ARCHITECTING SECURE PROCESSOR CACHES

A Dissertation
Presented to
The Academic Faculty

By

Gururaj Saileshwar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2022

ARCHITECTING SECURE PROCESSOR CACHES

Thesis committee:

Dr. Moinuddin Qureshi, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Christopher Fletcher
Department of Computer Science
University of Illinois Urbana-Champaign

Dr. Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Date approved: July 8, 2022

Genius is one percent inspiration, ninety-nine percent perspiration

Thomas Alva Edison

To my parents

ACKNOWLEDGMENTS

This PhD has been a long journey and I wouldn't be at the finish line without all the help and support I received from my mentors, family, and friends.

Mentors and Colleagues

First, I am deeply thankful to my advisor Prof. Moinuddin Qureshi. I knew very little about doing good research before I started working with him. So, I am extremely grateful he took a bet on me when he first offered me an internship and then mentored me through my PhD all these years. All of my research accomplishments, I owe to Moin.

I am as grateful to my mentor from my undergraduate program at IIT-Bombay, Prof. Bipin Rajendran. Without Bipin's encouragement to apply to graduate school and his timely recommendation to Moin, I would not be where I am today.

A lot of what I learned at Georgia Tech, I learned from my colleagues in the Memory Systems Lab, who over the years became friends and family. I am extremely grateful for the mentorship of Prashant Nair (friend, philosopher and guide) and the companionship of Swamit Tannu (labmate, roommate, coffee-buddy) – the path through my PhD became considerably easier because I followed in your footsteps. The PhD journey has been extremely enjoyable thanks to the great company of other colleagues ahead of me (Chia-Chen Chou, Jian Huang, Vinson Young) and also those after me (Mohammad Arjomand, Poulami Das, Sanjay Kariyappa, Anish Saxena, Aditya Rohan, Ramin Ayanzadeh, Narges Alavisamani).

I am deeply thankful to my collaborators who have taught me a lot more than I could have learned on my own – Prof. Chris Fletcher, who mentored me when I was just starting my work on side channels, and Prof. Taesoo Kim, with whom I wrote my first security conference paper. I am also very grateful to Prof. Daniel Gruss who welcomed me to his group in TU Graz, Austria for a summer internship – I learned a lot about offensive security research from him and his students.

I am also thankful to all my internship mentors – Wendy Elsasser, Prakash Ramrakhyani, and Jose Joao at ARM Research, Ken Grewal at Intel Labs, Muntaquim Chowdhury at Microsoft, and Rick Boivie, Alper Buyuktosunoglu, and Tong Chen at IBM Research – their guidance, advice, and industry perspective helped me grow considerably as a researcher.

I feel fortunate to have had great mentor figures in the computer architecture community. I am thankful for all the conversations about research I have had with Mengjia Yan, who has been a guiding light for a better part of my PhD. I am also thankful to Akshitha Sriraman, Wenjie Xiong, Dimitrios Skarlatos, Nader Sehatbaksh, and Saugata Ghose, for their advice during my job search.

Lastly, I am grateful to my thesis committee – Prof. Tushar Krishna, Prof. Milos Prvulovic, Prof. Taesoo Kim, and Prof. Chris Fletcher – I am glad to have had their mentorship and their constructive feedback throughout the PhD.

Friends and Well-Wishers

This PhD journey seems unimaginable without the support of my close friends, Ananda Samajdar, Poulami Das, Moumita Dey, Divyakiran Kadiyala, and earlier in my PhD, Karthik Rao, Swamit Tannu, Amruta Vidwans, and Vinson Young. You guys have been there with me through all the ups and downs. Also, life in the US would not have been as joyful without the friendship of Jayant and Vrushali in the initial days in Atlanta, or the companionship of Gaurav, Bhowmick, and Akshat on road trips to national parks. Special thanks to Gaurav and Anushree for welcoming me to their home and giving me the comforts of home-cooked food as I write this thesis.

I would also like to thank some of my favorite coffee shops – Octane, Amelie's, East Pole, and Chattahoochee Coffee in Atlanta, Mozart's and Spokesman in Austin, Insomnia in Hillsboro, and Zoka and Milstead in Seattle – their WiFi and electricity supported the writing of several of my papers and their delicious Cortados fueled the ideas in those papers.

Family

Lastly, my PhD would not have been possible without the relentless support and sacrifices from my family. I am forever thankful to my girlfriend and now-fiance, Anisha, for her support. She has been my rock throughout this journey, kept me grounded through the highs of paper acceptances and the lows of COVID, and helped me succeed in this endeavor.

I am also thankful to my brother, Siddesh. His early encouragement and support were instrumental in my going to graduate school and getting a PhD. He has been an inspiration and one of my biggest cheerleaders throughout this journey.

To my parents, I dedicate this thesis. They always encouraged me to follow my dreams; they supported my decision to quit my job and pursue graduate studies abroad. Through ups and downs, and in sickness and in health, they've continued their unwavering support and given me the courage to pursue my ambitions. I am forever indebted to them for the values they instilled in me, that kept me going throughout this PhD and in life. Amma and Appa – thank you for everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Caches in modern processors enable fast access to data and help alleviate the performance overheads from slow access to DRAM main-memory. While sharing of cache resources between multiple cores, especially the last-level cache, boosts cache utilization and improves system performance, it has been shown to cause serious security vulnerabilities in the form cache side-channel attacks. Different cores of a system can simultaneously run sensitive and malicious applications which can contend for the shared cache space. As a result, accesses of a sensitive application can influence the cache utilization and the execution time of a malicious application, introducing a side-channel of information leakage.

Such cache interactions between a sensitive victim and a malicious spy have been shown to allow leakage of encryption keys, user-sensitive data such as files or browsing histories, confidential intellectual property such as machine-learning models, etc. Similarly, such cache interactions can also be used as a channel for covert communication between two colluding malicious applications, when direct communication via network ports is disabled. The focus of this thesis is to develop principled and practical mitigation for such cache side channel and covert channel attacks.

To develop principled defenses, it is necessary to develop a deep understanding of attacks. So, first, this thesis investigates the capabilities of attackers and in the process develops a new cache covert channel attack called Streamline, which is considerably faster than current state-of-the-art attacks, with fewer requirements. With an asynchronous and flushless information transmission protocol, Streamline reaches bit-rates of more than 1 MB/s while being applicable to all ISAs and micro-architectures. This demonstrates the need for effective defenses against cache attacks across all platforms.

Second, this thesis develops new principled and practical defenses utilizing cache location randomization. Randomized caches obfuscate the mappings of addresses to cache locations to prevent malicious programs from inferring contention patterns on shared last-

level caches with victim programs. However, successive defenses relying on randomization have been broken by recent attacks. To end the arms race in randomized caches, this thesis proposes a principled defense, MIRAGE, which provides the security of a fully-associative design in a practical manner for randomized caches. This eliminates set-conflicts and set-conflict based cache attacks in a future-proof manner.

Third, this thesis explores cache-partitioning based defenses to eliminate all potential cache side channels through shared last-level caches. Such defenses map mistrusting applications to isolated cache partitions, thus preventing any information leakage across applications through cache state changes. However, existing solutions are not scalable or do not allow flexible usage of DRAM and cache resources. To address these problems, this thesis provides a scalable and flexible cache-isolation framework, Bespoke Cache Enclaves, supporting hundreds of partitions independent of memory utilization. This work enables practical adoption of cache-isolation defenses against cache side-channel attacks.

Lastly, this thesis develops techniques to secure caches against exploitation in transient execution attacks. Attacks like Spectre and Meltdown exploit processor speculation to illegally access secrets and leak these out through cache covert channels, *i.e.*, making transient changes to processor caches. This thesis enables CleanupSpec, one of the first defenses against such attacks, which reverses speculative modifications to caches on misspeculations, to limit such transient information leakage via caches. This solution prevents caches from being exploited by attacks like Spectre with minimal overheads.

Overall, this thesis enables several techniques that provide principled yet practical security for processor caches against side channels and covert channels. These techniques can potentially enable the wide adoption of secure cache designs in future processors and support efforts to enable confidential computing in systems.

CHAPTER 1
INTRODUCTION

Over the past several decades, relentless transistor scaling has enabled orders of magnitude gains in performance and efficiency for computer hardware. This has enabled computers with smaller form factors to perform extremely complex tasks and computing devices to become pervasively embedded in our daily lives. Today, we store some of our most personal and private data in cloud computers, and generate and consume such data on personal computing devices (like mobiles and laptops). As a result, data security and privacy have become a foundational requirement[1] for all computing systems, mobile to cloud.

Traditionally, computing hardware has been optimized not for security but for performance. An important performance optimization in processors is caching frequently used data on-chip for fast access (in a few ns), to avoid slow access to data from the main-memory (close to 100 ns) each time. While caching can significantly boost performance, it also introduces variation in execution time of programs which can leak information through a new security vulnerability, *cache side-channel attacks*.

Side channels are observable side-effects of execution of sensitive applications that can leak secret information. Side channels exist in real-life: for example, a safe cracker can guess the secret combination of a safe by listening to the sound the dial of a safe makes as it moves. Similarly, as sensitive code executes on a system, its secrets can be leaked based on secret-dependent side-effects observable by an adversary, such as variations in its execution time or other micro-architectural changes. A classic example of such vulnerabilities is cache side channels, which result from the timing variation introduced by processor caches. Through cache interactions with sensitive programs, malicious programs have been shown capable of leaking users' secret encryption keys [2], users' private data like web-browsing

[1]The annual cost of cyber-crime is over $1 trillion (1% of the world's economy) in 2020, and growing. [1]

history [3], and even confidential intellectual property like ML models [4]. This highlights the potency of cache side-channel attacks. Thus, security, particularly against cache side-channel attacks, has become an important consideration for future hardware designs.

1.1 Understanding the Problem of Cache Side-Channel Attacks

Modern processors have a multi-level cache hierarchy, typically consisting of private L1 and L2 caches per core and a larger Last-Level-Cache (LLC) shared across multiple cores for high utilization. Thus, a malicious spy process and a victim process can run simultaneously on different cores and contend for the shared cache, leading to side-channel leakage.

In a cache side-channel attack, a spy can monitor the cache access patterns of the sensitive victim program to infer its secrets. For example, the secret-dependent addresses accessed by the victim change the shared LLC state (*e.g.*, evict addresses of the spy process). The spy can infer these cache state changes (*e.g.*, based on a slow cache miss for its addresses) and learn the access pattern and any dependent secrets of the victim.

Alternatively, the cache may also be exploited as a covert channel, *i.e.*, used as a medium to covertly transmit information between two attacker-controlled entities. For example, information can be transmitted based on patterns of cache hits and misses on shared cache lines (*e.g.*, bit value 1 for a hit, 0 for a miss). Such cache covert channels have been heavily used in transient execution attacks like Spectre and Meltdown [5, 6] to leak secrets.

1.2 Challenges in Mitigating Cache Side-Channel Attacks

Although cache side-channel attacks have been known for more than a decade, principled yet practical defenses that can eliminate such attacks have been elusive. In recent years, two main classes of defenses have emerged: randomization and partitioning. However, several challenges have limited the adoption of such defenses. On the one hand, new attacks have emerged that break existing defenses. On the other hand, several effective defenses have limited scalability and practicality. Below, we describe these challenges in detail.

***Challenge-1*. Limited Understanding of Attack Requirements.** Although cache attacks have been known for several years, the architectural primitives needed for cache attacks are not fully understood. Recent defenses have targeted primitives used in current cache attacks. For instance, defenses like SHARP [7] advocate for disabling unprivileged usage of `clflush` instruction in the x86 ISA, used by popular attacks like Flush+Reload [8] for eviction of targeted addresses. The ARM ISA, which does not permit unprivileged usage of cache line flush instructions, is thought to be immune to such attacks. This poses the question, can alternative eviction mechanisms (like cache thrashing) enable attacks as fast or as effective as current attacks? If so, current defenses would provide a false sense of security. This motivates the need for a more holistic understanding of attacks.

***Challenge-2*. Successive Randomization-Based Defenses Broken.** Recently, cache randomization [9, 10, 11] has emerged as a promising solution to mitigate cache attacks. By randomizing the mapping of addresses to cache sets, these defenses attempt to obfuscate the cache contention patterns between distrusting entities. However, successive randomized cache defenses have recently been broken by newer adaptive attacks [10, 12, 13]. This prompts the need for exploring principled randomization-based defenses that are capable of providing durable and future-proof security against cache attacks.

***Challenge-3*. Limited Scalability of Cache-Partitioning Defenses.** A principled defense against cache attacks is cache partitioning [14, 15, 16, 17, 18, 19, 20]. Such defenses map distrusting entities to disjoint portions of the cache to eliminate any cache contention between them and mitigate cache attacks. However, existing solutions are limited in scalability and flexibility due to constraints imposed by the cache design. Solutions using way-partitioning [16, 17, 18, 19, 20], only support as many partitions as the LLC associativity (16 - 32), and are not scalable to systems with hundreds of cores. Page-Coloring [14, 15, 21, 22] based solutions can only allocate memory to processes in the same ratio as cache sets, and this inflexibility is limiting at scale. To enable wide adoption, there is a need for new practical and scalable cache-partitioning based defenses.

***Challenge-4*. Limited Protection for Caches Against Transient Leaks.** Existing defenses against cache side-channels primarily protect against cross-core or cross-process information leakage. However, with the discovery of transient execution attacks like Spectre [5], Meltdown [6], and others, a new threat model has become relevant where information transmission happens within the same program, through a covert channel between transiently executed code and architecturally visible code. As caches are the dominant covert channel used in such attacks, it is critical to develop solutions to prevent such transient exploitation of caches to leak information. Yet, existing defenses against side-channel attacks provide limited protection against transient data leakage. This prompts the need for practical defenses for caches against transient leaks.

1.3 Goals of this Thesis

The goal of this thesis is to architect secure caches, and protect systems against cache side channels and covert channel attacks in a future-proof manner. To achieve this goal, this thesis aims to address the gap in the understanding of cache attacks, and the lack of principled and practical defenses. Specifically, this thesis has the following goals:

1. **Understand the capabilities of cache attacks**, through a study of existing cache covert channel attacks and the development of new attacks.

2. **Develop principled bottom-up randomization-based defenses** that can eliminate the most powerful cache side-channel attacks with minimal software support.

3. **Develop principled top-down isolation-based defenses** against all known cache side-channel attacks exploiting cache state, with support from system software.

4. **Harden caches against transient information leakage** to prevent exploitation in transient execution based attacks.

1.4 Thesis Statement

This thesis demonstrates that "*cache attacks can affect shared caches of all CPUs and micro-architectures, and principled defenses against such cache attacks can be practically implemented in next-generation hardware, by intelligently re-designing caches for security with minimal impact on system performance*".

1.5 Thesis Overview and Contributions

Towards enabling principled and practical security for caches, this thesis first studies existing attacks and their limitations and develops new attack strategies to highlight the real requirements of attacks. Then, this thesis develops new strategies for effective defenses.

1.5.1 Developing the Fastest Cache Covert Channel Attack with Fewer Limitations

First, this thesis attempts to understand the limitations of current attacks, by studying existing attacks in the covert channel setting. The fastest cache covert channel attacks are the Flush+Reload [8] and Flush+Flush [23] which achieve covert-channel communication bit-rates of 300-500 KB/s. These attacks require cacheline flush instructions (`clflush` on x86) for resetting the cache line state, which are available for unprivileged usage in x86 ISA. In the process of understanding the limitations of such attacks, this thesis discovers a new attack strategy for covert channels that results in a faster attack with fewer limitations.

The thesis observes that the bit-rate of existing attacks is bounded by a synchronous protocol requiring synchronization between sender and receiver after each bit. To design a faster attack, this thesis develops a new asynchronous transmission protocol (without the synchronization overheads). Moreover, the resetting of the cache state occurs automatically by the act of transmission, through cache thrashing, without flushes. This strategy enables the Streamline attack [24], with a bit-rate of 1.8 MB/s, which is faster than all existing attacks (due to its asynchronous nature) and is broadly applicable (due to lack of flushes).

The Streamline attack is the fastest known cache attack with a bit-rate 3× higher than the prior best attack and is applicable to all ISAs and micro-architectures. This highlights the need for effective and principled defenses on all platforms.

1.5.2 Designing Principled Randomization-Based Defenses

Second, this thesis designs principled randomization cache defenses, capable of providing future-proof security. While recent randomized cache defenses [9, 10, 11, 25, 26] attempt to provide security against cache side-channel attacks with randomized mappings of addresses to cache sets, they have successively been broken by advances in attacks [27, 10, 28, 29, 12, 30]. This thesis observes that the root cause of the vulnerability in recent randomized caches is that they allow deterministic evictions from the same set where a new line is installed. This allows the address-to-set mappings to be de-obfuscated, which renders such defenses vulnerable to set-conflict based cache attacks. To eliminate set-conflict based attacks, it is essential to eliminate set-conflict based evictions.

Towards a principled randomized cache defense, this thesis enables a fully-associative randomized design for last-level caches called MIRAGE [31], where new line installs result in evictions of randomly selected lines from the entire cache. This eliminates set-associative evictions based on installed addresses, eliminating the information leakage in randomized caches, and eliminates any conflict-based attacks. Moreover, MIRAGE implements such principled fully-associative randomization while retaining practical set-associative cache lookups, using a combination of over-provisioning of the cache and load-balancing hashing, based on Power-of-2-choices [32].

MIRAGE eliminates conflict-based side-channel attacks while incurring less than 2% slowdown and modest storage overhead of 17%-20%. While 2018-2020 saw 5 defenses get broken by 6 attacks, MIRAGE has remained unbroken since 2020, promising an end to the arms race in set-conflict based cache attacks.

1.5.3 Designing Scalable Cache-Partitioning Defenses

While randomized caches can prevent set-conflict based attacks, they still allow a spy to infer subtle information about the victim such as the LLC fraction used by a victim application, and are vulnerable to cache-occupancy attacks [3]. Cache-partitioning based defenses [14, 15, 16, 17, 18, 19, 20] can partition the cache into disjoint regions, allot each region to a different process, and prevent any contention for cache space between distrusting processes to prevent cache attacks. However, existing LLC partitioning mechanisms face practical limitations: way-partitioning [16, 17, 18, 19, 20] only supports a few tens of partitions and page-coloring [14, 15, 21, 22] is inflexible in terms of memory capacity.

To provide a scalable and flexible solution, this thesis proposes Bespoke Cache Enclaves [33], a cache design that enables fine-grain customizable partitions of the LLC along sets. The key idea of Bespoke is to leverage a dynamic cache set indexing logic that guides addresses of a particular program to only the allocated sets, regardless of the memory mappings. As a result, Bespoke can support hundreds of cache partitions simultaneously in the LLC, that can be managed independent of memory allocations. Moreover, it can even improve performance for applications with a customizable size of partitions, which are free from interference from other applications. Such a design can enable practical adoption of cache-partitioning as a principled defense against cache attacks.

Bespoke is a principled yet practical mitigation for cache attacks, providing scalable, flexible and customizable partitions in the LLC. It provides up to 512 isolated LLC partitions while incurring negligible storage overheads (2%) and slowdown (1%).

1.5.4 Securing Caches Against Transient Information Leakage

Transient execution attacks like Spectre [5], Meltdown [6], and others exploit processor speculation to access secrets and typically leak these out through transient changes to processor caches, *i.e.*, through a cache covert channel. Existing cache randomization or parti-

tioning based defenses do not prevent such attacks as they do not prevent transient changes to the cache state. Prior defenses against such attacks, such as InvisiSpec [34], that make speculation invisible to caches, incur considerable performance overheads.

This thesis observes that on mis-speculation, just as processors clear speculative register state, reversing speculative modifications to caches additionally can limit transient leakage via caches. With this observation, this thesis proposes CleanupSpec [35]. CleanupSpec tracks and reverses transient cache state changes, or randomizes them to obfuscate them. This technique protects the cache hierarchy against transient attacks with minimal performance impact (5% slowdown) and modest storage overhead of 1 kilobyte per core.

CleanupSpec provides low-overhead hardware mitigation for transient-execution attacks like Spectre. It hardens the memory system against speculative leaks with an average slowdown of 5%, which is 3x lower than prior solutions.

1.6 Organization of the thesis

The subsequent chapters of the thesis are organized as follows.

Chapter 2 provides background on prior attacks, covering cache side-channel and covert-channel attacks, and transient execution attacks.

Chapter 3 describes the design of the Streamline attack, the fastest known cache attack to covertly transmit information through caches.

Chapter 4 describes the design of MIRAGE, the principled randomized cache defense that eliminates set-conflict based cache attacks.

Chapter 5 describes Bespoke Cache Enclaves, the scalable cache-partitioning framework, that eliminates all cache side-channels exploiting cache state.

Chapter 6 describes the design of CleanupSpec, the technique to harden the memory hierarchy against transient execution attacks.

Chapter 7 concludes the thesis, and discusses potential directions for future work.

CHAPTER 2
BACKGROUND ON CACHE ATTACKS

This section provides a brief background on the attacks that are within the scope of this thesis: cache side channel and covert channel attacks, and transient execution attacks.

2.1 Cache Organization in Modern Processors

Processor caches are typically organized at the granularity of 64-byte cache lines. The cache consists of two structures – the *tag-store* and the *data-store*. For each cacheline, the metadata used for identification (e.g. address, valid-bit, dirty-bit) is called the *tag* and stored in the ”tag-store”, and there is a one-to-one mapping of the tag with the *data* of the line, which is stored in the ”data-store”. The tag-store is organized in a set-associative manner for efficient lookup where each address maps to a set (a group of contiguous locations), and each location in a set is called a way. Each set consists of w ways (typically 8 – 32).

2.2 Cache Side-Channel Attacks

Processors typically have a multi-level cache hierarchy for effectively caching frequently used data. Typically, L1 and L2 caches are private per core while the L3 cache or Last-Level-Cache (LLC) is shared across multiple cores. Different cores can simultaneously run distrusting programs of different origins: *e.g.*, a banking application on one core, and a web browser rendering a malicious website on another. The cache access pattern of a sensitive program on one core (victim) can change the shared LLC state and be inferred by a malicious program on another core (spy) based on variations in the latency of cache accesses of the spy, resulting in a side-channel leakage of information. Figure 2.1 shows the typical threat model for cache side channels targeting the last-level cache.

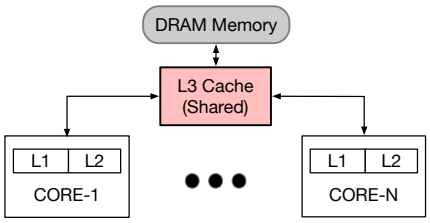


Figure 2.1: Threat Model: Shared LLC is the focus of cache attacks.

Cache side channel attacks were pioneered by Bernstein [36] and Percival [37] in 2004-2005. Since then, a large number of variants of cache attacks have been discovered. We classify these cache attacks broadly into three categories – conflict-based, shared-memory-based, and cache-occupancy based attacks.

2.2.1 Conflict-Based Cache Side-Channel Attacks

Such attacks exploit sharing of cache sets between programs running on different cores. In such attacks, the spy accesses addresses that map to the same cache set as the victim's address, and evict the victim address from the cache. Similarly, accesses by the victim can evict the spy's addresses mapping to the same set, allowing the spy to observe the victim's accesses. This episode of the spy and the victim evicting each other's addresses from a shared cache set is called a *set conflict*, and such attacks are called *conflict-based attacks*.

In the classic **Prime+Probe** [2, 38] attack shown in Figure 2.2, the spy primes (i.e. installs its lines in) cache sets shared with a victim, then allows the victim to execute and evict one of these spy lines. Later, the spy accesses its lines and measures the access time (using timer instructions such as `rdtsc` or `rdtscp` in x86), to infer which lines have a slow cache-miss due to eviction via set-conflict with victim accesses. This allows the spy to infer the set accessed by a victim and several bits of the victim's address, as the addresses have a deterministic mapping to cache sets. If these accesses were secret dependent, this also leaks the victim's secret. Such attacks can leak secret keys from AES T-table and RSA Square-Multiply algorithms [2], leak DNN models [4], and monitor user activity [39] among other malicious activities.

