN, gcc, mcf, nab
12	namd, bwaves, imagick, blender

FACE's set-partitioning maintains its fully associative design which helps cope with the imposed capacity limit of statically partitioning the cache. However, Way-based partitioning reduces both associativity and capacity which results in worse miss rates. Set partitioning scales with the number of cores, while way partitioning is limited by the number of LLC ways, which usually is less than 32. We note that partitioned caches sometimes outperform non-partitioned caches in cases where one or more benchmarks occupy a large portion of the cache with low impact on miss rate (i.e, has a lower marginal utility [61]) while other benchmarks benefit from additional capacity. Dynamically partitioning INTERFACE resulted in only a 1.7% drop due to limiting occupancy of benchmarks with low marginal utility.

5.2.1 Cache Size Sensitivity

We evaluated LLC sizes of 2, 4, 8 and 16MB using the eight 4-core mixes (Table 5.2) and plotted the geometric mean in the Figure 5.3. For cache sizes 2, 4 and 8MB, we observed a high miss rate for all the configurations due to capacity misses. For these sizes, the hit access latency overhead of set-partitioned INTERFACE results in a higher average access latency compared to the baseline way-partitioned configuration, and a performance drop of 0.5%, 0.8% and 0.3% respectively. For 16MB, set-partitioned INTERFACE has a lower miss-rate than way partitioned baseline, resulting a performance increase of 0.7%.

5.2.2 Performance Impact of Free-list in MT workloads

The importance of our Free-list goes beyond the initial booting phase. For MT workloads, disabling the Free-list beyond the initial booting phase means losing cache capacity due to blocks invalidated by coherence transactions. To evaluate the Free-list's performance impact, we used 8 PARSEC and 1(ep) NPB 4-thread workloads and configured non-partitioned

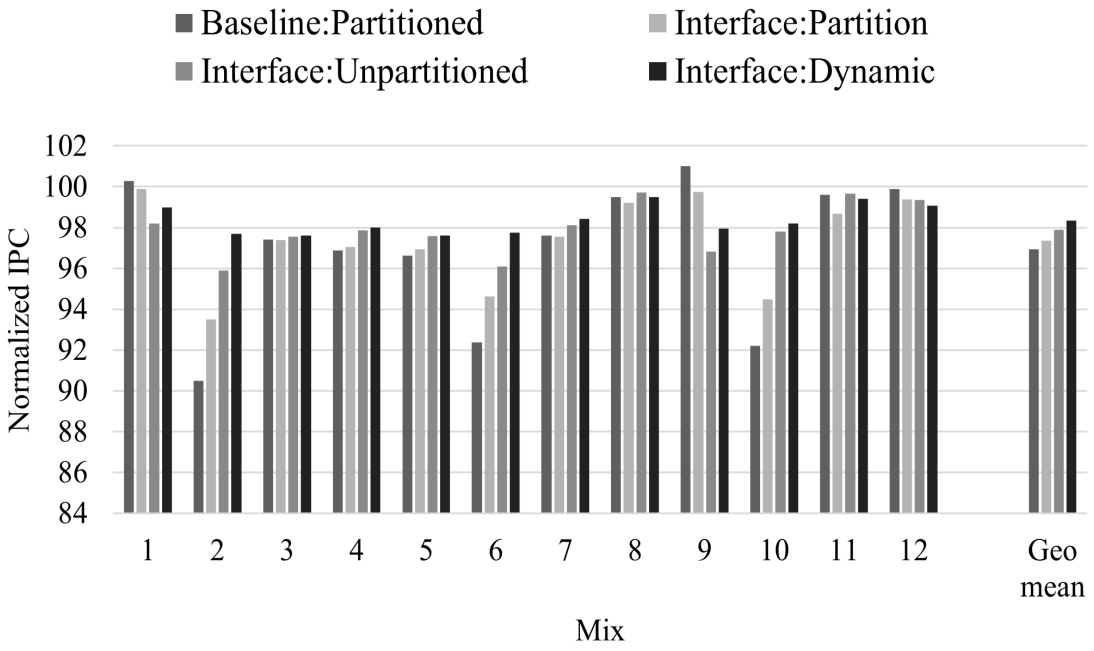


Figure 5.2: Normalized IPC of the way-partitioned baseline, set-partitioned INTERFACE, dynamically partitioned INTERFACE, and non-partitioned INTERFACE against non-partitioned baseline (16MB).

