

The Other Side of Secure Dynamic Cache Partitioning

Submitted in partial fulfillment of the requirements for the degree of
Master of Science (by Research)
by

Prerna Priyadarshini
23M2115
prerna@cse.iitb.ac.in

Supervisor:
Prof. Biswabandan Panda



Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2023-2026

Thesis Approval

This thesis entitled

The Other Side of Secure Dynamic Cache Partitioning

by

Prerna Priyadarshini

Roll No.: 23M2115

is approved for the degree of

Master of Science by Research in Computer Science and Engineering

Digital Signature
Biswabandan Panda (10001949)
18-Jul-26 07:34:39 PM

Prof. Biswabandan Panda
(Supervisor)

Digital Signature
Sayandeep Saha (10002140)
18-Jul-26 09:05:54 PM

Prof. Sayandeep Saha
(Examiner)


.....

Prof. Chester Rebeiro
(Examiner)

Digital Signature
Udayan Ganguly Ganguly (i10040)
.....19-Jul-26 10:37:47 AM.....

Prof. Udayan Ganguly
(Chairperson)

Date: July 18, 2026

Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented, fabricated, or falsified any idea, data, fact, or source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and may also invoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained when needed.

I declare that I have used the following AI tools and/or similar software tools: ChatGPT (GPT-4o mini), Claude Sonnet 4.5, Grammarly, and Overleaf (LaTeX editor) for language refinement, formatting assistance, and improving the clarity and coherence of the text, without generating or fabricating experimental results, data, or references.

Digital Signature Perna Priyadarshini (23m2115) 20-Jul-26 01:25:33 PM

Perna Priyadarshini
23M2115

Date: July 16, 2026

Nomenclature

AMAT	Average Memory Access Time
APKI	Accesses Per Kilo Instructions
BCE	Bespoke Cache Enclave
BTB	Branch Target Buffer
CBT	Cluster Bank Table
CC	Composite Cachelets
CFL	Cachelet Free List
CIM	Cluster Indirection Module
CP	Cache Partitioning
CPMU	Cache Partitioning Management Unit
CPTC	Cache Partition Table Context
CPU	Central Processing Unit
DAWG	Dynamically Allocated Way Guard
DRAM	Dynamic Random-Access Memory
ISA	Instruction Set Architecture
L1D	Level 1 Data Cache
L2C	Level 2 Cache
LBH	Load Balancing Hash
LCID	Logical Cluster ID
LLC	Last-Level Cache

MB	Megabyte
MSHR	Miss Status Holding Register
MT/s	MegaTransfers per Second
OS	Operating System
PCID	Physical Cluster ID
PQ	Prefetch Queue
ROB	Reorder Buffer
SCALE	Secure and Scalable Cache Partitioning
SDRAM	Synchronous Dynamic Random-Access Memory
SGX	Software Guard Extensions
TEE	Trusted Execution Environment
UCP	Utility-Based Cache Partitioning
VPT	Virtual Partition Table

Abstract

Last-Level Cache (LLC) side-channel attacks have emerged as a major security concern in modern multi-core systems due to the shared nature of cache hierarchies. To mitigate such attacks, several secure cache partitioning mechanisms have been proposed that dynamically partition LLC resources among mutually distrusted domains. Recent state-of-the-art techniques advocate dynamic cache partitioning as a practical balance between strong security guarantees and efficient cache utilization. However, the dynamic behavior introduced by these mechanisms itself may unintentionally expose new forms of information leakage. In this work we revisit the security assumptions underlying modern dynamic cache partitioning techniques and investigate whether their allocation mechanisms can themselves act as side channels. We first categorize existing secure dynamic partitioning schemes into two broad classes: application-demand-based techniques and LLC-usage-based techniques. Using representative designs from each category, namely Bespoke Cache Enclave (BCE) and SCALE, we analyze how runtime allocation behavior, reconfiguration policies, and cache occupancy interactions may leak sensitive information across isolated domains.

For application-demand-based partitioning, we demonstrate that dynamic allocation and deallocation behavior in BCE introduces observable leakage through allocation failures and remapping-induced timing variations. These effects enable covert communication and expose occupancy information despite the absence of traditional cache contention. We further explore the feasibility of process fingerprinting by analyzing cluster allocation behavior and identifying workload-dependent allocation patterns. For LLC-usage-based partitioning, we study SCALE, a randomized utility-driven cache allocation framework designed to resist occupancy and conflict-based attacks. Through theoretical analysis and experimental evaluation, we show that although SCALE introduces multiple layers of randomization, observable cache occupancy variations and allocation dynamics can still reveal application characteristics. We demonstrate a occupancy-based fingerprinting attack capable of distinguishing workloads using memorygram analysis and cache-usage patterns under dynamic randomized allocation.

Overall, this thesis highlights that eliminating direct cache contention alone is insufficient to guarantee security in dynamic partitioned cache architectures. Allocation decisions, reconfiguration timing, and runtime cache-management policies themselves can become sources of leakage. Our findings emphasize the need for future secure dynamic partitioned techniques to carefully consider the security implications of dynamic resource management in addition to traditional contention-based channels.

Contents

Thesis Approval	i
Declaration	ii
Nomenclature	iii
Abstract	v
1 Introduction	1
2 Background	9
2.1 Memory Hierarchy	9
2.1.1 Cache Hierarchy	10
2.1.2 Shared last-level cache	12
2.2 Last-Level Cache Attacks	13
2.2.1 Conflict-Based Attack	14
2.2.2 Shared-Memory Based Attack	17
2.2.3 Occupancy-Based attack	18
2.2.4 Security Implications of Process Fingerprinting	19
3 Related works	21
3.1 Mitigating LLC Attacks	21
3.2 Static Partitioning	22
3.3 Dynamic Partitioning	23
3.3.1 Application-Demand-Based Technique	25
3.3.2 LLC-Usage-Based Techniques	29
3.3.3 Selection of Representative Techniques	33
4 Revisiting Security in Application-Demand-Based Partitioning	35
4.1 Key Observations	36
4.1.1 Implications of Dynamic Allocation Leakage	37
4.2 Allocation/Deallocation Covert Channels	37
4.2.1 Attack Overview	38

4.2.2	Experimental Setup	38
4.2.3	Threat Model and Assumptions	39
4.2.4	Channel-I: Allocation Latency Channel	40
4.2.5	Channel-II: Memory Access Latency Channel	42
4.2.6	Comparison of the Proposed Covert Channels	44
4.3	A Preliminary Attempt at Process Fingerprinting	45
4.3.1	Attack Intuition	46
4.3.2	Fingerprinting Methodology	46
4.3.3	Cluster Allocation Patterns in BCE	47
4.3.4	Limitations of Occupancy-Based Fingerprinting	48
5	Revisiting Security in LLC-Usage-based Technique	50
5.1	Theoretical Analysis of Design Choice	50
5.2	Claims and Limits of SCALE	54
5.2.1	Claims of SCALE	55
5.2.2	Limits of SCALE	56
5.3	Application Fingerprinting Attack on SCALE	58
5.3.1	Experimental Setup	59
5.3.2	Threat Model	59
5.3.3	Occupancy Attack Methodology	61
5.3.4	Memorygrams Under SCALE	62
5.3.5	Feature Extraction	63
5.3.6	Classification Methodology	64
5.3.7	Impact of Noise and Reconfiguration Randomness	66
5.3.8	Sensitivity Analysis	67
5.3.9	Takeaway	67
6	Comparative Analysis of Dynamic Cache Partitioning Techniques	69
6.1	Comparison of Design Choice	69
6.2	Security Comparison	70
6.3	Performance Comparison	71
6.4	Implementation Complexity	71
6.5	Overall Comparison	72
7	Conclusion and Future Work	74
8	Acknowledgements	75

Chapter 1

Introduction

Modern processors rely heavily on multi-level cache hierarchies to bridge the widening performance gap between fast processors and comparatively slower main memory systems. Among these cache levels, the Last-Level Cache (LLC) plays a critical role in reducing expensive off-chip DRAM accesses by serving as the final on-chip buffering layer before memory requests are forwarded to main memory. In modern multi-core processors, the LLC is typically shared among multiple cores, enabling efficient utilization of cache resources and facilitating inter-core data sharing. However, this shared nature of the LLC also introduces serious security vulnerabilities. Since applications executing on different cores interact with the same LLC, their memory accesses can interfere with one another, creating observable timing differences between cache hits and cache misses. These timing variations form the basis of a broad class of microarchitectural attacks known as LLC-based side-channel and covert-channel attacks.

Over the years, researchers have demonstrated several forms of LLC attacks that exploit this shared resource. As shown in Figure 1.1, these attacks can be broadly categorized into three classes: *shared-memory-based attacks* like Flush+Reload [39], *conflict-based attacks* like Prime+Probe [40], and *occupancy-based attacks* [30]. Shared-memory attacks exploit shared cache lines between attacker and victim processes to infer victim accesses with fine granularity. Conflict-based attacks exploit cache-set contention by constructing eviction sets and observing eviction behavior. Occupancy-based attacks, in contrast, monitor aggregate cache usage patterns rather than individual cache sets, allowing attackers to infer application behavior through variations in LLC occupancy over time. Recent studies have shown that occupancy-based attacks can even enable process and website fingerprinting with minimal attacker capabilities. To mitigate LLC attacks, prior works have primarily explored two directions: LLC randomization [5, 11, 25, 26, 31, 38] and LLC par-

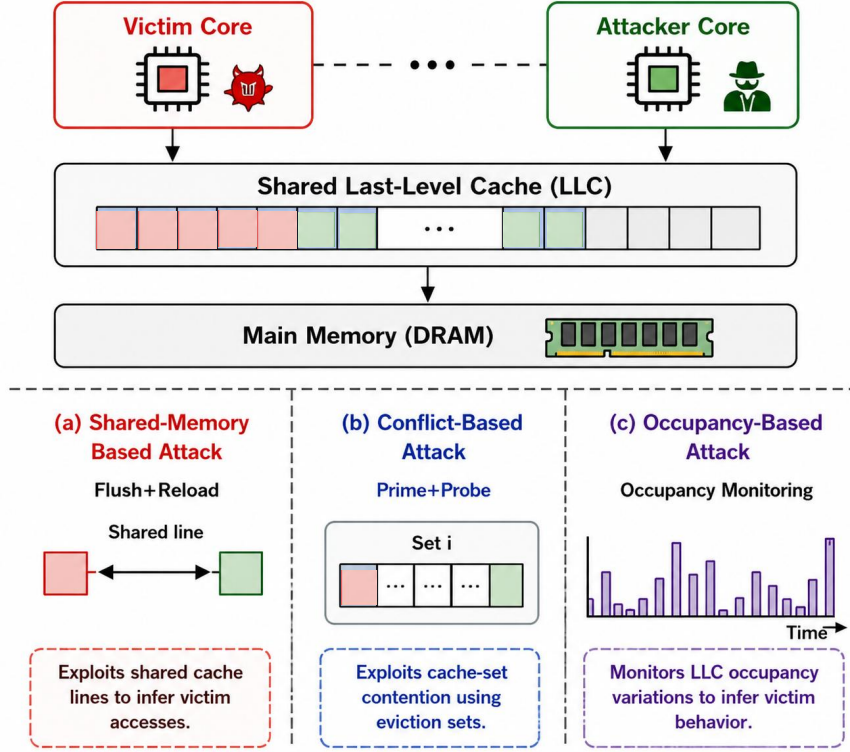


Figure 1.1: Classification of side-channel attacks on the shared LLC in multicore systems. (a) Shared-memory-based attacks exploit shared cache lines, (b) conflict-based attacks exploit cache-set contention, and (c) occupancy-based attacks infer victim behavior from LLC usage variations.

tioning [4, 8, 16, 18, 20, 28, 29, 32, 35, 36]. Randomization-based techniques attempt to make cache behavior unpredictable by randomizing address-to-set mappings or replacement behavior, thereby complicating the construction of eviction sets required for conflict-based attacks [34, 37]. However, such approaches do not fundamentally eliminate cache contention and therefore remain vulnerable to occupancy-based attacks [7] except for heavy-weight designs such as Sasscache [14]. In contrast, partitioning-based techniques attempt to enforce spatial isolation by allocating disjoint LLC regions to different applications or security domains. By preventing mutually distrusted domains from sharing cache space, partitioning techniques aim to eliminate contention-based leakage altogether.

These partitioning techniques provide isolation either via way partitioning [10, 18, 20] or set partitioning [6, 29] or both [4, 32]. Early cache partitioning proposals largely relied on static partitioning, where LLC regions were permanently assigned to specific domains. Although such approaches provide strong isolation guarantees, they often suffer from poor

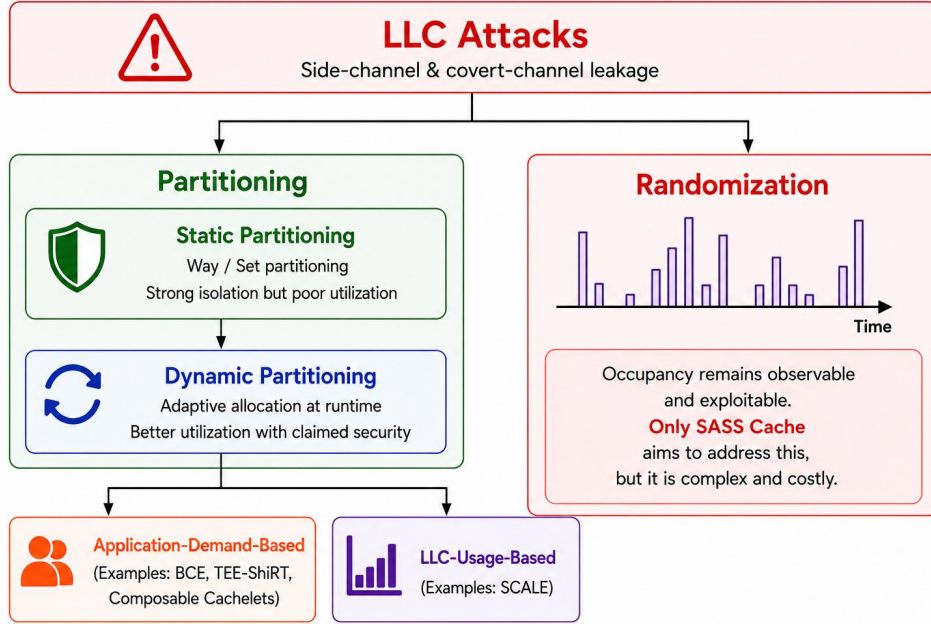


Figure 1.2: Classification of LLC defense mechanisms, highlighting dynamic partitioning techniques and its sub categories.

cache utilization and substantial performance degradation because unused cache capacity from one domain cannot be reused by another [5]. To overcome these limitations, recent research has shifted toward dynamic cache partitioning techniques that adapt cache allocations at runtime according to application demand or observed cache behavior. The objective of these approaches is to strike a balance between two conflicting goals: strong security isolation and efficient cache utilization. State-of-the-art dynamic partitioning techniques can broadly be categorized into two groups based on how LLC space allocation decisions are made, as illustrated in Figure 1.2. The first category consists of *application-demand-based techniques*, where applications or protection domains explicitly request LLC space, and the system dynamically allocates isolated cache regions accordingly. Techniques such as BCE [29], TEE-SHiRT [4], and Composable Cachelets [32] belong to this category. These proposals rely on explicit reservation and hardware-enforced isolation to ensure that mutually distrusted domains do not overlap in cache space.

Bespoke Cache Enclave (BCE) [29] is a representative application-demand-based dynamic partitioning technique that partitions the LLC at the granularity of clusters, where each cluster corresponds to a group of contiguous cache sets. BCE dynamically allocates and deallocates these clusters to security domains as shown in Figure 1.3. It uses privileged

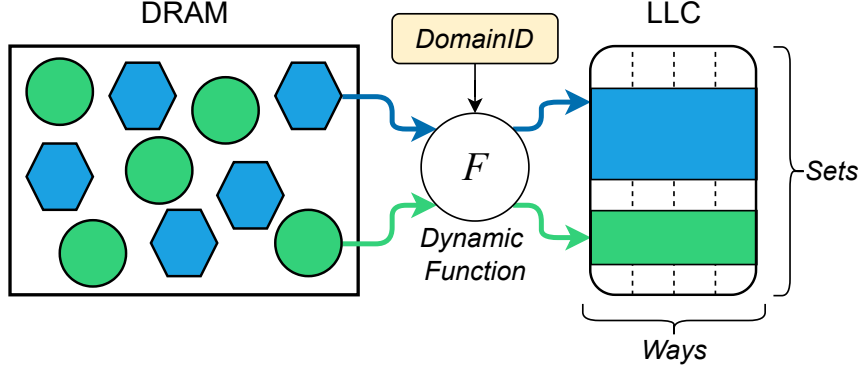


Figure 1.3: BCE provides isolation via set partitioning and allocates LLC sets dynamically. It uses dynamic indexing to map LLC lines to allocated LLC sets. Different colors show different application domains at the DRAM and LLC.

instructions such as `BCEAlloc` and `BCEDealloc` for allocation. By combining hardware-enforced set partitioning with operating-system support, BCE aims to provide both strong isolation and efficient cache utilization.

The second category consists of *LLC-usage-based dynamic partitioning techniques*, where cache allocations are determined according to observed runtime cache usage patterns rather than explicit application requests. These techniques continuously monitor cache behavior, such as miss rates or marginal utility per cache way, and dynamically redistribute LLC space to optimize performance while maintaining isolation. SCALE [16] is a prominent example of this category. SCALE performs way-based LLC partitioning distributed across LLC slices and leverages Utility-Based Cache Partitioning (UCP) [27] to estimate the usefulness of additional cache space for different applications as shown in Figure 1.4. Based on these utility estimates, SCALE dynamically reallocates LLC ways to reduce overall cache misses. To further prevent leakage through predictable allocation behavior, SCALE incorporates randomness using differential privacy mechanisms and randomized allocation noise. The design claims to provide security against occupancy-based attacks while simultaneously improving performance.

The pertinent question. Are state-of-the-art dynamic cache partitioning techniques [4, 16, 29, 32] truly secure against LLC occupancy-based attacks and process fingerprinting attacks? While these techniques claim to eliminate cross-domain interference through dynamic isolation and randomized allocation, the dynamic allocation behavior itself may unintentionally leak information about application activity, cache demand, or execution phases.

To answer this question, we analyze two representative dynamic partitioning techniques in

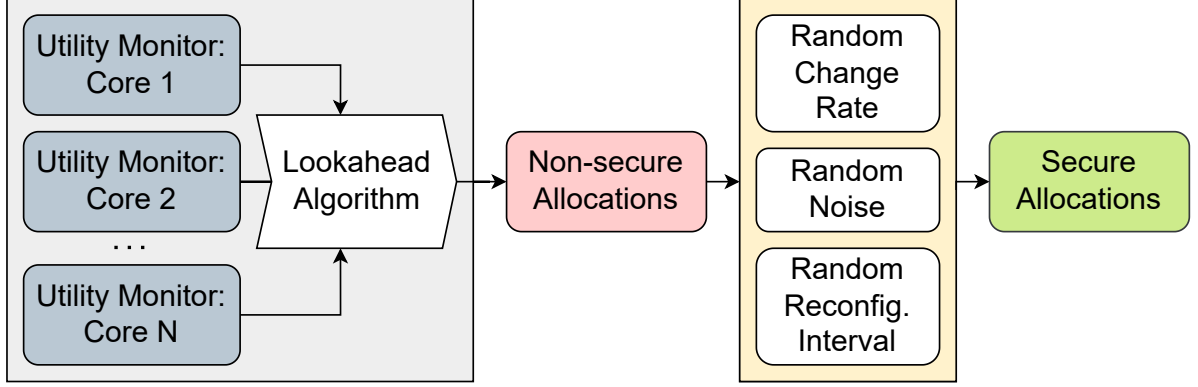


Figure 1.4: SCALE provides dynamic allocation using UCP with additional layers of randomness to make this allocation secure.

detail: BCE and SCALE. BCE represents the class of application-demand-based partitioning techniques where applications explicitly request LLC space. SCALE represents LLC-usage-based partitioning techniques that allocate cache space according to observed runtime cache utility. We select BCE because it is among the best-performing set-partitioning techniques with fine-grained dynamic allocation support when evaluated under modern multi-core workloads [5]. Similarly, we study SCALE because it attempts to jointly optimize security and performance through dynamic utility-driven allocation with additional randomization mechanisms.

Our Observations. We observe that several implementation choices in BCE and SCALE unintentionally leak information through dynamic cache allocation behavior.