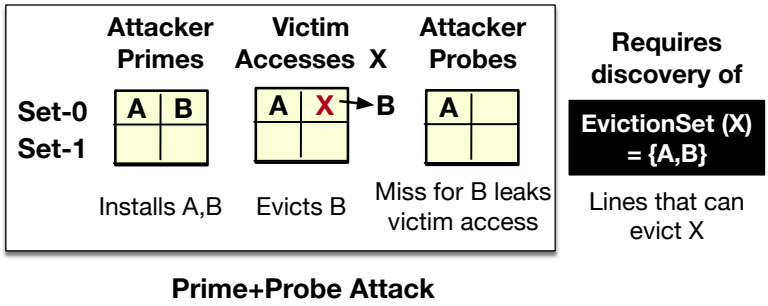


Figure 2.2: Example of Conflict-Based Attack (Prime+Probe).

Variants of Conflict-Based Attacks

Similar to the Prime+Probe attack, other attack variants exploit set conflicts.

Prime+Abort [40] is a variant that does not require the usage of timer instructions. Here the spy primes cache sets and waits in a transactional memory construct (*e.g.* Intel TSX transaction). If the victim evicts a spy line from one of the primed cache sets, it causes a transaction abort for the spy, allowing the spy to infer the victim access.

In **Evict+Time** [38], the spy uses set conflicts to evict specific victim lines and then allows the victim to execute and measures its execution time. If the victim execution is slowed due to a cache miss (instead of hit), the spy learns that the victim accessed that line.

Requirement of Conflict-Based Attacks: Eviction-Set Discovery

In all conflict-based attacks, as shown in Figure 2.2, the first step for an attacker is to generate an *eviction-set* for a victim address, *i.e.*, a minimal set of addresses mapping to the same cache set as the victim address which can evict the victim address from the cache. As commercial CPUs use undocumented mapping of addresses to sets, attacks algorithmically discover eviction-sets.

Liu et al. [2] first proposed an algorithm to discover eviction-sets by testing and eliminating addresses one at a time from a pool of randomly selected candidates, requiring $O(n^2)$ accesses to discover an eviction-set (where n is the number of lines in the cache).

Subsequent works by **Qureshi** [10] and **Vila et al.** [29] developed a faster algorithm taking $O(n)$ accesses, by eliminating groups of lines at a time from the set of potential candidates. **Song and Liu** [28] optimized these to reduce the constant factor for $O(n)$ algorithms.

2.2.2 Shared-Memory Based Cache Side-Channel Attacks

Such attacks exploit shared-memory addresses between the spy, which result in a single shared cache line in the cache common to both processes. Such shared cache lines can exist due to shared memory due to read-only sharing of shared libraries or de-duplication by the OS for saving memory such as in Linux KSM [41]. In such attacks, the spy can observe a cache hit on a shared cache line installed by the victim, and thus infer that a victim previously accessed that shared address.

In the classic **Flush+Reload** [8] attack shown in Figure 2.3, a spy first uses an instruction such as the `clflush` (x86) instruction to *flush i.e.*, evict a shared cacheline from the cache-hierarchy. Then, the victim executes and accesses the address, causing the shared cache line to be installed into the cache. The spy then *reloads*, *i.e.*, re-accesses the address and measures the latency: if this access is a cache hit, the spy can infer that the victim accessed the address; else, a cache miss indicates the victim did not access the address.

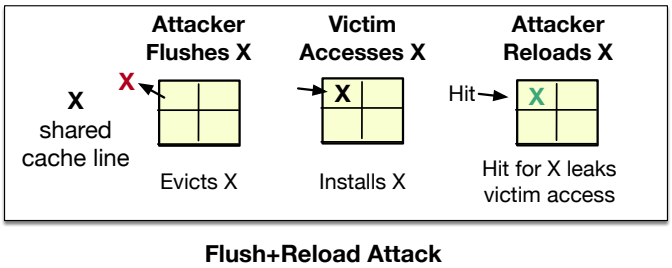


Figure 2.3: Example of Shared-Memory-Based Attack (Flush+Reload).

Evict+Reload [42] and **Thrash+Reload** [43] are variants of Flush+Reload, where the eviction of shared addresses from LLC is done either via set conflicts or via cache thrashing, *i.e.*, accessing a number of lines larger than the LLC capacity, respectively.

Flush+Flush [23] exploits latency variation for `clflush` based on whether a cache line is present or absent in the LLC. Here, the spy keeps flushing a shared address while measuring its latency; if the victim installs the line in LLC, the next flush is faster, else slower.

Other Attacks Exploiting Shared Memory

Other attacks exploit updates to the cache replacement policy to launch more stealthy attacks. For example, attacks like **Refresh+Reload** [44] and the one by **Xiaong and Szefer** [45] infer a victim access to a shared cache line, based on the replacement state update due to the victim access, which changes the order of any subsequent evictions of lines from the cache-set. Such attacks are stealthy as they do not introduce any additional cache-misses for the victim when the attack is in progress, and thus evade any detection attempts.

Take-a-way [46] exploits conflicts in the way-predictor tables in the L1 Caches of AMD CPUs to cause a variant of an Evict+ Reload (and also a Prime+Probe) attack. In this attack, a spy can evict a targeted shared cache line from the L1 Cache by accessing a conflicting address in the way-prediction table, which evicts the shared address from the table and also the L1 Cache. Subsequently, if the victim accesses the line, a hit can be observed on a reload by the spy, to complete the Evict+Reload attack.

2.2.3 Cache-Occupancy Based Side-Channel Attacks

Such attacks exploit the dynamic sharing of LLC space between a spy and the victim, where changes to the cache space used by the victim can leak information about the victim's execution. Such cache-occupancy based attacks exploit coarse-grained information (like cache space usage) unlike set-conflict or shared-memory based attacks that leak out the entire address trace of the victim in a fine-grained manner. Cache-occupancy based attacks allow subtle privacy breaches like fingerprinting websites browsed by a user.

For example, **Shusterman et al.** [3, 47] showed that the website browsed by a user can

be fingerprinted based on the changes to the web browser's cache working set over time while rendering the website. In such attacks, the spy periodically fills the LLC with its lines, then counts the number of its lines that get evicted over time due to the user's browser as it renders a website, to develop a website fingerprint. Subsequently, this fingerprint can be matched with an offline repository to identify the website.

A subsequent attack variant [47] showed that website fingerprinting attacks can be mounted with just HTML+CSS code in a malicious web page. As a result, these attacks can subvert even browser security measures like Chrome Site-Isolation [48], which renders web page content from different sources in separate processes, and Chrome Zero [49] which limits JavaScript and timers in web pages.

2.3 Cache Covert-Channel Attacks

Covert channels allow colluding processes to communicate without detection. Among the covert channels, cache covert-channels are typically the fastest and most reliable. These have been used in recent transient execution attacks like Spectre [5] and Meltdown [6], and covert transfer of information by malware [50]. An important consideration for covert-channel attacks is the bit-rate of information transmission and the bit-error-rate – a faster and more reliable channel allows for faster and more robust transmission. Table 2.1 shows the bit-rates and error-rates for recent cache covert-channel demonstrations.

Table 2.1: Prior Cache Covert Channels (Bit-Rate > 50 KB/s)

Attack	Attack Model	Targets Shared Memory	Bit-Rate	Bit Error Rate
Take-a-way [46]	Same-Core	✓	588 KB/s	1–3%
Flush+Flush [23]	Cross-Core	✓	496 KB/s	0.84%
Prime+Probe (L1) [37]	Same-Core	✗	400 KB/s	–
Flush+Reload [23]	Cross-Core	✓	298 KB/s	0%
Prime+Probe (LLC) [2]	Cross-Core	✗	75 KB/s	1%
Xiaong and Szefer [45]	Same-Core	✓	72 KB/s	<2%

2.3.1 Attacks Exploiting Shared Memory

Take-a-way [46] is the fastest same-core attack achieving a bit-rate of 588 KB/s between two hyper-threads running on the same physical core. In cross-core settings, **Flush+Flush** and **Flush+Reload** channels by Gruss et al. [23] are the fastest with bit-rates of 298 KB/s and 496 KB/s between two processes running on different physical cores. Lastly, the work by Xiaong and Szefer [45] exploits L1 cache replacement policies to transmit information at 72KB/s bit-rate between two hyper-threads on the same physical core.

2.3.2 Attacks Exploiting Set Conflicts (Without Shared Memory)

Prime+Probe based covert-channels were pioneered by Percival [37] on L1 Caches in a same-core setting. This work used an array as large as the L1 Cache and transmits a bit on each array-entry, to achieve a bit-rate of 400KB/s. Subsequent works [2, 51] demonstrate a Prime+Probe attack on the LLC in a cross-core setting, with a bit-rate of 75KB/s [2].

2.4 Transient Execution Attacks

Transient execution attacks exploit speculative execution in modern processors to illegally access secrets (not accessible via architecturally permissible data flows) and then encode these secrets through a covert channel to leak them out to an architecturally visible state (non-speculative code). Classic examples of such attacks include Spectre [5] and Meltdown [6], which access secrets by speculatively bypassing a bounds-check (by exploiting the branch predictor in case of Spectre) or a privilege check (in case of Meltdown), and transmit the data to non-speculative execution via a cache covert channel.

In such attacks, typically, the speculatively accessed secret is encoded as an address for a load instruction that installs the associated cache line transiently into the cache. Subsequently, the secret can be inferred by non-speculative code, by checking which address has a cache hit, as the cache state changes made transiently are preserved even after a

mis-speculation is detected and the correct-path execution starts.

The first wave of transient execution attacks included Meltdown and Spectre and their variants [52, 53, 54, 55]. The second wave of attacks included Micro-architectural Data-Sampling (MDS) attacks and their variants [56, 57, 58] which speculatively accessed secret data from load/store buffers. All of these attacks typically leak data from speculative to non-speculative execution through cache covert channels. Hence, this thesis focuses on mechanisms to provide protections for caches against such speculative leaks.

2.5 Goals of this Thesis

This thesis aims to improve the security of caches and improve the resilience of caches to side channels and covert channels. To that end, this thesis has the following goals:

1. Understand the capabilities of cache attacks by studying existing cache covert channel attacks and developing new attacks more capable than existing attacks.

2. Develop principled defenses against the most powerful cache side-channel attacks through bottom-up re-design of randomized caches.

3. Develop principled defenses against all known cache side-channel attacks exploiting cache state, through a top-down re-design of cache isolation frameworks.

4. Secure caches against transient information leakage to prevent their usage as a covert channel in transient execution attacks.

The subsequent chapters of this thesis describe how these goals are achieved.

CHAPTER 3

STREAMLINE - A NEW CACHE COVERT CHANNEL ATTACK

To design effective defenses, it is essential to analyze attacks and understand their limitations. To that end, this thesis first analyzes existing cache attacks and develops new attacks. As the fundamental mechanism used to leak data is the same in cache side channels and covert channels, for this chapter, we consider cache attacks in the covert channel scenario.

3.1 Context: Why Investigate Cache Covert Attack Capabilities?

Covert channels allow two colluding malicious processes (trojan and spy) to communicate without detection. Of all the covert channels, cache covert channels are the fastest and most reliable, and have been heavily used in recent attacks like Spectre [5], Meltdown [6], etc. Pushing the envelope on the bit-rate of cache covert-channel attacks helps re-evaluate the overall vulnerability of a system to data leakage via such channels, as faster covert channels allow exfiltration of payloads in shorter times. Moreover, investigating the limitations of such attacks helps inform defense strategies while designing future hardware.

As a result, this chapter [1] focuses on understanding the limitations in bit-rate for state-of-the-art cache covert-channel attacks and exploring the construction of newer attacks achieving higher bit-rates. Our default focus is on cross-core cache attacks, where a malicious sender and a receiver process execute on two different processor cores and attempt covert communication via accesses to the shared LLC, as such a setting is typical for a virtualized environment with a per-core resource allocation.

Limitations of Existing Attacks. The current fastest cross-core covert channels are flush-based attacks, such as Flush + Reload [8] and Flush + Flush [23] shown in Fig-

[1]This chapter is published as a paper "*Streamline: A Fast, Flushless Cache Covert-Channel Attack by Enabling Asynchronous Collusion*", appearing in the proceedings of ASPLOS 2021 [24].

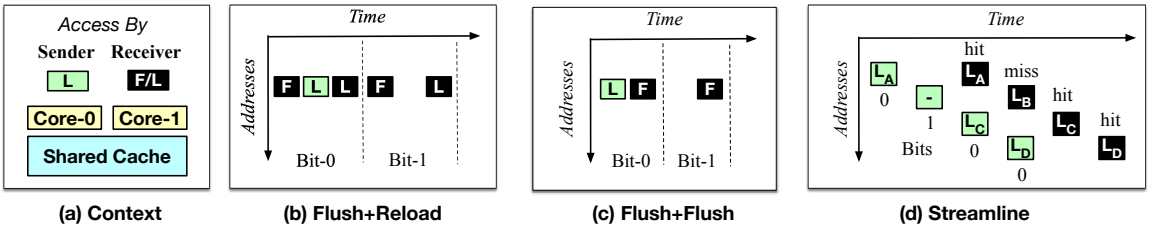


Figure 3.1: (a) Cache covert-channel attacks allow the colluding sender and receiver processes to transmit information via timing differences on accesses to shared caches. (b,c) Prior covert-channel attacks require synchronized transmission and flushes (F) in addition to loads (L) for each bit sent between the sender and the receiver: this limits the channel bit-rates to 298 KB/s (Flush+Reload) and 498 KB/s (Flush+Flush). (d) In Streamline, the sender and receiver asynchronously communicate on a large sequence of addresses without flushes (each bit transmitted on a new address), achieving a bit-rate of 1801 KB/s.

ure 3.1(b) and Figure 3.1(c). Gruss et al. [23] showed these attacks achieve bit-rates of 298 KB/s (Flush+Reload) and 496 KB/s (Flush+Flush) at less than 1% bit-error-rate.

The transmission rate of these channels is limited by two requirements: (1) synchronous communication between the sender and receiver, with each bit being transmitted during a coordinated time window, and (2) having to execute at least one flush and (one or two) load operations within the synchronous window for each bit. With such channels, it is difficult to obtain bandwidths higher than 1 MB/s (i.e. bit-period <125 ns) because the latency of flush is typically 50–70 ns and the latency of memory is approximately 100 ns. A bit period of less than 150 ns breaks down the channel due to loss of synchronization. Additionally, attacks requiring cacheline flush instructions are not universally applicable, especially for several ARM processors [59] where the unprivileged use of flush instructions is disabled by default (in ARM v8 ISA) or completely unsupported (in ARM v7 ISA).

3.2 Key Idea of Streamline - A Faster and Universal Cache Attack

Towards investigating the feasibility of a universal covert-channel attack, we enable *Streamline*, a flush-less attack that is also faster than all known covert-channel attacks. The key idea behind this attack is to enable the sender and receiver to communicate asynchronously over a large number of lines by using the cache to buffer data between the sender and the re-

ceiver, and relying on cache thrashing to naturally evict the resident lines (instead of using expensive clflush operations to explicitly evict lines).

The protocol starts with the sender and receiver sharing a large shared array a few tens of MBs in size (larger than the size of the LLC), rather than a single or a small number of addresses[2], like in prior attacks. The sender and the receiver have a pre-determined sequence of addresses within the array over which they transmit successive bits. The sender transmits on successive addresses without waiting for the receiver as long as the receiver follows behind accessing the same addresses in a streamlined manner, as shown in Figure 3.1(d). The encoding is similar to prior works, wherein the sender accesses an address to transmit Bit-0 and does not access it to transmit Bit-1, allowing the receiver to infer a 0 or 1 based on whether it observes an LLC-Hit/LLC-Miss respectively.

Potential Benefits. If the shared array is sufficiently larger than the LLC capacity, by the time the sender wraps around to the beginning of the array, the addresses installed during the previous iteration are automatically evicted due to cache-thrashing. This way eviction can be performed without the requirement of a flush instruction. Such an attack can also be significantly faster than previous attacks [23, 8], as it does not require synchronization every bit-period between the sender and receiver and only requires a single operation (one load) per bit. The bit-rate of this attack is only limited by how fast loads can be executed and measured.

Challenges. However, to orchestrate Streamline with low error-rates, we face two key challenges unique to asynchronous protocols:

Challenge-1. Ensuring addresses used for transmission occupy a significant fraction of the cache: It is critical that the addresses accessed by the sender map to diverse locations in the cache. Otherwise, successive addresses accessed by the sender may evict previous addresses before the receiver can access them, causing errors. The access sequence also

[2]Percival's [37] pioneering work on covert channels also used a sequence of array entries as large as the L1-cache for communication, but it is more than 4x slower than our channel because it still relies on synchronous operation (see Section 3.7.2)

needs to circumvent hardware optimizations, like the prefetcher, designed to predict memory access patterns and preemptively install lines in the cache, and the cache replacement policy, which preemptively evicts lines with low reuse from the cache. We present a general approach to generate an address sequence that occupies a significant fraction of the LLC, fools the prefetcher, and is resilient to the replacement policy, that is applicable to any asynchronous attack.

Challenge-2. Ensuring a bounded gap between the sender and receiver: For Streamline to succeed, it is critical that the sender remains ahead of the receiver, and both execute at similar rates. If the receiver is faster, it can overtake the sender and observe a spurious stream of cache misses. Whereas, if the sender goes too far ahead of the receiver, it can evict its own addresses before the receiver can access them. We develop strategies to balance the sender and receiver rates, including a pseudo-random channel encoding to match the rate of DRAM-accesses executed by the sender and the receiver, and matching the number of `rdtscp` executed. While these optimizations reduce the rate-mismatch, minor differences in execution speed between processes are expected in real systems which can cause the gap between the sender and the receiver to grow unbounded over time. As a fail-safe, we enforce coarse-grain synchronization (once every 200,000 bits) between them using a lower bandwidth covert channel, to limit the gap between the sender and the receiver.

Key Contributions. Overall, this work makes the following contributions:

1. To our knowledge, this is the first work to propose a high-bandwidth cache covert channel without relying on a synchronized protocol to transmit each bit. Our proposal, *Streamline*, uses the cache to buffer data between the sender and the receiver, and relies on thrashing to naturally evict the data from the cache post transmission.

2. We overcome obstacles for high-bandwidth attacks, by fooling hardware optimizations (the prefetcher and replacement policy) and ensuring a bounded gap between sender and receiver (via rate-matching and coarse-grained synchronization).

3. We demonstrate Streamline on Intel Xeon Skylake in a cross-core setting and achieve a bit-rate of 1801 KB/s at a bit-error-rate of 0.37%. Our bit-rate is 3.6x higher than Flush+Flush (496 KB/s), the prior-best cross-core attack, and 3x higher than Take-a-Way attack [46] (588 KB/s), the prior-fastest same-core cache attack.

4. We discover a fundamental limitation of existing load-latency measurement gadgets that prevents latency measurement of multiple loads in parallel and limits the potential bit-rate of Streamline and even other future attacks.

The Streamline attack code is open-sourced at: https://github.com/gururaj-s/streamline.

3.3 Background: Recent Cache Covert-Channel Attacks

3.3.1 Threat Model for Cache Covert Channels

Modern processors typically have multi-level caches, with core-private L1 and L2 caches and a Last-Level-Cache (LLC) shared among multiple cores. Consequently, a covert-channel attack can be *cross-core* [2], where the sender and receiver are in two processes running on different cores, or *same-core*, where they execute on the same physical core from within two SMT threads [46] or within the same process [5, 6]. While our attack is applicable to both attack models, for simplicity, we assume the cross-core attack model, applicable to a virtualized setting where resource-allocation occurs per core, as our default.

3.3.2 State-of-the-Art Cache Covert-Channels

State-of-the-art attacks operate the sender and receiver in a synchronous manner, where they communicate each bit within a synchronized epoch (corresponding to a bit-period) and wait till the epoch ends before communicating the next bit. Within each bit-period, the sender and receiver execute multiple operations to first encode a bit, then decode a bit, and finally reset the channel to be ready to communicate the next bit.

```
Sender                  Receiver
foreach(bit){           foreach(bit){
 if(bit == 0)            t = rdtscp
  load(x)                load(x)
 wait(end-epoch)         T = rdtscp-t
}                        bit = T<thresh?0:1
                         clflush(x)
                         wait(end-epoch)
                        }
```

(a) Flush+Reload Attack

```
Sender                  Receiver
foreach(bit){           foreach(bit){
 if(bit== 0)             t = rdtscp
  load(x)                clflush(x)
 wait(end-epoch)         T = rdtscp-t
}                        bit = T<thresh?0:1
                         wait(end-epoch)
                        }
```

(b) Flush+Flush Attack

```
Sender                  Receiver
foreach(bit){           foreach(bit){
 if(bit== 0)             t = rdtscp
  load(x)                load(x_conflict)
 wait(end-epoch)         T = rdtscp-t
}                        bit = T<thresh?1:0
                         wait(end-epoch)
                        }
```

(c) Take-a-Way Attack

Figure 3.2: State-of-the-art covert-channel attacks. All existing attacks require sender and receiver to communicate each bit in a synchronized epoch and wait till the epoch ends before starting the next bit. Cross-core attacks Flush+Reload and Flush+Flush achieve bit-rates of 298KB/s and 496KB/s, while same-core Take-a-way achieves 588 KB/s.

Cross-Core Flush+Reload Attack [8].

This attack (shown in Figure 3.2(a)) transmits information using the timing difference between an LLC-hit and an LLC-miss for a load to a shared address. The sender encodes each bit by executing or not-executing a load to the address to convey a bit value 0 or 1. The receiver decodes a bit, by executing a load to the same address and measuring its latency using `rdtscp` before and after the load. A bit value 0 or 1 is decoded based on whether it observes a cache-hit or a cache-miss. Subsequently, to reset the channel, the receiver issues a `clflush` to evict the address from the cache. The sender and receiver both wait till the end of a bit-period to ensure the other has finished its operations (synchronizing using `rdtscp` which provides a shared notion of time), before communicating the next bit on the same address. Gruss et al. [23] used this attack to achieve a bit-rate of 298 KB/s at $<0.05\%$ error-rate.

Cross-Core Flush+Flush Attack [23].

This attack is the current fastest cross-core attack and exploits the difference in execution time for a `clflush`, based on whether an address is cached or not. As shown in Figure 3.2(b), the sender's operations are identical to the Flush+Reload attack, i.e. executing or not-executing a load for encoding bit-0 or bit-1. To decode a bit, the receiver executes and measures the latency of a `clflush` to the same address: a faster execution implies

that the address was accessed by the sender and cached, and hence a bit-0 transmission, whereas a slower execution implies bit-1. Note that this attack does not require a separate operation to reset the channel, as the `clflush` in the receiver implicitly evicts the address from the cache, allowing the sender to transmit the next bit once the current epoch ends. As it requires one less operation than Flush+Reload, this attack achieves a higher bit-rate of 496 KB/s; but it has a higher error-rate of 0.84%, as `clflush` has a smaller timing difference ($\sim$10 cycles) compared to that of LLC-hits and misses ($\sim$200 cycles).

Same-Core Take-a-Way Attack [46].

This attack is the fastest same-core attack and exploits timing differences arising from the way-prediction technique used for L1-cache accesses in AMD CPUs. AMD's way-predictors are vulnerable to address conflicts, where accessing two addresses that map to the same way-predictor entry results in evictions of each other from the entry and also the L1-cache. Take-a-way attack exploits this as a covert channel, as shown in Figure 3.2(c). Every bit-period, the receiver issues a load to prime a predictor-entry and then reloads the address. Meanwhile, to transmit bit-0, the sender executes a load to a conflicting address that evicts the receiver address, causing the receiver a cache miss. Otherwise, for bit-1, the sender skips the load, causing the receiver a cache hit. The sender and receiver wait for the bit-period to end before resuming transmission for the next bit. Take-a-way uses this protocol to launch up to 80 synchronous channels and achieves a bit-rate of 588 KB/s.

3.3.3 Pitfalls of Existing Attacks

Synchronous Communication.

State-of-the-art covert-channel attacks require a synchronous transfer of bits. The operations to reset a bit, encode a bit, and decode a bit must be executed in a single synchronous window shared between the sender and the receiver, for each bit before moving on to transmit the next bit. As a result, the size of the synchronous window has to be sufficiently large

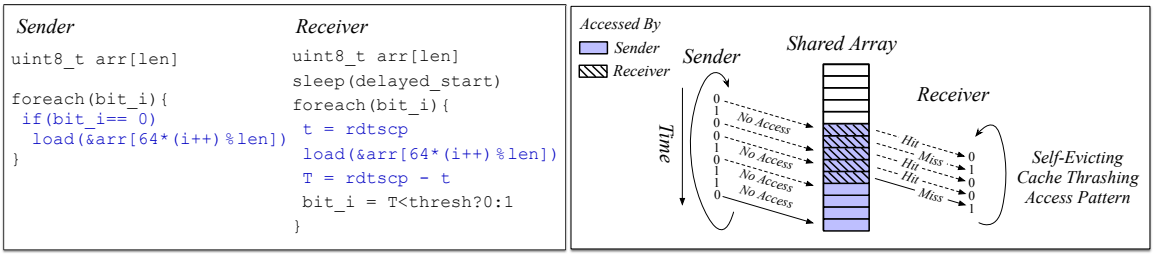
to accommodate all three operations. Any attempt to decrease the bit-period results in loss of synchronization, and breakdown of channel communication.