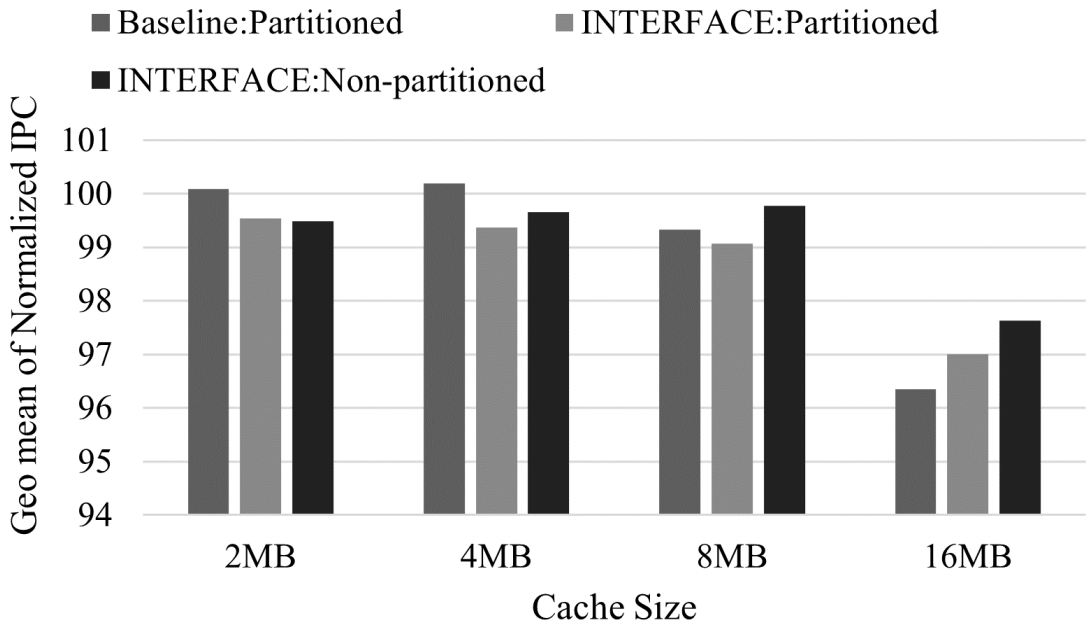


Figure 5.3: Geometric mean of Normalized IPC for 4-mix benchmarks at different cache sizes.

INTERFACE with and without Free-list. We simulated a 4-core, classic memory system (MOESI coherence) gem5 FS model and normalized IPC against a Baseline 16MB cache. Figure 5.4 shows a 13.2% mean performance drop due to disabling the free-list, which resulted in losing cache capacity. We saw 2x and 1.26x drops in streamcluster and ep respectively due to a large number of coherence invalidations reducing cache capacity. The remaining workloads exhibit lower data sharing, hence the performance drop is smaller.