Observation I: BCE offers dynamic LLC allocation by allowing application domains to allocate or deallocate LLC clusters using privileged instructions such as `BCEAlloc` and `BCEDealloc`. However, when one domain acquires a large fraction of LLC clusters, other domains may experience allocation failures or observe changes in allocation latency. This dynamic allocation behavior itself can become a source of information leakage.

Observation II: On an LLC allocation change in BCE, the allocation undergoes remapping of LLC clusters. Similar observations have also been noted in prior work, which states that *“even small allocation changes require invalidating and remapping all partition data to accommodate changes to the indexing logic”* [16]. Such remapping operations introduce timing variations observable by other domains and may indirectly leak application behavior.

Observation III: The dynamic cluster allocation behavior of BCE motivates an important question regarding process fingerprinting. Since different applications exhibit varying cache demands and cluster occupancy behavior, we introspect whether such allocation

patterns can indirectly reveal characteristics of co-running applications. In particular, we investigate whether occupancy-based observations in BCE can provide sufficient distinguishability among applications despite the presence of spatial isolation. However, because BCE enforces strict cluster-level isolation, the scope for occupancy-based fingerprinting appears limited compared to conventional shared LLC environments.

Observation IV: SCALE introduces randomized noise in LLC allocation to make the allocation policy less predictable. However, although individual allocations appear randomized, the long-term allocation behavior still remains correlated with the runtime utility estimated by UCP [27]. Memory-intensive applications tend to receive higher LLC allocations compared to lightweight workloads, creating distinguishable occupancy patterns.

Observation V: Although SCALE uses differential privacy [12, 13, 33] to provide information-theoretic guarantees [16], differential privacy does not inherently prevent leakage through aggregate statistical properties such as long-term occupancy trends or allocation patterns [33]. As a result, the runtime allocation behavior of SCALE may still enable occupancy-based process fingerprinting attacks.

New Channels and Fingerprinting Attacks. Based on these observations, we design new covert channels and investigate the feasibility of process fingerprinting attacks targeting dynamic LLC partitioning techniques.

- **Allocate/Deallocate Channels in BCE:** We design covert channels that exploit dynamic cluster allocation and deallocation behavior in BCE. In one variant, the sender occupies a large fraction of LLC clusters while the receiver measures the timing difference between successful and unsuccessful allocation requests. In another variant, the receiver infers allocation success indirectly through memory access latency differences caused by LLC hits and misses (Chapter 3).
- **Assessing the Scope of Process Fingerprinting in BCE:** We further investigate whether BCE’s dynamic cluster allocation behavior leaks sufficient occupancy information for process fingerprinting. By analyzing cluster allocation patterns and occupancy behavior across applications, we assess the extent to which co-running workloads can be distinguished under strict set-partitioned isolation. Our analysis suggests that although some workload-dependent allocation trends are observable, BCE significantly restricts the leakage available to occupancy-based fingerprinting attacks (Chapter 3).
- **Process Fingerprinting in SCALE:** We demonstrate that occupancy-based fingerprinting attacks remain feasible under SCALE despite its randomized allocation mech-

Technique	Isolation Type	Dynamism via	New Channels / Attacks	Key Takeaway
BCE	Set Partitioning	Application Demand Based	• Allocation/Deallocation Covert Channels • Explored Scope of Occupancy-Based Fingerprinting	High-bandwidth of 2.61Mbps and 4.46Kbps covert channels achievable; occupancy-based fingerprinting scope is limited under strict cluster-level isolation.
SCALE	Way Partitioning	LLC Usage Based	• Occupancy-Based Process Fingerprinting	Fingerprinting feasible despite randomized allocation and differential privacy mechanisms, with 96% accuracy achieved in process fingerprinting.

Table 1.1: Summary of our findings on covert channels and process fingerprinting in dynamic LLC partitioning techniques.

anisms. By constructing memorygrams and extracting occupancy-related features, we show that application behavior remains distinguishable through LLC occupancy variations over time (Chapter 4).

Key Results. We achieve an average covert channel bandwidth of 2.61Mbps and 4.46Kbps for two variants of allocate/deallocate channels, respectively, with an accuracy of 100%. Our experiments show that timing differences arising from allocation success and failure can be reliably exploited for covert communication with high accuracy.

We further analyze the feasibility of occupancy-based process fingerprinting under BCE and observe that strict cluster-level isolation significantly limits the amount of information leakage available to the attacker. Although application-dependent allocation behavior can still be observed to some extent, BCE considerably reduces the distinguishability achievable through occupancy monitoring. In contrast, we achieved an application fingerprinting accuracy of 96% in SCALE, demonstrating that occupancy-based fingerprinting attacks remain practical despite randomized dynamic allocation and differential privacy-based noise injection. Our results show that application-specific memory access characteristics continue to leak through long-term LLC occupancy patterns. Overall, our work demonstrates that while modern dynamic partitioning techniques significantly improve the balance between security and performance, the dynamic allocation mechanisms themselves can become new sources of information leakage. These findings highlight the need for future secure cache architectures

to carefully consider not only cache contention, but also the security implications of runtime allocation behavior itself.

Chapter 2

Background

This chapter discusses all the necessary background information required to understand the rest of the document.

2.1 Memory Hierarchy

In today's computing landscape, applications are increasingly driven by large volumes of data, placing significant pressure on memory systems. While processors continue to evolve rapidly in terms of speed and parallelism, the main memory, typically Dynamic Random Access Memory (DRAM), has not advanced at the same pace. Ideally, one would want a memory system that offers both extremely high speed and virtually unlimited capacity. In practice, however, this balance is difficult to achieve.

Modern DRAM technologies can provide bandwidths on the order of 8800 MT/s (Mega-Transfers per Second)(around 70.4 GB/s) [2, 3]. Meanwhile, contemporary processors are capable of issuing memory requests at rates that translate to peak bandwidths well beyond 500 GB/s.¹ The widening gap between processor capability and memory speed often forces the CPU to wait for data, resulting in pipeline stalls. This growing performance mismatch is widely known as the memory wall problem. Fortunately, program behavior is not entirely random. Many applications repeatedly access the same data during execution, for instance, elements of an array that are processed multiple times. This behavior is referred to as temporal locality, where recently accessed data is likely to be accessed again in the near future.

¹For example, a recent Intel i7 processor can generate two 64-bit memory references per core. With eight cores running at 3.4 GHz, this amounts to tens of billions of memory references per second, yielding a theoretical peak bandwidth of approximately 870.4 GB/s.

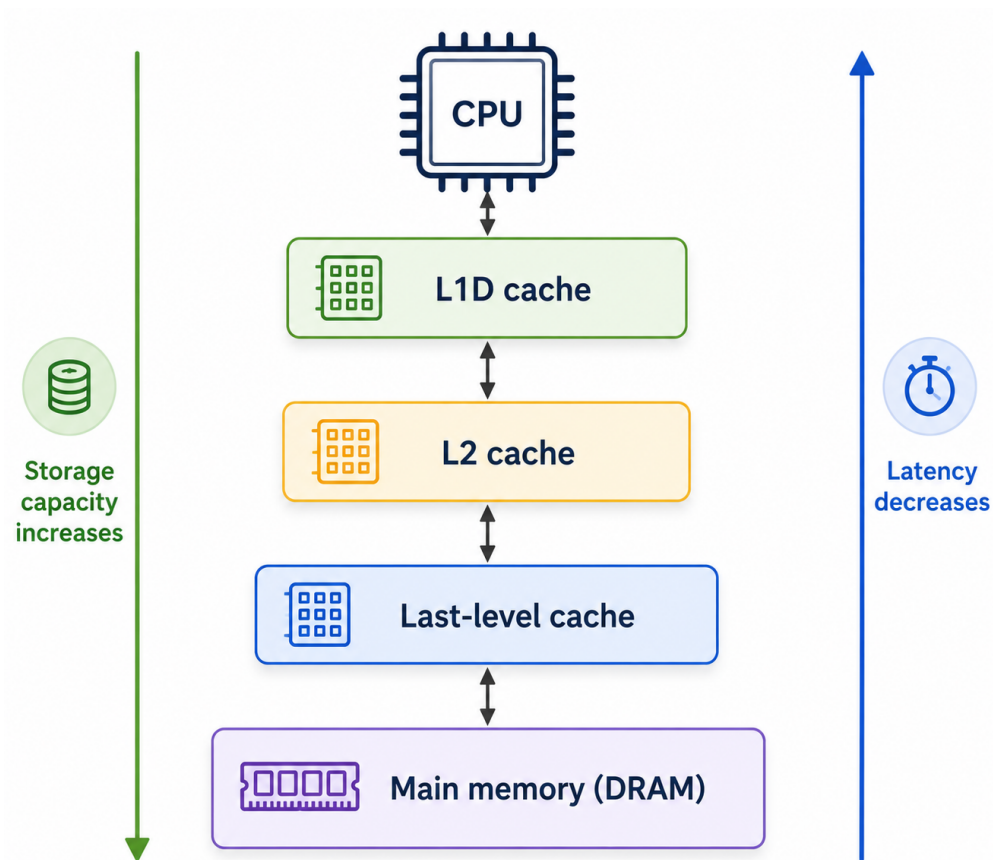


Figure 2.1: Memory hierarchy where storage capacity increases toward lower levels, while access speed increases toward higher levels.

Additionally, programs often access memory locations that are physically close to one another. This tendency, called spatial locality, implies that neighboring addresses are likely to be referenced soon after each other. These two forms of locality underpin the design of modern memory hierarchies and play a crucial role in reducing the impact of the memory wall.

2.1.1 Cache Hierarchy

One of the most effective responses to the processor–memory speed imbalance has been the introduction of cache memory. Instead of relying solely on main memory, systems incorporate small on-chip storage structures that operate at much higher speeds. These caches act as an intermediate layer, keeping recently or frequently accessed data closer to the processor so

that future accesses can be completed with minimal delay.

Caches store information in fixed-size units known as blocks (or cache lines). When the processor requests data, the cache checks whether the corresponding block is already present. If it is, the request is satisfied immediately—an event referred to as a cache hit. The time required to return data in this case is called the hit time, which is typically only a few processor cycles. However, because caches are designed to be extremely fast, they are also relatively small. As a result, they cannot accommodate the entire set of memory locations actively used by a program, commonly known as the program’s working set. When the requested block is absent from the cache, a cache miss occurs. The system must then retrieve the block from main memory (DRAM), a much slower operation that incurs what is known as the miss penalty. To quantify the performance of such a system, architects rely on a widely used metric called Average Memory Access Time (AMAT). This measure captures both fast hits and expensive misses in a single expression:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Designing a cache involves balancing competing factors. Enlarging the cache generally reduces the miss rate because more of the working set can be retained on-chip. However, increasing size often leads to longer access latency and higher energy consumption. This creates a fundamental design tension: a smaller cache offers extremely low latency but suffers from more frequent misses, whereas a larger cache reduces misses at the cost of slower access time. Modern systems resolve this trade-off by adopting a multi-level cache hierarchy rather than relying on a single cache structure. The first-level cache (L1) is positioned closest to the processor and is optimized for minimal latency to keep up with the CPU’s clock frequency. Behind it sits the second-level cache (L2), which is larger and slightly slower, absorbing many of the misses generated at L1. In many architectures, a third level—often called the Last Level Cache (LLC)—provides additional storage capacity before memory requests are forwarded to DRAM. Within this hierarchy, the level closest to the processor is considered the lowest level (L1), while levels farther away are regarded as progressively higher. The LLC, being nearest to main memory, forms the final on-chip buffering layer. A typical organization of this memory hierarchy, along with representative size and latency trends across levels, is illustrated in Figure 2.1.

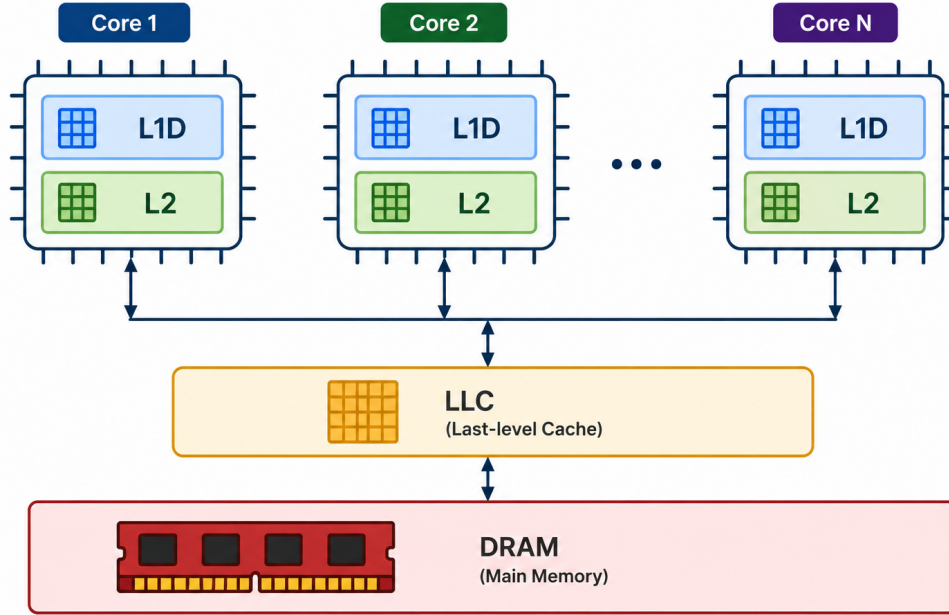


Figure 2.2: Each core has private L1 and L2 cache, whereas the last-level cache is shared across all cores.

2.1.2 Shared last-level cache

In a typical multi-core processor, the first two levels of cache, L1 and L2, are private to each core, meaning that each core maintains its own dedicated copy of these caches. In contrast, the Last Level Cache (LLC) is commonly shared among all cores in the system as shown in Figure 2.2. The primary motivation for sharing the LLC is to improve overall cache utilization. In heterogeneous or workload-diverse multi-core environments, different cores often exhibit varying cache demands. Some cores may run memory-intensive applications with high cache requirements, while others may execute lighter workloads that utilize only a small portion of their allocated cache space. If the last-level cache were private among cores, unused capacity in one core could not be leveraged by another core experiencing higher demand. By allowing all cores to access a common LLC, the system enables more flexible and efficient use of cache resources. Cores with greater memory pressure can effectively utilize more of the shared capacity, while cores with lower demand naturally consume less. This sharing improves overall performance by reducing unnecessary misses that might otherwise occur due to rigid cache allocation. Another important advantage of sharing is

that it facilitates data sharing between cores. When multiple cores operate on shared data structures, a shared LLC can reduce redundant data replication across private caches and improve coherence efficiency.

2.2 Last-Level Cache Attacks

Caches are designed to reduce the high latency associated with off-chip memory accesses. When a processor core requests data and the required block is already present in the cache, the access is completed quickly. In contrast, if the data is absent from the cache, a cache miss, the request must be forwarded to lower levels of the memory hierarchy, potentially all the way to DRAM. Such accesses incur significantly higher latency. Consequently, cache hits and cache misses exhibit observable timing differences.

Under normal circumstances, these timing variations are simply performance characteristics of the memory system. However, in shared cache environments, particularly at the LLC, these latency differences can unintentionally expose information. Since the LLC is shared among multiple cores, the cache activity of one core can influence the cache state observed by another core. By carefully measuring access times, a core can infer whether certain data was recently accessed or evicted by another core. This timing-based observation mechanism forms the basis of what are known as cache timing channels. Over the past decade, numerous studies have demonstrated how such timing channels can be exploited to extract sensitive information [21, 23, 39]. Depending on the threat model and the relationship between the communicating entities, these exploits are broadly categorized as side-channel attacks or covert-channel attacks

A *covert channel* refers to a communication mechanism that deliberately exploits shared system resources in ways they were not originally designed to support. Instead of using legitimate inter-process communication mechanisms, two cooperating processes belonging to *different security domains* encode and transmit information through shared hardware structures, thereby bypassing the intended isolation enforced by the system. In the context of caches, one process (the sender) intentionally manipulates the cache state, for example by accessing specific memory locations or occupying cache resources, to encode information. Another cooperating process (the receiver), executing in a different security domain, observes changes in the shared cache through timing measurements and decodes the transmitted information. Although covert channels enable unauthorized communication across security boundaries, they rely on active cooperation between the sender and receiver.

In contrast, a **side-channel attack** results in a direct breach of confidentiality, where an adversary extracts sensitive information from a victim process without its knowledge or cooperation. Although side channels also exploit hardware resources in unintended ways—such as timing behavior, cache states, or other microarchitectural effects—the key distinction lies in the absence of collaboration between the involved parties. In a side-channel scenario, the victim performs normal computations, unaware that its execution is influencing shared system resources. The attacker passively observes measurable effects, such as variations in access latency, to infer confidential information. Unlike covert channels, where both sender and receiver deliberately encode and decode data, a side-channel attack involves only a malicious observer extracting information from an unsuspecting target.

Therefore, while both covert and side channels rely on unintended communication mediums, side-channel attacks pose a more serious security threat because they directly undermine system confidentiality. When considering the Last Level Cache (LLC), side-channel attacks can be classified based on the mechanism through which the attacker observes or manipulates the cache state. Since the LLC is shared across cores, it provides a common resource through which microarchitectural interactions can be monitored. Depending on how the attacker leverages this shared structure, LLC-based side-channel attacks are broadly categorized into three groups: (i) shared-memory based attacks (ii) conflict-based attacks (iii) occupancy-based attacks.

2.2.1 Conflict-Based Attack

Conflict-based attacks exploit the set-associative organization of modern caches and the fact that the Last Level Cache (LLC) is shared across cores. In a set-associative cache, each memory block maps to a specific cache set determined by certain bits of its physical address. Each set can store only a limited number of blocks, equal to the associativity. When this limit is exceeded, one of the existing blocks must be evicted according to the cache’s replacement policy. This inevitable eviction under contention is the core mechanism leveraged in conflict-based attacks.

In this class of attacks, the adversary does not monitor a specific shared cache line. Instead, the attacker focuses on entire cache sets. The idea is to deliberately occupy all ways of selected cache sets and then observe whether any of those lines have been displaced during the victim’s execution. Because both the victim and the attacker access memory that maps to the same cache sets, their activities interfere with one another. By detecting

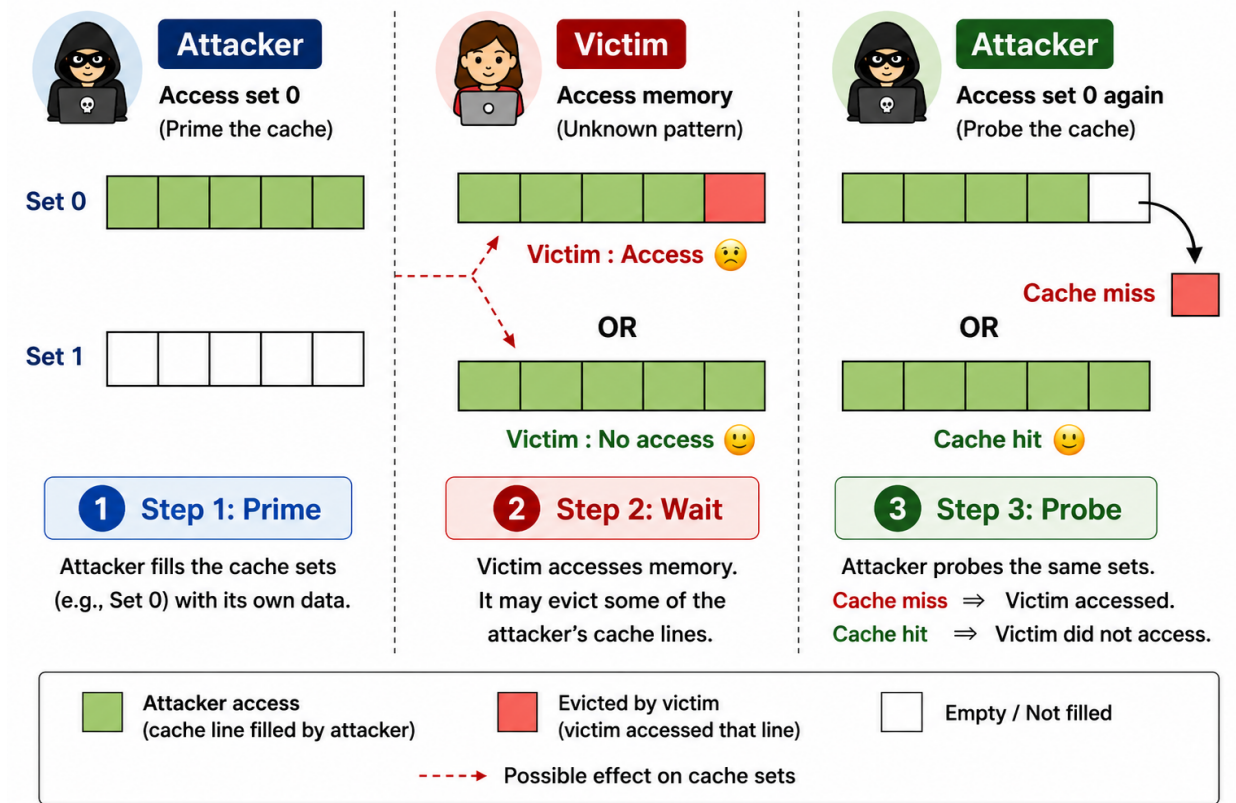


Figure 2.3: An overview of prime+probe attack as a three-step process where attacker primes the set and then waits for the victim’s access, finally probes the set to infer hit/miss to leak necessary information about victim.

such interference through timing measurements, the attacker can infer which cache sets the victim accessed and, consequently, obtain information about its memory access behavior.