ISA or Micro-architecture Specific Requirements.

All existing fast covert-channel attacks suffer from limited applicability. For example, Flush-based attacks like Flush+Reload and Flush+Flush, require the usage of a cacheline flush instruction for transmission of each bit. While the x86 ISA supports unprivileged usage of `clflush` instruction, the ARMv8 ISA disables such unprivileged by default. ARMv7 and below do not even support such cacheline flush instructions, making such attacks infeasible on several mobile processors [59]. On the other hand, attacks like Take-a-way exploit features like L1-cache way-prediction only in AMD processors, making such attacks infeasible on other micro-architectures. Although attacks like Prime+Probe [2] that exploit the generic set-associative structure of caches are widely applicable (they do not require flushes or shared memory), they are considerably slower. For example, Liu et al. [2] achieve a bit-rate of 75 KB/s, which is 7x slower than the fastest known flush-based attacks.

3.3.4 Goal: A Fast and Universal Attack

Our goal is to investigate whether bit-rate of cache covert channels can be significantly improved. To be faster than state-of-the-art, an attack should not require the synchronous operation of the sender and receiver while encoding and decoding bits. At the same time, we investigate if such an attack may be universally applicable to processors of all architectures and micro-architectures; the only requirements for such an attack must then be the existence of shared memory and timing difference between fast shared-cache accesses and slow memory accesses. Such an attack operating without cacheline flushes could also highlight the vulnerability of defenses such as SHARP [7] (that rely on disabling the use of flush instructions) and inform the design of future defenses. To that end, we design Streamline as a fast, flush-less, and asynchronous covert channel.



(a) Streamline Attack Algorithm (b) Access Pattern of Sender and Receiver to Shared Array

Figure 3.3: Overview of the Streamline Attack. The sender and receiver communicate asynchronously via accesses to a shared array `arr` (larger than the LLC). The sender keeps transmitting on sequential entries of the array, without waiting for the receiver to decode. By the time the sequential access wraps around to the start of the array, the entries accessed in the previous iteration are evicted from the LLC due to the cache-thrashing access pattern.

3.4 Deep-Dive: Design of the Streamline Attack

We first intuitively describe Streamline, then the challenges in enabling it with low error-rates, and how we overcome them.

3.4.1 High-Level Idea of Streamline

Algorithm: Streamline achieves fast asynchronous communication by transmitting each bit on a different address of a large shared array. As shown in Figure 3.3, for each bit transmission, the sender chooses a successive entry (cache line) of the shared array and installs the entry into the LLC if the bit is 0, else it skips that entry. Then, without waiting for the receiver to access that entry, the sender moves to the next bit. The receiver follows behind in the same sequence loading each entry. If a load is an LLC-Hit, that implies the sender installed that entry into the LLC, and hence the corresponding bit was 0; else if the load is a DRAM-access, the sender skipped that entry, and the bit was 1.

When the sender wraps around to the start of the array, the addresses used in the previous iteration must be evicted from the LLC before they can be reused. Unlike prior attacks that use conflicts [2, 46] or flushes [23, 8] to evict addresses, the access pattern of Streamline on the large array (larger than LLC capacity) implicitly induces cache thrashing and previously accessed addresses get automatically evicted to accommodate new addresses.

Potential Bit-rate: Streamline does not require any extra operations, such as flushes, to evict previously used addresses because of the cache-thrashing pattern of its accesses.[3] Moreover, for each bit, the sender or receiver does not have to wait for the epoch to complete, but can continue to the next bit. As long as the receiver follows behind the sender in a rate-matched manner, the bit-rate is only limited by the receiver's throughput in executing and measuring loads. For example, on an Intel Skylake system where a DRAM access takes approximately 300 cycles, this channel can potentially exceed a bit-rate of 1.5 MB/s.

Challenges: Ensuring that the asynchronous communication is also error-free is challenging for the following reasons:

1. **We need to ensure the sender is consistently ahead of the receiver:** If the sender falls behind the receiver at any time, the receiver continuously observes spurious LLC-misses, and erroneously decodes bits as all-1s. We need to ensure the receiver is slower and always follows the sender.

2. **We need to tolerate slack between the sender and the receiver:** If the portion of the cache used for communication is too small, the sender can self-evict addresses it previously installed in the LLC via cache-thrashing, before the receiver can access them, causing bits to be decoded erroneously. To prevent premature eviction, we need to ensure the addresses installed by the sender spread over the entire LLC, and are protected from interference by replacement policy, and the prefetcher.

3. **We need to prevent the sender from going too far ahead of the receiver:** Since the cache (that acts as a buffer for our communication) is of a limited size, we need to prevent the sender from wrapping around and lapping the receiver. To that end, we need coarse-grain synchronization (*e.g.*, once in hundreds of thousands of bits) to prevent the sender-receiver gap from growing beyond tolerable limits.

[3]Thrash+Reload attack [43] also uses cache-thrashing to evict addresses, but it is a synchronous attack that waits till thrashing evicts an address before transmitting the next bit with the same address. Hence it has a bandwidth of only 4 bits/minute, which is more than a million times slower than Streamline.

Next, we describe how we address each of these challenges.

3.4.2 Channel Encoding For Sender-Rate > Receiver-Rate

Figure 3.4 shows how a naive algorithm for transmission (sender issues load for payload-bit 0, and skips the load for payload-bit 1) can result in burst errors, due to a payload-dependent rate-mismatch between the sender and receiver. If the payload bits are mostly 0s, the sender can slow down due to slow DRAM accesses and fall behind the receiver. This can cause the receiver to erroneously decode all-1s, as it gets LLC-Misses for addresses not accessed yet by the sender. If the payload is mostly 1s, then the sender can skip several addresses and end up considerably ahead of the receiver. In this scenario, the addresses installed by the sender can get evicted even before the receiver can access them, causing a channel breakdown.

Payload	*Sender*	*Receiver*
0	LLC Miss	LLC Hit
1	-	LLC Miss
0	LLC Miss	LLC Hit
0	LLC Miss	LLC Hit
0	LLC Miss	LLC Hit
0	LLC Miss	LLC Hit
0	*Receiver goes ahead of Sender (causing errors)*	LLC Miss
1		LLC Miss
0		LLC Miss

(a) Payload with Many-0s
(Receiver goes beyond Sender)

Payload	*Sender*	*Receiver*
1	-	LLC Miss
1	-	LLC Miss
1	-	LLC Miss
Many 1s	*Skipped*	*Misses prematurely evict sender addresses*
1	-	LLC Miss
0	LLC Miss	LLC Miss
0	LLC Miss	LLC Miss

(a) Payload with Many-1s
(Receiver falls far behind Sender)

Figure 3.4: A naive channel encoding scheme causes a rate-mismatch between the sender and receiver. If the receiver goes ahead of the sender or falls too far behind, it observes erroneous LLC-Misses (in red), leading to errors.

To keep the sender and receiver rates payload-independent, we use a pseudo-random channel encoding. As shown in Figure 3.5, the sender uses a pseudo-random number generator (PRNG) whose seed is known to both sender and receiver, to modulate payload bits with a sequence of random 0s and 1s. For each payload bit (PB-i), the sender transmits a bit (TB-i) as TB-i = PB-$i \oplus$ PRNG-i. On receiving TB-i, the receiver is able to reconstruct the payload (PB-i) as PB-i = TB-$i \oplus$ PRNG-i, as the PRNG seed is known to it.

Payload (PB-i)	*PRNG-i*	*Transmitted (TB-i) (PB-i ^ PRNG-i)*	*Sender*	*Receiver*	
0	1	1	-	LLC Miss	
0	0	0	LLC Miss	LLC Hit	**Receiver slower than Sender**
0	0	0	LLC Miss	LLC Hit	
0	1	1	-	LLC Miss	
1	1	0	LLC Miss	LLC Hit	
1	1	0	LLC Miss	LLC Hit	
1	0	1	-	LLC Miss	
1	0	1	-	LLC Miss	

Figure 3.5: Modulating payload bits (PB-*i*) with a random sequence (PRNG-*i*) and then transmitting (TB-*i*) ensures the Receiver is slower than Sender (with equal LLC-misses, but more LLC-Accesses), regardless of payload values.

The PRNG-based channel encoding equalizes the number of 0s and 1s transmitted (TB-i) in expectation irrespective of the actual payload-bit values (PB-i), as long as the PB-i and PRNG-values are drawn from independent distributions. In this scenario, the sender and the receiver have a comparable number of DRAM accesses, as the sender has a DRAM access when TB-i is 0, while the receiver has a DRAM access when TB-i is 1. However, the receiver additionally incurs LLC-Hits when TB-i is 0, which makes the receiver execute at a slower rate than the sender. This ensures that the receiver always follows behind the sender, and the gap between them grows at a deterministic rate.

3.4.3 Access-Pattern to Tolerate Sender-Receiver Slack

As the sender transmits at a faster rate than the receiver, the gap between the address being accessed by the sender and that being accessed by the receiver, at each moment in time, keeps widening. As the number of addresses that an LLC can store is limited to a finite value, the gap between the sender and the receiver can theoretically increase up to this limit while maintaining a low error rate. To tolerate a sender-receiver gap close to this theoretical limit, we design the sequence of addresses used by Streamline ensuring:

1. The sequence maps to a large majority of LLC sets. Additionally, the sequence should not be predictable by the cache prefetcher, which can disrupt the channel by prefetching addresses into the LLC, irrespective of the payload.

2. The sequence uses all the ways within a particular LLC set, and fools the LLC replacement policy. This is essential to ensure the addresses installed by the sender are not prematurely evicted because of replacement decisions.

Achieving High Set Coverage and Fooling Prefetcher.

Simple sequences such as accessing sequentially contiguous cachelines have high set coverage, but are easily predicted by the prefetcher (Intel CPUs have a next-line prefetcher, sequential stream prefetchers, and stride prefetcher [60, 61]) and the channel can be disrupted. Other sequences used in prior works [23], that access one cacheline per 4KB page to fool the prefetcher, have very poor cache set coverage. To identify an optimal access pattern with high set coverage that also fools the prefetcher, we devise the following experiment. We systematically generate sequences of N addresses that access every x-th cacheline in a page and lines from y pages are accessed before the next line from the same page, and measure the latency of each access in the sequence. We repeat this experiment 5 times for $N = 1000$ and $x, y = \{1, 2, 3, 4, 5\}$, and report the miss-rate, i.e. the number of cache-misses observed out of N accesses for each sequence. A higher cache miss rate for a sequence indicates that it is more effective in fooling the prefetcher.

As shown in Table 3.1, the prefetcher learns access patterns (and lowers miss-rates) if accesses are strided within a single page ($y = 1$) for any stride ($x \geq 1$), or if the accesses are sequential ($x = 1$) irrespective of the number of pages ($y \geq 1$) across which lines are accessed before the same page is re-accessed. However, a strided access pattern ($x > 2$) that is spread across more than one page ($y \geq 2$) is highly effective in fooling the prefetchers, with the miss-rate for such a sequence being $> 90\%$. We believe this is because the stride-tracking mechanism operates at page granularity (as prefetched addresses do not cross page boundaries) and is overwhelmed by the back-to-back accesses across pages.

For Streamline, we pick the access pattern that best fools the prefetchers, *i.e.* a stride of 3 spread over 2 pages ($x = 3, y = 2$). This pattern covers 1/3rd of the LLC sets

Table 3.1: LLC Miss-Rate for a Sequence accessing every xth cacheline in a page, with y pages accessed at a time. (Higher miss-rate implies sequence fools prefetcher better)

x \ y	1	2	3	4	5
1	1.8%	3.7%	2.7 %	2.5%	2.2%
2	6.6%	7.3%	6.7%	6.9%	7.0%
3	11.6%	99.5%	98.9%	90.9%	88.0%
4	15.3%	97.5%	97.8%	95.7%	90.5%
5	17.3%	98.8%	91.8%	91.6%	90.6%

as it accesses every 3rd line within a page, which is significantly better than accessing one cacheline per 4KB page (as in prior work [23]). We also empirically observe that sequences that start from the middle of a 4KB page are better at fooling the prefetcher stride-tracking (e.g. 14th cacheline). The exact equation for calculating the index of the shared byte-array for each bit (i) of the payload is given by Equation (3.1)–Equation (3.3).

$$Pg\text{-}num = 2 * int(3 * i/128) \quad + \quad i\%2 \tag{3.1}$$

$$Cl\text{-}num = (14 + 3 * int(i/2))\ \%\ 64 \tag{3.2}$$

$$array\text{-}index = (Pg\text{-}num * 4096 + Cl\text{-}num * 64)\ \%\ arr\text{-}sz \tag{3.3}$$

Covering LLC-Ways by Fooling Replacement Policy.

To ensure the addresses accessed in Streamline occupy a majority of the LLC-ways, and are protected from premature eviction from a set, we need to fool the LLC replacement policy. Prior work [62] reverse engineered the LLC replacement policy in Intel CPUs, showing it to maintain 2-bit age values per cacheline for tracking re-use (similar to re-use bits in RRIP Replacement [63]). A new line is assigned an age value of 2 or 3 (based on CPU generation), and subsequent LLC hits to such lines decrements their ages, till they saturate

at 0. A line with age-3 within a set is evicted, when a new line is to be installed to the set; if no such line exists, all the ages in the set are incremented till an age-3 line is found.

To fool the replacement policy and avoid premature eviction of addresses installed by Streamline, we need to prevent them from being the oldest in the set. To that end, we issue extra LLC hits to the addresses installed in the LLC by the sender, to decrement their age and prevent their preemptive eviction. We achieve this by making the sender re-access previously installed addresses, after a lag of 5000 bits, to ensure this trailing access causes an LLC hit (the address is likely to be evicted from the L1 and L2 cache within 5000 bits).

Figure 3.6 shows the bit-error-rate for Streamline versus the sender-receiver gap (in number of bits), for different access patterns: a naive sequence accessing one cacheline per page, a sequence with high LLC set coverage from Section 3.4.3, and a sequence with high coverage of LLC sets and ways from Section 3.4.3. The naive sequence shows an increase in error-rate beyond a sender-receiver gap of 1000 bits, the sequence with high LLC set coverage beyond 4000 bits, and the sequence with high coverage of LLC sets and ways retains low error-rates till a gap of 40,000 bits between sender and receiver. With this, Streamline tolerates a sender-receiver gap of 1/3rd the LLC capacity (128,000 addresses).

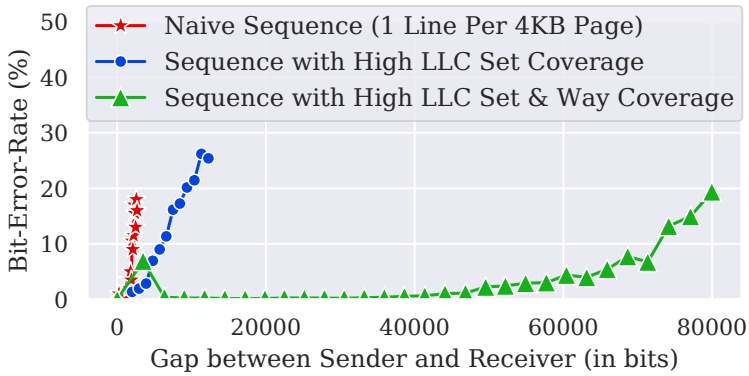


Figure 3.6: Error-rate versus Sender-Receiver Gap. With a sequence of addresses that covers a majority of LLC sets and ways, Streamline builds considerable tolerance to slack between the sender and receiver.

3.4.4 Techniques to Bound Sender-Receiver Slack

In an ideal scenario where the sender and receiver are perfectly rate-matched, the gap between the sender and receiver would remain constant, and never go beyond the tolerable slack. To that end, we attempt to rate-match the sender and receiver as best as possible and then explicitly bound the maximum slack between the sender and receiver, to ensure the sender never wraps around the array and laps the receiver.

Matching Sender and Receiver Load Execution Rates.

One of the key reasons for the rate mismatch between the sender and the receiver is that the receiver measures the latency of the loads it executes while the sender does not. To measure the load latency, the receiver uses `rdtscp` instructions in the sequence `rdtscp; load; rdtscp;`. Such usage of `rdtscp` serializes the execution of loads of different bits for the receiver, whereas the sender (without such serializing instructions) can issue loads of multiple bits in parallel. As a result, despite issuing two loads per bit (one for transmission and one for fooling the replacement policy), the sender can execute at a faster rate than the receiver. Hence, to throttle the sender, we add a load-serializing `rdtscp` per bit in the sender that limits its load execution rate and reduces its mismatch with the receiver.

Periodic Coarse-Grain Synchronization

In a realistic setting, the sender and the receiver will always have a non-zero drift. To prevent the sender-receiver gap from exceeding a tolerable limit due to this, we synchronize the sender and receiver at a coarse granularity (e.g. every 200,000 bits) using a separate low-bandwidth covert channel. When the sender reaches the end of an epoch of 200,000 bits, it stops transmitting and waits. Once the receiver completes 195,000 bits of the epoch, it communicates a bit on the synchronization channel to the sender, to permit the sender to resume. As synchronization is extremely infrequent (e.g. once in 200,000 bits), any low-

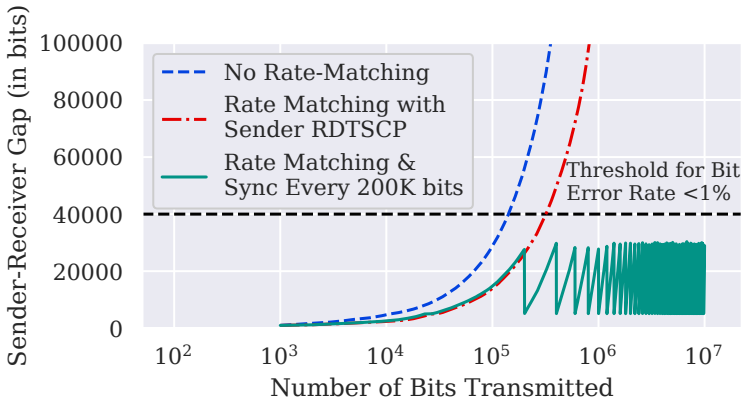


Figure 3.7: Gap between Sender and Receiver vs Number of bits transmitted. Rate-limiting the sender to match its rate with the receiver and using coarse-grain synchronization to halt the sender every 200,000 bits, ensures the gap is maintained below 40,000 bits (within this threshold, the error-rate stays below 1%).

bandwidth covert channel can be used without any bandwidth loss (we use a Flush+Reload channel for synchronization).

To justify our choice of synchronizing every 200,000 bits, Figure 3.7 shows the sender-receiver gap (in bits) as the bits transmitted increase. We compare three access patterns: (a) using the tailored access pattern from the previous section that covers a majority of the cache, (b) the tailored access pattern with the addition of a rate-limiting `rdtscp` for the sender, and (c) the further addition of halting the sender periodically based on synchronization every 200,000 bits. With the first access pattern, the sender-receiver gap crosses a threshold of 40,000 bits (beyond which error-rate goes above 1%) within a transmission of 100,000 bits. With the second access pattern that rate-limits the sender, the sender-receiver gap remains within the threshold till 400,000 bits. To ensure that the sender-receiver gap is below the threshold (*i.e.,* the channel operates within the threshold of 1% error-rate) indefinitely while also keeping some head-room, we add synchronization between the sender and receiver every 200,000 bits transmitted (*i.e.* the third access pattern). With this, Streamline can maintain the channel error rate below 1% for billions of bits transmitted.

3.4.5 Overall Algorithm for Streamline

Figure 3.8 shows the algorithm for Streamline incorporating the error-mitigating techniques from Section 3.4.3 and Section 3.4.4. The modulation of the payload with the PRNG-sequence (Section 3.4.2) happens off the critical path and hence is not shown.

Sender

```
foreach(bit_tx[i]){

 //to rate-limit
 rdtscp

 //to transmit bit
 if(bit_tx[i] == 0)
  load(&arr[index(i)])

 //to beat real-policy
 if(bit_tx[i-5000] == 0)
  load(&arr[index(i-5000)])

 //synchronize every 200K
 if(i%200000 == 199999)
   FR_Sync_With_Receiver()
}
```

Receiver

```
sleep(delayed_start)

foreach(bit_rx[i]){

 //to receive bit
 t = rdtscp
 load(&arr[index(i)])
 T = rdtscp - t
 bit_rx[i]=T<thresh?0:1

 //synchronize every 200K
 if(i%200000 == 195000)
  FR_Sync_With_Sender()
}
```

Figure 3.8: Algorithm for Sender and Receiver in Streamline to achieve fast and asynchronous communication, incorporating techniques to ensure low error-rates.

3.5 Results

In this section, we evaluate the covert-channel transmission bit-rate and bit-error-rate that Streamline achieves.

3.5.1 Methodology

We run our experiments on a 4-core Intel Skylake CPU (Intel Xeon E3-1270), with an 8MB LLC, running at a frequency of 3.9 GHz (the results were also successfully reproduced on Intel Core i7-8700K Kaby Lake and Core i5-9400 Coffee Lake CPUs). For our system, we measure the average LLC-Hit latency to be 95 cycles, and LLC-Miss latency to be 285 cycles. So, we use a threshold of 180 cycles for the receiver to determine if a load is an LLC-Hit or Miss. We use large pages for the sender and receiver, to minimize any effects due to TLB misses. We pin the sender and receiver processes to two different

cores, to ensure all communication is through the LLC. We assume the sender and receiver share an array (default size of 64 MB) with read-only permissions that Streamline uses for communication; we analyze other array sizes in Section 3.5.4.

3.5.2 Streamline Channel Bit Rate and Error Rate

Figure 3.9 shows the bit-rate in KB/s and bit-error-rate for Streamline, plotted against the size of the payload that is transmitted (in bits). As a high-speed covert channel typically has more utility at large payload sizes, we evaluate Streamline for payload sizes of 200,000 bits to 1 billion bits. We report the bit-rate by measuring time from start to end for the receiver, divided by the number of bits transmitted, averaged over 5 runs. Streamline achieves a steady-state bit-rate of 1801 KB/s at a bit-error-rate of 0.37%. This corresponds to a bit-period of 265 CPU cycles, which is in between the latency for an LLC Hit and DRAM access on our system.

Streamline's bandwidth is limited by how fast the receiver executes and measures a load for each bit, and is not limited by synchronization as in prior works. Although DDR4-DRAM has a bandwidth of 150 Million accesses/second, Streamline is still limited to a bandwidth of <2 MB/s, due to serialization of loads by the `rdtscp` used to measure load-latency at the receiver. As the receiver cannot execute multiple loads in parallel, it is forced to wait till each load completes (incurring raw LLC-Hit/ DRAM-access latency per bit-period), before issuing the next load. We discuss how this bandwidth limitation is fundamental to all future attacks in Section 3.6.2.

In Figure 3.9, the bit-error-rate for a payload size of 200,000 bits is ∼2% and higher than the stable bit-error-rate of 0.37% for larger payload sizes. This is because of high error-rates incurred during the first 5000 bits, when the trailing accesses to fool the replacement policy have not started (during the first 5000 bits, the error-rate goes up to 20% as shown in Figure 3.6). However, these errors become imperceptible as payload size increases, and the error-rate stabilizes at 0.3%.

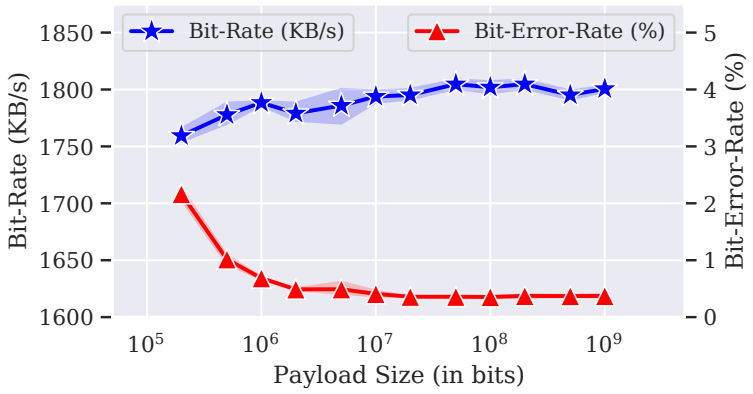


Figure 3.9: Covert-channel Bit-rate and Bit-error-rate vs Payload Size (shaded regions represent 95% CI, *i.e.* confidence intervals). Streamline has a bit-rate of 1801 KB/s at an error-rate of 0.37% (note the non-zero start of the Y-Axis for bit-rate).