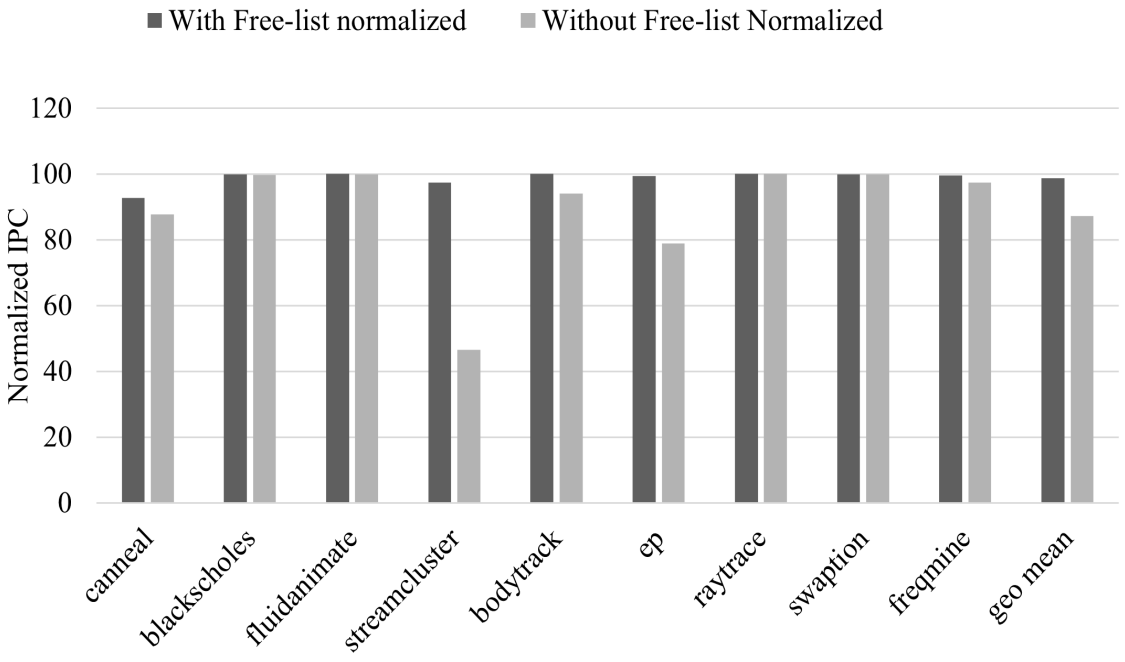


Figure 5.4: Normalized IPC of Non-partitioned INTERFACE with and without a Free-list.

5.3 Latency, Storage and Power Overheads

The latency overhead of INTERFACE can be attributed to the indexing hash function logic, the wider tag bits (tag + data store pointer + status bits) and the sequential indirection to access a data store block. To get latency and power estimates of INTERFACE, we used Cacti 7.0 [62]. We modeled the baseline and INTERFACE caches in Table 5.1. Modeling the baseline 16MB cache was simple in CACTI using the default NUCA configuration. However, modeling INTERFACE and MIRAGE was challenging since Cacti doesn't support a different associativity for tag and data stores.

5.3.1 Latency

For each cache access, INTERFACE makes 32 tag comparisons. To compute the latency upper bound, we modeled a 32MB, 32-way sequential cache with 62 tag bits. At 3GHz

using 22nm technology, our model required one more cycle for data access compared to the baseline. Since the primary and secondary skews are accessed in parallel, we need 6 indexing hash-function blocks. At 3GHz clock frequency, MIRAGE [21, 63] reports the PRINCE block cipher logic can be pipelined into 3 stages for a total of 3 cycles. Our partition table, modeled using Cacti, is accessed in parallel with the cipher logic and takes 1 cycle to complete. Lastly, the masking and adding operations are incorporated into the last stage of our cipher logic. In total, INTERFACE requires 4 additional cycles per cache access: 3 cycles for indexing logic and 1 cycle for the indirection to data store logic and tag comparison. Latency is similar to MIRAGE since we use the same hash functions, indirection logic and smaller tag bit width.