Prime+Probe [21] is the most well-known example of a conflict-based attack. Figure 2.3 highlights the attack as three conceptual phases. In the prime phase, the attacker fills one or more target cache sets with its own data by accessing addresses that map to those sets. Since the attacker accesses enough addresses to occupy all available ways, the selected sets become fully populated with the attacker’s cache lines. Next, during the victim’s execution phase, the victim performs its normal computation. If the victim accesses memory addresses that map to any of the same cache sets, the cache’s replacement mechanism will evict some of the attacker’s lines to accommodate the victim’s data. Finally, in the probe phase, the attacker re-accesses the same addresses used in the priming step and measures the access latency. If the access is fast, the corresponding line was not evicted; if it is slow, eviction

has occurred. By observing which sets experience evictions, the attacker can infer patterns in the victim’s memory accesses. Prime+Probe does not require knowledge of the victim’s exact addresses. Instead, it reveals which cache sets are being used, providing information at the granularity of cache-set activity. Depending on how much structural information the attacker has about the cache indexing function, this set-level information can often be correlated with specific memory regions or algorithmic behavior. More advanced variants go beyond simply detecting eviction and attempt to infer changes in replacement state or eviction ordering, thereby extracting finer-grained behavioral information.

Eviction set

A fundamental building block of conflict-based attacks is the eviction set. An eviction set is a collection of memory addresses that all map to the same cache set. Because a cache set can hold only a fixed number of lines—equal to its associativity—accessing a sufficient number of addresses that map to that set guarantees eviction of any other line currently residing there. For example, consider an LLC that is 16-way set-associative. This means each cache set can store at most 16 cache lines simultaneously. If an attacker identifies 16 distinct memory addresses that map to the same cache set and accesses them sequentially, those 16 lines will occupy all ways of that set. Any additional access to a different address mapping to the same set will necessarily evict one of the existing lines. Thus, by repeatedly accessing all addresses in the eviction set, the attacker can reliably clear or overwrite that specific cache set.

Constructing an eviction set requires identifying addresses that collide in the same cache index. Since the mapping from physical addresses to cache sets depends on specific address bits (and sometimes undocumented hashing functions), attackers typically use empirical timing analysis. They allocate a large pool of memory and measure which addresses interfere with one another. Addresses that consistently cause increased access latency—indicating mutual eviction—are grouped together. Through iterative refinement, the attacker obtains a minimal group of addresses that collectively fill all ways of a single cache set. This group constitutes a functional eviction set. Once constructed, eviction sets allow the attacker to create controlled cache contention without relying on privileged instructions. They are therefore central to enabling reliable Prime+Probe-style attacks in real-world systems.

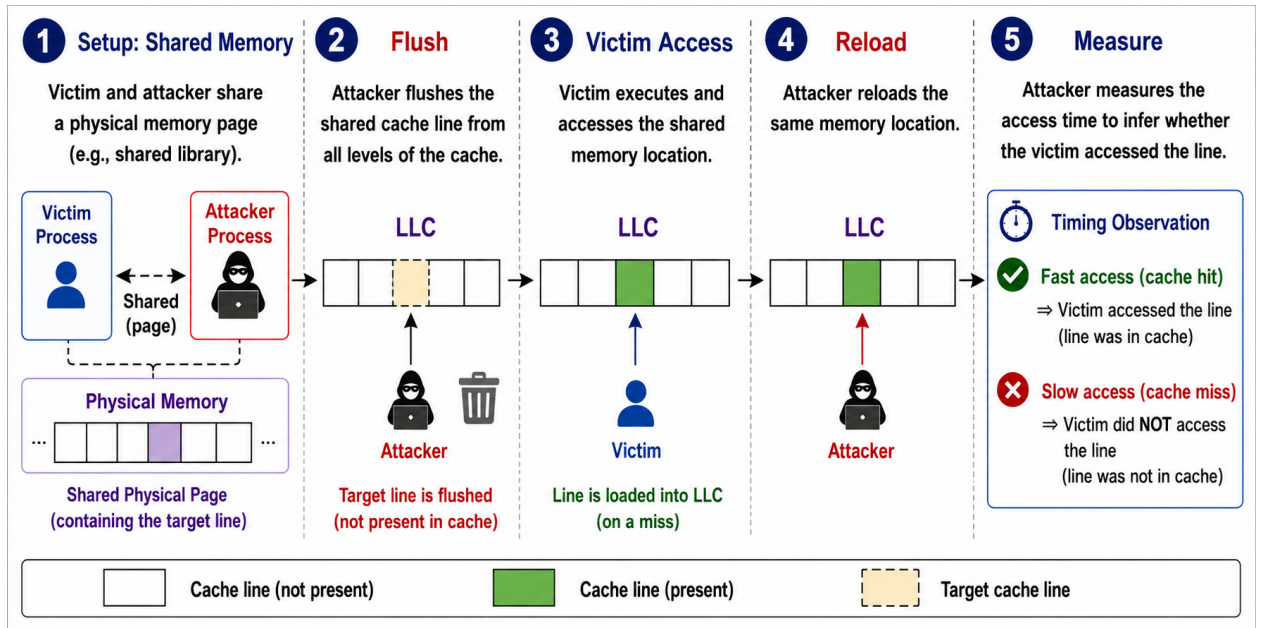


Figure 2.4: An overview of flush+reload attack. The attacker flushes a shared cache line, waits for victim execution, and reloads the same line to infer victim accesses from cache-hit and cache-miss timing differences.

2.2.2 Shared-Memory Based Attack

Shared-memory based cache attacks exploit scenarios in which an attacker and a victim process share at least some portion of physical memory. This sharing typically arises from read-only shared libraries (e.g., common system libraries loaded into multiple processes) or from memory deduplication techniques employed by the operating system, where identical memory pages are merged to conserve space. Because the underlying physical memory is shared, the corresponding cache lines are also shared in the Last Level Cache (LLC).

The key idea behind these attacks is that the attacker (often called the spy) monitors the cache state of shared memory locations to determine whether the victim has accessed them. Since cache hits and misses exhibit measurable timing differences, observing the access latency to a shared cache line reveals whether it was recently used by the victim.

Flush+Reload [39] is a well-known example of this class of attacks, and its workflow is illustrated in Figure 2.4. This attack takes advantage of a hardware-supported cache flush instruction (such as `clflush` in x86 architectures) that invalidates a specific cache line from all levels of the cache hierarchy. The attacker begins by flushing a shared cache line, ensuring that it is absent from the cache. The victim is then allowed to execute. If

the victim accesses that memory location during its execution, the corresponding cache line is automatically fetched back into the cache. The attacker subsequently reloads the same memory address and measures the access latency. A fast access time indicates that the victim accessed the line (resulting in a cache hit), whereas a slow access suggests that the victim did not use it. Because the attacker directly monitors a specific shared line, Flush+Reload provides very fine-grained information, often at cache-line resolution.

A fundamental requirement for all shared-memory attacks is the existence of shared physical cache lines between mutually untrusted processes. Without shared memory, the attacker cannot directly observe whether the victim’s specific data has been cached.

2.2.3 Occupancy-Based attack

Cache occupancy-based attacks [7, 30] take a more coarse-grained approach compared to other LLC-based side-channel techniques. Rather than targeting specific cache sets or individual cache lines, these attacks infer information by monitoring overall changes in cache usage. The underlying observation is that, because the Last Level Cache (LLC) is shared across cores and processes, activity from one process can reduce the effective cache space available to another. By tracking variations in its own cache occupancy over time, an attacker can indirectly estimate the victim’s cache usage patterns.

The key idea is that when a victim process executes and consumes cache capacity, it displaces some of the attacker’s cache lines from the LLC. If the attacker periodically measures how much of its own data remains in the cache, it can approximate how much cache space the victim has used during that interval. Unlike conflict-based attacks, which attempt to identify activity in specific cache sets, occupancy-based attacks focus on aggregate cache pressure. Figure 2.5 illustrates the workflow of a typical occupancy-based fingerprinting attack. Attacker repeatedly fills the LLC with its own data. After thrashing the cache, it waits for the victim to execute. As the victim loads content and performs computations, it consumes cache space, evicting some of the attacker’s lines. The attacker then measures how many of its lines remain cached by re-accessing its data and counting the number of misses. By performing this measurement periodically, the attacker constructs a time-series pattern representing the victim’s cache usage over time. This pattern serves as a “fingerprint” of the website being loaded. By comparing the observed pattern against a pre-collected repository of fingerprints, the attacker can identify which website the victim is visiting (in website fingerprinting) or which application is running on the victim core (in application fingerprinting).

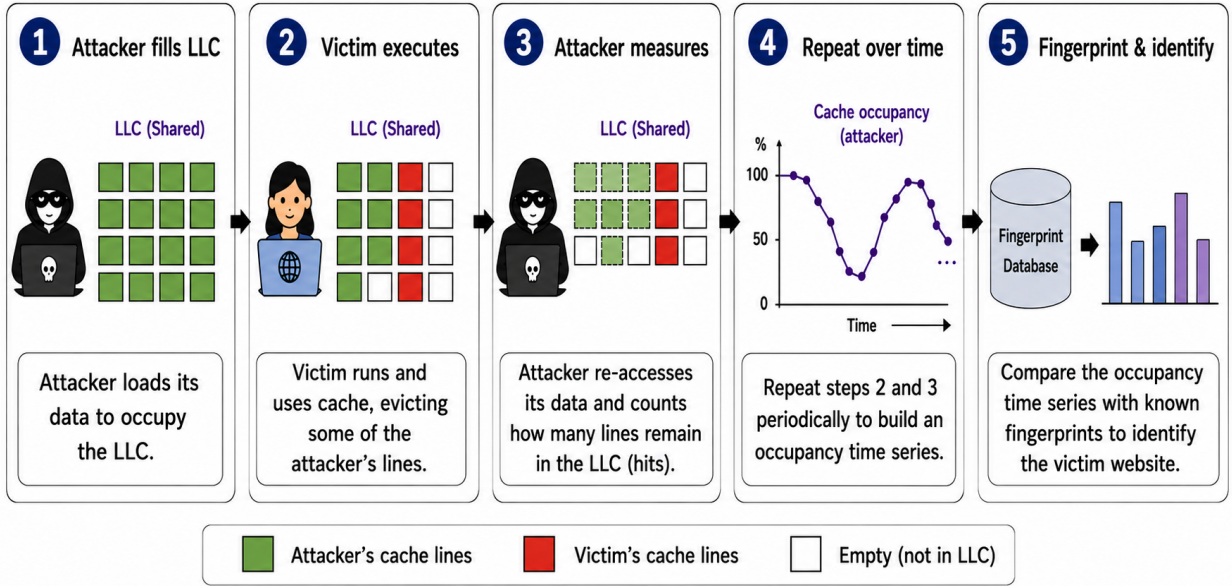


Figure 2.5: An overview of occupancy-based attack. The attacker fills the LLC with its own data, measures cache occupancy changes caused by victim’s execution, and constructs a time-series fingerprint to identify victim’s activity.

Subsequent work demonstrated that such fingerprinting attacks can be mounted with minimal capabilities—requiring only standard HTML and CSS code executed within a malicious web page in the same browser. Even without high-resolution timers or sophisticated scripting, observable differences in cache occupancy can be exploited. As a result, these attacks remain effective despite browser-level defenses such as site isolation mechanisms that separate web pages into distinct processes, or restrictions that limit the precision of JavaScript timers. The fundamental vulnerability arises not from direct memory sharing, but from the dynamic sharing of the LLC’s capacity.

A defining characteristic of occupancy-based attacks is that they require only that the attacker and victim share the LLC resource at runtime. No shared memory pages or precise knowledge of cache indexing is necessary. As long as both processes compete for the same cache capacity, the attacker can infer information from variations in its own cache retention.

2.2.4 Security Implications of Process Fingerprinting

Process fingerprinting poses significant security and privacy risks in shared computing environments by allowing adversaries to infer high-level application behavior from low-level

hardware observations. Such information can reveal the identity of running applications, expose sensitive workloads (e.g., cryptographic applications), and leak execution phases, thereby compromising user privacy. In multi-tenant environments, fingerprinting further violates isolation guarantees by enabling the detection of co-located workloads across security domains. More importantly, fingerprinting often serves as a reconnaissance step for advanced side-channel attacks, allowing attackers to identify suitable targets and precisely time attacks such as Prime+Probe, thereby significantly improving their effectiveness. Although fingerprinting alone may not directly disclose secret data, it substantially amplifies the impact of other microarchitectural attacks by reducing the attacker’s uncertainty about the victim’s execution. Overall, process fingerprinting undermines the abstraction between hardware and software, enabling adversaries to infer high-level program behavior from low-level signals. This highlights the need for stronger isolation and obfuscation mechanisms in modern computing systems.

Chapter 3

Related works

3.1 Mitigating LLC Attacks

This chapter explores the various defense mechanisms proposed to counter the three classes of LLC-based attacks discussed earlier: shared-memory attacks, conflict-based attacks, and occupancy-based attacks. Since all these attacks fundamentally exploit contention and shared resource behavior in the Last Level Cache (LLC), most mitigation strategies aim to either obscure cache behavior or enforce stronger isolation among competing processes. Broadly, proposed defenses can be categorized into two major approaches: randomization-based techniques and partitioning-based techniques. Both approaches attempt to limit the attacker’s ability to observe or manipulate cache state, but they differ significantly in philosophy and effectiveness.

A substantial body of prior work has focused on LLC randomization techniques [5, 14]. These techniques typically randomize aspects of cache organization, such as address-to-set mapping or replacement behavior. The primary objective is to make it difficult for an attacker to predict cache indexing functions and construct reliable eviction sets. Since conflict-based attacks, such as Prime+Probe, rely heavily on precise knowledge of cache set mappings, randomization increases uncertainty and complicates the attacker’s ability to create controlled contention. In particular, randomized indexing functions significantly raise the difficulty of building eviction sets, which are central to mounting conflict-based attacks. However, while randomization can disrupt fine-grained set-level attacks, it does not eliminate contention itself. In occupancy-based attacks, the attacker does not require precise knowledge of cache sets. Instead, it monitors aggregate cache usage. Because randomization does not prevent dynamic sharing of cache capacity, these techniques remain vulnerable to occupancy-based

attacks. An exception is SassCache [14], which was specifically designed to mitigate both conflict-based and occupancy-based cache timing attacks by combining randomized indexing with additional cache management mechanisms. Although these design choices substantially improve security, they also introduce significantly greater implementation complexity compared to conventional randomization techniques. Thus, randomization primarily increases attack complexity rather than fully eliminating leakage. In contrast, LLC partitioning techniques aim to enforce strict spatial isolation within the cache. Partitioning mechanisms allocate disjoint regions of the LLC to different processes or security domains, thereby preventing mutual eviction and contention. By ensuring that one process cannot evict or displace another’s cache lines, partitioning directly mitigates contention-based leakage. State-of-the-art partitioning approaches are designed to provide isolation guarantees that prevent all forms of LLC contention-based attacks, including conflict-based and occupancy-based techniques, by eliminating shared cache space between mutually distrusting entities.

3.2 Static Partitioning

Although this thesis focuses on secure cache partitioning, cache partitioning was originally proposed to improve system performance rather than security. In multi-core processors, co-running applications often exhibit diverse cache demands. Allowing memory-intensive applications to utilize a larger fraction of the shared Last-Level Cache (LLC), while restricting cache-insensitive applications, can reduce destructive cache interference and improve overall system throughput. Several hardware and software techniques have exploited this idea. For example, Intel’s Cache Allocation Technology (CAT) enables software to reserve different portions of the LLC for different applications or virtual machines, providing quality-of-service (QoS) guarantees and reducing interference among co-running workloads. Similarly, cache partitioning has been widely used in real-time systems to reserve cache resources for latency-critical tasks, thereby improving execution predictability while preventing best-effort applications from degrading their performance. Beyond performance optimization, researchers later recognized that cache partitioning can also provide strong security benefits. By spatially isolating mutually distrusted applications within the LLC, cache partitioning naturally eliminates many forms of cache contention that are exploited by cache timing attacks. Consequently, cache partitioning evolved from a performance optimization technique into an effective defense mechanism against cache side-channel attacks.

Early security-oriented cache partitioning proposals relied on *static cache partitioning*.

The fundamental idea behind static partitioning is to divide the LLC into disjoint regions and permanently assign each region to a specific security domain. This isolation can be enforced either through way partitioning, where cache ways are exclusively assigned, or set partitioning, where different domains are mapped to separate cache sets. Because domains do not overlap in cache space, static partitioning provides strong and straightforward security guarantees by eliminating cross-domain interference. However, this strict isolation comes at a significant performance cost. Static partitioning prevents flexible cache sharing, often leading to inefficient LLC utilization. When one domain underutilizes its allocated cache space, other domains cannot reclaim or use the idle capacity. As a result, overall performance may degrade substantially, particularly under dynamic and heterogeneous workloads. In effect, static partitioning undermines the original architectural motivation for a shared LLC—efficient and adaptive resource sharing.

To address this limitation, modern state-of-the-art designs have shifted toward dynamic cache partitioning. Instead of permanently dividing the LLC, dynamic partitioning adjusts cache allocations at runtime based on application behavior or declared demand. The primary objective is to strike a balance between security and performance: preserve isolation properties where necessary while retaining the efficiency benefits of shared cache utilization. By adapting allocations over time, dynamic partitioning seeks to provide a practical middle ground—a “sweet spot” between strong security guarantees and high performance efficiency.

Given this evolution in design philosophy, dynamic partitioning has emerged as the practical direction for secure cache architectures. Static partitioning, while secure, is often considered overly conservative due to its significant performance trade-offs. In contrast, dynamic policies aim to retain the performance advantages of a shared LLC while introducing mechanisms to mitigate security risks. Therefore, in this work, we focus exclusively on dynamic partitioning techniques, as they represent the modern state-of-the-art and embody the central design challenge of balancing security with performance. Our goal is to critically evaluate whether these dynamic mechanisms, despite their adaptive and randomized designs, can truly uphold their intended security guarantees under adversarial observation.

3.3 Dynamic Partitioning

To mitigate Last-Level Cache (LLC) side-channel attacks, several dynamic partitioning techniques have been proposed that distribute LLC space among mutually distrusted domains at runtime. The fundamental objective of these approaches is to ensure that two adversar-

ial domains do not simultaneously access the same cache region, thereby enforcing spatial isolation and preventing information leakage through shared cache contention. Unlike static partitioning, which allocates fixed cache regions irrespective of workload behavior and often incurs significant performance degradation due to underutilization, dynamic techniques aim to provide both security isolation and performance efficiency by adapting allocations over time. Over the years, numerous proposals have adopted this dynamic strategy to reduce the performance penalty traditionally associated with static cache partitioning. These techniques introduce runtime adaptability by reallocating cache resources based on application behavior, system state, or explicit domain requirements. While they claim to provide strong isolation guarantees, it remains unclear whether such dynamism introduces subtle security vulnerabilities. Therefore, in this work, we systematically examine the design principles underlying modern dynamic partitioning mechanisms to evaluate whether their security guarantees truly hold under adversarial conditions.

Through our analysis, we observe that state-of-the-art dynamic partitioning techniques can be broadly categorized into two groups based on how they determine and enforce LLC space allocation. In the first category, the control of cache allocation is largely driven by the application (or domain) itself, typically with support from the operating system or hardware mechanisms. The application explicitly requests a certain amount of cache space, and the system enforces isolation accordingly. Techniques such as BCE [15], TeeSHirT [4], Ceviche Cache [17], and Composite Cachlet [32] fall under this category. In these proposals, cache allocation decisions are driven by application requests. The requested cache allocation may be determined through offline profiling or programmer annotations rather than being explicitly known by the application itself. The system then dynamically allocates isolated cache regions to satisfy these requests while maintaining spatial isolation among protection domains. Although application-guided allocation provides fine-grained control over cache resources, it may require cache requirements to be estimated or profiled for the target workload. Consequently, the effectiveness of the requested allocation can depend on the underlying hardware configuration and workload characteristics. However, the allocation mechanism itself is independent of a particular hardware platform, as the operating system or runtime can adapt the requested cache allocation to the available cache resources.