3.5.3 Analysis of Errors and Error-Correction

Table 3.2 shows the breakdown of error-rates by type – 0 to 1 bit errors and 1 to 0 bit errors, for different payload-sizes. As payload size increases, the 1 to 0 errors (which form a significant fraction of the total errors for smaller payloads) drop considerably. At the same time, 0 to 1 errors stay the same, and become a dominant fraction for larger payloads.

0 to 1 bit errors typically manifest when an address accessed by the sender gets evicted from the LLC before the receiver can access it, because the gap between receiver and sender grew too large or because of cache usage by other system processes. In either scenario, we observe these errors appear in bursts that are hard to correct without re-transmission. On the other hand, we observe that 1 to 0 errors occur when a DRAM access is faster than our LLC-hit threshold, resulting in a false declaration of an LLC-Hit. We expect these errors to be randomly distributed as these accesses form the tail of the DRAM latency distribution, with a high chance of them being single-bit errors. Hence, we develop an error-correction scheme for Streamline that corrects single-bit errors.

To add error-correction, we break our payload into 8-byte packets and append each packet with a (72,64) Hamming Code before transmission, that can correct 1-bit errors and detect 2-bit errors occurring during transmission. We pick this specific design with

Table 3.2: Breakup of Error-Rates for Different Payload Sizes

Payload Size (in bits)	2×10^5	10^6	10^7	10^8	10^9
Total Bit-Error-Rate	2.16%	0.68%	0.41%	0.35%	0.37%
Rate of 1 to 0 Errors	1.95%	0.44%	0.12%	0.09%	0.11%
Rate of 0 to 1 Errors	0.22%	0.25%	0.29%	0.26%	0.27%

a 12.5% overhead (1-byte code per 8-byte data) to maintain equivalence with the error-mitigation framework developed by Gruss et al. [23] (a more robust mechanism including re-transmission) that incurs a similar 12% overhead for the Flush+Flush attack.

Table 3.3 shows the impact of our error-correction scheme on the bit-rate and bit-error-rate for transmitting 1 billion bits. With error-correction, the effective data bit-rate goes down by almost 10% to 1598 KB/s, while the bit-error-rate goes down to 0.12%. The remaining bit-errors are due to uncorrected or miscorrected bits because of multi-bit errors within a packet. We observe a similar 0.2% drop in error-rate for all payload sizes.

Table 3.3: Streamline with and without Error-Correction (parenthesis includes margin-of-error for 95% confidence interval)

Configuration	**Bit-rate**	**Bit-Error-Rate**
Without Error-Correction	1801 KB/s ($\pm$ 3)	0.37% ($\pm$ 0.04%)
With (72,64) Hamming Code	1598 KB/s ($\pm$ 2)	0.12% ($\pm$ 0.01%)

3.5.4 Sensitivity to Shared Array Size

Table 3.4 shows the error-rate of Streamline with a payload of 100 million bits, as the size of the shared array used for covert communication varies. As our system has an 8MB LLC, we evaluate shared array sizes of 8MB to 64MB (1x to 8x the LLC size). As the size decreases from the default value of 64MB to 32MB, the error-rate stays close to 0.3%. However, it increases considerably below 32MB, reaching 3.2% for 16MB and 28% for 8MB. Having a sufficiently large shared array is critical for the robustness of the channel, as Streamline relies on the cache-thrashing behavior of its accesses to evict addresses previously used for communication. Without evictions via cache-thrashing, Streamline cannot effectively

reuse array addresses on a wrap-around, resulting in high error rates. Streamline needs an array that is at least 3x the size of the LLC for inducing effective cache-thrashing, as its access pattern only loads every 3rd cache line of the array to fool the prefetcher. Hence, Streamline incurs higher error rates for array-sizes 2x (16MB) and 1x (8MB) the LLC-size.

Table 3.4: Streamline with Different Shared Array Sizes (parenthesis includes margin-of-error for 95% confidence interval)

Shared Array Size	**Bit-Error-Rate**
64MB (default)	0.35% (± 0.02%)
32MB	0.33% ± 0.01%
16MB	3.2% (± 0.1%)
8MB	27.5% (± 0.1%)

3.5.5 Sensitivity to Synchronization Period

Table 3.5 shows channel characteristics for a payload of 100 million bits, as the synchronization period is varied. Across all periods, the bit-rate remains above 1780 KB/s because the synchronization overhead is negligible. However, the bit-error-rate increases to 0.7% if the synchronization frequency decreases once every 500,000 bits, as the sender-receiver gap exceeds tolerable limits (going beyond 500,000 bits leads to channel breakdown). On the other hand, increasing synchronization frequency to once every 25,000 results in minor differences in error-rates (increase by 0.1% versus our default of once every 200,000 bits), but we observe these tend to be single-bit errors that are easily correctable.

Table 3.5: Streamline with Different Synchronization Periods (parenthesis includes margin-of-error for 95% confidence interval)

Synchronization Period	**Bit-rate**	**Bit-Error-Rate**
Every 500,000 bits	1818 KB/s (± 5)	0.65% (± 0.05%)
Every 200,000 bits (default)	1802 KB/s (± 7)	0.35% (± 0.02%)
Every 100,000 bits	1797 KB/s (± 6)	0.37% (± 0.03%)
Every 50,000 bits	1783 KB/s (± 10)	0.40% (± 0.02%)
Every 25,000 bits	1791 KB/s (± 5)	0.46% (± 0.01%)

3.6 Discussion

3.6.1 Resilience to System Noise

In Streamline, the sender buffers bits for the receiver in LLC locations to enable asynchronous communication. However, considerable cache-activity from co-running processes on the system can cause lines installed by the sender to get evicted before the receiver accesses them, adding noise to the channel. However, Streamline can achieve noise resilience by limiting the time window where cache lines installed by the sender are vulnerable to eviction (*i.e.* the time window for which the line is installed by the sender but not yet accessed by the receiver). This can be achieved by reducing the maximum sender-receiver gap, by using a smaller synchronization period for the sender and receiver.

Our default implementation uses a synchronization period of 200,000 bits that limits the sender-receiver gap to a maximum of 40,000 bits. Reducing the synchronization period to once every 50,000 bits limits the maximum sender-receiver gap to 8,000 bits (as shown in Figure 3.7) without affecting the bit-rate (as shown in Table 3.5). In this case, as the buffer size used at any given time is no more than 8000 bits, i.e. 6% of our 8MB LLC (131,000 lines), the potential for interference from co-running processes is significantly diminished.

To evaluate the attack fidelity under noise, we evaluate Streamline while running applications stressing the CPU caches simultaneously on a different core. We use `stress-ng` (configurable kernels that stress system resources) [64] and run applications that stress the CPU caches from `"--class cpu-cache"`.

Figure 3.10 shows the error-rate of Streamline transmitting a payload of 100 million bits (averaged over 5 runs) using synchronization periods of 200,000 and 50,000 bits, while each application from `stress-ng` is co-running (pinned to an adjacent core). While the error-rate of the channel reaches a worst case of 15% with the synchronization period of 200,000 bits, it is limited to a worst-case of 0.8% when the sync-period is reduced to 50,000 bits and relatively noise-resilient (close to the error-rate of 0.3% under noise-free

setting). On the other hand, Streamline's bit-rate with co-running `stress-ng` applications is slightly diminished and varies from 1500-1800 KB/s due to increased memory latency and queuing delays.

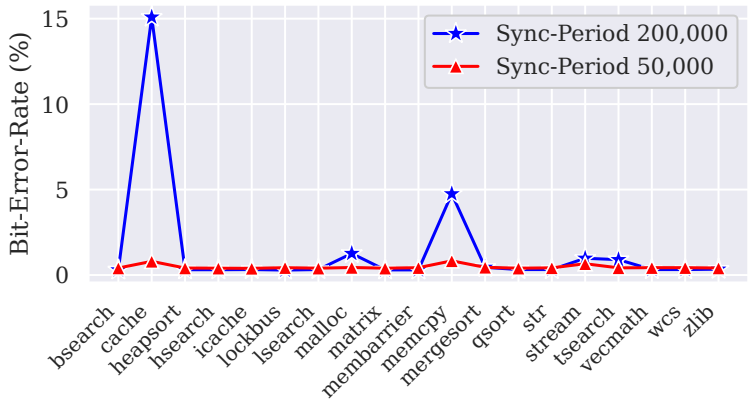


Figure 3.10: Error-Rate of Streamline under co-running `stress-ng` workloads, for sender-receiver synchronization periods of 200,000 and 50,000 bits.

We also tested Streamline while simultaneously running the Chromium-60 web browser streaming Youtube videos and found error-rate to be 0.5-0.6% (no impact on bit-rate). Note that infrequent spurious noise evicting line at such low rates can be mitigated using error correction codes (discussed in Section 3.5.3).

3.6.2 Limiter for Covert-Channel Bit-rate

Streamline mitigates two key bottlenecks faced by prior covert channels: (a) the transmission bottleneck (the requirement of loading and resetting an address with separate operations every bit), and (b) the synchronization bottleneck (sender waits until the receiver has decoded a bit before transmitting the next bit). Thus, we improve the covert-channel bit-rate by >3x compared to state-of-the-art. Our work, however, exposes a new bottleneck for covert channels – *the measurement bottleneck*.

To see the problem, consider that the bit-rate in Streamline is determined by the rate at which the receiver can execute and measure loads. All existing methods to measure load latency on x86 systems result in load serialization for accurate latency measurement. For example, we use the sequence `rdtscp; load; rdtscp;` (other works use

`rdtsc;lfence;` instead of `rdtscp`). Fundamentally, these sequences need to ensure that the second timer instruction samples the time after the load returns (which they do with either an explicit `lfence` or fence-like semantics as in `rdtscp`). This fencing implies that the next load cannot execute until the previous one returns. This loss of parallelism means that each bit-period is limited by the load latency to access DRAM or LLC (between 100 – 300 cycles). Thus, Streamline is limited to a bit-period of 267 cycles, as is any future attack requiring timing loads.

Note that using gadgets like sibling counting-threads [65, 66], for measuring time without `rdtscp`, is not viable for measuring the latency of multiple loads executing in parallel in x86. Executing and measuring loads in parallel with such timer-variables leads to re-ordering of timer-loads and potential TSO violations on Intel CPUs [67] – these cause speculative timer-loads to be squashed, resulting in incorrect latency measurements. We leave the study of new methods to measure the load latency of multiple loads in parallel, to reach higher bit-rates, for future work.

3.6.3 Real-World Applicability

Potential Applications. Streamline is applicable to the classic covert-channel setting [50, 68, 69] between trojan and spy processes (both controlled by an adversary), where a trojan malware has infiltrated a sand-box and gained access to secrets, and needs to communicate with a spy program with access to network ports (capable of exfiltrating data or communicating with a command-and-control server). Streamline can enable transmission of high bandwidth payloads (such as video streams) in such a setting that is not possible with existing covert channels. Moreover Streamline is also applicable as a covert-channel for transient-execution attacks [6, 5], particularly in controlled environments like Javascript, where instructions for flushing cache lines may not be accessible.

Shared-Memory Requirement. Streamline requires a shared array that is 2-4x the size of the LLC. This can be easily achieved by the colluding processes with an `mmap` of

shared libraries, either using a single large shared library (*e.g.*, libQtWebKit.so is ∼32MB) or by chaining a sequence of smaller libraries (*e.g.*, libc.so, libcrypto.so are ∼2MB each). Memory may also be shared between processes via OS-based deduplication (*e.g.*, Linux KSM [41]), as in prior attacks [69]. Future work could also explore asynchronous communication without shared-memory (*e.g.*, using conflicts in shared sets for bit transmission) to avoid this requirement.

Applicability to Hyper-Threads. While we evaluate Streamline between processes running on different cores, it is also applicable to hyper-threads simultaneously running on the same core sharing the L1/L2-cache. An advantage of such a setting is that a smaller shared array is required for thrashing L1/L2 caches, but the smaller difference in hit-vs-miss latencies for these caches is a challenge. For these reasons, in a cross-thread setting, the L2 cache is a more suitable target for Streamline than the L1 cache.

3.7 Comparison with Prior Work

In this section, we compare Streamline with an implementation of the Flush+Reload [8] covert channel on our system and also with other prior covert channels.

3.7.1 Comparing Flush+Reload and Streamline

Figure 3.11 shows the bit error rate versus bit rate for the Flush + Reload [8] covert channel (averaged over 5 runs), generated on our system using the code from the Arch-Sec tutorial at ISCA-2019 [70] (note that this is not a fully optimized implementation, so the trend is more representative than the actual values). To obtain the error-rates at different bit-rates for Flush+Reload, we reduce the transmission window per bit (time for which the sender and receiver perform accesses to communicate a bit) from 32,768 cycles to 256 cycles, while artificially ensuring that the synchronization-related errors remain negligible so that the errors are only due to transmission (note that the error-rate obtained is a lower-bound). We observe the error-rate stays low (below 1%) until 200 KB/s (bit period of 2000 cycles),

but beyond 200 KB/s the error-rate considerably increases beyond 10%. This is because Flush+Reload requires multiple operations (loads for transmission, flush for reset) to be executed within progressively shorter bit periods. In comparison, Streamline achieves an error-rate of 0.3% with a bit period of 265 cycles as it only requires a single operation per bit for transmission and also does not require synchronization for every bit.

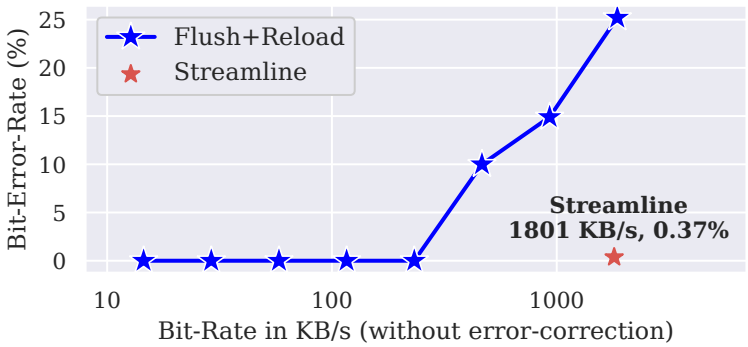


Figure 3.11: Bit-rate and bit-error-rate (without error-correction) of Flush+Reload attack versus Streamline.

3.7.2 Comparison with other Covert-Channels

Table 3.6 shows bit-rates and error-rates of cache covert-channel attacks in recent works, compared with Streamline. As state-of-the-art bit-rates for these attacks are difficult to achieve without hand-tuned assembly implementations (not publicly available to our knowledge), we present results reported in the respective papers.

Table 3.6: Comparison with Prior Cache Covert Channels (Bit-Rate>50 KB/s)

Attack	**Attack Model**	**Bit-Rate**	**Bit Error Rate**
Take-a-way [46]	Same-Core	588 KB/s	1–3%
Flush+Flush [23]	Cross-Core	496 KB/s	0.84%
Prime+Probe (L1) [37]	Same-Core	400 KB/s	–
Flush+Reload [23]	Cross-Core	298 KB/s	0%
Prime+Probe (LLC) [2]	Cross-Core	75 KB/s	1%
Xiaong and Szefer [45]	Same-Core	72 KB/s	<2%
Streamline (this work)	**Cross-Core**	**1801 KB/s**	**0.37%**

Flush+Flush and **Flush+Reload** implementations by Gruss et al. [23] were the prior fastest cross-core covert channels, with bit-rates of 298 KB/s and 496 KB/s. Streamline achieves 3.6x and 6x higher bit-rates at comparable error-rate as it (a) is asynchronous (the sender does not wait for the receiver each bit), and (b) only requires a load per bit (no flushes). Note that the hardware used in Streamline (Intel Xeon Skylake) has approximately 15% speedup compared to the hardware in prior attacks (Intel i7 Haswell) [71] and Streamline's bit-rate increase of 3.6x far outweighs any potential hardware speedup. In fact, Streamline is likely to have similar bit-rates across recent processor generations as its bit-period is bottlenecked by LLC/DRAM latencies which have largely stayed similar. Moreover, Streamline is more universally applicable as it does not rely on cacheline flush instructions (required by flush-based attacks), that are unavailable for unprivileged use on ARM CPUs by default.

Prime+Probe attacks exploiting set conflicts were pioneered by Percival [37] on L1-Caches. This work used an array as large as the L1-Cache for communication, transmitting a bit per array-entry (similar to our proposal). However, it used a synchronous protocol, where the sender accesses L1-cache sets to transmit while the receiver waits, and only once it completes the receiver accesses all L1-cache lines and checks for conflicts while sender waits, that limits its bit-rate to 400KB/s [37]. Subsequent works [2, 51] demonstrated a synchronous cross-core Prime+Probe attack on slower LLCs, achieving a bit-rate up to 75KB/s [2]. Streamline is much faster than these attacks, primarily due its asynchronous operation, where the sender and receiver do not have to wait on the other frequently (although Streamline requires shared-memory; Prime+Probe does not). The strategies in Streamline may also be used to enable a faster asynchronous Prime+Probe attack in future works, where the bit-rate is not limited by synchronization.

Other attacks exploit the replacement policy metadata in L1 caches [45] or in LLCs [62], or even coherence protocols [72, 69] to transmit information more stealthily. However, they suffer a synchronization bottleneck like Prime+Probe or Flush+Reload that limits their bit-

rate. Streamline uses insights from prior replacement policy attacks to enable a considerably faster asynchronous attack.

Thrash+Reload [43], a variant of Flush+Reload attack, uses cache-thrashing instead of flush to evict an address each bit. Unfortunately, it is quite slow due to its synchronous nature: the sender waits for the receiver to thrash the cache by accessing more addresses than LLC capacity before it transmits the next bit (bit-rate of 4 bits/min over the network [43] and up to 1000 bits/s natively). In Streamline, the transmission itself induces cache-thrashing, and eviction of previous addresses is automatic (without needing the sender or receiver to wait), helping it achieve 14000x higher bit-rate.

Take-a-way [46] on AMD machines has the highest known bit-rate (588 KB/s) for a same-core covert channel attack. It achieves this by communicating over 80 concurrent synchronous channels, leveraging different L1 cache sets. On the other hand, Streamline transmits over a large number of addresses in a single asynchronous channel and achieves a 3x higher bit-rate. Streamline is also more general as it only relies on generic LLC properties like sharing among cores and thrashing, unlike Take-a-way that exploits way-prediction features only known to be exploitable in AMD CPUs.

3.8 Mitigation Strategy

Defenses that restrict the unprivileged usage of cacheline flush instructions to mitigate flush-based attacks, such as SHARP [7] or ARM ISA, are incapable of mitigating Streamline. Three main mitigation strategies that might be used to restrict Streamline: detection, noise injection, or isolation. We describe these below.

Detection based approaches attempt to identify attacks either by profiling them using performance-counters available in commodity hardware [73, 74, 75] or by using specialized hardware [68, 76] to detect contention-patterns prevalent in such attacks. Performance-counter based detection is unlikely to specifically detect Streamline as its cache-access rates

and cache-miss rates are quite similar to generic memory-intensive applications (e.g. those processing streams of data). Detectors using specialized hardware, that have the capability to infer detailed cache re-use and contention patterns, have a higher chance of detecting Streamline. For example, record-replay techniques [76] can profile the distribution of cache hits and misses and potentially infer the cache-access pattern used in Streamline. However, detection-tools can also be easily fooled by an adaptive attack that shapes its distribution of cache-hits/misses (using extra LLC-accesses) to match a benign workload and avoid detection. Thus, a detection-based approach is not a fool-proof mitigation strategy.

Noise-injection based strategies can attempt to reduce the fidelity of any cache covert channel by dislodging the cachelines used by the sender and receiver for communication via cache accesses from a co-running application. While most prior attacks only use a single or a small group of addresses for communication and are highly vulnerable to disruption if noise injection is targeted at these addresses, Streamline utilizes a sequence of addresses (that can be made unpredictable) that makes targeted noise injection more difficult. Moreover, for the noise injection to be successful, addresses installed by a sender need to be dislodged before the receiver accesses them. Reducing the number of bits buffered in the LLC at a time and the time window for which each address is buffered, by reducing the synchronization period (as discussed in Section 3.6.1), limits sensitivity to noise.

Hardware designs like randomized prefetching [77] or randomized cache fills [78] may naturally hinder Streamline operation by introducing noise, although such designs also cause slowdown for benign applications. On the other hand, random replacement can add noise to the channel, but is unlikely to fully prevent Streamline. If shorter synchronization periods are in use where less than 10% of the LLC space is actively used as a buffer at any given time (Section 3.6.1), even random replacements due to co-running process activity are unlikely to dislodge a significant portion of the lines between the sender and receiver accesses to disrupt the channel. Streamline can tolerate infrequent interference due to random replacement using ECC (as discussed in Section 3.5.3).

Isolation based approaches prevent processes in different trust-domains from sharing cache locations and are highly effective at mitigating shared-memory based covert-channel attacks. Such cache isolation can be achieved by disabling shared-memory or deduplication (e.g. disabling Linux KSM [41]), or by using cache-partitioning techniques that eliminate cross-domain hits required for transmission in Streamline: by allocating disjoint groups of LLC sets to processes in different trust-domains [22, 79], or by duplicating shared cachelines across trust-domains [16, 11]. Such techniques eliminate Streamline and all cache attacks exploiting hits on shared cachelines. However, all such solutions either have performance costs or face scalability challenges (cache-partitioning requires allocation at the limited granularity of ways or sets) or require support from system-software to classify processes into trust-domains (needed for cache-partitioning) that limit their applicability.

3.9 Impact of this Research

3.9.1 Demonstrating Vulnerability of Existing Defenses

Streamline made fast cache covert channel attacks universally applicable and highlighted that ISAs like ARM and environments like Javascript are vulnerable to such attacks. Prior to this work, ISAs like ARM and environments like Javascript were presumed to be immune to fast cache covert-channel attacks Flush+Reload [8] as they do not provide user-space access to an instruction to flush cachelines.

Moreover, prior defenses like SHARP [7] advocated for disabling unprivileged usage of flush instructions in user-space for future ISAs, as a solution for defending against attacks like Flush+Reload. Streamline highlights that such a strategy of disabling flush instruction is broken, as it demonstrates a fast covert-channel attack without cache line flush instructions (using thrashing behavior).

3.9.2 New Cache Attacks and Defenses

Within 4 months of its publication, Streamline was adopted in offensive research in a transient-execution attack exploit in Javascript, Rage-Against-Machine-Clear [80]. The inaccessibility of cacheline flush functionality in Javascript, makes Streamline an extremely practical covert channel to use for exploitation in this environment.

At the same time, Streamline with its universal applicability highlights the fact that cache timing channels are a universal problem affecting all micro-architectures, ISAs, and runtime environments. This encourages the development of principled cache defenses like cache-partitioning and duplication of shared cache lines across security domains as defense strategies. Such strategies are attractive for deployment across CPU manufacturers, to prevent all shared memory based attacks. We explore these strategies in subsequent chapters.

CHAPTER 4

MIRAGE - A FULLY-ASSOCIATIVE RANDOMIZED CACHE DEFENSE

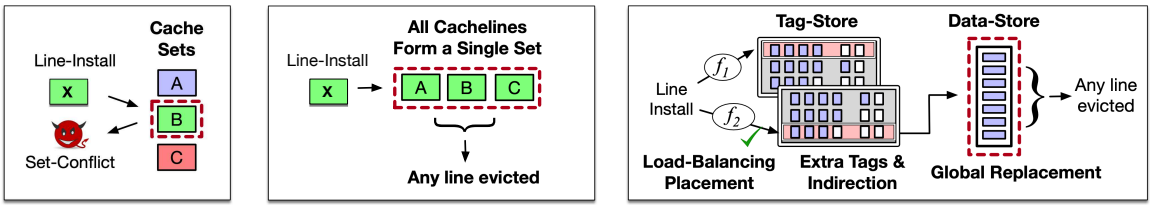
In this chapter, we study defenses against cache attacks. The focus of this chapter is to develop principled defenses building on cache randomization, a promising technique for security against set-conflict based cache attacks.