5.3.2 Power

We modeled 6 tag arrays (2 primary and 4 secondary) using Cacti to represent INTERFACE's 6 parallel tag array accesses for each read/write request. We obtained the total dynamic power by adding the dynamic power consumed by the 6 tag arrays (including routing and wiring dynamic power) to the dynamic power consumed by a 16MB data array. INTERFACE consumes more dynamic power than MIRAGE due to the 4 extra tag array accesses. However, for LLCs dynamic power accounts for a small fraction of total power which is dominated by static power. The static power (leakage) overhead of INTERFACE is due to the 1.5x increase in the number of extra tag store entries and the 2.06x increase in the number of tag bits per entry. To estimate static power, we modeled the tag store size of INTERFACE using a 24MB set associative cache with 62 tag bits per entry, and used a 16MB set-associative cache with 30 tag bits as a baseline cache. To estimate the tag store leakage power of both configurations, we multiplied the per-bank tag array leakage power by the bank count. Since INTERFACE's data store configuration is the same as the baseline, we used the baseline's per-bank data array leakage power multiplied by the bank count to estimate the data store leakage power. The optimal bank layout for both the baseline cache and the 24MB cache was 4x4, hence we added the wire leakage and router leakage powers of the respective caches to the total array leakage power (tag array leakage + data array leakage) to get the overall estimated leakage power of INTERFACE and the baseline. The leakage power contributed by our Partition Table is negligible because it only accounts for 0.002% of the cache capacity. Free-list reads and writes to the data store incur more dynamic power than MIRAGE's dedicated Free-list table reads and writes since INTERFACE reads longer bit-lines. However, the LLC is dominated by static power, and static power saved by removing a dedicated hardware free-list table is significantly larger than the additional dynamic power incurred from reading longer bit-lines. Table 5.3 shows that INTERFACE increases total power by 16.9% vs. baseline, compared to a 21.9% increase for MIRAGE.

Table 5.3: Power Overhead Compared to Baseline.

	Baseline	**MIRAGE**	**INTERFACE**
Dynamic Energy/access(nJ)	0.776	1.305	1.870
Dyn. Power (W)	0.041	0.065	0.093
Leak. Power (W)	5.837	7.101	6.781
Total Power (W)	5.878	7.166	6.874

5.3.3 Storage

INTERFACE requires more storage than the baseline due to partition registers, extra tag store entries, per-tag forward pointer (for indirection), and extra address tag bits for physical address reconstruction. Compared to a baseline 16MB 16-way cache (Table 5.4), INTERFACE increases tag store size by 3.1X leading to a 11.6% LLC storage overhead. These overheads are computed for a 16MB 16-way set-associative baseline. Comparing with 32-way MIRAGE, 32-way INTERFACE will have 4 extra ways instead of 6 and will require 10^{30} insertions per spill instead of 10^{25} insertions per spill, with 8.3% less storage. Overheads for MIRAGE and INTERFACE are slightly higher for larger caches and slightly lower for smaller caches. However, compared to the state-of-the-art secure design (MIRAGE), INTERFACE requires 33% fewer extra ways, removes data store reverse pointers (608KB), Process ID bits (448KB) and free-list pointers (576KB). Consequently, statically set-partitioned INTERFACE has a 26.2% smaller tag storage and 7% smaller data storage for an overall of 10.4% smaller total LLC capacity than MIRAGE. Dynamically-partitioned INTERFACE requires process ID bits in the tag to enable our eviction policy, so the tag storage is 14.3% smaller than MIRAGE, and the total LLC capacity is only 8.1% smaller than MIRAGE.

Table 5.4: Baseline, MIRAGE and INTERFACE cache storage sizes for a 16 MB cache with 4 extra ways in the secondary skews of INTERFACE, and 48bit physical addresses.

	Baseline	**MIRAGE**	**INTERFACE (Static)**	**INTERFACE (Dynamic)**
Partition Reg./Core(bits)	0	0	32	32
Total Partition Table(bytes)	**0**	**0**	**384(96 cores)**	**384**
Tag Bits	28	42	42	42
Status Bits	2	2	2	2
Data Store Pointer (bits)	0	18	18	18
PID Bits	0	10	0	10
Bits/tag entry	30	72	62	72
Total Tag Store (bytes)	**960KB**	**4032KB**	**2976KB**	**3456KB**
Data Store entry (bits)	512	531	512	512
Free-list (bytes)	**0**	**576KB**	**5**	**5**
Total Data Store (bytes)	**16,384KB**	**17,568KB**	**16,384KB**	**16,384KB**
Total (bytes)	**17,344KB**	**21,600KB**	**19,360KB**	**19,840KB**