The second category adopts a fundamentally different approach. Instead of relying on explicit requests from applications, these techniques monitor runtime cache usage patterns and dynamically distribute LLC space to optimize overall performance, typically by minimizing total cache misses across domains. In this model, cache ways or sets are reallocated

periodically based on observed utility metrics, such as marginal reduction in misses per additional way. The underlying goal is to achieve near-optimal performance while still maintaining isolation boundaries between domains. Allocation decisions are therefore driven by measured cache behavior rather than explicit application requests. We refer to this class as *LLC-Usage-Based Dynamic Partitioning*. This categorization highlights an important design distinction: while both classes enforce isolation dynamically, they differ in who drives the allocation decision, the application itself or the runtime cache-usage characteristics. In the remainder of this study, we analyze whether these distinct approaches to dynamism introduce new security implications, particularly in adversarial settings where malicious domains may manipulate allocation mechanisms to infer sensitive information.

3.3.1 Application-Demand-Based Technique

Application-demand-based dynamic partitioning techniques [4, 17, 29, 32] are built on the principle that cache isolation should be explicitly controlled by the protection domains themselves. In this class of designs, security-sensitive applications (or trusted runtime components acting on their behalf) specify their cache requirements, and the system dynamically allocates dedicated LLC regions accordingly. The objective is to ensure strong spatial isolation between mutually distrusted domains while retaining flexibility to adapt allocations over time. These techniques typically integrate hardware support (e.g., way partitioning, set partitioning, cache coloring, or secure enclaving primitives) with operating system coordination to enforce strict ownership of cache regions. Once a portion of the LLC is allocated to a domain, other domains are prevented from accessing that region concurrently. Unlike rigid static schemes, these mechanisms introduce dynamism by allowing allocations to be adjusted at runtime, either in response to application phase changes, varying workload demands, or security policies. As a result, the system can better utilize available cache resources without permanently dedicating fixed partitions.

A defining characteristic of this category is the notion of explicit reservation: isolation is achieved by honoring application-level requests and enforcing non-overlapping cache assignments. The effectiveness of such approaches depends on several architectural and system-level design choices, including how applications communicate their cache demands, how the system arbitrates competing requests, how reallocation is performed, and whether transitions between allocations preserve strict isolation guarantees. In the following subsections, we examine representative techniques in this category, detailing their allocation mechanisms,

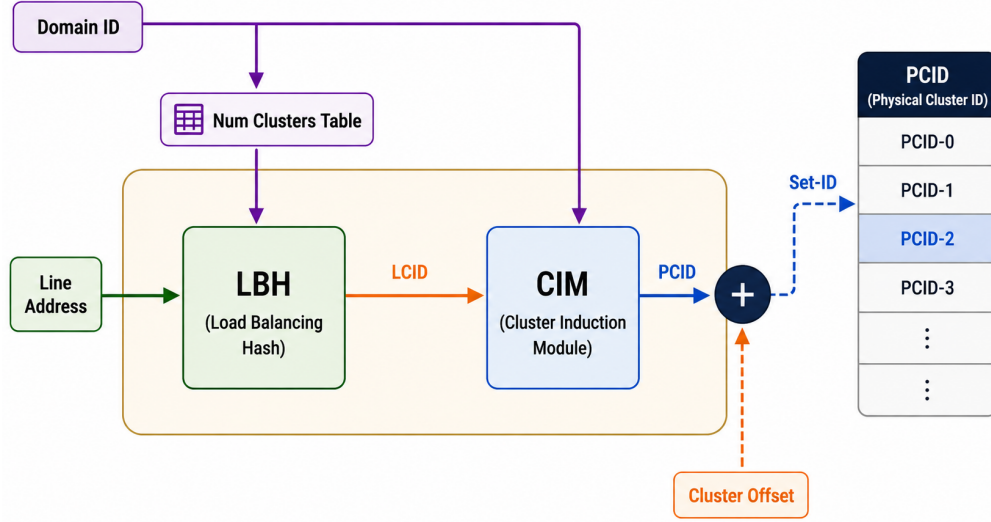


Figure 3.1: BCE uses domain identifiers to ensure that incoming cache lines are mapped only to the clusters allocated to their respective domains. It also maintains an allocation table that tracks the number of LLC clusters assigned to each domain.

enforcement strategies, and security implications.

Bespoke Cache Enclave (BCE)

Bespoke Cache Enclave (BCE) [29] is a set-partitioning technique designed to provide strong isolation in the Last Level Cache (LLC) while overcoming limitations of earlier partitioning approaches such as way partitioning and page coloring. Unlike way partitioning, which divides the cache along the associativity dimension, BCE partitions the LLC at the granularity of *clusters*, where each cluster consists of a group of contiguous cache sets. In BCE, the LLC is logically divided into multiple clusters, and each cluster represents a fixed number of adjacent cache sets. For example, if the cluster size is chosen to be 64 sets, then a 32 MB, 16-way LLC containing 32,768 sets can be divided into:

$$\frac{32,768}{64} = 512 \text{ clusters.}$$

Each cluster can be exclusively assigned to a distinct security domain. Since clusters do not overlap, no two domains share the same cache sets, thereby eliminating inter-domain set contention. With 512 clusters, the system can theoretically support up to 512 isolated domains simultaneously, assuming one cluster per domain.

An important advantage of BCE over page coloring is that it *decouples cache allocation*

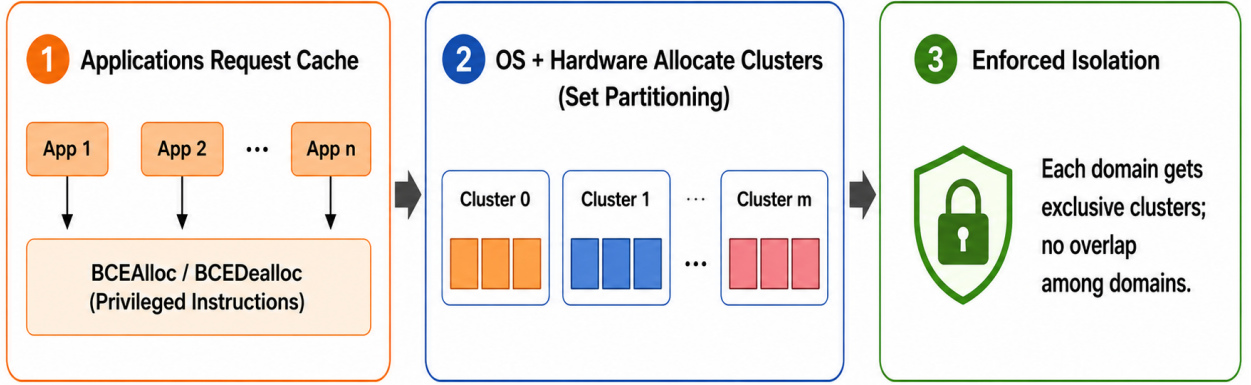


Figure 3.2: BCE exposes two privileged instructions (BCEAlloc/BCEDealloc) for domains to manage their clusters, in order to do so, it takes support from OS.

from DRAM allocation. In page coloring, cache isolation is achieved indirectly through physical page placement, which tightly couples memory capacity and cache capacity. In contrast, BCE directly partitions cache sets at the hardware level, allowing cache space to be allocated independently of memory allocation. Furthermore, unlike way partitioning approaches such as DAWG [18], BCE is not limited by the number of cache ways, making it more scalable in terms of the number of supported domains.

Enforcement Policy and Cache Indexing. Figure 3.1 illustrates how an incoming line address from a particular domain is enforced only to the set allocated to it. To enforce cluster-based isolation, BCE assigns each cluster in the LLC a unique identifier called a *Physical Cluster ID (PCID)*. This identifier represents the actual physical location of the cluster in the cache. When a process belonging to a specific security domain issues a memory access, the address is processed to generate a *Logical Cluster ID (LCID)* using a hashing mechanism. The LCID is then translated to the corresponding PCID, ensuring that the access is restricted to clusters allocated to that domain. This mapping mechanism guarantees that cache lookups, evictions, and replacement decisions occur only within the domain’s assigned clusters, preventing cross-domain interference. A key feature of BCE is its flexibility in cluster allocation. Domains are not restricted to a fixed number of clusters; instead, clusters can be dynamically allocated or deallocated based on workload requirements. This enables fine-grained control over cache capacity and allows the system to tailor cache allocation according to the performance and security needs of each domain.

Operating System Support. BCE relies on cooperation from the operating system (OS) to manage cluster allocation as shown in Figure 3.2. The hardware exposes software

interfaces that allow the OS to request isolated LLC partitions of specific sizes. To facilitate this, BCE introduces two privileged instructions:

- **BCEAlloc**: Allocates one or more clusters to a security domain.
- **BCEDealloc**: Releases previously allocated clusters.

These instructions are accessible only to trusted software such as the OS kernel. The OS is responsible for managing cluster ownership and enforcing domain isolation. It may further expose system calls to applications, enabling them to request customized LLC allocations based on their performance or security requirements. By combining hardware-enforced set partitioning with OS-level management, BCE provides flexible and secure cache isolation while allowing dynamic and efficient utilization of LLC capacity.

TEE-SHirT

TEE-SHirT [4] is a secure cache hierarchy designed for Trusted Execution Environments (TEEs) that prevents cache side-channel attacks through hardware-enforced cache partitioning. Instead of allowing enclaves and non-enclave applications to share cache resources, TEE-SHirT dynamically allocates isolated cache partitions to each security domain using a hardware Cache Partition Management Unit (CPMU). The design further incorporates partition sanitization, cache coherence support, and virtualization of partition metadata to efficiently support multiple enclaves while maintaining strong isolation. By combining dynamic partition allocation with hardware-enforced ownership, TEE-SHirT eliminates contention between mutually distrusted domains and prevents leakage through shared cache resources.

Ceviche Cache

Ceviche Cache [17] is a capability-based secure cache architecture that provides fine-grained isolation by associating every cache line with a hardware capability. Instead of relying solely on cache partitioning, Ceviche enforces access control through capability tokens that determine which domain can access a particular cache line. Cache space is allocated dynamically based on application demand by granting capabilities that define a domain's cache quota, while hardware ensures that only authorized domains can insert, access, or evict cache lines. This combination of explicit allocation and hardware-enforced ownership provides strong spatial isolation while maintaining flexibility and efficient cache utilization.

Composable Cachelets

Composable Cachelets (CC) [32] is a fine-grained hardware cache partitioning mechanism that divides the last-level cache into small allocation units called cachelets. These cachelets are dynamically assigned to enclaves or security domains and can be composed to form larger virtual cache partitions as application requirements change. Hardware manages allocation, remapping, and replacement within these cachelets, ensuring that mutually distrusted domains never share cache resources simultaneously. By combining dynamic allocation with fine-grained hardware isolation, Composable Cachelets eliminate cache interference while achieving higher cache utilization than conventional way-based partitioning techniques.

3.3.2 LLC-Usage-Based Techniques

LLC-usage-based dynamic partitioning techniques allocate cache resources according to the observed runtime behavior of applications rather than explicit application requests. In this class of designs, the system continuously monitors cache utilization metrics — such as miss rates, marginal utility per cache way, reuse behavior, or occupancy patterns — and dynamically redistributes LLC space to improve overall efficiency while maintaining isolation between mutually distrusted domains. The central objective is to strike a balance between security and performance: domains are assigned disjoint cache regions, but the size of these regions is periodically adjusted based on measured demand and predicted benefit.

These techniques typically operate in epochs. During each epoch, the hardware collects per-domain statistics and evaluates the utility of assigning additional cache ways or sets. At the end of the monitoring interval, a reallocation decision is made — often guided by heuristics that aim to minimize total cache misses or maximize system throughput. The reallocation is then enforced by updating way masks, partition tables, or other hardware control structures to ensure that domains do not overlap in their assigned regions. By adapting to workload phase changes and varying working-set sizes, LLC-usage-based approaches seek to avoid the inefficiencies of fixed partitioning while preserving spatial separation across domains. A defining characteristic of this category is that allocation decisions are policy-driven and measurement-based rather than application-declared. Consequently, the security and robustness of these schemes depend critically on the correctness of monitoring mechanisms, the stability of utility estimation, and the safety of transitions during reallocation. In the following subsections, we examine representative techniques in this category, analyze how they compute allocation decisions, and evaluate the implications of their dynamic behavior

on isolation guarantees.

SCALE: Secure and Scalable Cache Partitioning

Traditional cache partitioning mechanisms rely on two fundamental components: (i) an *allocation policy*, which determines how cache space is divided among applications, and (ii) an *enforcement policy*, which ensures that applications access only their assigned cache regions. While many prior works focused on maximizing performance through intelligent allocation, the security implications of allocation decisions have received comparatively less attention. However, allocation decisions themselves can act as a side channel, since observable changes in partition sizes may leak information about application behavior. To improve both cache utilization and security, SCALE dynamically allocates cache ways according to the runtime cache utility of each application. The allocation decisions are based on Utility-Based Cache Partitioning (UCP), which estimates the benefit of assigning additional cache resources to individual applications.

Specifically, UCP employs *Auxiliary Tag Directories (ATDs)* together with utility monitors to estimate the marginal reduction in cache misses that would result from allocating additional cache ways to an application. The ATDs emulate multiple cache allocations by tracking only cache tags, enabling the hardware to estimate the utility of additional cache capacity without storing actual cache data. Based on these utility estimates, a lookahead algorithm periodically computes an efficient cache allocation for all applications. Unlike hardware performance counters (HPCs), which provide aggregate execution statistics such as cache misses or instructions retired, ATDs continuously estimate the expected performance benefit of different cache allocations. SCALE then combines these utility estimates with differential privacy and randomized allocation mechanisms to improve cache utilization while mitigating information leakage through predictable allocation decisions.

Utility-Based Cache Partitioning (UCP)

Utility-Based Cache Partitioning (UCP) is a performance-driven dynamic partitioning mechanism that aims to maximize overall system throughput. UCP operates by estimating the marginal utility of assigning additional cache ways to each application. It uses *utility monitors* to track how many misses would be reduced if an application were given extra cache capacity. At periodic intervals, UCP employs a *Lookahead algorithm* that evaluates these marginal utilities and computes an allocation of cache ways that maximizes aggregate per-

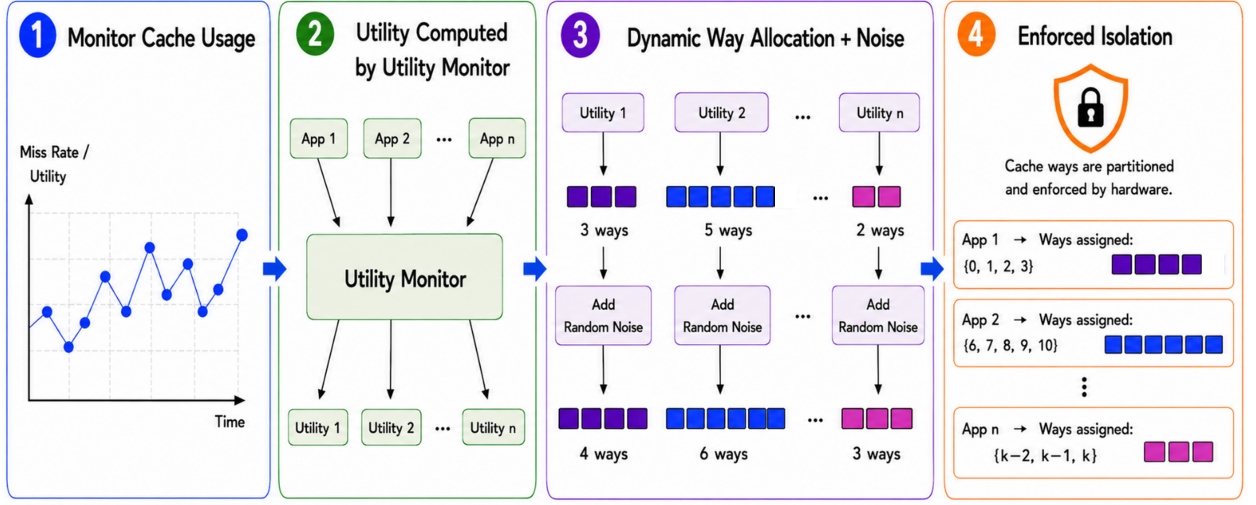


Figure 3.3: SCALE exploits UCP to find utility of each application and it adds random noise derived from zero-mean laplace distribution to these allocation to remove the tackle the predictability of UCP.

formance. The allocation is deterministic: given the same utility measurements, UCP will always produce the same partitioning decision. Although UCP is highly effective from a performance standpoint, it does not consider security. Its deterministic nature introduces predictability into allocation transitions. Because cache partition sizes directly influence observable behavior such as hit rate, miss rate, and occupancy, an attacker can monitor these changes and potentially correlate them with victim activity. In adversarial environments, this makes UCP *unsafe*, as it may leak sensitive information through allocation-based side channels.

Allocation Policy

SCALE builds upon the performance foundation of UCP but introduces security into the allocation process. The central insight behind SCALE is that deterministic allocation decisions enable inference attacks. To mitigate this vulnerability, SCALE injects controlled randomness into the allocation process while preserving performance benefits. As shown in Figure 3.3, SCALE retains UCP’s utility monitors and Lookahead mechanism to compute an ideal performance-oriented allocation. However, instead of directly applying this deterministic allocation, SCALE modifies it using a multi-layered randomized defense.

Three-Layer Randomized Defense SCALE incorporates three complementary layers of randomization:

1. **Allocation Noise Injection:** Random perturbations are added to the allocation suggested by the deterministic Lookahead algorithm. Rather than assigning exactly the computed number of ways, SCALE slightly adjusts allocations within a bounded range (Figure 3.3). This masks precise allocation decisions and reduces information leakage.
2. **Randomized Allocation Change Rate:** The magnitude of allocation adjustments between reconfiguration periods is randomized. Instead of allowing abrupt and predictable transitions, changes occur gradually and unpredictably over multiple intervals, increasing uncertainty.
3. **Randomized Reconfiguration Timing:** The timing of allocation updates is randomized. Reconfiguration intervals are not strictly periodic; instead, they occur at unpredictable times. This prevents attackers from correlating observed cache behavior with specific allocation events.

Together, these mechanisms introduce uncertainty across spatial (which ways), magnitude (how much change), and temporal (when change occurs) dimensions. By eliminating deterministic allocation patterns, SCALE mitigates both conflict-based and occupancy-based attacks that rely on predictable partition behavior.

Enforcement Policy and Indexing

In addition to secure allocation, SCALE provides hardware-level enforcement to guarantee isolation between domains.

When a core issues a memory request, it first consults a *Cluster Bank Table (CBT)* to determine the relevant cache bank corresponding to the target set index. The CBT maps set indices to banks within the LLC. After identifying the bank, the core consults a *Way Partition Table (WPT)*. The WPT maintains domain-specific allocation information and specifies which cache ways within the bank are assigned to each domain. Using the domain ID as a key, the WPT restricts access to only the allocated ways. This ensures that cache hits, evictions, and replacement updates occur strictly within the assigned partition.

To prevent leakage through replacement metadata or residual state, SCALE enforces a strict sanitization policy. When a cache way is reassigned from one domain to another,

all associated replacement state is flushed. This guarantees that no historical access information remains that could leak across domains. Overall, SCALE transforms a deterministic, performance-driven partitioning policy into a secure, scalable, and locality-aware framework. By combining utility-based monitoring with multi-layered randomization and hardware-enforced isolation, SCALE protects the LLC against timing-based side-channel attacks while preserving high performance.

3.3.3 Selection of Representative Techniques

As discussed in the previous sections, existing secure dynamic cache partitioning techniques can be broadly classified into two categories based on how cache allocation decisions are made: application-demand-based techniques and LLC-usage-based techniques. Although several proposals have been developed within each category, many of them share similar architectural principles and allocation strategies.