4.1 Context: Why Study Randomized Cache Defenses?

Set-conflict based cache attacks (e.g. *Prime+Probe* [38]) are potent side channel attacks that exploit the set-associative design in caches. In such designs, addresses map to a small group of cache locations called a *cache set*, for efficient lookup. If the addresses of both the victim and the attacker map to the same set, then they can evict each other from the cache (such an episode is called a *set-conflict*). The attacker uses such set-conflict based evictions to monitor secret-dependent access patterns of the victim, and leak victim secrets.

Recent proposals for *Randomized LLCs* [9, 11, 10, 25] attempt to mitigate set-conflict based attacks in LLCs by randomizing locations of cachelines. With a randomized address-to-cache-set mapping that is unpredictable to an adversary, these designs obfuscate locations of set-conflicts to prevent eviction-set discovery and conflict-based attacks. A recent design [11] also uses the process-ID or a trust-domain-ID as an input to the randomized set mappings to duplicate cache lines for shared addresses across processes and mitigate shared-memory attacks. Such designs are practical to implement requiring simple changes to the LLC set-index derivation functions (IDF) and having a modest performance impact.

Arms Race with Recent Defenses and Attacks. Unfortunately, in recent years, successive randomized cache defenses [9, 11, 10, 25] have been broken by advances in attacks. This is because, although such defenses obfuscate the cachelines selected for eviction, the evicted lines continue to be deterministic and from a small number of locations in the cache (equal to the cache associativity), as shown in Figure 4.1(a). Thus set-conflicts continue



(a) Set-Associative Eviction (b) Goal: Fully-Associative Global Eviction (c) Our Proposal Mirage

Figure 4.1: (a) Traditional LLCs have set-associative evictions (SAE) which leak information of installed addresses. (b) MIRAGE provides Global Evictions (GLE) to provide an abstraction similar to a fully-associative cache to eliminate this information leakage. (c) Mirage enables GLEs and eliminates SAEs with practical set-associative lookups.

to occur and using these set-conflicts, new attacks [29, 10, 81] with more efficient algorithms were able to de-obfuscate the set mapping in these defenses and continue to launch set-conflict based attacks, thus rendering these defenses broken.

Towards a Principled Randomized Cache Defense. In this work[1], we address the root cause of set-conflict based attacks – *set-conflicts*. Our goal is to eliminate set-conflicts and attacks exploiting them, by redesigning the cache with *global evictions*, i.e the victims for eviction are chosen (randomly) from among *all* the lines in the cache. With global evictions, any line resident in the cache can get evicted when a new address is installed into the cache, as shown in Figure 4.1(b), and an adversary observing an eviction gains no information about the installed address, thus eliminating set-conflict based cache attacks.

A fully associative cache design naturally provides such global evictions. However, a key challenge is ensuring practical cache lookup. As a line can reside in any cache location, a cache lookup requires searching through the entire fully-associative LLC (containing tens of thousands of lines) and can be slower than even a memory access. Towards a practical defense, this chapter explores a design called MIRAGE[2], that provides the security of a fully-associative design (with global evictions), and the practical lookup of a set-associative design. Mirage achieves this with a combination of over-provisioning, tag-to-data indirection, and load-balancing, as shown in Figure 4.1(c). Before diving into Mirage's design, we first explore the background of the arms race in randomized caches to motivate our design.

[1]This chapter was published as a part of the paper "*MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design*", appearing in the proceedings of USENIX Security 2021 [31].

[2]MIRAGE stands for Multi-Index Randomized Cache with Global Evictions.

4.2 Background: History of the Arms Race in Randomized Caches

Last-level caches (LLCs) are shared among multiple cores for performance. So cachelines of different processes can contend for the space in a cache set and evict each other from the cache. Such ”set-conflicts” are exploited in side-channel attacks to evict victim cachelines.

4.2.1 Threat Model

We assume a threat model where the attacker and victim execute simultaneously on different physical cores sharing an LLC, that is inclusive of the L1/L2 caches private to each core. We focus on conflict-based cache side-channel attacks where the attacker causes set-conflicts to evict a victim's line and monitor the access pattern of the victim. For simplicity, we assume no shared memory between victim and attacker, as existing solutions [11] are effective at mitigating possible attacks on shared lines. In Section 4.5, we discuss how Mirage defends against shared-memory based attacks like Flush+Reload or Evict+Reload.

4.2.2 Eviction Set Discovery in Conflict-Based Cache Attacks

The Prime+Probe attack [38] is a classic example of set-conflict based cache attacks. Here, the attacker primes a set with its addresses, allows the victim to evict an attacker line due to set-conflicts, and then probes addresses to check if there is a miss due to the eviction. This allows the spy to infer that the victim accessed that set. Before launching such attacks, the first step for an attacker is to generate an *eviction-set* for a victim address, i.e. a minimal set of addresses that can evict the victim address by mapping to the same cache set.

4.2.3 Advances in Attacks and Defenses

Given how critical eviction-set discovery is for such attacks, recent defenses have proposed randomized caches to obfuscate the address to set mapping and make it harder to learn eviction sets. At the same time, recent attacks have enabled faster algorithms for eviction

set discovery despite these defenses. As a result, this has fueled an arms race between attack and defense. Next, we describe the moves made by recent attacks and defenses.

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

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

Move-2: Defense via Encryption and Remapping

CEASER [9] (shown in Figure 4.2(a)) proposed randomizing the address to set mapping by accessing the cache with an encrypted line address. By enabling dynamic re-keying, it ensures that the mapping changes before an eviction-set can be discovered with an algorithm that requires $O(n^2)$ accesses.

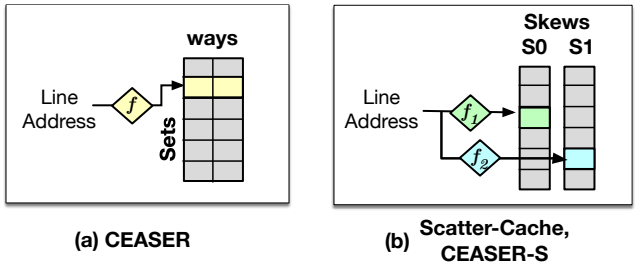


Figure 4.2: Recent Works on Randomized Caches

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

Subsequent works [10, 29] developed faster algorithms to discover eviction-sets in $O(n)$ accesses, eliminating groups of lines from potential candidates rather than one line at a time. CEASER is unable to prevent eviction-set discovery in time with such algorithms.

Move-4: Defense via Skewed Associativity

Scatter-Cache [11] and CEASER-S [10] adopt skewed associativity in addition to randomized mapping of addresses to sets, to further obfuscate the LLC evictions. As shown in Figure 4.2(b), such designs partition the cache across ways into multiple skews, with each skew having a different set-mapping, and a new address is installed in a randomly selected

skew. Such a design provides greater obfuscation as eviction sets get decided by the line to skew mapping as well. These designs were shown to be immune to faster eviction set discovery algorithms [10, 29] that require $O(n)$ steps.

Move-5: Attack by Probabilistic Eviction Set Discovery

A recent work [81] showed that faster eviction-set discovery in Scatter-Cache is possible with an intelligent choice of initial conditions, that boosts the probability of observing conflicts. This allows the discovery of partial eviction-sets (lines that evict a target in a subset of the ways) within 140K accesses in Scatter-Cache, enabling a conflict-based attack.

Pitfalls of Prior Defenses

The security of defenses has hinged on obfuscation of eviction-sets. However, newer algorithms enabling faster eviction-set discovery are capable of de-obfuscating these defenses. Ideally, we seek a defense that eliminates *Set-Associative Evictions (SAE)*, which leak information about mappings and allow de-obfuscation of eviction-sets. Eliminating SAE would not only safeguard against current eviction-set discovery algorithms but also a hypothetical oracular algorithm that can learn an eviction-set with just a single SAE.

4.2.4 Goal: A Practical Fully-Associative LLC

As a principled defense against conflict-based attacks, we seek to design a cache that provides *Global Eviction (GLE)*, *i.e.*, the eviction candidates are selected from among all of the addresses resident in the cache when new addresses are installed. Such a defense would eliminate SAE and be immune to eviction-set discovery, as evicted addresses are independent of the addresses installed and leak no information about installed addresses. While a fully-associative design provides global evictions, it incurs prohibitive latency and power overheads when adopted for an LLC.[3] The goal of this work is to develop an LLC design that guarantees global evictions while retaining practical set-associative lookup.

[3]Recent works propose fully-associative designs for a subset of the cache (Hybcache [82]) or for L1-Caches (RPCache [83], NewCache[84]). These approaches are impractical for LLCs (see Section 4.10.1).

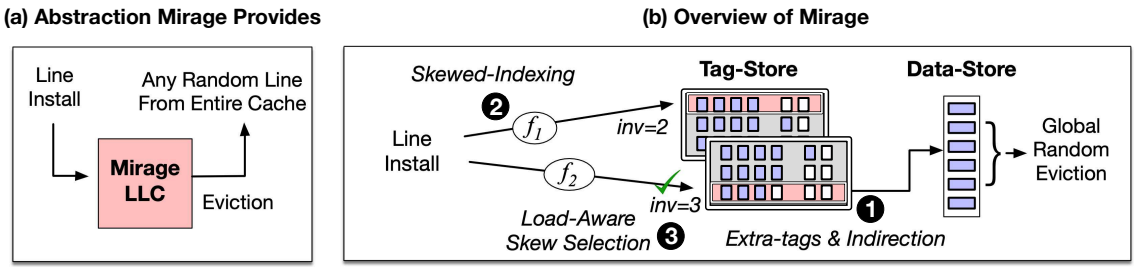


Figure 4.3: (a) Mirage provides the abstraction of a fully-associative design with globally random evictions. (b) It achieves this by using extra tags and indirection between tags and data blocks, skewed indexing, and load-aware skew-selection.

4.3 Design: Full Associativity via MIRAGE

To guarantee global evictions practically, we propose *Mirage (Multi-Index Randomized Cache with Global Evictions)*. Mirage provides the abstraction of a fully associative cache with random replacement, as shown in Figure 4.3(a). It has the property that on a cache miss, a line selected randomly from among all resident lines is evicted. This ensures the evicted line is independent of the incoming line and leaks no information.

4.3.1 Overview of Mirage

Mirage has three key components, as shown in Figure 4.3(b). First, it uses a cache organization that decouples tag and data location and uses indirection to link tag and data entries (❶ in Figure 4.3(b)). Provisioning extra invalid tags allows accommodating new lines in indexed sets without tag conflicts, and indirection between tags and data blocks allows victim selection from the data-store in a global manner. Second, Mirage uses a tag-store design that splits the tag entries into two structures (skews) and accesses each of them with a different hashing function (❷ in Figure 4.3(b)). Finally, to maximize the likelihood of getting an invalid tag on cache install, Mirage uses a load-balancing policy for skew-selection leveraging the ”power of 2 choices” [32] (❸ in Figure 4.3(b)), which ensures no SAE occurs in the system lifetime and all evictions are global. We describe each component next.

4.3.2 Tag-to-Data Indirection and Extra Tags

V-way Cache Substrate: Figure 4.4 shows the tag and data store organization using pointer-based indirection in Mirage, which is inspired by the V-way cache [85]. V-way originally used this substrate to reduce LLC conflict-misses and improve performance. Here, the tag-store is over-provisioned to include extra invalid tags, which can accommodate the metadata of a new line without a set-associative eviction (SAE). Each tag-store entry has a forward pointer (FPTR) to allow it to map to an arbitrary data-store entry.[4] On a cache-miss, two types of evictions are possible: if the incoming line finds an invalid tag, a Global Eviction (GLE) is performed; else, an SAE is performed to invalidate a tag (and its corresponding data entry) from the set where the new line is to be installed. On a GLE, V-way cache evicts a data entry selected from the entire data-store and also the corresponding tag identified using a reverse pointer (RPTR) stored with each data entry. In both cases, the RPTR of the invalidated data entry is reset to invalid. This data entry and the invalid tag in the original set are used by the incoming line.

Repurposing V-way Cache for Security: Mirage adopts the V-way cache substrate with extra tags and indirection to enable GLE, but with an important modification: it ensures the data-entry victim on a GLE is selected randomly from the entire data-store (using a hardware PRNG) to ensure that it leaks no information. Despite this addition, the V-way cache by itself is not secure, as it only reduces but does not eliminate SAE. For example, if an adversary has arbitrary control over the placement of new lines in specific sets, they can map a large number of lines to a certain set and deplete the extra invalid tags provisioned in that set. When a new (victim) line is to be installed to this set, the cache is then forced to evict a valid tag from the same set and incur an SAE. Thus, an adversary who can discover the address to set mapping can force an SAE on each miss, making a naive adoption of the V-way Cache vulnerable to the same attacks in conventional set-associative caches.

[4]While indirection requires a cache lookup to serially access the tag and data entries, commercial processors [86, 87, 88] since the last two decades already employ such serial tag and data access for the LLC to save power (this allows the design to only access the data way corresponding to the hit).

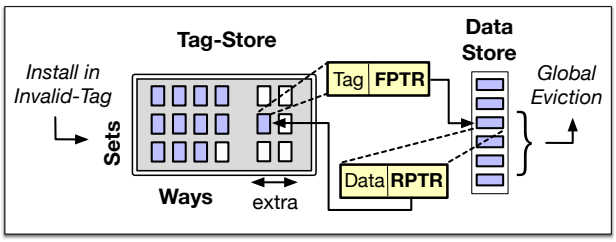


Figure 4.4: Overview of the cache substrate used by Mirage with indirection and extra tags (inspired by V-Way Cache).

4.3.3 Skewed-Associative Tag-Store Design

To ensure GLE on each line install, Mirage reshapes the tag organization. To allow an address to map to multiple sets in the tag store and increase the probability of obtaining an invalid tag, Mirage architects the tag-store as a skewed-associative structure [89]. The tag store is split into two partitions or skews, and a different randomizing hash function is used to map addresses to sets in each skew. The hash function[5] to map addresses to sets is constructed using a 12-round PRINCE cipher [90], which is a low-latency 64-bit block-cipher using 128-bit keys. Note that prior work [27] used a reduced round version of PRINCE cipher for randomized cache indexing.

Unlike prior defenses using skewed-associativity [10, 11], each skew in Mirage contains invalid tags. Offering the flexibility for a new line to map to two sets (one in each skew) in the presence of invalid tags significantly increases the chance of finding an invalid tag in which it can be installed and avoiding an SAE. Moreover, the cryptographically generated address-to-set mapping ensures that the adversary (without knowing the secret key) cannot arbitrarily deplete these invalid tags within a set.

4.3.4 Load-Aware Skew Selection

Natural imbalance in usage of tags across sets can deplete invalid tags across sets and cause an SAE. On a line install, the skew-selection policy, that decides the skew in which the line

[5]The hash-function construction is similar to Scatter-Cache (SCv1) [11], where set-index bits are sliced from a cipher-text encrypted using a plaintext of physical line-address concatenated with a Security-Domain-ID, and the set-index for each skew is computed using a different secret key.

is installed, determines the distribution of invalid tags across sets. Prior works, including Scatter-Cache [11] and CEASER-S [10], use random skew-selection, which randomly picks one of the two skews on a line install. With invalid tags, this policy can result in imbalanced sets – some with many invalid tags and others with none (that incur SAE). Our analysis, using a buckets-and-balls model we describe in Section 4.4.1, indicates that a random skew-selection policy results in an SAE every few misses (every 2600 misses with 6 extra ways/skew), and provides robustness only for microseconds.

To guarantee the availability of invalid tags across sets and eliminate SAE, Mirage uses a load-aware skew selection policy inspired by *"Power of 2 Choices"* [32, 91], a load-balancing technique used in hash-tables. As indicated by ❸ in Figure 4.3, this policy makes an intelligent choice between the two skews, installing the line in the skew where the indexed set has a higher number of invalid tags. In the case of a tie between the two sets, one of the two skews is randomly selected. With this policy, an SAE occurs only if the indexed sets in both skews do not have invalid tags, that is a rare occurrence as this policy actively promotes balanced usage of tags across sets. Table 4.1 shows the rate of SAE for Mirage with load-aware skew selection policy, as the number of extra tags per skew is increased from 0 to 6. Mirage with 14-ways per skew (75% extra tags) encounters an SAE once in 10^{34} cache installs, or equivalently 10^{17} years, ensuring no SAE throughout the system lifetime. We derive these bounds analytically in Section 4.4.3.

4.4 Security Analysis of Mirage

In this section, we analyze set-conflict-based attacks in a setting where the attacker and the victim do not have shared memory (shared-memory attacks are analyzed in Section 4.5). All existing set-conflict based attacks, such as Prime+Probe [38], Prime+Abort [40], Evict+Time [38], etc., exploit eviction-sets to surgically evict targeted victim addresses, and all eviction-set discovery algorithms require the attacker to observe evictions dependent on the addresses accessed by the victim. In Mirage, two types of evictions are possible – a

Table 4.1: Frequency of Set-Associative Eviction (SAE) in Mirage as extra ways per skew increase (assuming a baseline of 16-MB LLC with 16-ways and 1ns per install)

Ways in each Skew (Base + Extra)	**Installs per SAE**	**Time per SAE**
8 + 0	1	1 ns
8 + 1	4	4 ns
8 + 2	60	60 ns
8 + 3	8000	8 us
8 + 4	2×10^8	0.16 s
8 + 5	7×10^{16}	2 years
8 + 6 (default Mirage)	10^{34}	10^{17} years

global eviction, where the eviction candidate is selected randomly from all the lines in the data-store, that leaks no information about installed addresses; or a set-associative eviction (SAE), where the eviction candidate is selected from the same set as the installed line due to a tag-conflict, that leaks information. To justify how Mirage eliminates conflict-based attacks, we estimate the rate of SAE and show how an SAE is unlikely in system lifetime.

Our security analysis makes the following assumptions:

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

2. **Even a single SAE is sufficient to break the security.** The number of accesses required to construct an eviction-set has reduced due to recent advances, with the state-of-the-art [2, 10, 29] requiring at least a few hundred SAE to construct eviction-sets. To potentially mitigate even future advances in eviction-set discovery, we consider a powerful hypothetical adversary that can construct an eviction-set with just a single SAE (the theoretical minimum), unlike previous defenses [9, 10, 11] that only consider existing eviction-set discovery algorithms.

Table 4.2: Parameters for Buckets and Balls Modeling

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

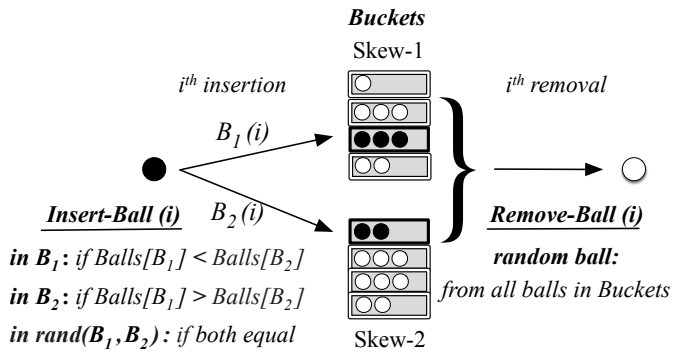


Figure 4.5: Buckets-and-balls model for Mirage with 32K buckets (divided into 2 skews), holding 256K balls in total to model a 16MB cache. The bucket capacity is varied from 8-to-14 to model 8-to-14 ways per skew in Mirage.

4.4.1 Bucket-And-Balls Model

To estimate the rate of SAE, we model the operation of Mirage as a buckets-and-balls problem, as shown in Figure 4.5. Here each bucket models a cache set and each ball throw represents a new address installed into the cache. Each ball picks from 2 randomly chosen buckets, one from each skew, and is installed in the bucket with more free capacity, modeling the skew selection in Mirage. If both buckets have the same number of balls, one of the two buckets is randomly picked.[6] If both buckets are full, an insertion will cause a *bucket spill*, equivalent to an SAE in Mirage. Otherwise, on every ball throw, we randomly remove a ball from among all the balls in buckets to model Global Eviction. The parameters of our model are shown in Table 4.2. We initialize the buckets by inserting as many balls as

[6]A biased tie-breaking policy [92] that always picks Skew-1 on ties further reduces the frequency of bucket-spills by a few orders of magnitude compared to random tie-breaks. However, to keep our analysis simple, we use a random tie-breaking policy.

cache capacity (in number of lines) and then perform 10 trillion ball insertions and removals to measure the frequency of bucket spills (equivalent to SAE). Note that having fewer lines in the cache than the capacity is detrimental to an attacker, as the probability of a spill would be lower; so we model the best-case scenario for the attacker.

4.4.2 Empirical Results for Frequency of Spills

Figure 4.6 shows the average number of balls thrown per bucket spill, analogous to the number of line installs required to cause an SAE on average. As bucket capacity increases from 8 to 14, there is a considerable reduction in the frequency of spills. When the bucket capacity is 8, there is a spill on every throw as each bucket has 8 balls on average. As bucket capacity increases to 9 / 10 / 11 / 12, the spill frequency decreases to once every 4 / 60 / 8000 / 160Mn balls. For bucket capacities of 13 and 14, we observe no bucket spills even after 10 trillion ball throws. These results show that as the number of extra tags increases, the probability of an SAE in Mirage decreases super-exponentially (better than squaring on every extra way). With 12 ways/skew (50% extra tags), Mirage has an SAE every 160 million installs (equivalent to every 0.16 seconds).

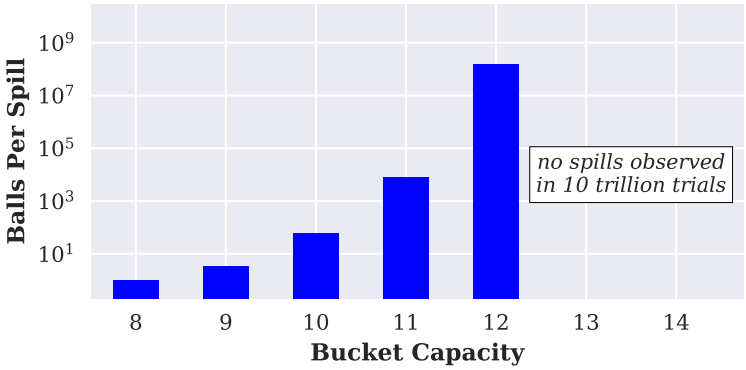


Figure 4.6: Frequency of bucket spills, as bucket capacity is varied. As bucket-capacity increases from 8 to 14 (i.e. extra-tags per set increase from 0% to 75%), bucket spills (equivalent to SAE) become more infrequent.

While an empirical analysis is useful for estimating the probability of an SAE with up to 12 ways/skew, increasing the ways/skew further makes the frequency of SAE super-

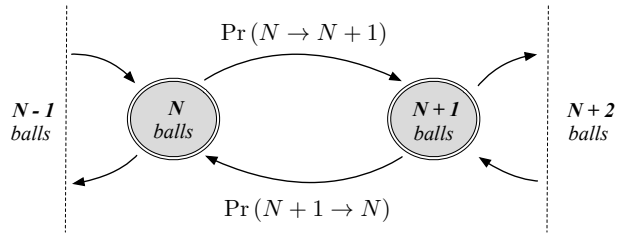


Figure 4.7: Bucket state modeled as a Birth-Death chain, a Markov Chain where the state variable N (number of balls in a bucket) increases or decreases by one at a time, due to a birth (insertion) or death (deletion) of a ball.

exponentially less. Hence, it is impractical to empirically compute the probability of SAE in a reasonable amount of time beyond 12 ways/skew (an experiment with 10 trillion ball throws already takes a few days to simulate). To estimate the probability of SAE for a Mirage design with 14 ways/skew, we develop an analytical model as described next.

Table 4.3: Terminology used in the analytical model

Symbol	Meaning
$\Pr(n = N)$	Probability that a Bucket contains N balls
$\Pr(n \leq N)$	Probability that a Bucket contains $\leq N$ balls
$\Pr(X \to Y)$	Probability that a Bucket with X balls transitions to Y balls
W	Capacity of a Bucket (beyond which there is a spill)
B_{tot}	Total number of Buckets (32K)
b_{tot}	Total number of Balls (256K)

4.4.3 Analytical Model for Bucket Spills

To estimate the probability of bucket spills analytically, we start by modeling the behavior of our buckets and balls system in a spill-free scenario (assuming unlimited capacity buckets). We model the bucket-state, i.e. the number of balls in a bucket, as a Birth-Death chain [93], a type of Markov chain where the state-variable (number of balls in a bucket) only increases or decreases by 1 at a time due to birth or death events (ball insertion or deletions), as shown in Figure 4.7.