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we showed that shared last-level caches are vulnerable to conflict- and occupancy-based attacks. Prior mitigation approaches (e.g., partitioning and randomized caches) suffer from high performance overheads or low security guarantees. MIRAGE [21] proposed a fully-associative approach using a set-associative tag store with extra entries and a fully-associative data store with global eviction to defend against conflict-based attacks. However, MIRAGE does not prevent occupancy attacks, incurs area and power overheads due to a large number of extra tags, and a higher implementation complexity due to using custom data store designs with reverse pointers and a separate hardware free list.

INTERFACE uses a novel architecture that increases the number of tag skews and partitions the tag store at a lower area and power overhead while using a simpler data store design. *Interface defends against both conflict- and occupancy-based attacks.* Compared to MIRAGE, INTERFACE prevents occupancy attacks, reduces extra tags by 33%, reduces tag store area/leakage by 26.2% and data store area/leakage by 7% with a 2.7% average performance overhead over a set-associative 16MB baseline. Compared to partitioned caches, INTERFACE prevents intra-process conflict attacks and provides comparable performance. Dynamically-partitioned INTERFACE only has a 1.7% performance overhead, compared to 2.7% drop of set-partitioned INTERFACE and 3.1% drop of way-partitioned baseline, vs. the un-partitioned baseline and 8.1% lower LLC area/leakage vs. MIRAGE with a small added vulnerability to occupancy attacks.

Compared to most prior works, INTERFACE does not require burdening the OS or the user to classify execution contexts, can support a large number of concurrent processes by using a static Core ID-based set partitioning, and does not need attack pattern detection logic to provide robust security guarantees. By adding a limited number of extra tag store entries, two registers, a small Partition table and using an encrypted indexing hash function, INTERFACE provides strong security guarantees with small performance, area and power overheads.

6.2 Future Work

In the future, we would like to extend the concept of security aware hardware optimizations, which requires addressing several challenges. One of the challenges of security aware hardware designs is performance degradation. To minimize the performance loss of using static and strong partitioning principles, we are extending our current work by loosening the strict security requirements without compromising future security guarantees. This involves exploring different adaptive approaches without introducing exploitable characteristics. Current mitigation approaches also introduce significant area and power overheads, reducing the overheads introduced by these approaches is an another challenge that needs to be addressed.

Most importantly, reducing the complexity of security aware designs to enable quick industry adaptation is another challenge we partially addressed in this work, and want to explore further opportunities to minimize validation cost of new designs. Lastly, we want to introduce mitigation approach which principally address the root cause of these hardware vulnerabilities transparently without burdening the software developer with new programming paradigms.

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*, 2015, pp. 605–622.

[2] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[3] D. Gruss, C. Maurice, and K. Wagner, "Flush+flush: A stealthier last-level cache attack," *CoRR*, vol. abs/1511.04594, 2015. [Online]. Available: http://arxiv.org/abs/1511.04594

[4] 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: USENIX Association, Aug. 2019, pp. 639–656. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/shusterman

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 1–20. [Online]. Available: https://doi.org/10.1007/11605805_1

[6] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Website fingerprinting through the cache occupancy channel and its real world practicality," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2042–2060, 2021.

[7] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 570–581. [Online]. Available: https://doi.org/10.1145/2660267.2660294

[8] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 199–212. [Online]. Available: https://doi.org/10.1145/1653662.1653687

[9] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 494–505. [Online]. Available: https://doi.org/10.1145/1250662.1250723

[10] 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)*, 2018, pp. 775–787.

[11] ——, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 360–371.

[12] 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.

[13] 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)*, 2020, pp. 1110–1123.