Among application-demand-based techniques, we select BCE as the representative design because it provides fine-grained dynamic set partitioning through explicit allocation and deallocation primitives while enforcing strong spatial isolation between protection domains. Moreover, several subsequent techniques, including TEE-SHirT, Composable Cachelets, and Ceviche Cache, follow a similar design philosophy in which applications (or the operating system) explicitly manage cache resources. Consequently, analyzing BCE enables us to study the security implications of the broader class of application-demand-based partitioning techniques. For the LLC-usage-based category, SCALE was selected because, to the best of our knowledge, it is the only secure dynamic cache partitioning technique that dynamically determines cache allocations based on runtime LLC usage while incorporating differential privacy and randomized allocation mechanisms to defend against occupancy-based attacks. To achieve this, SCALE injects carefully calibrated random noise into the utility values computed by UCP before determining the final cache allocation. This random perturbation, referred to as *differential privacy noise*, is sampled from a Laplace distribution and obscures small variations in application cache demand. As a result, consecutive allocation decisions become statistically difficult to distinguish while preserving the overall utility estimates required for efficient cache allocation. By leveraging Utility-Based Cache Partitioning (UCP), SCALE continuously reallocates cache resources according to application demand, thereby improving overall cache utilization and system performance while maintaining security. Unlike application-demand-based schemes, SCALE performs autonomous hardware-

driven cache allocation, making it representative of a fundamentally different design philosophy. Therefore, rather than evaluating every existing dynamic partitioning scheme individually, this thesis focuses on BCE and SCALE as representative designs for the two major classes of dynamic cache partitioning. Studying these two techniques enables us to investigate whether the identified vulnerabilities arise from specific implementations or from the underlying allocation strategies adopted by each class.

Chapter 4

Revisiting Security in Application-Demand-Based Partitioning

With the increasing reliance on distributed and on-demand computing systems, ensuring strong security guarantees has become an important concern. Various application demand-based techniques have been proposed to improve system efficiency while dynamically adapting to application requirements. These techniques aim to improve cache utilization and performance isolation by allowing applications to request cache resources based on their runtime demand. However, the dynamic nature of such allocation mechanisms can also introduce new forms of information leakage that are absent in static partitioning schemes.

To better understand the security implications of application demand-based partitioning techniques, we analyze BCE as a representative design from this class of approaches. BCE dynamically allocates and deallocates LLC clusters to application domains while attempting to maintain isolation across co-running workloads. By studying the interaction between dynamic LLC allocation and shared cache behavior, we identify several observable side effects that can potentially be exploited for covert communication and occupancy inference. In this chapter, we first analyze the design choices of BCE and derive key security observations. We then discuss the broader implications of these observations and show how they can be exploited to construct covert channels and occupancy-based fingerprinting attacks.

4.1 Key Observations

Observation-I: Allocation Failure as a Leakage Source

To improve resource utilization and performance isolation, application demand-based techniques such as BCE allow domains to dynamically manage their LLC allocations. In BCE, application domains can allocate or deallocate LLC sets using privileged instructions such as `BCE_alloc` and `BCE_dealloc`. This enables domains to request additional cache space when needed and release it when the demand decreases. However, since the LLC capacity is limited and shared among domains, an imbalance in allocation can occur. If one domain acquires a large number of LLC clusters, leaving only a few or none available for other domains, the remaining domains may experience allocation failures when attempting to obtain additional cache clusters. This creates an important leakage source. The success or failure of a new allocation request indirectly reveals information about the current LLC occupancy state. Consequently, a malicious domain may infer whether another co-running domain currently occupies a large fraction of the LLC simply by observing the outcome of its own allocation requests. This observation raises an important question: can such allocation failures be exploited to establish covert communication or infer occupancy information across domains?

Observation-II: Remapping-Induced Timing Leakage

In BCE, any modification to LLC allocations triggers changes in the mapping of cache sets assigned to application domains. Since the indexing logic depends on the current allocation configuration, changes in allocation require the cache to undergo a remapping process. As discussed in prior work, *"even small allocation changes require invalidating and remapping all partition data to accommodate changes to the indexing logic"* [16]. Consequently, previously stored cache data must be invalidated and reorganized according to the updated mapping configuration. This remapping process introduces additional latency during successful allocations. In contrast, unsuccessful allocations terminate without requiring remapping operations. Therefore, successful and unsuccessful allocations exhibit different execution latencies. These timing differences create another observable side effect that may potentially leak information about LLC occupancy and allocation activities across domains.

4.1.1 Implications of Dynamic Allocation Leakage

The observations discussed above reveal that dynamic allocation behavior in BCE creates externally observable signals that can unintentionally leak occupancy information across isolated domains. Although BCE prevents direct cross-core cache evictions and set-level interference, the allocation mechanism itself exposes indirect side channels through allocation outcomes and remapping-induced latency variations. Importantly, these leakage sources do not rely on traditional cache attacks such as Prime+Probe or Flush+Reload. Instead, the attacker exploits the behavior of the allocation framework itself. Since allocation success depends on the currently available LLC space, and successful allocations trigger expensive remapping operations, allocation requests become a source of observable information about the occupancy state of co-running applications.

These observations have two important security implications. First, allocation outcomes can potentially serve as a communication medium between mutually isolated domains, thereby enabling covert channels. Second, long-term occupancy information inferred through allocation behavior can reveal application-specific cache footprints, enabling process fingerprinting attacks. Motivated by these implications, we next demonstrate two covert channels that exploit BCE’s dynamic allocation behavior.

4.2 Allocation/Deallocation Covert Channels

BCE claims that it mitigates all three classes of cross-core attacks, including occupancy-based attacks. Its security argument relies on the assumption that partition sizes are allocated independently of secret-dependent accesses and therefore prevent occupancy inference [29]. However, BCE does not consider the scenario where one application has already occupied a significant fraction of LLC space, thereby influencing the allocation outcomes observed by other domains. BCE further states that *“If the requested number of LLC clusters are available, then the instruction returns success, otherwise failure”* [29]. We show that these allocation outcomes themselves can become a communication medium between domains. Specifically, we propose two covert channels:

- **Channel-I:** A timing-based covert channel that exploits the latency difference between successful and unsuccessful allocations.
- **Channel-II:** A memory-access-based covert channel that infers allocation outcomes through LLC hit and miss behavior.

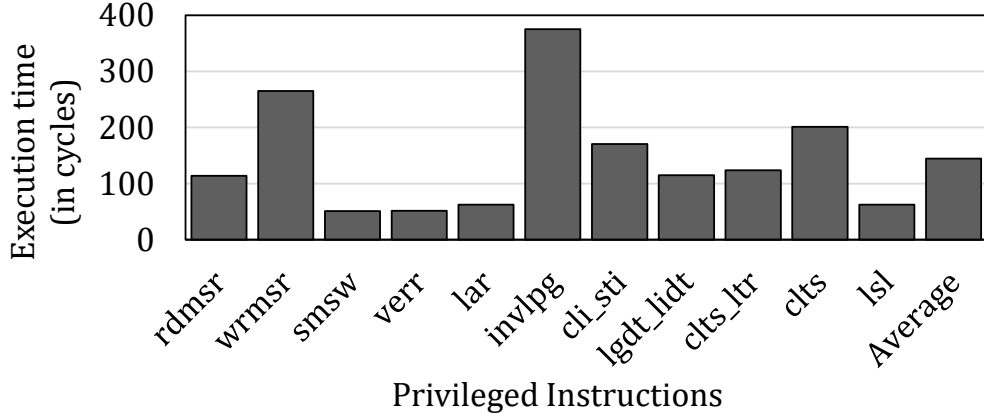


Figure 4.1: Execution time of privileged instructions in x86.

4.2.1 Attack Overview

Both covert channels rely on the same underlying principle: the sender influences the occupancy state of the LLC, while the receiver infers this occupancy indirectly through allocation behavior. The channels differ only in the observable signal used for inference.

In Channel-I, the receiver directly measures the latency of allocation requests. Since successful allocations trigger cache remapping operations whereas unsuccessful allocations do not, the receiver can distinguish allocation outcomes using timing differences. In Channel-II, the receiver does not directly monitor allocation latency. Instead, it infers whether the allocation succeeded or failed by observing memory access latency. Successful allocations allow memory accesses to hit in the LLC, whereas failed allocations force accesses to DRAM, thereby exposing allocation outcomes indirectly through cache hit and miss behavior. Together, these channels demonstrate that dynamic allocation behavior itself can become a medium for covert communication despite cache partitioning.

4.2.2 Experimental Setup

To evaluate the practicality of these covert channels and quantify their leakage potential, we use the ChampSim [1] cycle-accurate microarchitectural simulator, which has also been used in prior cache security studies [5, 9, 19]. We simulate a two-core system whose memory hierarchy parameters are modeled after Intel Sunny Cove. The processor consists of an out-of-order 4GHz core with a 6-issue width, 4-retire width, and a 352-entry reorder buffer. The private cache hierarchy consists of a 48KB L1D cache and a 512KB L2 cache, while the LLC is organized as a 2MB non-inclusive 16-way cache per core.

Core	Out-of-order, 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
L1I	32 KB, 8-way, 4 cycles, 8 MSHRs, LRU
L1D	48 KB, 12-way, 5 cycles, 16 MSHRs, LRU
L2	512 KB, 8-way, 15 cycles, 32 MSHRs, LRU, non-inclusive
LLC	2 MB per core, 16-way, 30 cycles, 64 MSHRs, RRIP, non-inclusive, load balancing hash [29]

Table 4.1: Simulation parameters of the baseline system.

Impact of Allocation Invocation Overhead The allocation and deallocation operations incur additional latency due to their privileged execution. To account for this overhead, we experimentally measured the latency of privileged instruction on an x86-based Intel server running the latest version (6.8.0-52-generic) of Linux kernel as shown in Figure 4.1 and incorporated this latency into all covert-channel experiments. Consequently, the reported communication bandwidths already include the overhead associated with invoking the allocation primitives and therefore reflect the effective end-to-end communication rate rather than an idealized hardware-only latency. We observe that privileged instructions take approximately 150 cycles on average.

Although the privileged invocation contributes a constant overhead to each transmitted bit, it affects both successful and unsuccessful allocation requests equally. Since the receiver distinguishes transmitted bits based on the relative difference between allocation outcomes, this overhead primarily reduces the achievable bandwidth without affecting the correctness of communication. This observation is consistent with our experimental results, which achieve 100% communication accuracy despite accounting for the invocation overhead. Table 4.1 summarizes the system configuration used throughout our evaluation.

4.2.3 Threat Model and Assumptions

Our threat model assumes a system with two active processes, namely the sender and the receiver, executing on separate cores. The attack surface consists of the LLC, which is

dynamically partitioned using BCE. Since BCE prevents direct cross-core cache evictions, the sender and receiver cannot directly create LLC eviction sets. Both processes can issue memory requests to their own data but cannot access cache lines outside their address space or outside their allocated LLC clusters. However, both the sender and receiver can accurately distinguish LLC hits and misses for their own accesses. In addition, they can infer whether an LLC allocation request succeeds or fails. We further assume that the sender and receiver are unaware of BCE’s internal LLC indexing and allocation policies. Despite this limited visibility, we show that observable allocation outcomes alone are sufficient to establish covert communication channels.

4.2.4 Channel-I: Allocation Latency Channel

Our first covert channel directly exploits the timing asymmetry between successful and unsuccessful LLC allocations in BCE. As discussed earlier, successful allocations trigger cache invalidation and remapping operations, whereas unsuccessful allocations terminate without remapping. This difference creates a measurable timing signal that can be used for covert communication between isolated domains.

Root Cause of Timing Leakage

As shown in Figure 4.1, privileged instructions take around 150 cycles on average on an x86 processor. When the requested LLC clusters are available, BCE performs the allocation and subsequently invalidates and remaps all affected partition data according to the updated indexing configuration. In contrast, unsuccessful allocations only check cluster availability and return without performing remapping. Consequently, successful allocations incur significantly higher latency than unsuccessful allocations. The latency overhead approximately scales with:

$$\text{Number of allocated clusters} \times \text{Cluster size} \times \text{Number of ways} \times \text{LLC access latency}$$

If an application domain currently occupies k clusters and requests allocation of another LLC cluster, the total remapping overhead can reach approximately:

$$30 \times k \text{ kilo cycles}$$

Algorithm 1 Channel-1 Sender

```
for bit in message [ ] do
  if bit = 1 then
    | Allocate(sender, maxClusters - 1)
  else
    | Deallocate(sender)
  wait()
```

This creates a large timing gap between successful and unsuccessful allocations, making the allocation outcome externally observable.

Transmission Protocol

Using this timing asymmetry as a communication primitive, the sender and receiver establish the following covert communication protocol. Algorithms 1 and 2 outline the pseudo-code for the sender and receiver processes, respectively. Function `Allocate(x,y)` denotes an allocation request where `x` is the domain-id of the requesting domain and `y` is the number of clusters requested. Function `Deallocate(x)` denotes a deallocation request where `x` is the domain-id of the requesting domain. Initially, the receiver allocates only one LLC cluster. To transmit a bit value of ‘1’, the sender allocates all remaining LLC clusters. To transmit a bit value of ‘0’, the sender deallocates all of its currently assigned clusters.

Figure 4.2 illustrates the communication sequence. In step ①, the sender either occupies almost the entire LLC or releases its allocations depending on the transmitted bit. After transmitting a bit, the sender waits for a fixed synchronization interval. In step ②, the receiver repeatedly requests allocation of one less than the total number of LLC clusters and measures the allocation latency. Since the receiver already knows the expected latency of successful and unsuccessful allocations, it can infer the transmitted bit based on a predefined threshold. For our experiments, we use a threshold of 200 cycles. If the measured latency is below the threshold, the allocation is interpreted as unsuccessful, implying that the sender currently occupies most LLC clusters. The receiver therefore infers a transmitted bit value of ‘1’. Otherwise, the allocation is interpreted as successful and the receiver infers a transmitted bit value of ‘0’.

Algorithm 2 Channel-1 Receiver

```
for bit in message[ ] do
  startTimer  $\leftarrow$  getCurrentTime()
  Allocate(receiver, maxClusters - 1)
  endTime  $\leftarrow$  getCurrentTime()
  allocationTime  $\leftarrow$  endTime - startTimer
  if allocationTime < allocDeallocThreshold then
    | bit  $\leftarrow$  1
  else
    | bit  $\leftarrow$  0
  Deallocate(receiver)
  Allocate(receiver, 1)
```

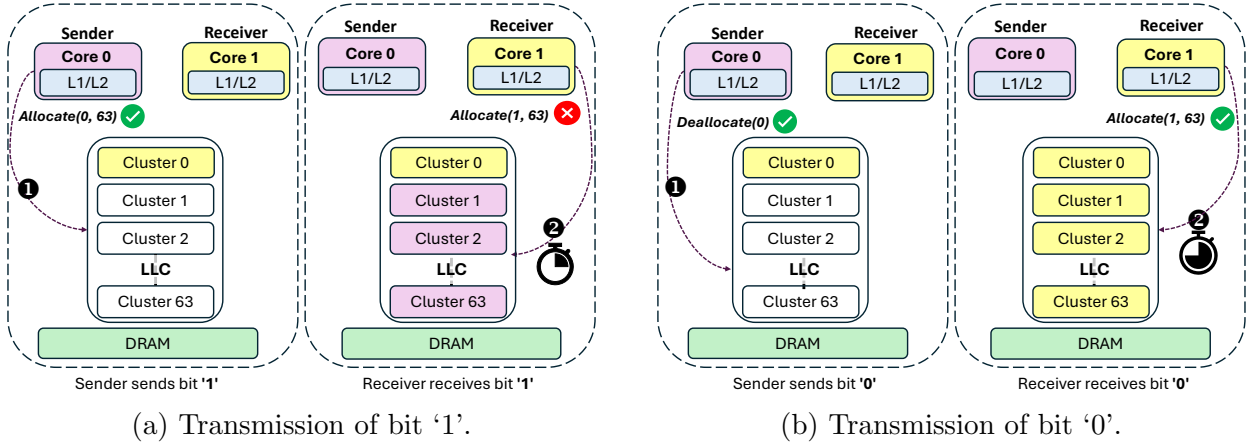


Figure 4.2: Steps involved in allocation/deallocation channel-I. White indicates unallocated clusters; pink and yellow denote sender and receiver allocations.

4.2.5 Channel-II: Memory Access Latency Channel

Although Channel-I directly exploits the latency difference between successful and unsuccessful allocation requests, such timing measurements may be restricted in practical systems due to operating system monitoring or hardware performance isolation mechanisms. Therefore, we next present an alternative covert channel that indirectly infers allocation outcomes through memory access latency behavior.

Motivation

The key intuition behind this channel is that successful and unsuccessful allocations affect the receiver's memory access behavior differently.

Algorithm 3 Channel-2 Sender

```
for bit in message[ ] do
  if bit = 1 then
    | Allocate(sender, maxClusters)
  else
    | Deallocate(sender)
  wait()
```

When the receiver successfully allocates LLC clusters, a portion of its memory accesses are served from the LLC. In contrast, when allocation fails because the sender occupies most LLC clusters, the receiver experiences significantly more DRAM accesses. Consequently, successful and unsuccessful allocations expose different average memory access latencies. Unlike Channel-I, this covert channel does not require directly measuring the execution latency of privileged instructions. Instead, it relies on the observable difference between LLC hits and DRAM accesses after allocation attempts.

Transmission Protocol

Initially, neither the sender nor the receiver occupies any LLC clusters. Both processes know the total LLC size and the cluster allocation granularity used by BCE. To transmit a bit value of ‘1’, the sender allocates all available LLC clusters. To transmit a bit value of ‘0’, the sender deallocates all its currently assigned clusters. Figure 4.3 illustrates the communication process. In step ❶, the sender either occupies the LLC or releases its allocations depending on the transmitted bit. In step ❷, the receiver requests allocation of the maximum possible number of LLC clusters. However, unlike Channel-I, the receiver does not directly monitor allocation latency. Instead, it allocates an array larger than the L1 and L2 caches but smaller than the LLC and repeatedly accesses this array to measure the average memory access latency. If the allocation succeeds, a significant fraction of the receiver’s memory accesses hit in the LLC, resulting in lower average access latency. Conversely, if the allocation fails because the sender occupies most LLC clusters, the receiver experiences substantially more DRAM accesses, increasing the observed memory access latency.

The receiver compares the measured access latency against a predefined threshold to infer the transmitted bit. If the latency exceeds the threshold, the receiver concludes that the allocation failed and infers that the sender currently occupies most LLC clusters, corresponding to a transmitted bit value of ‘1’. Otherwise, the receiver infers a transmitted bit value of ‘0’.

Algorithm 4 Channel-2 Receiver

```
for bit in message[ ] do
  Allocate(receiver, maxClusters)
  latency  $\leftarrow$  measureMemoryLatency()

  if latency > memoryThreshold then
    | bit  $\leftarrow$  1
  else
    | bit  $\leftarrow$  0
  Deallocate(receiver)
```

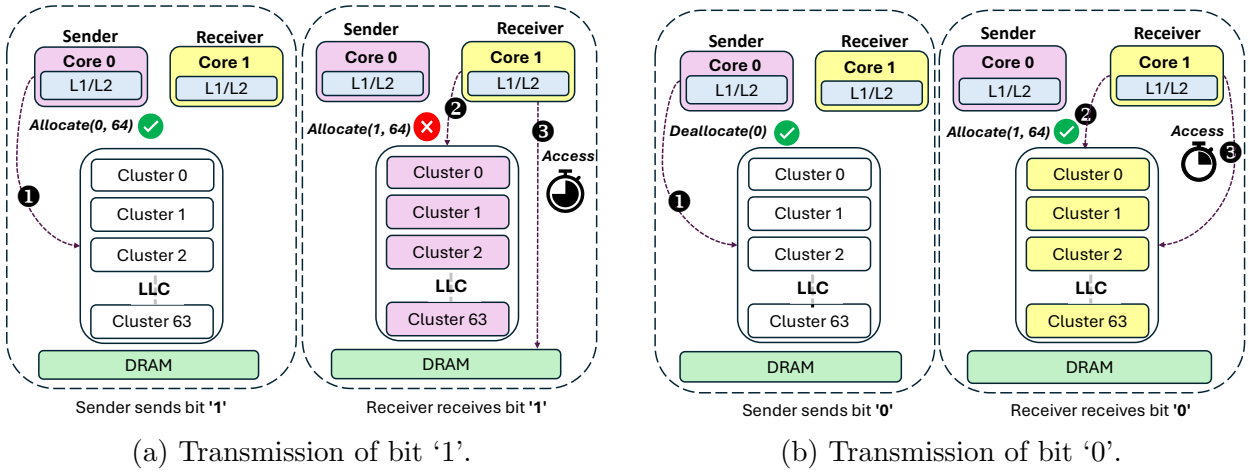


Figure 4.3: Steps involved in allocation/deallocation channel-II. White indicates unallocated clusters; pink and yellow denote sender and receiver allocations.

Algorithms 3 and 4 summarize the sender and receiver pseudocode for Channel-II.