We use a classic result for Birth-Death chains, that in the steady-state, the probability of each state converges to a steady value and the net rate of conversion between any two states

becomes zero. Applying this result to our model in Figure 4.7, we can equate the probability of a bucket with N balls transitioning to N+1 balls and vice-versa to get Equation (4.1). The terminology used in our model is shown in Table 4.3.

$$\Pr(N \rightarrow N+1) = \Pr(N+1 \rightarrow N) \tag{4.1}$$

To calculate $\Pr(N \rightarrow N+1)$, we note that a bucket with N balls transitions to N+1 balls on a ball insertion if: (1) the buckets chosen from both Skew-1 and Skew-2 have N balls; *or* (2) bucket chosen from Skew-1 has N balls and from Skew-2 has more than N balls; *or* (3) bucket chosen from Skew-2 has N balls and from Skew-1 has more than N balls. Thus, if the probability of a bucket with N balls is $\Pr(n = N)$, the probability it transitions to N+1 balls is given by Equation (4.2).

$$\Pr(N \rightarrow N+1) = \Pr(n=N)^2 + 2 * \Pr(n=N) * \Pr(n>N) \tag{4.2}$$

To calculate $\Pr(N+1 \rightarrow N)$, we note that a bucket with N+1 balls transitions to N balls only on a ball removal. As a random ball is selected for removal from all the balls, the probability that a ball in a bucket with $N+1$ balls is selected for removal equals the fraction of balls in such buckets. If the number of buckets equals B_{tot} and the number of balls is b_{tot}, the probability of a bucket with $N+1$ balls losing a ball (i.e. the fraction of balls in such buckets), is given by Equation (4.3).

$$\Pr(N+1 \rightarrow N) = \frac{\Pr(n = N+1) * B_{tot} * (N+1)}{b_{tot}} \tag{4.3}$$

Combining Equation (4.1),Equation (4.2), Equation (4.3), and placing $B_{tot}/b_{tot} = 1/8$, (the number of buckets/balls) we get the probability of a bucket with N+1 balls, as given by Equation (4.4) and Equation (4.5).

$$\Pr(n{=}N{+}1){=}\frac{8}{N{+}1}*\Big(\Pr(n{=}N)^2 \\ +2*\Pr(n{=}N)*\Pr(n{>}N)\Big) \tag{4.4}$$

$$=\frac{8}{N{+}1}*\Big(\Pr(n{=}N)^2{+}2*\Pr(n{=}N) \\ -2*\Pr(n{=}N)*\Pr(n{\leq}N)\Big) \tag{4.5}$$

As n grows, $\Pr(n = N) \to 0$ and $\Pr(n > N) \ll \Pr(n = N)$ given our empirical observation that these probabilities reduce super-exponentially. Using these conditions Equation (4.4) can be simplified to Equation (4.6) for larger n.

$$\Pr(n = N + 1) = \frac{8}{N + 1} * \Pr(n = N)^2 \tag{4.6}$$

From our simulation of 10 trillion balls, we obtain probability of a bucket with no balls as $\Pr_{obs}(n = 0) = 4 \times 10^{-6}$. Using this value in Equation (4.5), we recursively calculate $\Pr_{est}(n = N + 1)$ for $N \in [1, 10]$ and use Equation (4.6) for $N \in [11, 14]$, when the probabilities become less than 0.01. Figure 4.8 shows the empirical ($\Pr_{obs}$) and analytically estimated ($\Pr_{est}$) probability of a bucket having N balls. $\Pr_{est}$ matches $\Pr_{obs}$ for all available data points.

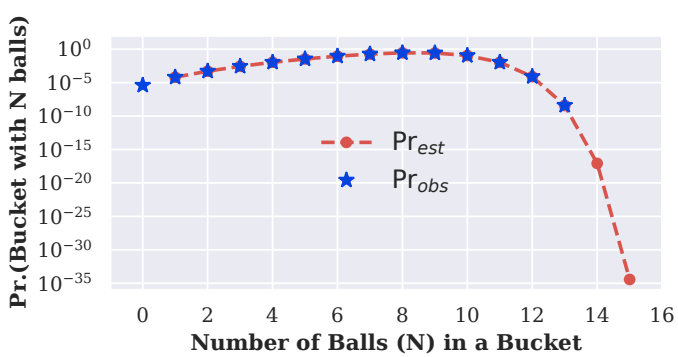


Figure 4.8: Probability of a Bucket having N balls – Estimated analytically ($\Pr_{est}$) and Observed ($\Pr_{obs}$)

Figure 4.8 shows that the probability of a set having N lines decreases double-exponentially beyond 8 lines per set (the average number of data-lines per set). For N = 13 / 14 / 15, the probability reaches 10^{-9} / 10^{-17} / 10^{-35}. This behavior is due to two reasons – (a) for a set to get to N+1 lines, a new line must map to two sets with at least N lines; (b) a set with a higher number of lines is more likely to lose a line due to random global eviction. Using these probabilities, we estimate the frequency of SAE in the next section.

4.4.4 Analytical Results for Frequency of Spills

For a bucket of capacity *W*, the spill-probability (without relocation) is the probability that a bucket with W balls gets to $W + 1$ balls. By setting $N = W$ in Equation (4.2) and $\Pr(n > W) = 0$, we get the spill-probability as Equation (4.7).

$$\text{Pr}_{spill} = \Pr(W \rightarrow W + 1) = \Pr(n = W)^2 \tag{4.7}$$

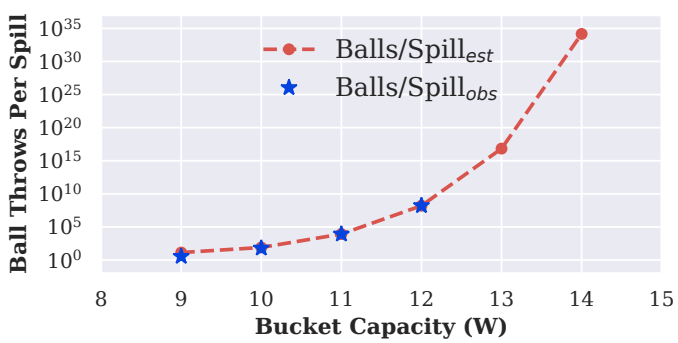


Figure 4.9: Frequency of bucket-spill, as bucket-capacity varies – both analytically estimated (Balls/Spill$_{est}$) and empirically observed (Balls/Spill$_{obs}$) results are shown.

Figure 4.9 shows the frequency of bucket-spills (SAE) estimated by using $\Pr_{est}(n = W)$, from Figure 4.8, in Equation (4.7). The estimated values (Balls/Spill$_{est}$) closely match the empirically observed values (Balls/Spill$_{obs}$) from Section 4.4.2. As the number of tags per set, i.e. bucket-capacity (W) increases, the rate of SAE, i.e. the frequency of bucket-spills shows a double-exponential reduction (which means the exponent itself is increasing exponentially). The probability of a spill with x extra ways is of the form $P^{(2^x)}$; therefore with

5-6 extra ways, we get an extremely small probability of spill as the exponent term reaches 32 – 64. For W = 12 / 13 / 14, an SAE occurs every 10^8 / 10^{16} / 10^{34} line installs. Thus, the default Mirage design with 14-ways per set, with a rate of one SAE in 10^{34} line installs (i.e. once in 10^{17} years), effectively provides the security of a fully associative cache.

4.5 Protecting against Shared-Memory Attacks

Thus far, we have focused primarily on attacks that cause eviction via set conflicts and without any shared data between the victim and the attacker. If there is shared-memory between the victim and the attacker, attacks such as Flush +Reload [8], Flush+Flush [23], Invalidate+Transfer [94], Flush+Prefetch [95], Thrash+Reload [43], Evict+Reload [42], etc. are possible, where an attacker evicts the shared line from the cache using *clflush* instruction or cache-thrashing [43] or by accessing the line's eviction-set [42], and issues subsequent loads or flushes [23] to the line while measuring its latency to monitor victim accesses to that line. We describe how Mirage is protected against these attacks based on the type of shared memory being attacked.

Shared Read-only Memory: Attacks on shared read-only memory addresses are prevented in Mirage by placing distrusting programs (victim and attacker) in different security domains and maintaining duplicate copies of shared lines in the cache for each security domain. Such duplication ensures that a load on a shared address from one domain does not hit on the copy of another domain (similarly flush from one domain does not evict another's copy) and has been used in several prior secure cache works [16, 11, 82]. For example, Scatter-Cache (SCv1) [11] uses Security-Domain-ID (SDID) concatenated with the physical line-address as input to the set index derivation function (IDF), allowing a shared address to map to different sets for different domains and get duplicated. Mirage uses an IDF construction identical to Scatter-Cache SCv1 and similarly duplicates shared lines across domains.

However, we observe that relying on the IDF to create duplicate copies has a weakness: it can allow a shared-memory address in two different SDIDs to map to the same set in a skew with a small probability ($1/number\text{-}of\text{-}sets$), which can result in a single copy of the line. To guarantee duplicate copies of a line across domains even in this scenario, Mirage stores the SDID of the domain installing the line along with the tag of the line, so that a load (or a flush) of a domain hits on (or evicts) a cache line only if the SDID matches along with the tag-match. Mirage stores 8-bit SDID supporting up to 256 security domains (similar to DAWG [16]), which adds $<3\%$ LLC storage overhead; however more or fewer SDID can be supported without any limitations in Mirage.

Shared Writable Memory: It is infeasible to duplicate shared writeable memory across domains, as such a design is incompatible with cache-coherence protocols [11, 16]. To avoid attacks on such memory, we require that writable shared-memory is not used for any sensitive computations and only used for data transfers incapable of leaking information.

4.6 Discussion

4.6.1 Requirements for Randomizing Function

The randomizing function used to map addresses to cache sets in each skew is critical in ensuring balanced availability of invalid tags across sets and eliminating SAE. We use a cryptographic function (computed with a secret key in hardware) so that an adversary cannot arbitrarily target specific sets. This is also robust to *shortcut attacks* [12], which can exploit vulnerabilities in the algorithm to deterministically engineer collisions. Furthermore, the random mapping for each skew must be mutually independent to ensure effective load-balancing and minimize naturally occurring collisions, as required by power-of-2-choices hashing [96]. We satisfy both requirements using a cryptographic hash function constructed using the PRINCE cipher, using separate keys for each skew. Other ciphers and cryptographic hashes that satisfy these requirements may also be used for Mirage.

4.6.2 Key Management in Mirage

The secret keys used for the randomizing set-index derivation function are stored in hardware and not visible to software including the OS. As no information about the mapping function leaks in the absence of SAE in Mirage, by default Mirage does not require continuous key-refreshes like CEASER / CEASER-S [9, 10] or keys provisioned per domain like Scatter-Cache[11]). We recommend that the keys for Mirage be generated at boot-time within the cache controller (using a secure pseudorandom number generator in hardware).

Mirage can also have the capability to refresh the keys (with a cache flush) in the event of any key or mapping leakage. All prior randomized cache designs become vulnerable to conflict-based attacks if the adversary guesses the key (1 in 2^{64} chance) or if the mappings leak via unknown attacks, as they have no means of detecting such a breakdown in security. On the other hand, Mirage can automatically detect leakage of mappings or keys via even hypothetical future attacks, as any subsequent conflict-based attack requires an SAE, which is not expected to occur under normal operation in Mirage. If multiple SAEs are encountered indicating that the mapping is no longer secret, Mirage can refresh its keys and mappings (with a cache flush) and ensure continued security.

4.6.3 Security for Sliced LLC Designs

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

4.6.4 Security as Baseline Associativity Varies

The rate of SAE strongly depends on the number of ways provisioned in the tag-store. Table 4.4 shows the rate of SAE for a 16MB LLC, as the baseline associativity varies from 8 ways – 32 ways. As the baseline associativity varies, with just 1 extra way per skew, the different configurations have an SAE every 13 – 14 installs. However, adding each extra way squares the rate successively as per Equation (4.7). Following the double-exponential curve of Figure 4.9, the rate of an SAE goes beyond once in 10^{12} years (well beyond system lifetime) for all three configurations within 5–6 extra ways.

Table 4.4: Cacheline installs Per SAE in Mirage as the baseline associativity of the LLC tag-store varies

LLC Associativity	**8-ways**	**16-ways (default)**	**32-ways**
1 extra way/skew	13 ($<$ 20ns)	14 ($<$ 20ns)	14 ($<$ 20ns)
5 extra ways/skew	10^{21} (10^{4} yrs)	10^{16} (2 yrs)	10^{14} (3 days)
6 extra ways/skew	10^{43} (10^{26} yrs)	10^{34} (10^{17} yrs)	10^{29} (10^{12} yrs)

4.6.5 Implications for Other Cache Attacks

Replacement Policy Attacks: Reload+Refresh [44] attack exploited the LLC replacement policy to influence eviction decisions within a set, and enable a side-channel stealthier than Prime+Probe or Flush+Reload. Mirage guarantees global evictions with random replacement, that has no access-dependent state. This ensures that an adversary cannot influence the replacement decisions via its accesses, making Mirage immune to such attacks.

Cache-Occupancy Attacks: Mirage prevents an adversary that observes an eviction from gaining any information about the address of an installed line. However, the fact that an eviction occurred continues to be observable, similar to prior works such as Scatter-Cache [11] and HybCache [82]. Consequently, Mirage and these prior works, are vulnerable to attacks that monitor the cache-occupancy of a victim by measuring the number of evictions, like a recent attack [3] that used cache-occupancy as a signature for website

fingerprinting. The only known way to effectively mitigate such attacks is static partitioning of the cache space. We discuss how cache partitioning schemes can be designed in a practical manner in the next chapter of this thesis.

4.7 Mirage with Cuckoo-Relocation

The default design for Mirage consists of 6 extra ways / skew (75% extra tags) that avoids SAE for well beyond the system lifetime. If Mirage is implemented with fewer extra tags (e.g. 4 extra ways/skew or 50% extra tags), it can encounter SAE as frequently as once in 0.16 seconds. To avoid an SAE even if only 50% extra tags are provisioned in Mirage, we propose an extension of Mirage that relocates conflicting lines to alternative sets in the other skew, much like Cuckoo Hashing [97]. We call this extension *Cuckoo-Relocation*.

4.7.1 Design of Cuckoo-Relocation

The design of Cuckoo-Relocation is shown using an example in Figure 4.10. An SAE is required when an incoming line (Line Z) gets mapped in both skews to sets that have no invalid tags (Figure 4.10(a)). To avoid an SAE, we need an invalid tag in either of these sets. To create such an invalid tag, we randomly select a candidate line (Figure 4.10(b)) from either of these sets and relocate it to its alternative location in the other skew. If this candidate maps to a set with an invalid tag in the other skew, the relocation leaves behind an invalid tag in the original set, in which the line to be installed can be accommodated without an SAE, as shown in Figure 4.10(c). If the relocation fails as the alternative set is full, it can be attempted again with successive candidates till a certain number of maximum tries, after which an SAE is incurred. For Mirage with 50% extra tags, an SAE is infrequent even without relocation (less than once in 100 million installs). So in the scenario where an SAE is required, it is likely that other sets have invalid tags and relocation succeeds.

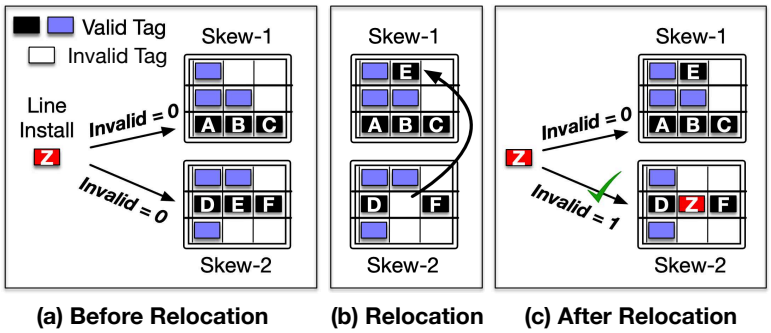


Figure 4.10: Cuckoo Relocation, a technique to avoid an SAE if Mirage is implemented with 50% extra tags.

4.7.2 Results: Impact of Relocation on SAE

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

Table 4.5 shows the rate of SAE for Mirage with 50% extra tags and Cuckoo-Relocation, as the maximum number of relocation attempts is varied. Attempting relocation for up to 3 lines is sufficient to ensure that an SAE does not occur in system lifetime (SAE occurs once in 22000 years). We note that attempting relocation for up to 3 lines can be done in the shadow of a memory access on a cache-miss.

Table 4.5: Frequency of SAE in Mirage with 50% extra tags (4 extra ways/skew) as the number of relocation attempts increase

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

4.7.3 Security Implications of Relocation

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

4.8 Performance Analysis

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

4.8.1 Methodology

Similar to prior works on randomized caches [11, 9, 10, 25], we use a micro-architecture simulator to evaluate performance. We use an in-house simulator that models an inclusive 3-level cache hierarchy (with private L1/L2 caches and shared L3 cache) and DRAM in detail, and has in-order x86 cores supporting a subset of the instruction-set. The simulator input is a 1 billion instructions long program execution-trace (consisting of instructions and memory addresses), chosen from a representative phase of a program using the Simpoints sampling methodology [98] and obtained using an Intel Pintool [99].

As our baseline, we use a non-secure 16-way, 16MB set-associative LLC configured as shown in Table 4.6. For Mirage, we estimate the LLC access latency using RTL-synthesis

of the cache-lookup circuit (Section 4.8.2) and Cacti-6.0 [100] (a tool that reports timing, area, and power for caches), and show that it requires 4 extra cycles compared to the baseline (3-cycles for PRINCE cipher and 1 extra cycle for tag and data lookup). For comparisons with the prior state-of-the-art, we implement Scatter-Cache with 2-skews, 8 ways/skew and use PRINCE cipher for the hash function for set-index derivation, which adds 3 cycles to lookups compared to baseline (to avoid an unfair advantage to Mirage, as Scatter-Cache [11] originally used a 5-cycle QARMA-64 cipher).

We evaluate 58 workloads, including all 29 SPEC CPU2006 benchmarks (each has 8 duplicate copies running on 8 cores) and 29 mixed workloads (each has 8 randomly chosen SPEC benchmarks).

Table 4.6: Baseline System Configuration

Processor and Last-level Cache	
Core	8-cores, In-order Execution, 3GHz
L1 and L2 Cache Per Core	L1-32KB, L2-256KB, 8-way, 64B line size
LLC (shared across cores)	16MB, 16-way Set-Associative, 64B line size LRU Replacement Policy, 24 cycle lookup
DRAM Memory-System	
Frequency, tCL-tRCD-tRP	800 MHz (DDR 1.6 GHz), 9-9-9 ns
DRAM Organization	2-channel (8-Banks each), 2KB Row-Buffer

4.8.2 Synthesis Results for Cache Access Latency

Compared to the baseline, the cache access in Mirage additionally requires (a) set-index computation using the PRINCE cipher based hash-function, (b) look-up of 8-12 extra ways of the tag-store, and (c) FPTR-based indirection on a hit to access the data. We synthesized the RTL for the set-index derivation function with a 12-round PRINCE cipher [90] based on a public VHDL implementation [101], using Synopsys Design Compiler and FreePDK 15nm gate library [102]. A 3-stage pipelined implementation (with 4 cipher rounds/stage) has a delay of 320ps per stage (which is less than a cycle period). Hence, we add 3 cycles to the LLC access latency for Mirage (and Scatter-Cache), compared to the baseline.

We also synthesized the FPTR-indirection circuit consisting of AND and OR gates to select the FPTR of the hitting way, and a 4-to-16 decoder to select the data-store way using the lower 4-bits of the FPTR (the remaining FPTR-bits form the data-store set-index). This circuit has a maximum delay of 72ps. Using Cactii-6.0 [100], we estimate that lookup of 16 extra ways from the tag-store further adds 200ps delay in 32nm technology. To accommodate the indirection and tag access delays, we increase the LLC-access latency for Mirage further by 1 cycle (333ps). Overall, Mirage incurs 4 extra cycles for cache accesses compared to the baseline. Note that the RPTR-lookup and the skew selection logic (counting valid bits in the indexed sets) required on a cache miss need simple circuitry (delay less than a cycle) and are used only in the background while a DRAM access completes.

Table 4.7: Average LLC MPKI of Mirage and Scatter-Cache

Workloads	Baseline	Mirage	Scatter-Cache
SpecInt-12	10.79	11.23	11.23
SpecFp-17	8.82	8.51	8.51
Mix-29	9.52	9.97	9.97
All-58	9.58	9.80	9.80

4.8.3 Impact on Cache Misses

Table 4.7 shows LLC Misses Per 1000 Instructions (MPKI) for the non-secure Baseline, Mirage, and Scatter-Cache averaged for each workload suite. We observe that all LLC-misses in Mirage in all workloads result in Global Evictions (no SAE), in line with our security analysis.[7] Compared to the Baseline, Mirage incurs 2.4% more misses on average (0.2 MPKI extra) as the globally-random evictions from the data-store lack the intelligence of the baseline LRU policy that preserves addresses likely to be re-used. The miss count for Scatter-Cache is similar to Mirage as it uses randomized set-indexing that causes randomized evictions with similar performance implications (however note that all its evictions

[7]Workloads typically do not always access random addresses. But the randomized cache-set mapping used in Mirage ensures accesses always map to random cache-sets, which allows the load-balancing skew-selection to maintain the availability of invalid tags across sets and prevent any SAE.

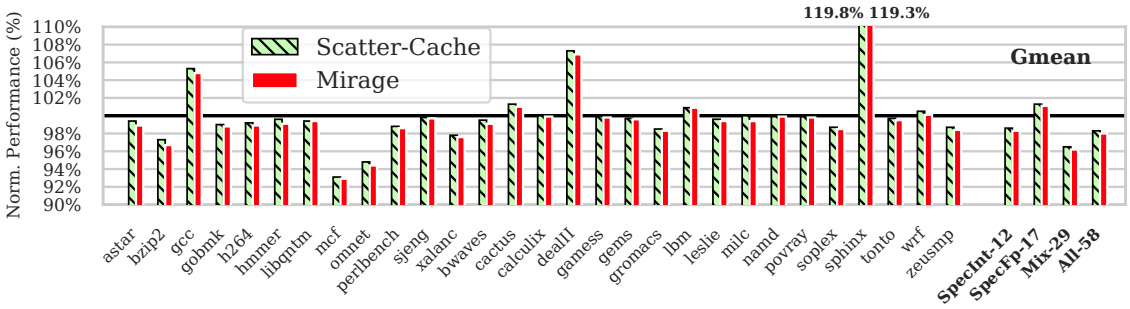


Figure 4.11: Performance of Mirage and Scatter-Cache normalized to Non-Secure Baseline (using weighted speedup metric). Over 58 workloads, Mirage has a slowdown of 2%, while Scatter-Cache has a slowdown of 1.7% compared to the Non-Secure LLC.

are SAE that leak information). We observe that randomization can increase or decrease conflict misses for different workloads: e.g., Mirage and Scatter-Cache increase misses by 7% for *mcf* and *xalanc* while reducing them by 30% for *sphinx* compared to baseline.

4.8.4 Impact on Performance

Figure 4.11 shows the relative performance for Mirage and Scatter-Cache normalized to the non-secure baseline (based on the *weighted speedup*[8] metric). On average, Mirage incurs a 2% slowdown due to two factors: increased LLC misses and a 4 cycle higher LLC access latency compared to the baseline. For workloads such as *mcf* or *omnet*, Mirage increases both the LLC misses and access latency compared to a non-secure LLC and hence causes a 6% to 7% slowdown. On the other hand, for workloads such as *sphinx*, *dealII* and *gcc*, Mirage reduces LLC-misses and improves performance by 5% to 19%. In comparison, Scatter-Cache has a lower slowdown of 1.7% on average despite having similar cache misses, as it requires 1 cycle less than Mirage for cache accesses (while both incur the cipher latency for set-index calculation, Mirage requires an extra cycle for additional tag lookups and indirection).

[8] $Weighted\text{-}Speedup = \sum_{i=0}^{N-1} IPC\text{-}MC_i / IPC\text{-}SC_i$ is a popular throughput metric for fair evaluation of N-program workloads [103], where *IPC* stands for Instructions per Cycle, $IPC\text{-}MC_i$ is the IPC of a program-i in a multi-program setting, and $IPC\text{-}SC_i$ is the IPC of program-i running alone on the system. Using *Raw-IPC* as the throughput metric, the slowdown decreases by 0.2%.