[14] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[15] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-cache: On-demand and scalable cache isolation for security architectures," *arXiv preprint arXiv:2110.08139*, 2021.

[16] 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)*, 2018, pp. 974–987.

[17] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[18] Intel, "Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2016.

[19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[20] 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.

[21] 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)*. USENIX Association, Aug. 2021, pp. 1379–1396. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[22] 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)*, 2005, pp. 544–555.

[23] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 107–116. [Online]. Available: https://doi.org/10.1145/339647.339660

[24] 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*, 2015, pp. 605–622.

[25] 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)*. USENIX Association, Aug. 2020, pp. 1967–1984. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[26] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 51–67. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[27] 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)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[28] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 353–364. [Online]. Available: https://doi.org/10.1145/2897845.2897867

[29] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic prime+probe attacks and covert channels in scattercache," 2019. [Online]. Available: https://arxiv.org/abs/1908.03383

[30] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. M. Colbert, and P. L. Yu, "Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[31] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "Papp: Prefetcher-aware prime and probe side-channel attack," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317877

[32] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2897937.2897962

[33] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 139–152.

[34] 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)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 427–442. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/song

[35] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 2nd ACM Workshop on Computer Security Architectures*, ser. CSAW '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 25–34. [Online]. Available: https://doi.org/10.1145/1456508.1456514

[36] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 771–788. [Online]. Available: https://doi.org/10.1145/3582016.3582033

[37] K. Wang, F. Yuan, L. Zhao, R. Hou, Z. Ji, and D. Meng, "Secure hybrid replacement policy: Mitigating conflict-based cache side channel attacks," *Microprocess. Microsyst.*, vol. 89, no. C, mar 2022. [Online]. Available: https://doi.org/10.1016/j.micpro.2021.104420

[38] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 347–360.

[39] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 871–882. [Online]. Available: https://doi.org/10.1145/2976749.2978324

[40] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 406–418.

[41] 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)*. USENIX Association, Aug. 2020, pp. 451–468. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky

[42] T. Bourgeat, I. 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*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 42–56. [Online]. Available: https://doi.org/10.1145/3352460.3358310

[43] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, ser. CCSW '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 77–84. [Online]. Available: https://doi.org/10.1145/1655008.1655019

[44] 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: USENIX Association, Aug. 2019, pp. 675–692. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/werner

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

[46] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 169–178.

[47] 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)*. IEEE, 2021, pp. 987–1002.

[48] 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," *CoRR*, vol. abs/2008.01957, 2020. [Online]. Available: https://arxiv.org/abs/2008.01957

[49] 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)*. IEEE, sep 2022. [Online]. Available: https://doi.org/10.1109%2Fseed55351.2022.00009

[50] 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.

[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 - extended abstract," in *ASIACRYPT*, 2012.

[52] D. J. Bernstein, "Cache-timing attacks on aes," *Technical Report*, 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414

[53] S. Banescu, "Cache timing attacks," 2011. [Online]. Available: https://www.researchgate.net/publication/235339284_Cache_Timing_Attacks

[54] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006, pp. 201–215.

[55] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of des implemented on computers with cache," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2003, pp. 62–76.

[56] P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking Secrets Through Compressed Caches," in *Proceedings of the 25th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-25)*, March 2020.

[57] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, p. 1094–1104, oct 2001. [Online]. Available: https://doi.org/10.1109/71.963420

[58] P. W. Deutsch, W. T. Na, T. Bourgeat, J. S. Emer, and M. Yan, "Metior: A comprehensive model to evaluate obfuscating side-channel defense schemes," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589073

[59] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[60] SPEC, "Spec cpu 2017," https://www.spec.org/cpu2017/, 2017.

[61] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 423–432.

[62] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, jun 2017. [Online]. Available: https://doi.org/10.1145/3085572

[63] J. Harttung, "Implementation of the prince lightweight block cipher in vhdl, https://github.com/huljar/prince-vhdl."