4.2.6 Comparison of the Proposed Covert Channels

The two proposed covert channels exploit different aspects of BCE’s dynamic allocation mechanism. While both enable reliable communication between mutually distrusted domains, they differ in the way information is observed and consequently exhibit different communication characteristics. Covert channels are commonly evaluated using several metrics, including communication bandwidth, bit error rate, robustness against system noise, synchronization overhead, and detectability. Since both proposed channels are evaluated under the same experimental setup, threat model, and synchronization mechanism, this work primarily compares them using communication bandwidth and accuracy. These metrics

directly capture the communication capacity and reliability of the channels, making them sufficient for evaluating their relative effectiveness.

Table 4.2 summarizes the comparison between the two channels.

Table 4.2: Comparison of the proposed covert channels.

Metric	Channel I	Channel II
Observation Mechanism	Allocation latency	Memory access latency
Bandwidth	2.61 Mbps	4.46 Kbps
Communication Accuracy	100%	100%
Receiver Observation	Direct	Indirect

As shown in Table 4.2, Channel I provides substantially higher communication bandwidth while maintaining the same communication accuracy as Channel II. This improvement is primarily because the receiver directly observes the latency difference between successful and unsuccessful allocation requests, resulting in a stronger timing signal and lower communication latency. In contrast, Channel II infers allocation success indirectly through memory access latency. Since each transmitted bit requires additional memory accesses to determine whether the allocated cluster is available, the overall communication latency increases considerably, leading to lower bandwidth. Nevertheless, Channel II remains significant because it demonstrates that reliable covert communication is feasible even when the receiver cannot directly observe allocation outcomes. Therefore, although Channel I represents the higher-capacity covert channel, Channel II illustrates that indirect cache observations are also sufficient to establish covert communication across isolated security domains.

4.3 A Preliminary Attempt at Process Fingerprinting

The covert channels discussed in the previous section demonstrate that allocation outcomes in BCE leak information about the LLC occupancy state across domains. Beyond covert communication, such leakage can potentially be exploited to infer characteristics of co-running applications.

Motivated by this observation, we next explore whether LLC occupancy information exposed through BCE’s allocation behavior can be used for process fingerprinting. In particu-

lar, we investigate whether an attacker can estimate the LLC footprint of a victim application and use this occupancy information to infer its identity.

4.3.1 Attack Intuition

The core intuition behind the fingerprinting attack is that different applications occupy different numbers of LLC clusters depending on their cache demand. Memory-intensive applications typically occupy larger LLC regions, whereas cache-light applications require fewer clusters. If an attacker can estimate how many LLC clusters are currently occupied, it can potentially infer the identity of the co-running application.

The attacker therefore repeatedly issues allocation requests and observes whether they succeed or fail. By gradually adjusting the requested number of clusters, the attacker converges to an estimate of the currently unoccupied LLC space. Since the total number of LLC clusters is known, the attacker can estimate the number of clusters occupied by the victim application. If applications exhibit sufficiently distinct occupancy footprints, this occupancy estimate can act as an application fingerprint.

4.3.2 Fingerprinting Methodology

Algorithm 5 outlines the recursive fingerprinting attack used to infer the number of LLC clusters occupied by a co-running application in a system where the LLC is partitioned using BCE. The attacker initially requests allocation of x LLC clusters and measures the time required for the allocation. If the latency is below a predefined threshold (`allocDeallocThreshold`), the allocation is interpreted as successful, indicating that sufficient LLC space is currently available. Otherwise, the allocation is treated as unsuccessful, implying that the requested number of clusters could not be allocated because they are already occupied by the victim application. Based on the observed allocation outcome, the attacker recursively adjusts its allocation request:

- If the allocation succeeds and no prior failure has occurred, the attacker doubles the requested clusters and retries the allocation.
- If the allocation fails, the attacker records that the allocation limit has been exceeded and reduces the request size.
- Once the attacker observes a successful allocation after a previous failure, it concludes that the current allocation approximately reflects the amount of unoccupied LLC space.

Algorithm 5 Fingerprinting Attack

flag $\leftarrow$ 0

Function *attack*(*x*):

startTimer $\leftarrow$ *getCurrentTime*()

Allocate(*x*)

endTimer $\leftarrow$ *getCurrentTime*()

allocationTime $\leftarrow$ *endTimer* - *startTimer*

outcome $\leftarrow$ (*allocationTime* > *allocDeallocThreshold*) ? success : failure

if *outcome* == *success* **then**

if *flag* == 1 **then**

return *x*

else

attack($2 \times x$)

if *outcome* == *failure* **then**

 flag $\leftarrow$ 1 **attack**($x/2$)

This recursive search process gradually converges to the largest allocation size that can be successfully accommodated in the LLC. Since the total number of LLC clusters is known, the attacker can estimate the number of clusters occupied by the co-running victim application.

4.3.3 Cluster Allocation Patterns in BCE

To understand how occupancy information translates into application identification, we next examine the LLC cluster allocation patterns assigned to different applications in BCE. The system under consideration employs BCE to partition the LLC into multiple clusters, allowing fine-grained control over cache allocation to applications. As shown in Table 4.3, each benchmark is assigned a different number of LLC clusters depending on its memory intensity and cache demand.

Memory-intensive workloads such as *mcf*, *xalancbmk*, and *gcc* receive significantly larger allocations than lightweight applications such as *perl* and *leela*. These differentiated allocation patterns improve cache utilization and performance isolation by allocating larger cache footprints to applications with higher memory demand. However, these allocation differences also create observable occupancy signatures. If an attacker can estimate the number of occupied clusters using the methodology described above, the occupancy estimate itself may reveal the identity or characteristics of the co-running application.

Benchmark	Clusters
gcc	15
perl	1
mcf	30
lbm	2
wrf	5
roms	3
xalancbmk	17
cactusBSSN	12
omnetpp	16
x264	15
leela	1
exchange2	5
nab	3

Table 4.3: LLC cluster allocation for various SPEC CPU2017 benchmarks under BCE-based LLC partitioning.

4.3.4 Limitations of Occupancy-Based Fingerprinting

Although LLC occupancy information leaks coarse-grained application characteristics, our analysis shows that occupancy alone is insufficient for reliable process fingerprinting in realistic systems. The primary limitation arises because multiple applications often exhibit identical or highly similar LLC occupancy footprints. As shown in Table 4.3, several benchmarks map to the same or closely overlapping cluster counts. For example:

- `perl` and `leela` both occupy 1 cluster,
- `roms` and `nab` each occupy 3 clusters,
- `wrf` and `exchange2` both occupy 5 clusters, and

- `gcc` and `x264` each occupy 15 clusters.

Consequently, the observed cluster count is not a unique identifier, but rather a coarse-grained occupancy characteristic shared by multiple applications. As a result, the attacker may narrow the victim down to a small candidate set, but cannot reliably identify the exact process. This ambiguity becomes even more pronounced in realistic multi-tenant environments where multiple applications execute concurrently. As the number of co-running workloads increases, the probability of different applications sharing identical or overlapping occupancy footprints also increases. Therefore, the mapping between observed LLC occupancy and candidate applications gradually becomes many-to-one rather than one-to-one. This directly reduces the discriminative power of occupancy-based fingerprinting and increases the probability of misclassification. In other words, while occupancy leakage may reveal coarse-grained workload characteristics, the fingerprinting accuracy degrades significantly as the number of co-running applications increases. These observations highlight that although BCE’s allocation behavior leaks occupancy information, the leakage is often insufficient for reliable fine-grained fingerprinting in realistic multi-application systems.

Chapter 5

Revisiting Security in LLC-Usage-based Technique

SCALE uses UCP [27], a dynamic LLC partitioning technique that divides the LLC ways among co-running applications to maximize the combined utility of the system. As discussed, SCALE adds several layers of randomness to UCP allocation to make it secure as per the Equation 5.1, where $A(t)$ is the amount of LLC allocated to an application for a reconfiguration interval (t) and $\Delta_{alloc}(t) = UCP_{allocation} - A(t)$ is the difference between the current allocation and UCP's newly calculated allocation.

$$\begin{aligned} A(t) = & randRound(A(t-1) + randRound(\Delta_{alloc}(t-1) \\ & \times changeRate) + noise) \end{aligned} \quad (5.1)$$

5.1 Theoretical Analysis of Design Choice

In this section, we closely examine each randomization layer added by SCALE, to theoretically analyze the contribution and impact of each layer in ensuring security. The *randRound* function in Equation 5.1 rounds an input value up or down with equal probability. For any input $X \in \mathbb{R}$, the function can be expressed as:

$$randRound(X) = \begin{cases} X & \text{if } X \in \mathbb{Z} \\ \lfloor X \rfloor + choose(\{0, 1\}) & \text{otherwise} \end{cases} \quad (5.2)$$

where $\lfloor . \rfloor$ is the floor (round down) function, *choose*($\{0, 1\}$) returns either 0 or 1 with equal probability, and $\mathbb{Z}$ and $\mathbb{R}$ denote the set of all integers and real numbers, respectively. Since,

the probability of $X \in \mathbb{Z}$, given $X \in \mathbb{R}$, is essentially zero, the *randRound* function effectively becomes:

$$\text{randRound}(X) \approx \lfloor X \rfloor + \text{choose}(\{0, 1\}) \quad (5.3)$$

The Equation 5.1, therefore, can be equivalently transformed into Equation 5.4. This transition shows that despite multiple levels of randomness, the effect produced by the randomizing components is rather a separable one, and the effects of these can be studied independently.

$$A(t) = A(t-1) + \lfloor \Delta_{\text{alloc}}(t-1) \times \text{changeRate} \rfloor + \lfloor \text{noise} \rfloor + \text{choose}(\{0, 1, 1, 2\}) \quad (5.4)$$

Random rounding. The effect of random rounding, as seen in the Equation 5.4, is a combination of rounding down of the calculated values and a random selection from a set of $\{0, 1, 1, 2\}$. The expected value of the random selection is 1 and the expected value of rounding down a random variable X is bounded as $\mathbb{E}[X] - 1 < \mathbb{E}[\lfloor X \rfloor] \leq \mathbb{E}[X]$.

Noise. SCALE derives its noise component from a zero-mean Laplace distribution with a noise parameter b .

$$\text{Lap}(x|0, b) = \frac{1}{2b} \exp\left\{\left(-\frac{|x|}{b}\right)\right\} \quad (5.5)$$

Note that the expected value of any noise derived from the above probability distribution function (PDF) will be 0, which means that if we observe the noises over a long interval, their average behaviour will effectively be zero. SCALE uses memorized values of noise, which are forced to be non-zero. But this does not affect the noise's zero-mean behaviour; hence, the expectation remains the same.

Change rate. The *changeRate* is derived from a uniform random distribution such that $\text{changeRate} \sim \mathcal{U}(a_1, a_2)$, where $0 \leq a_1 < a_2 \leq 1$. The change rate has an expected value of $(a_1 + a_2)/2$. As per SCALE, *changeRate* alone is ineffective against low bandwidth attacks as the allocation reaches close to $UCP_{\text{allocation}}$ after some time, where the rate of saturation depends on the parameters a_1 and a_2 . The *changeRate* is used as zero if $UCP_{\text{allocation}}$ in the current and the last reconfiguration interval is the same, $A(t) = \text{randRound}(A(t-1) + \text{noise})$.

Random reconfiguration intervals. UCP reallocates partitions at a fixed reconfiguration interval, which can be configured as per the demand. However, it suggests invoking the partitioning algorithm once every five million clock cycles based on their simulation results [27]. On the other hand, SCALE proposes to randomize the reconfiguration interval, whose properties can be configured by the user.

Combined effect. From the above discussion, it is evident that the expected value of

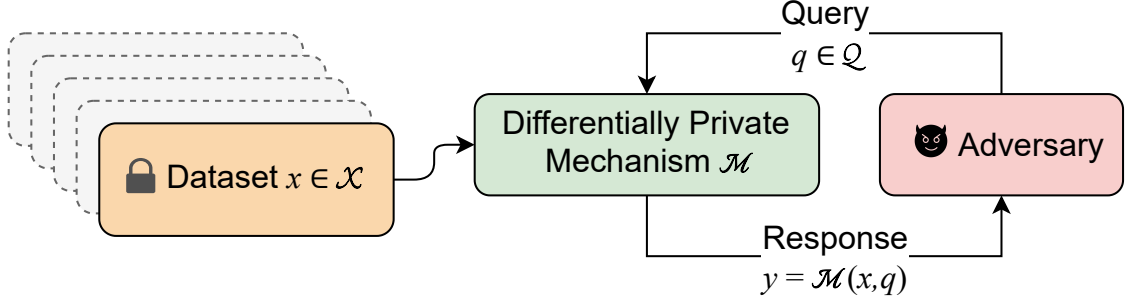


Figure 5.1: Information flow between the random mechanism $\mathcal{M}$, the dataset x , and the adversary.

$A(t)$ at time t , given the allocation at the instance $t - 1$ and the $UCP_{allocation}$, can be approximated with a δ error ($-1 < \delta \leq 1$) as:

$$\mathbb{E}[A(t)] = A(t - 1) + \left(\frac{a_1 + a_2}{2} \right) \Delta_{alloc}(t - 1) + \delta \quad (5.6)$$

This evaluation shows us that despite various layers of randomness, the stochastically expected behavior of SCALE's allocation policy remains predictable. The added noise is prone to averaging, and the expectation of $A(t)$ indicates that the average allocation behavior follows an unrandomized path with an average *changeRate*, leading to saturation after some time. This motivates a covert channel that exploits average allocation change over a sufficiently long interval.

Does ϵ -Differential Privacy Help?

SCALE uses differential privacy to provide information-theoretic security guarantees. The $UCP_{allocation}$ values are treated as a confidential dataset, while SCALE acts as the differentially private algorithm based on the Laplace Mechanism to produce random LLC allocations. In this section, we expand on the impact of differential privacy on SCALE's allocations and assess the security guarantees provided by it.

Differential privacy requires that an individual's data does not impact the output seen by any adversary. Suppose a dataset x , modeled as $x \in \mathcal{X}^n$, containing sensitive information about n individuals, where $\mathcal{X}$ is a data universe, is being shared to an adversary through queries to an algorithm or mechanism $\mathcal{M} : \mathcal{X}^n \times \mathcal{Q} \rightarrow \mathcal{Y}$. Here $\mathcal{Q}$ is the query space, and $\mathcal{Y}$ is the output space or the range of the algorithm $\mathcal{M}$. The information flow is represented

in Figure 5.1. For such a setup, we define the differential privacy as follows.

ε -Differential Privacy [12]: For some $\varepsilon \geq 0$, a randomized algorithm $\mathcal{M} : \mathcal{X}^n \times \mathcal{Q} \rightarrow \mathcal{Y}$ is said to be ε -differentially private if for each pair of datasets $x, x' \in \mathcal{X}^n$ ($x \sim x'$, varying only at one row) and every query $q \in \mathcal{Q}$, $\mathcal{M}$ satisfies:

$$\forall y \in \mathcal{Y}, \quad \mathbb{P}[\mathcal{M}(x, q) = y] \leq e^\varepsilon \cdot \mathbb{P}[\mathcal{M}(x', q) = y] \quad (5.7)$$

The parameter ε is often called the privacy parameter or the privacy budget. For applications with strict privacy requirements, smaller values of ε are recommended.

Laplace Mechanism [12, 33]: For a query $q : \mathcal{X}^n \rightarrow \mathbb{R}^d$ and a scale parameter b , a Laplace Mechanism $\mathcal{M}_{q,b}(x)$ defined over all $x \in \mathcal{X}^n$ is given by:

$$\mathcal{M}_{q,b}(x) = q(x) + \text{Lap}(0, b)^d \quad (5.8)$$

The Laplace Mechanism appends the query results with an additive zero-mean Laplacian noise. It satisfies the ε -differential privacy framework if $\varepsilon \cdot b \geq \Delta q$ (global sensitivity of q , Equation 5.9) or simply, $\varepsilon = \frac{\Delta q}{b}$. It makes every individual query result randomly different from their original value, protecting the individual privacy, while also keeping the mean of the query results very close to the original mean. Because of the random difference generated, the adversary cannot figure out, with certainty, the individual values, but the aggregated information is transmitted without any issue.

$$\Delta q = \max_{x \sim x'} \|q(x) - q(x')\| \quad (5.9)$$

Differential Privacy in SCALE. To prevent attackers from inferring application behavior from cache allocation decisions, SCALE incorporates differential privacy by perturbing the cache utility estimates before determining the final allocation. Specifically, random values sampled from a Laplace distribution are added to the utility values computed by Utility-Based Cache Partitioning (UCP). These random perturbations, referred to as *differential privacy noise*, obscure small differences in application cache demand, making consecutive allocation decisions statistically difficult to distinguish while largely preserving the performance benefits of utility-based allocation.

Consider an adversary trying to infer bits of a message by reading the values of LLC allocations ($UCP_{allocation}$). SCALE configures its global sensitivity (Δq) as the difference in $UCP_{allocation}$ values used to represent bits 1 and 0. Let us consider an adversary that

sends a query for observing a set of τ number of sequential allocations. SCALE’S algorithm will return each allocation randomly different from the $UCP_{allocation}$ values. The maximum distance between any two such queries will happen if one of the allocation sets is full of bit 1s and the other is the allocations corresponding to all bit 0s. Since SCALE configures its noise parameter b with the sensitivity for one-bit change, the difference between the above two queries will be $\Delta q\sqrt{\tau}$ for the Euclidean norm. For any $\tau > 1$, this distance is always bigger than Δq , even if the difference in $UCP_{allocation}$ for transmitting message bits 1 and 0 is the same. As the number of points in the observation set increases, the distance increases, making it harder to safeguard. This motivates the idea that one bit must be sent across multiple reconfiguration intervals to achieve better accuracy.

Moreover, SCALE sets *changeRate* to 0 if $UCP_{allocation}$ values for two consecutive reconfigurations are the same. To counter this, any application aiming to transmit a message by encoding it in the LLC occupancy needs to keep the target $UCP_{allocation}$ varying. For a change (δ) in the $UCP_{allocation}$, we want δ to be as small as possible. If $UCP_{allocation} = \alpha$, for a small δ , by its definition, ε -differential privacy suppresses the difference between α and $\alpha + \delta$.

5.2 Claims and Limits of SCALE

Although SCALE promises strong security guarantees using differential privacy principles, it is important to carefully analyze the implications of its design choices under long-term occupancy observation. SCALE primarily aims to protect LLC allocation information by introducing controlled randomness into cache partitioning decisions, thereby making instantaneous occupancy observations statistically indistinguishable. While this randomized behavior significantly complicates short-term occupancy inference, it remains unclear whether such guarantees continue to hold when an attacker performs long-term statistical monitoring.

In this section, we first discuss the key security claims made by SCALE and explain the intuition behind its differential privacy-based protection mechanism. We then analyze the limitations of these guarantees and present our observations demonstrating that, despite randomized LLC allocations, long-term occupancy statistics can still leak stable application characteristics. Our analysis shows that although SCALE successfully obscures individual allocation instances, statistical aggregation over extended execution intervals can still enable occupancy-based application fingerprinting.

5.2.1 Claims of SCALE

Claim-I: Security Within the Protected Range

For allocation changes within the *protected range*, SCALE provides strong security guarantees. A strong security guarantee specifically refers to a small ϵ in the differential privacy mechanism, meaning that individual allocation changes within the protected range (Δf) are statistically indistinguishable.

The key idea behind this claim is that LLC allocations corresponding to nearby application demands become probabilistically similar after perturbation. Since SCALE injects Laplace noise into the allocation decision, the resulting allocation observed by the attacker does not directly correspond to the true allocation demand of the victim application. As a result, short-term LLC occupancy observations become unreliable indicators of the victim’s actual cache demand. Even if the attacker observes a particular LLC occupancy value, it cannot confidently determine whether the victim’s true allocation was slightly higher or lower within the protected range.

Therefore, SCALE argues that fine-grained occupancy inference attacks become significantly harder because the mapping between application demand and observed LLC occupancy is intentionally obscured.

Claim-II: Randomness Across Repeated Executions

The probability of observing the same allocation fingerprint across repeated executions approaches zero as the number of reconfiguration periods increases. SCALE periodically reconfigures LLC allocations and injects fresh randomness during each reconfiguration interval. Since the injected noise varies independently across intervals, the resulting sequence of LLC allocations also changes across repeated executions of the same application.

This behavior prevents the attacker from observing a deterministic occupancy pattern over time. Even when the same workload is executed repeatedly, the occupancy trace generated by SCALE appears different due to the continuous injection of random noise. Consequently, SCALE claims that repeated executions of the same application do not generate stable fingerprints that can be easily matched by an attacker. This temporal randomness is intended to prevent long-term occupancy correlation attacks and repeated-trace fingerprinting attacks.