4.8.5 Sensitivity to Cache Size

Figure 4.12 shows the performance of Mirage and Scatter-Cache for LLC sizes of 2MB to 64MB, each normalized to a non-secure design of the same size. As cache size increases, the slowdown for Mirage increases from 0.7% for a 2MB cache to 3.2% for a 64MB cache. This is because larger caches have a higher fraction of faster cache hits that causes the increase in access latency to have a higher performance impact. Similarly, the slowdown for Scatter-Cache increases from 0.5% to 2.8% and is always within 0.4% of Mirage.

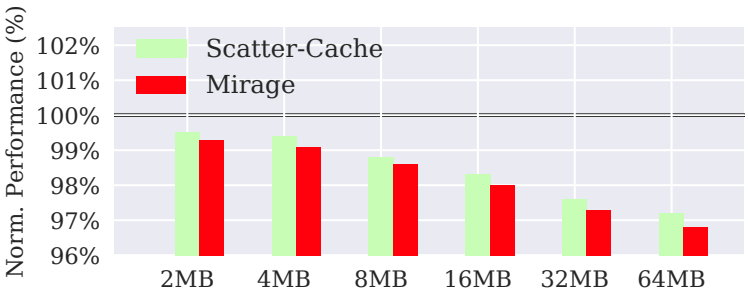


Figure 4.12: Sensitivity of Performance to Cache-Size.

4.8.6 Sensitivity to Cipher Latency

Figure 4.13 shows the performance of Mirage and Scatter-Cache normalized to a non-secure baseline LLC, as the latency of the cipher (used to compute the randomized hash of addresses) varies from 1 to 5 cycles. By default, Mirage and Scatter-Cache evaluations use a 3-cycle PRINCE-cipher [90] (as described in Section 4.8.2), resulting in slowdowns of 2% and 1.7% respectively. Alternatively, a cipher like QARMA-64 [104] (that was used in Scatter-Cache and assumed to have 5 cycle latency [11]) can also be used in Mirage; this causes Mirage and Scatter-Cache to have higher slowdowns of 2.4% and 2.2%. Similarly, future works may design faster randomizing functions for set-index calculations in randomized caches; a 1-cycle latency randomizing function can reduce slowdown of Mirage and Scatter-Cache to 1.5% and 1.2% respectively. Future works can study faster randomizing hashes that are also sufficiently robust (e.g., not susceptible to *shortcut attacks* [12]).

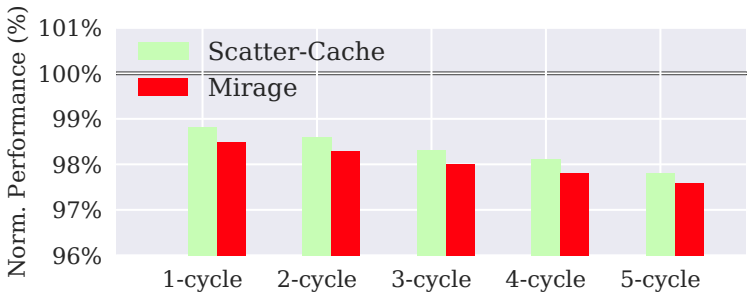


Figure 4.13: Sensitivity of Performance to Cipher Latency.

4.9 Cost Analysis

For analyzing the storage and power overheads of Mirage, we distinguish the two versions of our design as, *Mirage* (default design with 75% extra tags) and *Mirage-Lite* (with 50% extra tags and relocation).

4.9.1 Storage Overheads

The storage overheads in Mirage are due to (1) extra tag entries, and (2) FPTR and RPTR, the pointers between tag/data entries, and (3) tag-bits storing full 40-bit line address (for 46-bit physical address space) to enable address generation for write-backs. This causes a storage overhead of 20% for Mirage and 17% for Mirage-Lite compared to the non-secure baseline, as shown in Table 4.8. These overheads are dependent on cache line size as the relative size of tag-store compared to the data-store reduces at a larger line size. While we use a 64B line size, a 128B line size like IBM's Power9 CPUs [105] would reduce these overheads to 9-10% and a 256B line size would reduce these to 4-5%.

The storage overhead in Mirage is the main driver behind the area overhead, as the extra storage requires millions of gates, whereas all other extra logic for FPTR/RPTR-indirection, PRINCE cipher, etc., can be implemented in a few thousand gates (as shown in Section 4.9.3). Using CACTI-6.0 [100], we estimate that an LLC requiring 20% extra storage consumes 22% extra area. In terms of a storage-neutral comparison, Mirage has an average slowdown of <3.5% compared to a non-secure LLC with 20% more capacity.

Table 4.8: Storage Overheads in Mirage for 64B line size

Cache Size 16MB (16,384 Sets)		**Baseline** Set Associative	**Mirage** 2 skews x 14 ways/skew	**Mirage-Lite** 2 skews x 12 ways/skew
Tag Entry	Tag-Bits	26	40	40
	Status(V,D)	2	2	2
	FPTR	–	18	18
	SDID	–	8	8
	Bits/Entry	28	68	68
	Tag Entries	262,144	458,752	393,216
Tag-Store Size		**896 KB**	**3808 KB**	**3264 KB**
Data Entry	Data-Bits	512	512	512
	RPTR	–	19	19
	Bits/Entry	512	531	531
	Data Entries	262,144	262,144	262,144
Data-Store Size		**16,384 KB**	**16,992 KB**	**16,992 KB**
Total Storage		**17,280 KB (100%)**	**20,800 KB (120%)**	**20,256 KB (117%)**

4.9.2 Power Consumption Overheads

The larger tag-store in Mirage has a higher static leakage power when idle and also consumes more energy per read/write access. Table 4.9 shows the static and dynamic power consumption for Mirage in 32nm technology estimated using CACTI-6.0 [100], which reports the energy/access and static leakage power consumption for different cache organizations. We observe the LLC power is largely dominated by the static leakage power compared to dynamic power (in line with prior CPU power modeling studies [106]). The static power in Mirage (reported by CACTI) increases by 3.5-4.1W (18%-21%) in proportion to the storage overheads, whereas the dynamic power, calculated by multiplying the energy/access (from CACTI) by the total LLC-accesses per second (from our simulations), shows an insignificant increase of 0.02W on average. The increase in LLC power consumption of 4W (21%) in Mirage is quite small compared to the overall chip power budget, with comparable modern 8-core Intel/AMD CPUs having power budgets of 95-140W [107].

Table 4.9: Energy and Power Consumption for Mirage

Design	Energy / Access (nJ)	Dynamic Power (W)	Static Leakage Power (W)	Total Power (W)
Baseline	0.61	0.06	19.2	19.3
Mirage	0.78	0.08	23.3	23.4
Mirage-Lite	0.73	0.08	22.7	22.8

4.9.3 Logic Overheads

Mirage requires extra logic for the set-index computation using the randomizing hash-function and FPTR-indirection on cache-lookups, and for load-aware skew-selection and RPTR-indirection based tag-invalidation on a cache-miss. Our synthesis results in 15nm technology show that the PRINCE-based randomizing hash-function occupies 5460 um^2 area or 27766 Gate-Equivalents (GE - number of equivalent 2-input NAND gates) and the FPTR-indirection based lookup circuit requires 132 um^2 area or 670 GE. The load-aware skew-selection circuit (counting 1s among valid bits of 14 tags from the indexed set in each skew, followed by a 4-bit comparison) requires 60 um^2 or 307 GE, while the RPTR-lookup circuit complexity is similar to the FPTR-lookup. Overall, all of the extra logic (including the extra control state-machine) can fit in less than 35,000 GE, occupying a negligible area compared to the several million gates required for the LLC.

4.10 Related Work

Cache design for reducing conflicts (for performance or security) has been an active area of research. In this section, we compare and contrast Mirage with closely related proposals.

4.10.1 Secure Caches with High Associativity

The concept of cache location randomization for guarding against cache attacks was pioneered almost a decade ago, with **RPCache** [83] and **NewCache** [84], for protecting L1 caches. Conceptually, such designs have an indirection table that is consulted on each

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

Phantom-Cache [25] installs an incoming line in 1 of 8 randomly chosen sets in the cache, each with 16-ways, conceptually increasing the associativity to 128. However, this design requires accessing 128 locations on each cache access to check if an address is in the cache or not, resulting in a high power overhead of 67% [25]. Moreover, this design is potentially vulnerable to future eviction set discovery algorithms as it selects a victim line from only a subset of the cache lines. In comparison, Mirage provides the security of a fully-associative cache where any eviction-set discovery is futile, with practical overheads.

HybCache [82] is a recent design providing fully-associative mapping for a subset of the cache (1–3 ways), to make a subset of the processes that map their data to this cache region immune to eviction-set discovery. However, the authors state that "applying such a design to an LLC or a large cache in general is expensive" [82]. For example, implementing a fully-associative mapping in 1 way of the LLC would require parallel access to >2000 locations per cache lookup that would considerably increase the cache power and access latency). In contrast, Mirage provides security of a fully-associative design for the LLC with practical overheads, while accessing only 24–28 locations per lookup.

4.10.2 Cache Associativity for Performance

V-Way Cache [85], which is the inspiration for our design, also uses pointer-based indirection and extra tags to reduce set-conflicts – but it does not eliminate them. V-Way Cache uses a set-associative tag-store, which means it is still vulnerable to set-conflict based attacks, identical to a traditional set-associative cache. Mirage builds on this design and incorporates skewed associativity and load-balancing skew-selection to ensure set-conflicts do not occur in system lifetime.

Z-Cache [108] increases associativity by generating a larger pool of replacement candidates using a tag-store walk and performing a sequence of line relocations to evict the best victim. However, this design still selects replacement candidates from a small number of resident lines (up to 64), limited by the number of relocations it can perform at a time. As a result, a few lines can still form an eviction set which can be learned by attacks. Whereas, Mirage selects victims globally from all lines in the cache, eliminating eviction-sets.

Indirect Index Cache [109] is a fully-associative design that uses indirection to decouple the tag-store from data blocks and has a tag-store designed as a hash-table with chaining to avoid tag-conflicts. However, such a design introduces variable latency for cache hits and hence is not secure. While Mirage also uses indirection, it leverages extra tags and power of 2 choices based load-balancing, to provide security by eliminating tag-conflicts and retaining constant hit latency.

Cuckoo Directory [110] enables high associativity for cache directories by provisioning extra entries similar to our work and using cuckoo-hashing to reduce set-conflicts. **SecDir** [111] also applies cuckoo-hashing to protect directories from conflict-based attacks [112]. However, cuckoo-hashing alone is insufficient for conflict elimination. Such designs impose a limit on the maximum number of cuckoo relocations they attempt (e.g. 32), beyond which they still incur an SAE. In comparison, load-balancing skew selection, the primary mechanism for conflict elimination in Mirage, is more robust at eliminating conflicts as it can ensure no SAE is likely to occur in system lifetime with 75% extra tags.

4.10.3 Isolation-based Defenses for Set-Conflicts

Isolation-based defenses isolate the victim lines in the cache and prevent conflicts with the attacker lines. Prior approaches have partitioned the cache by sets [14, 15] or ways [83, 20, 16, 113] to isolate security-critical processes from potential adversaries. However, such approaches result in sub-optimal usage of cache space and are unlikely to scale as the number of cores on a system grows (for example, 16-way cache for a 64-core system).

Other mechanisms explicitly lock security-critical lines in the cache [83, 22], or leverage hardware transactional memory [114] or replacement policy [7] to preserve security-critical lines in the cache. However, such approaches require the classification of security-critical processes to be performed by the user or by the Operating-System. In contrast to all these approaches, Mirage provides robust and low-overhead security through randomization and global evictions, without relying on partitioning or OS-intervention.

4.11 Key Contributions and Impact of this Research

4.11.1 Key Contributions

To summarize, this work makes the following contributions:

1. We observe that conflict-based attacks can be eliminated by designing caches with global evictions, that consider all lines in the cache for eviction.

2. We propose *Mirage*, which provides global evictions of a fully associative cache with practical set-associative lookup. Mirage uses tag-to-data indirection and a load-balancing tag-store to ensure global evictions for the system lifetime and set-associative evictions occur once in 10^{17} years with 75% extra tags.

3. We propose *Mirage with Cuckoo Relocation*, where set-associative evictions in the tag store are avoided by relocating entries to alternative locations. This ensures set-associative evictions occur once in 22,000 years while reducing extra tags to 50%.

4. Mirage incurs a modest slowdown of 2%, and storage overhead of 17% to 20% for the extra tags and indirection, compared to a non-secure set-associative design.

5. Mirage thus provides a principled defense with the guarantee that the evicted lines are independent of the addresses installed. By eliminating set-conflicts, the root cause of cache attacks, it eliminates current and future conflict-based attacks.

6. Mirage's implementation is open-sourced at https://github.com/gururaj-s/mirage.

4.11.2 Potential Impact of this Research

Ending the Arms Race in Randomized Caches. There has been tremendous interest in randomized LLCs (more than 6 defenses from 2018 - 2021 across top security and architecture conferences). However, all prior defenses have been shown vulnerable via adaptive attacks. This is because prior defenses merely *obfuscate* set-conflicts and do not *eliminate* them. Moreover, each successive defense incrementally increases the level of obfuscation: so more intelligent attacks can break them. Because Mirage makes evicted lines independent of installed addresses, it fundamentally *eliminates* set-conflict related information leakage. This promises an end to the arms race in randomized cache defenses. While 2018-19 had 2 defenses getting broken by 3 attacks, and 2019-20 had 3 defenses broken by 3 attacks, since its public release in September-2020, Mirage has remained unbroken.

Enable Confidential Computing in Future CPUs. Cloud service providers seek to provide confidential computing services to their users. While resource allocations in the cloud typically occur per core, the last-level caches in cloud systems are typically shared across users. This can leak addresses accessed by one user to another causing privacy breaches in the cloud. Mirage re-designs the last-level cache bottom-up with global evictions to eliminate set-conflict and shared-memory based cache attacks across cores. Moreover, it requires negligible software support and has modest performance and storage overhead. Thus, Mirage is well-suited for practical adoption in future server CPU designs.

CHAPTER 5

BESPOKE CACHE ENCLAVES FOR SCALABLE CACHE PARTITIONING

This chapter of the thesis focuses on holistic defenses against cache side-channel attacks. Unfortunately, defenses like cache randomization developed earlier in this thesis only prevent a subset of the cache attacks, as they continue to allow contention of cache space leaking coarse-grained information. Hence, this chapter develops cache partitioning based defenses that can mitigate all cache side-channel attacks affecting the cache state.

5.1 Introduction

Cache attacks can be classified as *conflict-based attacks* (*e.g.,* Prime+ Probe [38]), *shared-memory-based attacks* (*e.g.,* Flush+ Reload [8]), and *cache-occupancy based attacks* [3, 47]. Randomized LLCs [9, 10, 11, 31, 30] defend against conflict-based attacks and even shared-memory attacks [11, 31], but they continue to be vulnerable to cache-occupancy attacks. This is because occupancy-based attacks leak information based on changes to the space used by the victim in the shared LLC and randomized caches allow contention for cache space. These attacks are powerful enough to cause privacy breaches like fingerprinting a user's browsing activity. Hence, we seek a principled solution that defeats all known cache side channels which exploit changes to the cache state.

Cache partitioning [14, 15, 16, 17, 18, 19, 20, 82] provides a principled defense against all three classes of cache attacks, by fully isolating the cache usage of victim and spy programs. These defenses achieve isolation by allocating non-overlapping cache regions to distrusting domains (cores, VMs, processes, or enclaves in a process). Without loss of generality, we assume domains to be at the granularity of processes in this work.

Design Goals. An ideal LLC partitioning defense should have the following properties:

1. *Security:* The scheme should prevent the spy from monitoring hits and misses of the

victim by disallowing any accesses outside its allocated cache space and also prevent inferences from shared state like replacement policy.

2. *Scalability:* The scheme should support hundreds of isolated LLC partitions. This is because the LLC could be shared between 64 – 128 cores on server systems; even on client systems, applications like Chrome Browser seeking isolation between web pages can spawn 50 - 100 processes, as characterized by Chrome Site Isolation[48].
3. *Fine-Grained Allocations and Flexibility:* The scheme should provide fine-grained allocations as sensitive programs like encryption algorithms can have small cache working sets (*e.g.,* AES T-Tables are 8KB). LLC allocations should also be independent of memory allocations and flexibly manageable, as the need for memory (footprint) and cache capacity (locality) may not be correlated.

Unfortunately, no existing cache partitioning defense satisfies all three properties. Way-partitioning solutions [16, 20, 18, 17] shown in Figure 5.1 (a) suffer from limited scalability (only supporting as many allocations as ways), while page-coloring based solutions [15, 14], shown in Figure 5.1 (b), suffer from fixed size and inflexible memory allocations.

Towards Practical Cache Partitioning Defenses. To develop a scalable and flexible solution, this thesis explores a secure cache isolation framework, Bespoke Cache Enclaves (BCE)[1], which allows large number of fine-grained cache allocations and flexible memory usage. BCE achieves this using a dynamic cache indexing function, which ensures cache lines of a domain are only directed to sets in the allocated clusters of a domain, as shown in Figure 5.1(c). Before discussing BCE's design, we discuss the background of cache partitioning and challenges with prior solutions to motivate our design.

[1]This chapter was published as a paper, "*Bespoke Cache Enclaves: Fine-Grained and Scalable Isolation from Cache Side-Channels via Flexible Set-Partitioning*", appearing in the proceedings of SEED 2021 [33]

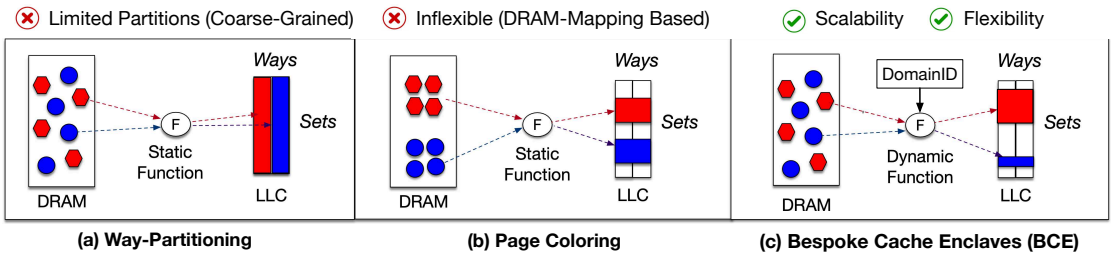


Figure 5.1: (a) Prior way-partitioning solutions provide few partitions, restricted by the LLC associativity (b) Prior page-coloring solutions have finer allocations but do not allow flexible use of DRAM and LLC in different ratios. (c) Our solution BCE seeks to allow a large number of fine-grained cache allocations and flexible memory usage, with dynamic indexing to guide lines to allocated cache regions.

5.2 Background on Cache Partitioning

We first describe the threat model for the adversary for our defense, and then discuss prior cache partitioning solutions and their limitations in depth.

5.2.1 Threat Model

Our threat model, shown in Figure 5.2, assumes a victim and spy running on different cores as resource-allocation often occurs at the granularity of a physical core in virtualized or cloud settings. Here, L1 and L2 caches are typically core-private and sharing of caches occurs at the LLC (e.g. L3-cache); hence we focus on side-channel attacks via LLCs. In case the L1 and L2 caches are time-shared or spatially shared among a few distrusting hyper-threads, we assume they are partitioned along ways or flushed on context-switches.

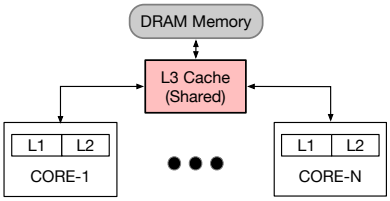


Figure 5.2: Threat model focuses on shared LLC attacks.

Similar to prior cache partitioning works [16, 20, 15], we limit our focus to stateful cache attacks, where the spy observes latency variation due to LLC state changes by the

victim (e.g., install, replacement-state update) which remain in the cache and leak information much after the victim execution completes. Like prior works, our threat model excludes attacks relying on bandwidth contention on NoCs [115, 116] or LLC ports as they are transient in nature, are quite susceptible to noise as they typically cause relatively small latency differences, and are thus less concerning.

Our goal is to address the threat from all three classes of stateful cache side-channel attacks: conflict-based, shared-memory based, and cache-occupancy based attacks.

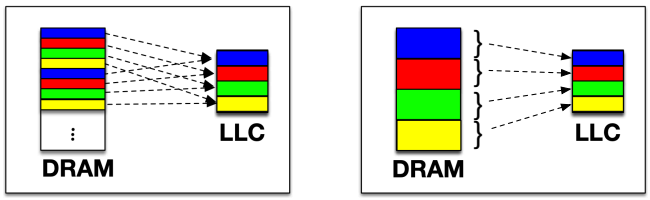
5.2.2 Prior Cache-Partitioning Based Defenses

Partitioning the LLC across distrusting applications can prevent all three classes of cache attacks, as it provides a notion of strong isolation, i.e. the cache state observable by a program is solely a result of its own execution and is unaffected by any other program's cache accesses. Current LLC-partitioning schemes broadly fall under two categories.

Way-Partitioning

Such solutions partition the LLC based on ways, allocating one or more non-overlapping ways to processes in different trust-domains, as shown in Figure 5.1 (a). The state-of-the-art way-partitioning solution, DAWG [16], stores a software-configurable bit-mask for each domain in the cache controller, which is used to determine the ways to be checked for cache-hit determination on LLC-accesses and eviction-decisions on LLC-misses. This allows DAWG to provide isolation across cache partitions of different trust-domains and prevent cross-domain hits or evictions or replacement policy updates. Along with OS support to ensure processes across different trust-domains only share read-only lines which are duplicated in each cache partition, this design prevents any sharing of cache space or lines or sets between domains.

Pitfall: Unfortunately, the number of simultaneous trust-domains in such a design is limited to the LLC associativity (number of ways). With associativity being limited to 16



(a) Round-Robin Colors (b) MI6

Figure 5.3: Page-coloring based cache partitioning. (a) Typical schemes assign consecutive colors to consecutive physical pages. (b) MI6 [14] partitions the DRAM and cache sets into 64 contiguous regions and assigns a DRAM and cache region a single color.

– 32 for LLCs as the number of cores continues to increase, way-partitioning does not scale to a large number of trust-domains. Moreover, the partitions available are coarse-grained (2MB for a 32MB 16-way LLC), which results in inefficient cache allocations for applications with small cache working sets (*e.g.* AES encryption). Ideally, LLC allocations should scale to a large number of domains and be fine-grained in size.

Page Coloring

Page Coloring [15, 21] defenses, shown in Figure 5.1 (b), partition the LLC along sets by dividing memory pages into colors based on the cache sets they map to and allocating pages of different colors to distrusting processes. Typically, such schemes assign distinct colors to successive 4KB pages, as shown in Figure 5.3(a), and the OS allocates pages of as many colors to a process as the amount of cache space desired. This supports a large number of colors (a 4KB page of one color maps to 64 LLC sets) and up to 512 trust-domains with a 32MB, 16-way LLC. However, such solutions are not compatible with large pages (*e.g.,* a 2MB page covers the entire 32MB 16-way LLC with a single color), which can lead to significant TLB pressure.

MI6 [14] makes page-coloring friendly for large pages. It statically partitions the DRAM and LLC space into 64 contiguous regions and modifies the cache indexing to have a static 1-to-1 mapping of each DRAM and LLC region, as shown in Figure 5.3(b). As each LLC and DRAM region have one color, MI6 supports up to 64 colors and 64 trust domains. While domains may be assigned more than one color, to effectively utilize multiple cache

regions, a domain requires its working-set to be spread by the OS uniformly across all allotted DRAM regions. However, MI6 observes that OS typically allocates pages contiguously causing clustering of hot data in one region and inefficient LLC utilization.

Pitfall: All page-coloring solutions cannot manage DRAM and cache space independently as they require allocations of both in the same ratio (*e.g.,* to claim half the cache, a domain must be given half the DRAM space). This results in inefficient use of cache space or memory space: an application with a large memory footprint but poor cache locality (*e.g.,* graph applications) needs significant DRAM space but not much cache space; meanwhile, a data-intensive kernel with high locality but small memory footprint (e.g. encryption or small matrix-multiply) may not need much DRAM space but benefits from larger cache space. Ideally, cache partitioning should be independent of the memory allocations.