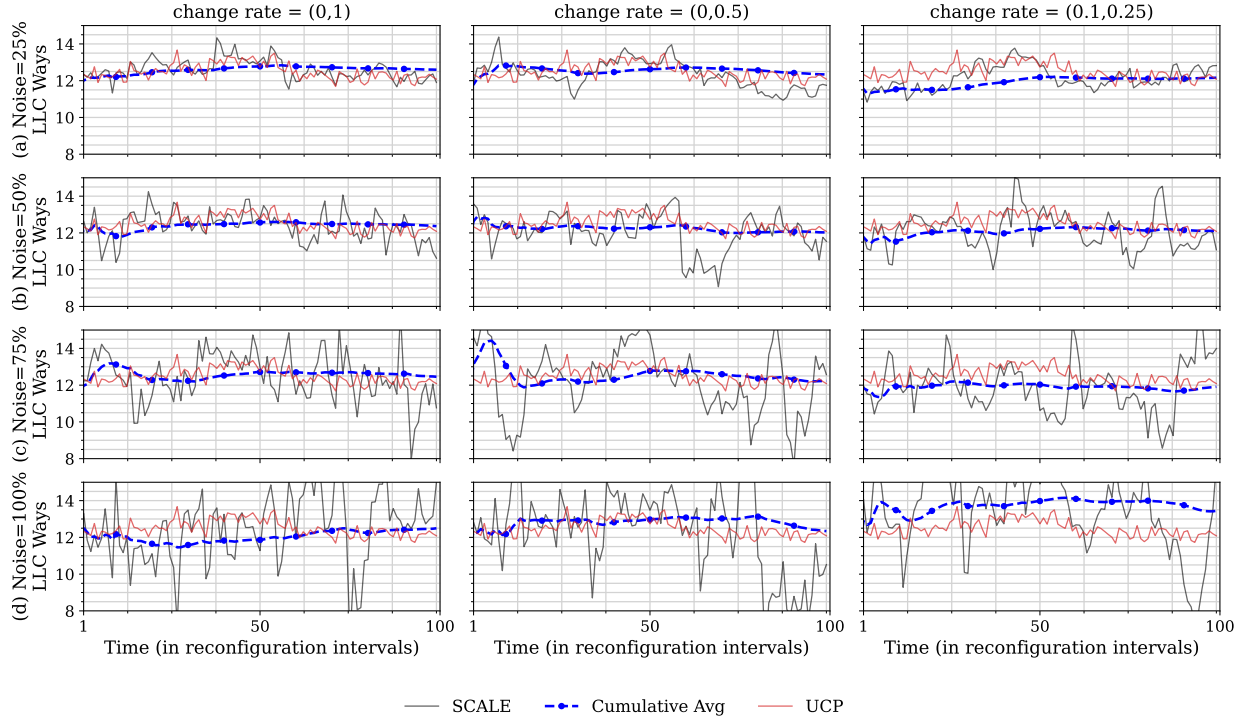


Figure 5.2: LLC ways allocation with SCALE and UCP over 100 reconfiguration intervals on a 4MB 16-way LLC for `mcf_1554B` running with `xalancbmk_10B` under different SCALE noise levels: (a) 25%, (b) 50%, (c) 75%, and (d) 100%.

5.2.2 Limits of SCALE

Although SCALE introduces randomness using differential privacy principles, we argue that these guarantees primarily hold for *individual* LLC allocation instances observed over short intervals. An occupancy attacker observing LLC allocations over a sufficiently long duration can still recover useful statistical information and make application fingerprinting feasible, even when SCALE operates entirely within its protected range. The fundamental limitation arises because differential privacy protects individual observations, but does not inherently guarantee privacy for statistical summaries derived from a large number of observations.

Long-Term Leakage Through Statistical Aggregation

Figure 5.2 validates our observation across different SCALE configurations for the SPEC CPU2017 benchmark `mcf_1554B` co-running with `xalancbmk_10B` on a 2-core, 16-way, 4MB LLC system.

Our experiments show that the cumulative average of SCALE’s LLC allocations gradu-

ally converges toward the allocation produced by UCP across all evaluated noise levels and reconfiguration-rate settings. Although larger noise values increase short-term fluctuations in LLC allocations, and lower change rates slow down the convergence process, neither mechanism prevents long-term statistical convergence. In the short term, SCALE successfully obscures the instantaneous LLC allocation through randomized perturbations. However, as the attacker accumulates observations across many reconfiguration intervals, the randomness introduced by differential privacy begins to average out. Consequently, the long-term average LLC allocation remains recoverable despite the injected noise.

Why Does This Happen?

The injected noise in SCALE is sampled from a Laplace distribution whose expected mean is zero. Therefore, while each individual noisy allocation appears random and unpredictable, the aggregate behavior across many samples follows well-known statistical properties. According to the law of large numbers, the average of a sufficiently large number of noisy observations converges toward the true mean of the underlying distribution. Thus, although a single LLC allocation instance may remain protected, long-term statistical measures such as the average LLC allocation gradually become visible to the attacker.

This creates an important distinction between:

- protecting *individual LLC allocations*, and
- protecting *statistical properties* of LLC allocations.

SCALE primarily guarantees the former, but not necessarily the latter.

Implication for Occupancy-Based Fingerprinting

The key insight is that differential privacy successfully obscures short-term allocation behavior, yet still leaks sufficient long-term information for occupancy-based fingerprinting attacks. Although SCALE prevents attackers from confidently inferring a single LLC allocation instance, it does not fully prevent attackers from recovering aggregate allocation behavior over extended execution intervals. Over time, the attacker can statistically estimate the underlying allocation characteristics of the victim application despite the presence of randomized noise. Therefore, the security guarantees provided by SCALE are fundamentally limited against attackers capable of performing long-term occupancy monitoring and statistical aggregation. This reveals an important gap between:

1. protecting instantaneous cache allocations, and
2. preventing statistical recovery of application behavior over time.

Our observations demonstrate that randomness alone is insufficient to completely eliminate occupancy-based fingerprinting if long-term statistical information remains observable. An occupancy attacker can continuously monitor LLC occupancy over many reconfiguration periods and derive stable statistical features such as:

- mean LLC occupancy,
- occupancy variance,
- long-term allocation trends,
- temporal allocation distributions, and
- convergence behavior of allocations.

These statistical summaries can themselves act as robust application fingerprints, even when instantaneous allocations remain randomized.

5.3 Application Fingerprinting Attack on SCALE

Although SCALE introduces randomized LLC allocations using differential privacy principles, we observe that long-term occupancy behavior still leaks stable application characteristics. In particular, while SCALE obscures instantaneous LLC allocations, an attacker observing occupancy behavior over extended execution intervals can still recover statistical properties that remain highly consistent across runs. In this section, we first describe the attacker model and the occupancy-based attack methodology used against SCALE. We then explain how long-term occupancy traces can be transformed into stable statistical fingerprints that enable reliable application classification despite randomized LLC allocations.

Occupancy-based side-channel attacks infer victim behavior by observing how much cache space is occupied over time. Unlike traditional Prime+Probe attacks that focus on set-level conflicts, occupancy attacks exploit coarse-grained variations in LLC utilization. The fundamental intuition behind these attacks is straightforward. When a victim application occupies a larger fraction of the LLC, the attacker experiences more cache misses while accessing its own memory buffer. Conversely, when the victim occupies a smaller portion

of the LLC, the attacker experiences more cache hits. By continuously monitoring these occupancy variations, the attacker can infer workload-specific execution patterns. Prior work has demonstrated that occupancy traces can reveal sensitive information such as website activity, application identity, and execution phases [30].

5.3.1 Experimental Setup

We use the ChampSim [1] simulator to evaluate the security of SCALE under occupancy-based fingerprinting attacks. Our evaluation models a two-core out-of-order processor with a Sunny Cove-like memory hierarchy. Each core is connected to a 2MB non-inclusive LLC bank with 16 cache ways. The private cache hierarchy consists of a 48KB 8-way L1D cache and a 512KB 12-way L2 cache. The processor operates at 4GHz and is modeled with a 352-entry reorder buffer (ROB) and a retire width of four instructions per cycle. We evaluate SPEC CPU2017 benchmarks co-running under SCALE-based LLC partitioning to analyze whether long-term occupancy behavior leaks distinguishable application characteristics despite randomized LLC allocations. Table 5.1 summarizes the experimental configuration used throughout our evaluation.

5.3.2 Threat Model

Our threat model follows prior occupancy-based side-channel attacks [30], where an attacker and a victim application are co-located on the same physical machine and share the Last-Level Cache (LLC). The attacker and victim execute on separate processor cores but compete for the same shared LLC resources managed by SCALE. Since the LLC remains a shared microarchitectural structure, the attacker aims to observe variations in the victim’s cache usage through changes in its own cache access behavior.

The attacker does not have access to privileged system information, direct victim memory contents, or explicit allocation metadata from SCALE. Instead, the attacker relies solely on timing-based occupancy observations obtained through its own memory accesses. Specifically, the attacker continuously monitors how the victim’s LLC occupancy affects its cache hit and miss behavior over time. We assume a *closed-world* attack setting, which is commonly used in application and website fingerprinting literature [30]. In this setting, the attacker already knows the set of possible victim applications beforehand and has previously profiled them offline under similar hardware and system conditions. During profiling, the attacker records occupancy traces corresponding to each candidate application and learns

Table 5.1: Experimental setup used for evaluating occupancy-based fingerprinting attacks on SCALE.

Component	Configuration
Simulator	ChampSim [1]
Processor Model	Out-of-order core
Clock Frequency	4 GHz
Number of Cores	2
ROB Size	352 entries
Retire Width	4 instructions/cycle
L1D Cache	48KB, 8-way
L2 Cache	512KB, 12-way
LLC Organization	Non-inclusive
LLC Size	2MB per core
LLC Associativity	16-way
Partitioning Technique	SCALE
Workloads	SPEC CPU2017 benchmarks

their statistical characteristics. During the actual attack phase, the attacker observes the occupancy behavior of an unknown co-running victim and attempts to match the observed trace against the previously learned profiles. The attacker’s objective is therefore not to reconstruct the exact LLC allocation sequence, but rather to identify the victim application by exploiting stable long-term occupancy characteristics.

This distinction is important because SCALE primarily attempts to prevent leakage of instantaneous LLC allocation information through randomized perturbations. However, our attacker focuses on aggregate statistical behavior accumulated over long execution intervals. Consequently, even if individual LLC allocations appear highly randomized, stable statistical properties may still reveal the identity of the victim application.

Algorithm 6 Attacker Pseudocode

```
Initialize memoryGram[sampleSize], buffer[bufSize] // bufSize is equal to the
LLC size
while true do
    counter  $\leftarrow$  0 t_start  $\leftarrow$  time()
    repeat
        counter  $\leftarrow$  counter + 1
        for i  $\leftarrow$  0 to  $\frac{\text{bufSize}}{\text{cacheBlockSize}} - 1$  do
            tmp  $\leftarrow$  buffer[i  $\times$  cacheBlockSize]
        until time() - t_start  $\geq$  reconfInterval
    memoryGram[t_start]  $\leftarrow$  counter
```

5.3.3 Occupancy Attack Methodology

To infer the victim application’s behavior, the attacker performs a sweep-counting occupancy attack [30]. The attack exploits the fact that LLC occupancy directly affects the attacker’s cache access latency and throughput. The attacker allocates a memory buffer whose size is equal to the LLC capacity and continuously sweeps through this buffer by repeatedly accessing one cache block from each cache line. During every observation interval, the attacker counts how many complete sweeps of the buffer can be performed. We refer to this observation duration as the *sampling interval*, which is set to 5M cycles in our evaluation.

The intuition behind the attack is straightforward. If the victim occupies a larger fraction of the LLC, a larger portion of the attacker’s buffer gets evicted from the cache. Consequently, the attacker experiences more LLC misses while sweeping through the buffer, reducing the number of completed sweeps during the sampling interval. In contrast, if the victim occupies a smaller fraction of the LLC, the attacker experiences more LLC hits, allowing more sweeps to complete within the same duration.

Thus, the number of completed sweeps indirectly reflects the victim’s LLC occupancy at that time. By continuously recording these measurements across multiple sampling intervals, the attacker constructs a temporal occupancy trace known as a *memorygram*. This memorygram captures how the victim’s cache occupancy evolves over time and serves as the primary source of information for fingerprinting attacks. Algorithm 6 shows the pseudocode of the attacker. The attacker first initializes a probing buffer whose size matches the LLC capacity. During every sampling interval, the attacker repeatedly sweeps through the entire buffer and increments a counter after each completed sweep. Each sweep accesses one block per cache line to ensure that the entire LLC capacity is probed uniformly.

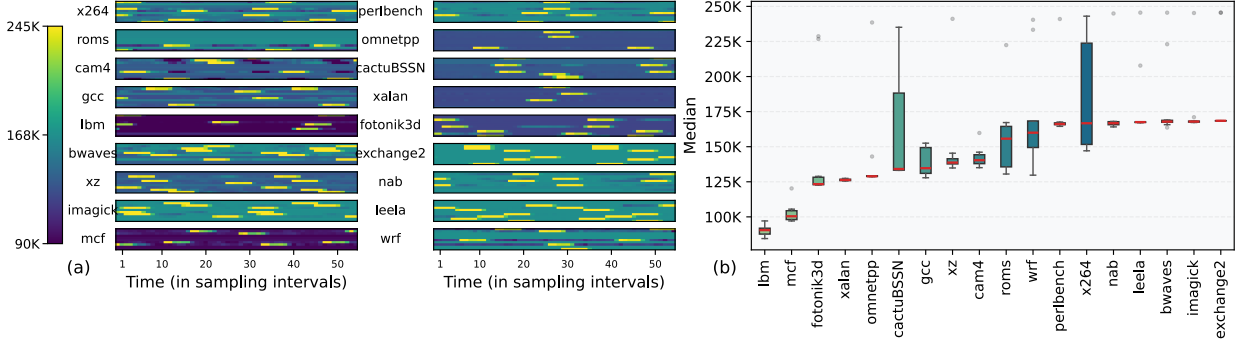


Figure 5.3: Attacker’s occupancy when co-running with SPEC CPU2017 benchmarks across runs over 55 sampling intervals. (a) Memorygram with each row as a run (y-axis omitted). (b) Boxplot of median counter values after median filtering; boxes show the interquartile range, the center line marks the median, and whiskers denote the non-outlier range.

At the end of the sampling interval, the attacker stores the total number of completed sweeps in the memorygram. This process repeats continuously throughout execution, generating a sequence of occupancy measurements over time. Importantly, the attacker does not require privileged access, direct knowledge of the victim’s memory accesses, or access to SCALE’s internal allocation metadata. The attack relies solely on timing differences caused by shared LLC occupancy behavior, making it practical even under randomized LLC partitioning mechanisms such as SCALE.

5.3.4 Memorygrams Under SCALE

Figure 5.3(a) shows the memorygrams generated for SPEC CPU2017 benchmarks when running under SCALE. Each memorygram represents the temporal LLC occupancy behavior observed by the attacker over multiple sampling intervals. At first glance, the occupancy traces appear highly noisy and irregular. This behavior is expected because SCALE continuously randomizes LLC allocations using differential privacy-based perturbations. As a result, the instantaneous occupancy patterns of different applications overlap significantly, making applications difficult to distinguish from short-term observations alone. In this sense, the memorygrams initially appear to support SCALE’s security intuition that randomized allocations can obscure application behavior.

However, a different trend emerges when the occupancy traces are analyzed over longer execution intervals. Although individual occupancy samples fluctuate due to injected randomness, the overall statistical behavior of each application remains relatively stable across

runs. Figure 5.3(b) illustrates this observation using statistical summaries derived from the occupancy traces. Despite the noisy instantaneous behavior, different applications exhibit distinguishable long-term occupancy characteristics. In particular, applications differ in their average occupancy levels, occupancy ranges, and temporal variability patterns. Consequently, the occupancy distributions form relatively separable clusters even under randomized LLC allocations. These observations indicate that while SCALE successfully obscures instantaneous LLC occupancy values, it does not eliminate stable long-term statistical signatures. An attacker observing occupancy behavior over sufficiently long intervals can therefore still distinguish applications using aggregate statistical properties of the memorygrams.

5.3.5 Feature Extraction

The occupancy traces obtained from SCALE contain significant short-term fluctuations due to the randomized perturbations introduced by differential privacy. While these perturbations obscure instantaneous LLC occupancy behavior, they also introduce sharp transient variations that are not useful for identifying long-term application characteristics. To reduce the impact of such short-term noise, we first apply a median filter over five reconfiguration intervals. This filtering step smooths sudden occupancy spikes while preserving the underlying long-term occupancy trend of the application.

After filtering, each memorygram is summarized using a set of statistical features that collectively characterize the application’s LLC occupancy behavior over time. Rather than relying on exact occupancy sequences, the attacker uses these aggregate statistical properties as stable fingerprints for classification. The extracted features and their significance are summarized below:

- **Mean Occupancy:** The mean captures the average LLC occupancy throughout execution. Applications with consistently high cache demand exhibit larger mean occupancy values, while cache-light workloads produce lower averages. This feature provides a coarse characterization of the application’s overall cache usage behavior.
- **Median Occupancy:** The median represents the central tendency of occupancy values while being robust to occasional spikes and abrupt perturbations caused by differential privacy noise. Since SCALE can introduce sharp short-term fluctuations, the median provides a more stable representation of the application’s typical occupancy behavior.

- **Standard Deviation:** The standard deviation measures the variability of occupancy over time. Some applications exhibit stable cache usage patterns, whereas others show highly dynamic behavior due to changing execution phases and memory access patterns. This feature helps distinguish workloads based on occupancy stability.
- **Minimum Occupancy:** The minimum occupancy captures the lowest observed LLC occupancy during execution. It reflects periods where the application experiences reduced cache demand and helps characterize the lower bound of the application’s occupancy behavior.
- **Maximum Occupancy:** The maximum occupancy captures the highest observed LLC occupancy during execution. Applications with aggressive memory behavior tend to exhibit larger peak occupancies, making this feature useful for identifying workloads with bursty cache demand.
- **25th Percentile Occupancy:** The 25th percentile represents the occupancy level below which 25% of the observations lie. This feature captures the lower stable region of the occupancy distribution while being less sensitive to outliers than the minimum occupancy.
- **75th Percentile Occupancy:** The 75th percentile represents the occupancy level below which 75% of the observations lie. This feature characterizes the upper stable region of the occupancy distribution and helps capture the spread of occupancy behavior across execution intervals.

Collectively, these features capture the average LLC demand, occupancy stability, temporal variability, and overall occupancy distribution shape of the application. Importantly, although instantaneous occupancy values vary significantly across executions due to randomized LLC allocations, these aggregate statistical summaries remain highly stable for the same application. Consequently, they form robust fingerprints that enable reliable application classification despite the randomness introduced by SCALE.

5.3.6 Classification Methodology

After extracting statistical features from the occupancy traces, we use a decision tree classifier to identify the co-running victim application. The classifier learns application-specific

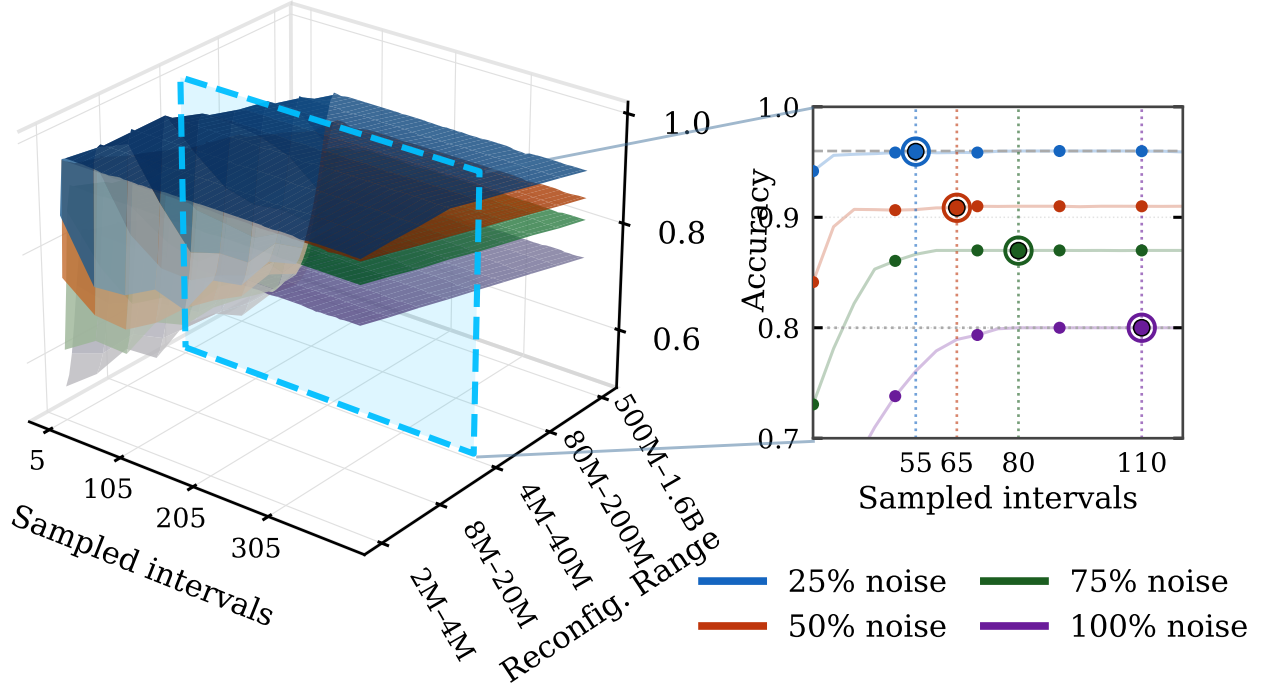


Figure 5.4: Classification accuracy with different reconfiguration interval ranges (in cycles), noise, and sampled intervals.

occupancy patterns from the extracted statistical summaries and uses them to distinguish workloads during the attack phase.

We choose a decision tree classifier for three primary reasons. First, decision trees learn interpretable threshold-based rules, making it easier to understand which occupancy characteristics contribute most to application differentiation. Second, they effectively separate applications using simple statistical features such as average occupancy and occupancy variability. Finally, the use of a lightweight classifier demonstrates that successful fingerprinting does not require sophisticated machine learning models; the leakage present in long-term occupancy statistics is already sufficiently strong for simple classifiers to achieve high accuracy.

During training, the classifier learns stable occupancy thresholds that separate applications based on their long-term LLC usage behavior. For example, applications with consistently higher LLC demand tend to form distinct regions in the feature space compared to applications with lower or highly variable occupancy behavior. During inference, the attacker extracts the same statistical features from an unknown victim’s occupancy trace and feeds them to the trained classifier. The classifier then predicts the most likely victim application based on the learned occupancy patterns.

5.3.7 Impact of Noise and Reconfiguration Randomness

Figure 5.4 shows the impact of differential privacy noise, reconfiguration interval randomness, and attacker observation duration on classification accuracy.

We observe that the classification accuracy consistently improves as the attacker monitors LLC occupancy over more sampling intervals. Initially, the injected randomness causes significant fluctuations in occupancy traces, making short-term application identification difficult. However, as the observation duration increases, the attacker accumulates a larger number of occupancy samples, allowing the long-term statistical behavior of the application to gradually emerge. Consequently, the classification accuracy eventually saturates after a sufficient number of observations. Higher differential privacy noise levels reduce the short-term distinguishability of applications by increasing fluctuations in LLC allocations. Similarly, larger reconfiguration ranges introduce additional temporal randomness by changing allocations less predictably over time. Both mechanisms therefore delay the saturation point because the attacker requires more samples to average out the injected randomness and recover stable occupancy characteristics.

The highlighted plane in Figure 5.4 corresponds to SCALE operating with a reconfiguration interval varying between 4M and 40M cycles. This plane provides a zoomed-in view of how the required number of sampling intervals changes with different noise levels. We observe that larger noise values and wider reconfiguration ranges increase the observation duration required for the attack to converge, but do not fundamentally eliminate the leakage.

In particular, although increasing reconfiguration randomness slows convergence by reducing short-term correlation between consecutive allocations, the long-term occupancy statistics remain recoverable once the attacker observes sufficiently long execution intervals.

Therefore, neither increased noise nor higher reconfiguration randomness fundamentally prevents the attack. Instead, these mechanisms primarily increase the number of observations required for successful fingerprinting. Once the attacker observes sufficiently long occupancy traces, the underlying statistical properties of the application become recoverable despite the randomized perturbations. This observation highlights an important limitation of relying solely on randomized LLC allocations for security. While randomness can effectively obscure instantaneous occupancy behavior, it does not eliminate stable long-term statistical signatures that enable occupancy-based fingerprinting.

5.3.8 Sensitivity Analysis

To evaluate the robustness of the attack, we perform sensitivity studies across multiple hardware configurations by varying cache hierarchy parameters and hardware prefetching mechanisms. Specifically, we analyze the effectiveness of the attack under different L1D sizes, L2 sizes, LLC capacities, and in the presence of modern hardware prefetchers. We further evaluate the attack using two recent prefetching mechanisms, IPCP [24] and Berti [22], both of which significantly alter memory access behavior and cache occupancy patterns. Since prefetchers can modify cache usage characteristics by bringing data into the cache proactively, they could potentially interfere with occupancy-based fingerprinting by introducing additional noise into occupancy traces.

Despite these variations, the attacker consistently achieves more than 90% classification accuracy across most realistic system configurations. This observation indicates that the statistical occupancy characteristics exploited by the attack remain sufficiently stable even under different cache organizations and aggressive prefetching behavior. We observe a noticeable reduction in attack effectiveness only for extremely large LLC configurations, such as 100MB LLC capacity per core. In such cases, applications occupy a relatively smaller fraction of the total cache space, reducing occupancy contention and weakening the observable occupancy signal available to the attacker. However, LLC sizes of this scale are impractical for commodity multicore systems and are therefore unlikely to mitigate the attack in real-world deployments. Overall, these results demonstrate that the attack remains highly robust across a wide range of realistic hardware configurations and memory hierarchy behaviors.

5.3.9 Takeaway

SCALE’s security argument is fundamentally based on the assumption that continuous randomization prevents the reproduction of identical LLC allocation sequences across executions. Since differential privacy introduces independent noise during every reconfiguration interval, the probability of observing the exact same allocation trace becomes extremely small over long executions.

However, this reasoning implicitly assumes that successful fingerprinting requires exact sequence matching. Our results show that this assumption does not hold for occupancy-based attacks. In practice, the attacker does not rely on reproducing identical occupancy traces. Instead, the attack leverages long-term statistical properties of LLC occupancy, such as average occupancy, occupancy variability, and distributional characteristics, which remain

remarkably stable across executions despite randomized perturbations.

Consequently, although SCALE successfully obscures instantaneous LLC allocation behavior, it does not eliminate the underlying statistical structure of application occupancy patterns. Over sufficiently long observation intervals, the injected randomness gradually averages out, allowing the attacker to recover stable application fingerprints. This reveals a fundamental limitation of differential privacy-based cache randomization mechanisms: obscuring individual allocation instances is insufficient if aggregate statistical characteristics remain observable over time. Therefore, while SCALE reduces short-term occupancy leakage, it does not fully prevent long-term occupancy-based application fingerprinting.

Chapter 6

Comparative Analysis of Dynamic Cache Partitioning Techniques

The previous two chapters independently analyzed two representative secure dynamic cache partitioning techniques: *Bespoke Cache Enclave (BCE)*, representing application-demand-based partitioning, and *SCALE*, representing LLC-usage-based partitioning. Although both techniques were proposed to achieve strong security and efficient cache utilization, they adopt fundamentally different approaches for dynamic cache allocation.

BCE allows applications to explicitly request cache resources, whereas SCALE dynamically redistributes cache ways based on runtime cache utility. Consequently, the source of dynamism differs significantly between the two techniques, resulting in different forms of information leakage. This chapter provides a comparative analysis of these two representative techniques from the perspectives of security, performance, usability, hardware complexity, and overall practicality. Such a comparison provides a broader understanding of the trade-offs involved in designing secure dynamic cache partitioning mechanisms.

6.1 Comparison of Design Choice

Although both BCE and SCALE dynamically partition the Last-Level Cache (LLC), they fundamentally differ in how allocation decisions are made. BCE follows an **application-demand-based** approach, where applications explicitly request cache resources through privileged allocation and deallocation operations. Once allocated, the corresponding cache clusters remain exclusively owned by the requesting domain until explicitly released. In contrast, SCALE follows an **LLC-usage-based** approach, where applications do not directly

participate in cache allocation. Instead, hardware continuously monitors cache behavior using Utility-Based Cache Partitioning (UCP) and periodically redistributes cache ways to maximize overall cache utility. To prevent predictable allocation behavior, SCALE further incorporates randomized reconfiguration and differential privacy mechanisms.

Therefore, the primary distinction between the two techniques lies in the entity responsible for cache allocation:

- **BCE:** Allocation is explicitly controlled by the application.
- **SCALE:** Allocation is autonomously determined by hardware based on runtime cache behavior.

6.2 Security Comparison

Both techniques eliminate traditional contention-based attacks through cache partitioning. However, our analysis demonstrates that the dynamic allocation mechanisms themselves introduce new attack surfaces.

BCE enforces strict set-level isolation by allocating exclusive cache clusters to different security domains. Consequently, traditional conflict-based attack and occupancy-based attacks become significantly more difficult. However, the allocation mechanism itself leaks information. Our analysis identified two covert communication channels. These channels arise from observable differences between successful and unsuccessful cluster allocations. Experimental evaluation demonstrated average covert-channel bandwidths of 2.61 Mbps and 4.46 Kbps, respectively. We additionally investigated occupancy-based process fingerprinting under BCE. Since cache clusters remain exclusively owned by individual applications, only coarse allocation information becomes observable. As a result, fingerprinting opportunities are significantly limited.

SCALE, unlike BCE, continuously redistributes cache ways according to runtime cache utility. To conceal allocation behavior, SCALE introduces multiple randomized layer and provides security guarantee via differential privacy. Despite these protections, our experiments demonstrate that long-term cache occupancy remains strongly correlated with application memory behavior. By exploiting these occupancy patterns, we achieved approximately 96% application fingerprinting accuracy. Unlike BCE, SCALE does not expose explicit allocation interfaces. Nevertheless, continuous runtime adaptation unintentionally leaks workload characteristics through aggregate cache occupancy.

6.3 Performance Comparison

BCE primarily emphasizes strong cache isolation. Since applications explicitly reserve cache clusters, execution becomes highly predictable with minimal interference from co-running applications. However, cache utilization depends heavily on accurate estimation of application cache demand. Over-allocation or under-utilization may reduce overall system efficiency.

SCALE instead prioritizes overall cache utilization. By dynamically reallocating cache ways according to observed runtime utility, memory-intensive applications automatically receive larger cache allocations whenever beneficial. Consequently, **SCALE** adapts more effectively to changing workload characteristics and generally achieves higher cache utilization.

6.4 Implementation Complexity

The two techniques differ not only in their allocation policies but also in the level of software involvement and hardware support required for deployment.

BCE follows an application-demand-based approach in which applications explicitly request and release cache resources. Consequently, software is responsible for estimating cache requirements, invoking allocation and deallocation primitives, and managing cache ownership throughout execution. Incorrect estimation of cache demand may lead to inefficient cache utilization or performance degradation. While this explicit management provides greater software-level control, it also increases programming and system management overhead.

SCALE, in contrast, is largely transparent to software. Applications are unaware of the underlying cache partitioning mechanism, while the hardware autonomously monitors cache behavior, estimates utility using Utility-Based Cache Partitioning (UCP), and dynamically redistributes cache ways. As a result, **SCALE** requires minimal software involvement and can be integrated into existing systems without application modifications. This reduction in software complexity, however, comes at the expense of substantially higher hardware complexity. **SCALE** requires several additional hardware components.

Overall, **BCE** offers greater software-level control with comparatively simpler hardware support, whereas **SCALE** minimizes software involvement by relying on considerably more sophisticated hardware mechanisms. Therefore, the choice between the two depends on the desired trade-off between software flexibility, deployment simplicity, and hardware complexity.

6.5 Overall Comparison

Table 6.1 summarizes the qualitative comparison between BCE and SCALE across several architectural, security, performance, and implementation aspects. Although both techniques aim to provide secure dynamic cache partitioning while maintaining efficient cache utilization, they achieve these objectives through fundamentally different design philosophies, leading to distinct security-performance trade-offs.

Table 6.1: Comparison between BCE and SCALE

Metric	BCE	SCALE
Partitioning Tech- nique	Set Partitioning	Way Partitioning
Allocation Policy	Application-demand-based	LLC-usage-based
Allocation Control	Application / Operating System	Hardware
Primary Design Goal	Strong spatial isolation	Performance-aware secure allocation
Observed Information Leakage	Allocation/deallocation covert channels	Occupancy-based finger- printing
Covert Channel	Present	Not observed
Fingerprinting Resis- tance	High	Low (96% fingerprinting ac- curacy)
Performance Adapt- ability	Moderate	High
Software Involvement	High	Low
Hardware Complexity	Moderate	High
Overall Cache Utiliza- tion	Moderate	High

As shown in Table 6.1, the two representative classes of secure dynamic cache partitioning techniques expose fundamentally different security-performance trade-offs.

Application-demand-based techniques such as BCE provide stronger spatial isolation and significantly reduce occupancy-based information leakage. However, exposing explicit allocation and deallocation primitives unintentionally creates covert communication channels through observable allocation behavior. Consequently, while BCE effectively limits occupancy-based fingerprinting, it introduces new avenues for covert communication.

Conversely, LLC-usage-based techniques such as SCALE maximize cache utilization by continuously adapting cache allocations according to runtime application behavior. Although randomization and differential privacy considerably increase the difficulty of exploiting allocation decisions directly, our analysis demonstrates that long-term cache occupancy remains sufficiently correlated with application characteristics to enable process fingerprinting with high accuracy. These observations indicate that the principal source of information leakage has shifted from shared cache contention to the dynamic resource management mechanisms themselves. Therefore, future secure cache partitioning techniques should protect not only cache contents but also the runtime allocation policies and metadata associated with resource management. Overall, our comparative analysis leads to three important observations:

1. Application-demand-based partitioning provides stronger resistance against occupancy-based fingerprinting but introduces explicit covert channels through dynamic allocation operations.
2. LLC-usage-based partitioning achieves better cache utilization and requires minimal software involvement; however, its runtime allocation behavior continues to leak sufficient statistical information for accurate application fingerprinting.
3. Neither technique completely eliminates information leakage. Instead, both shift the leakage from cache contention to different aspects of dynamic resource management, highlighting the need for future secure cache partitioning architectures to jointly secure cache contents, allocation decisions, and runtime resource management.

Chapter 7

Conclusion and Future Work

We revisited the security assumptions of modern secure dynamic cache partitioning techniques and showed that runtime allocation behavior itself can introduce new forms of information leakage. We studied two representative techniques, BCE and SCALE, representing application-demand-based and LLC-usage-based dynamic partitioning respectively.

For BCE, we demonstrated that dynamic allocation and deallocation behavior can be exploited to construct covert channels using allocation latency and memory-access timing differences. Although BCE significantly limits occupancy-based fingerprinting through strict cluster-level isolation, observable allocation behavior still leaks information across domains. For SCALE, we showed that occupancy-based fingerprinting attacks remain feasible despite randomized allocation and differential privacy-based noise injection. Our evaluation achieved 96% application fingerprinting accuracy, indicating that long-term LLC occupancy patterns still reveal application-specific behavior. Overall, our results show that eliminating direct cache contention alone is insufficient to guarantee security in dynamic partitioned cache architectures. Runtime allocation behavior, reconfiguration policies, and occupancy variations themselves can become sources of leakage.

As future work, secure cache architectures should incorporate leakage-aware allocation policies that minimize observable runtime behavior during cache partitioning. Further investigation is also required to understand the interaction of secure cache partitioning with other shared microarchitectural components such as hardware prefetchers, MSHRs, coherence structures, and DRAM systems. Developing formal leakage models and adaptive security mechanisms for dynamic cache partitioning also remains an important direction for future research.

Chapter 8

Acknowledgements

I would like to sincerely thank my advisor, **Prof. Biswabandan Panda**, for his continuous guidance, encouragement, and support throughout my time at this institution. His mentorship, valuable insights, and patience greatly influenced this research and contributed significantly to my academic development.

I am also thankful to **Abhijaat Bharadwaj** for his collaboration and contributions to this work. His suggestions, technical knowledge, and constructive discussions helped improve the quality and depth of the research.

I would further like to acknowledge my fellow **CASPER** lab members, especially **Sweta** and **Shubham Roy**, for their constant support, insightful discussions, and brainstorming sessions, which greatly contributed to this work.

Prerna Priyadarshini

IIT Bombay

Bibliography

- [1] “ChampSim simulator,” <http://github.com/ChampSim/ChampSim>, May 2020.
- [2] “DDR5 SDRAM,” https://en.wikipedia.org/wiki/DDR5_SDRAM, 2020.
- [3] “DDR5 memory standard: An introduction to the next generation of dram module technology,” <https://www.kingston.com/en/blog/pc-performance/ddr5-overview>, Jan. 2024.
- [4] K. Arkan, A. Farrell, W. Z. Cen, J. McMahon, B. Williams, Y. D. Liu, N. Abu-Ghazaleh, and D. Ponomarev, “TEE-shirt: Scalable leakage-free cache hierarchies for tees,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2024, pp. 1–18.
- [5] A. Bhatla, Navneet, and B. Panda, “The maya cache: A storage-efficient and secure fully-associative last-level cache,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 32–44.
- [6] 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*, 2019, pp. 42—56.
- [7] A. Chakraborty, N. Mishra, S. Saha, S. Bhattacharya, and D. Mukhopadhyay, “On the amplification of cache occupancy attacks in randomized cache architectures,” *arXiv preprint arXiv:2310.05172*, pp. 1–19, 2023.
- [8] 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)*, 2020, pp. 451–468.

- [9] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, “CHUNKED-CACHE: On-demand and scalable cache isolation for security architectures,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022, pp. 1–24.
- [10] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, “Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1—21, 2012.
- [11] J. Drean, “Casa: end-to-end quantitative security analysis of randomly mapped caches,” Ph.D. dissertation, Massachusetts Institute of Technology, 2020.
- [12] C. Dwork, F. McSherry, K. Nissim, and A. Smith, “Calibrating noise to sensitivity in private data analysis,” in *Theory of Cryptography*, S. Halevi and T. Rabin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 265–284.
- [13] C. Dwork and A. Roth, “The algorithmic foundations of differential privacy,” *Found. Trends Theor. Comput. Sci.*, vol. 9, no. 3–4, pp. 211–407, Aug. 2014. [Online]. Available: <https://doi.org/10.1561/04000000042>
- [14] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, “Scatter and split securely: Defeating cache contention and occupancy attacks,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2273–2287.
- [15] Gururaj, Kariyappa, Sanjay, and Moinuddin, “Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning,” 09 2021, pp. 37–49.
- [16] N. R. Holtryd, M. Manivannan, and P. Stenström, “SCALE: Secure and scalable cache partitioning,” in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023, pp. 68–79.
- [17] A. Kalita, Y. Yang, A. K. Murali, and A. Venkat, “Ceviche: Capability-enhanced secure virtualization of caches,” in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 4082–4098.
- [18] 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)*. IEEE, 2018, pp. 974–987.
- [19] D. Kumar, S. Y. Chavhan, B. Panda, and V. Gupta, “How sharp is SHARP?” in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019, pp. 1–12.
 - [20] 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)*. IEEE, 2016, pp. 406–418.
 - [21] 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 (S&P)*, 2015, pp. 605–622.
 - [22] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, “Berti: An accurate local-delta data prefetcher,” in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 975–991.
 - [23] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of aes,” in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed., 2006, pp. 1–20.
 - [24] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.
 - [25] 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.
 - [26] 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)*. IEEE, 2018, pp. 775–787.
 - [27] 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.

- [28] 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*, 2009, pp. 77—84.
- [29] G. Saileshwar, S. Kariyappa, and M. Qureshi, “Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning,” in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021, pp. 37–49.
- [30] 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.
- [31] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, “Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 955–969.
- [32] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, “Composable cachelets: Protecting enclaves from cache Side-Channel attacks,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2839–2856.
- [33] S. Vadhan, *The Complexity of Differential Privacy*. Springer, Yehuda Lindell, ed., 2017, pp. 347–450. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-57048-8_7
- [34] 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.
- [35] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, “Secdcp: secure dynamic cache partitioning for efficient timing channel protection,” in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1—6.
- [36] 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*, 2007, pp. 494—505.

- [37] Wei, Boya, Zihan, Zhenzhen, Wenhao, and Peng, “Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it,” in *Proceedings - 2021 IEEE Symposium on Security and Privacy, SP 2021*, 2021, pp. 955–969.
- [38] 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)*, 2019, pp. 675–692.
- [39] 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)*, 2014, pp. 719—732.
- [40] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, “A new prime and probe cache side-channel attack for cloud computing,” in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 2015, pp. 1718—1724.