5.2.3 Goal: Scalable & Flexible LLC Isolation

The goal of this work is to develop a secure cache partitioning substrate for LLCs that scalably supports a large number of isolated cache partitions (without being restricted by the associativity of the cache) and performs fine-grained cache allocations. At the same time, it should be flexible in making cache allocations to a domain without placing any restrictions on the memory capacity allocation, virtual-to-physical mapping, and page sizes. To that end, we propose *Bespoke Cache Enclaves (BCE)*, a design satisfying all these goals, as shown in Figure 5.1 (c).

5.3 Design of Bespoke Cache Enclaves

To develop a scalable and flexible LLC partitioning scheme, we focus on the cache indexing function which governs the mapping of addresses to cache sets. Prior cache partitioning defenses, using way-partitioning or page-coloring (including MI6), have fixed indexing functions configured statically at design time. The key insight of our work, *Bespoke Cache Enclave (BCE)*, is that the cache indexing can be made dynamic to guide addresses of a

domain to only the isolated cache region (or cache "enclave") of the domain. We first provide an overview of BCE and then the design of its components.

5.3.1 Overview of BCE

BCE is a set-partitioning scheme that allocates cache space at the granularity of *clusters* (group of contiguous sets) and provides each security domain with an isolated cache partition consisting of one or more clusters. Without loss of generality, we use a cluster size of 64 sets (64KB) in our design to match the granularity of page-coloring (although our design can support clusters as small as a single set). Thus, our 32MB 16-way LLC is divided into 512 contiguous clusters (also referred to as *physical* clusters) and each of these is identified by a unique *Physical Cluster ID (PCID)*.

BCE allows each domain to be allocated a configurable number of clusters (also referred to as *logical* clusters), and these clusters could be located at any PCID in the LLC. For example, as shown in Figure 5.4, Domain-A (*e.g.,* a cache-sensitive workload like AES) could be allotted many LLC clusters and Domain-B (*e.g.*, a cache-insensitive graph application) could have only one LLC cluster, and these could be located at arbitrary physical locations in the LLC. Additionally, BCE allows independent LLC and memory allocations. For example, Domain-A could have a small memory footprint whereas Domain-B could have a large footprint, independent of their cache allocations. BCE allows flexibility in LLC management, supporting few domains with hundreds of clusters per domain or even hundreds of domains with few clusters per domain. The indexing hardware of BCE maps a given line address and Domain-ID into a cache set in one of the clusters of the domain.

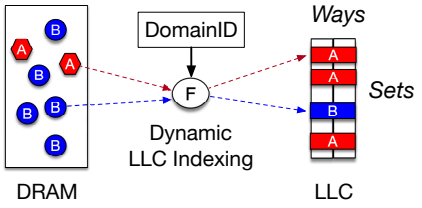


Figure 5.4: Overview of BCE Capabilities. BCE allows a configurable number of LLC clusters to be allocated to each domain, independent of memory allocations.

Challenges. As the number of clusters given to a domain can vary (from 1 to 512) and these clusters could be located anywhere in the cache, BCE faces a few challenges:

1. *Non-Contiguous Clusters:* First, how to store the mappings of where the clusters of each domain are physically located? Over time, the available clusters could be fragmented all over the LLC and the clusters allocated to a new domain could be non-contiguous and at arbitrary LLC locations. The set-indexing needs intelligent indirection to map lines to arbitrary LLC locations.
2. *Non-Power-of-2 Clusters Allocation:* Second, how to uniformly map addresses among the clusters of a domain to minimize set-conflicts? With different numbers of clusters per domain, some domains might be allocated a non-power-of-2 number of clusters to avoid leaving the clusters unallocated. The set-indexing needs intelligence to distribute lines uniformly among even non-power-of-2 clusters to minimize conflict misses.
3. *Constant Time Indexing:* Lastly, the security of isolated cache regions must be guaranteed by the indexing logic in hardware. As the set-indexing itself logic is shared by all domains, it needs to be constant time to prevent any new timing side-channels.

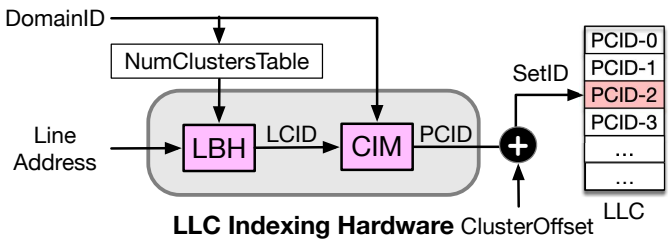


Figure 5.5: Overview of BCE Cache Indexing. The Load-Balancing Hash (LBH) uniformly hashes addresses among Logical Clusters of a Domain (LCID) and the Cluster-Indirection Module (CIM) maps LCIDs to Physical Cluster IDs (PCIDs) of the LLC. PCID and the cluster-offset together form the set index.

Solutions. BCE's cache indexing hardware addresses these challenges with two components shown in Figure 5.5. First, a *Cluster-Indirection Module (CIM)* to store and lookup the physical locations of the clusters of a domain (*i.e.,* logical clusters identified by LCID). Second, to determine which LCID an address maps to, a *Load-Balancing Hash (LBH)* is

used, which maps addresses of a domain uniformly across all its clusters, regardless of the number of clusters. On an LLC access, the LBH maps a line address and DomainID to an LCID and the CIM lookup using this LCID provides the physical LLC location of the cluster (PCID), where the set corresponding to the line address is located. The set within the PCID is identified by the *cluster-offset bits* (the 6 least-significant bits of the line address) and indexed by concatenating the PCID and cluster-offset bits. Note that BCE guarantees the isolation properties for security in hardware and is carefully designed to provide constant-time accesses to avoid new side channels, while software is only responsible for deciding the number of clusters for each domain. Next, we describe these components.

5.3.2 Cluster-Indirection Module: Maps Clusters

The Cluster-Indirection Module (CIM) is responsible for locating the LLC clusters allocated to a domain. To access a cache set, line addresses are hashed to logical clusters private to a domain (by the LBH), which may be located anywhere in the LLC. To locate these clusters, each logical cluster of a domain, *i.e. Logical Cluster ID (LCID)*, must be translated to its allocated LLC cluster, *i.e. Physical Cluster ID (PCID)*.

Design Constraints. The design of the CIM that can translate domain LCIDs to LLC PCIDs has two key requirements: *flexibility* and *fast lookups*. With our LLC divided into 512 clusters, there can be up to 512 domains each with one cluster or one domain with all 512 clusters. The CIM data structures storing the cluster mappings need to be flexible enough to store these mappings for allocations of up to 512 clusters, for up to 512 domains. Additionally, the CIM needs to support fast lookup as it is in the critical path of LLC accesses. One option for storing cluster-mappings is a hash-table. However, such a design is prone to hash-conflicts and variable access latency, which is a security vulnerability. As the CIM is shared by all domains, including mutually distrusting ones, the CIM itself can become the target of conflict-based attacks, where a spy in one domain observes variation in latency to access cluster mappings due to changes in mappings a victim in another domain.

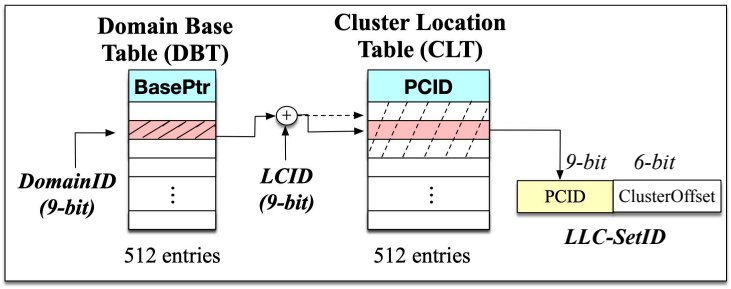


Figure 5.6: Design of the CIM. The DBT provides the base entry location (LCID-0) of a domain in the CLT. The base location added to the LCID, points to the CLT-entry containing the required PCID.

To prevent such new side-channels, we need to ensure the CIM has constant access latency. To that end, we design the CIM with two tables and indirection.

Design. Figure 5.6 shows the two-level CIM design. The *Cluster Location Table (CLT)* keeps an ordered list of valid LCID to PCID mappings for each domain with at least 1 valid cluster. The mappings for each domain are kept contiguous in the CLT, starting with the base entry (LCID-0) for each domain. The *Domain Base Table (DBT)* tracks the location of the base entry (LCID-0) in the CLT for each domain. To translate an LCID to PCID, the DBT is accessed with the DomainID to obtain a Pointer to the Base Entry (BasePtr) of that domain in the CLT. Adding the LCID to the BasePtr provides the CLT location for the LCID of that domain. Accessing this CLT location provides the required PCID, which is concatenated with the cluster-offset bits to get the set-index.

Isolation Guarantees. The CIM hardware, by design, provides the security guarantee of cache isolation, as only the LLC clusters allocated to a domain (those addressable by the CLT entries of the domain) are accessible to cache lookups from that domain. Only the CLT entries of a particular domain are accessible on cache lookups from that domain, as the LCID used to calculate the CLT index is always less than the number of clusters allocated to that domain (guaranteed by the LBH and the NumClustersTable in Figure 5.5). A more end-to-end security analysis is provided in Section 5.4.

Storage, Logic, and Latency. To support up to 512 PCIDs, the CLT needs at most 512 valid entries. Similarly, the DBT requires at most 512 entries for 512 domains. As both tables have 9-bit entries and a 1-bit valid-bit per entry, the tables require a total of 640 bytes of storage. Such small tables can be implemented as registers for fast lookup. Thus, the translation of LCID to PCID via CIM requires two register lookups and one combinational 9-bit adder, taking 2 CPU cycles.

5.3.3 Load-Balancing Hash: Maps Lines

While CIM ensures secure isolation between clusters of different domains, the performance of BCE relies on how uniformly addresses map to clusters of a domain (logical clusters) to minimize set-conflicts. The Load-Balancing Hash (LBH) is responsible for uniformly mapping line addresses to logical clusters of a domain (LCIDs).

The Problem of Load Balancing

In conventional LLCs, a modulo function is used as a uniform set-indexing function (`SetID = LineAddr % NumSets`), that is equivalent to selecting least significant bits of the line address for power-of-2 NumSets. However, in BCE, each domain could have a configurable number of clusters (from 1 to 512). These are not constrained to be a power-of-2, as that can result in under or over-provisioning of cache space for larger cache allocations. Ideally, we would like LBH to have the uniformity of the modulo function shown in Figure 5.7(a); but computing a modulo for non-power-of-2 divisors is expensive (requires tens of cycles)[2] for cache-indexing.

Imbalance with Simple Hashes. Figure 5.7(b) shows a simpler, single-cycle hash called *Linear-and-Invert*, which maps n-bits of a line address to an n-bit LCID (where n is log of NumClusters, rounded up to the next integer). If the n-bit Line Address value,

[2]Hardware implementation of modulo with non-powers-of-2 divisors requires recursive division with $O(b)$ pipeline stages, [117] where b is the number of bits of dividend (line address). For our design, this requires more than 10 stages and latency of tens of cycles.

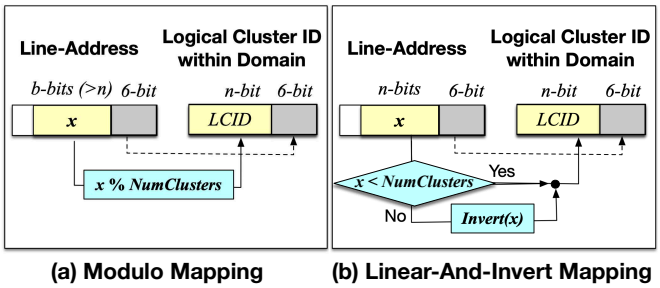


Figure 5.7: Candidates for LBH (mapping addresses to LCID): (a) Modulo mapping is uniform but requires multi-cycle hardware implementation. (b) Linear-And-Invert mapping is single-cycle but results in imbalance.

say x, is less than the NumClusters, x is used as the LCID, else x is inverted and used as the LCID. Thus, either x or Invert(x) is guaranteed to be less than NumClusters and can be used as the LCID. While this simple hash is low latency, it has significant non-uniformity. For example, a partition with 3 clusters, using two bits of line address as input to the hash, has one cluster receiving 50% of the lines and the other two clusters with 25% lines each.

We quantify this imbalance with a metric called *Load Imbalance*, *i.e.* the ratio of maximum to average lines per cluster. Figure 5.8 shows the load imbalance for the Modulo and Linear-And-Invert hashes while streaming a large number of lines, as clusters per domain vary from 1 to 512. Modulo mapping is close to the ideal value of 100%, whereas linear-and-invert has imbalance of 200%, *i.e.*, some clusters have 2x the average accesses.

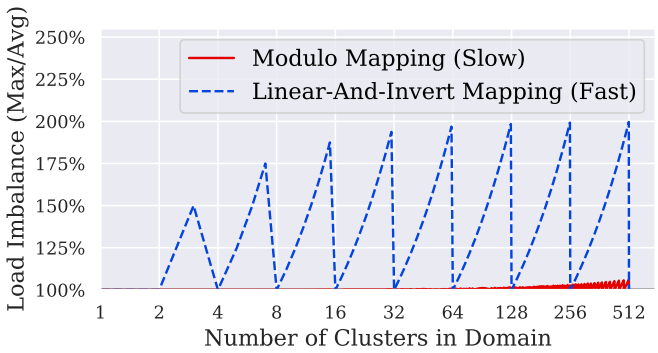


Figure 5.8: Load imbalance with Modulo and Linear-And-Invert mappings as the number of clusters in a domain varies from 1 to 512. Modulo is close to ideal but slow. Linear-And-Invert is fast but has up to 2x imbalance.

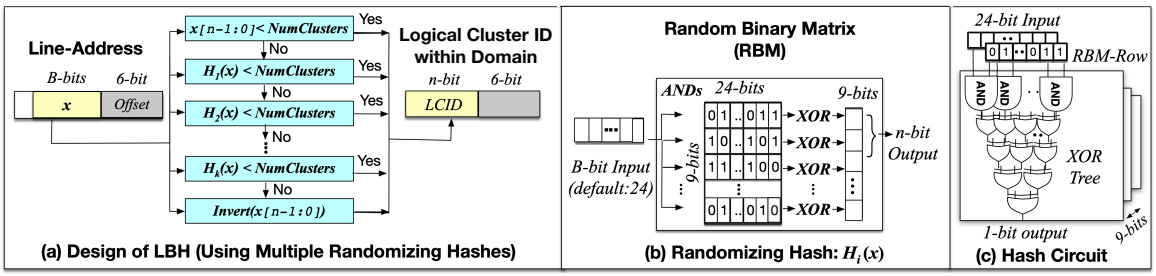


Figure 5.9: Design of Load-Balancing Hash (LBH). (a) Mapping of line addresses to LCID via multiple randomizing hashes (b) Implementation of randomizing hash $H_i(x)$ using Random-Binary-Matrix (RBM) (c) Hash Circuit Logic: Circuit has a critical path of 1 AND and 4 XORs (three 2 input and one 3-input in XOR Tree).

Solution: LBH with Randomizing Hashes

Design. To achieve near-ideal load imbalance with a single-cycle LBH, we construct it using multiple randomizing hashes as shown in Figure 5.9(a). To map a line address to LCID, the bottom n bits of the line address (x) are checked to see if it is less than NumClusters (where n is log of NumClusters rounded to next integer) and can be directly used as LCID (like linear mapping). If not, the line address is transformed using a low-latency randomizing hash, $H_1(x)$, and the bottom n bits are checked again to see if they can be used as LCID. This process repeats k times with k randomizing hashes; if LCID is still not obtained, then the inverted value ($\sim x[n-1:0]$) is used as the LCID. As each hash is expected to produce the LCID with a probability greater than half, the likelihood of using the inverted mapping after 3 - 5 hashes is quite small and the overall mapping is balanced.

Implementation. To ensure $H_1(x)$ to $H_k(x)$ are sufficiently uniform and independent, while also having a fast hardware implementation, we construct the hash-functions using Random Binary Matrices (RBM). As shown in Figure 5.9(b), each $H_i(x)$ consists of $B \times n$ bits sized matrix (default: 24×9) with bit values statically populated from a uniform random distribution. Each bit of the output hash is computed as bit-wise AND of the input-vector and the corresponding row of the RBM, followed by an XOR reduction. Successive hashes (H_1 to H_k) are generated with different matrix bit values, all sampled from the same uniform random distribution. Such a construction of $H_1(x)$ to $H_k(x)$ has the property

that any two functions selected at random only have a small probability of generating the same hash for the same input (such RBMs have been shown to create Universal Classes of Hashing Functions with this property [118]), and hence suited for our purpose.

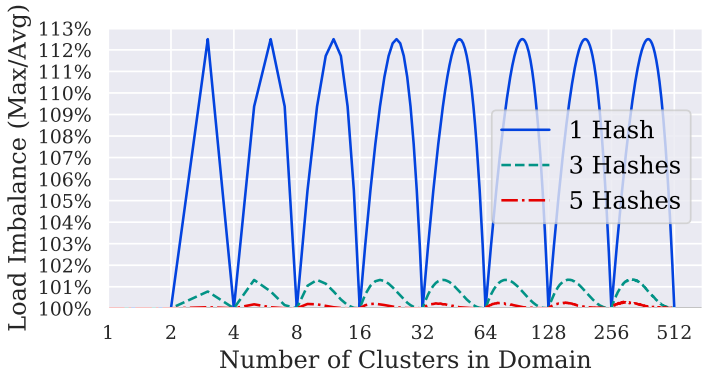


Figure 5.10: Load Imbalance with LBH using multiple randomizing hashes (24-bits of line address as input) as the number of clusters in a domain varies from 1 to 512. With 3–5 hash functions, imbalance is within 1% of ideal.

Results - Reduced Imbalance. Figure 5.10 shows the load imbalance with LBH using multiple randomizing hashes. We choose RBM dimensions of 24×9, to support LCID of up to 9-bits to support up to 512 NumClusters. Using 1 hash reduces the worst-case imbalance to 112.5%, 3 hashes reduce it to 101.3%, and 5 hashes to 100.3% (in comparison, using 0 hashes, which is equivalent to the linear-and-invert, results in 200% imbalance as shown in Figure 5.8). With 3 – 5 hashes, the worst-case load imbalance is very close to the ideal 100% and thus the mappings with LBH adds negligible conflict misses due to non-uniformity. Note that the randomizing LBH mapping is statically provisioned at design-time, deterministic, and only for performance: it can be disclosed to software if needed for software to customize data-structure layout for further minimizing set-conflicts.

Logic, Latency and Storage. The circuit for each hash, shown in Figure 5.9(c), has a critical path of 1 AND gate and 4-5 XOR gates (for XOR-reduction) from input to output. For LCID computation with LBH, the k hashes ($k = 3 \ldots 5$ is sufficient) can be computed in parallel, and hence the overall critical path has only one hash circuit (1 AND and 4 XOR gates) and $k + 1$ n-bit comparators to check if hash-values are less than NumClusters. This

computation incurs a latency of less than one CPU cycle (as cycle time in modern processors is typically designed for about 20 gate delays). The number of hash-functions and input-size (i.e. number of bits of the line address used as the input to the hashes) determine the storage overheads of LBH. Using more input-bits considerably reduces imbalance as LBH generates more uniform mappings over a larger input space; we find that using 5 hash functions of 24×9 bits is sufficient for minimizing the load imbalance and requires negligible storage of 135 bytes.

5.3.4 Software Interfaces to Request Clusters

Like prior cache partitioning defenses [16, 14], BCE provides software interfaces to request isolated LLC partitions of a given size from the hardware. BCE has two new privileged instructions, `BCE_alloc` and `BCE_dealloc` for the creation and deletion of LLC allocations for domains, usable by privileged software like OS responsible for resource allocation. The OS may provide system-calls that allow applications to request custom-sized LLC allocations (with appropriate fairness checks); this can spur research on real-world systems using customizable cache isolation (*e.g.,* extensions of Chrome Site Isolation [48] providing web pages bespoke cache isolation along with process isolation). We now describe the hardware operations performed on these instructions.

`BCE_alloc`: This instruction creates a new LLC partition for a trust domain. It takes two inputs: the DomainID and the number of clusters to be allocated to the domain. If the requested number of LLC clusters are available, then the instruction returns success, otherwise failure. If successful, the cache controller initializes a contiguous list of CLT entries (at the end of all valid CLT entries) with the PCIDs allocated for this domain. The DBT entry for the domain is initialized to point to the first CLT entry of the domain and the NumClustersTable is updated with the allocated clusters.

`BCE_dealloc`: This instruction deallocates an existing LLC partition. It takes a DomainID as an input, and invalidates the associated DBT and CLT entries. Subsequently,

the lines in the physical LLC clusters of the domain are flushed and PCIDs of that domain become free to be allocated to other domains. The invalidated CLT entries are also compacted and moved to the end of the CLT to ensure that a subsequent cache allocation can obtain a contiguous cluster of free CLT entries. Note that the CLT compaction takes tens of cycles and is much faster than the latency to flush contents of the freed cache clusters. Fortunately, deallocations are quite infrequent (only on domain termination, resizing, etc.).

Note that if a domain requires LLC space but the LLC is fully allocated, partitions of inactive domains (that are context-switched out) can be deallocated, flushed, and then re-allocated to a new domain (similar to prior works [16]).

5.3.5 Putting it Together: BCE Operation

Process-Start. When a new program is launched, it can be assigned to an existing domain or a new domain. For programs not desiring security, a single large non-secure LLC partition can be perpetually reserved. For a process desiring a new security domain, the OS uses `BCE_alloc` to allocate an isolated LLC partition and embeds the DomainID in the process context. Note that for the security of L1 and L2 caches, if they are private to a domain, they are flushed before a new domain begins; if they are shared among a few hyper-threads/cores, they must be way-partitioned.

LLC-Indexing. All read and write accesses to the LLC in BCE are accompanied with the DomainID from the thread-context. For an LLC-access, the LBH hashes the line address into the logical cluster ID (LCID) and then CIM translates the LCID into the physical cluster ID (PCID). The PCID in conjunction with the cluster offset bits determines the set index for the LLC access and the access then proceeds by checking this set for a tag-match like a conventional LLC.

LLC-Hit/Miss. On LLC-Hits, replacement state updates proceed unchanged using any intelligent replacement policy, as the entire set belongs to a single domain. On an LLC-miss, the eviction candidate selection is unchanged and is chosen from the entire set. On

dirty eviction, the line addresses for writebacks are generated by concatenating the tag with cluster offset of the set; tags are enlarged by 9 bits to support this.

Process-End. When a domain exits, the OS uses `BCE_dealloc` to reclaim the space.

5.3.6 Support Required from System Software

Similar to prior secure cache partitioning works [16, 14], BCE requires additional support from system software for ensuring correctness and security.

Page-Sharing: Processes that trust each other are allotted the same DomainID; they share cache-sets with each other like conventional caches and have read-write sharing of pages. Page sharing between distrusting processes in different domains has to be regulated by the OS like in [11], to ensure no read-write sharing between them and to prevent direct information leakage. For read-only shared pages (whose shared cache lines get duplicated in each domain), the OS is responsible for flushing such pages from each of the LLC domains when such pages are swapped out or unmapped (just as the OS updates the page-tables for each of the processes).

Inter-Process-Communication: Data transfer between processes in a single trust-domain with read-write shared memory is unchanged. Data transfers between processes in different domains have to be marshaled by the OS with special system calls to ensure no secret-dependent data leaks via such transfers. Also, kernel memory-copying functions performing data transfer from user-space processes to kernel, like `copy_from_user`, need to be modified to ensure the data is read from user's cache-partition and written to kernel's cache-partition (vice-versa for `copy_to_user`), like in [16].

Coherence: For cache coherence within processes of the same domain, coherence protocol is unchanged, except that all coherence packets include the Domain-ID to ensure correct LLC sets are accessed on snoop/invalidation requests.

5.4 Security Analysis

Assumptions. The security focus of BCE is on preventing cross-domain side-channels relying on LLC state changes, where a spy observes timing differences due to cache state modified by secret-dependent activity of the victim. These include conflict-based attacks [38, 40, 45, 44] exploiting cross-domain evictions from shared cache-sets, shared-memory-based attacks [8, 23, 94, 69, 24] exploiting cross-domain hits on shared cache lines, and cache-occupancy based attacks [3, 47] exploiting changes to LLC space used per domain. While the analysis uses the cross-process setting for BCE (distrusting processes mapped to different domains) for its arguments, it is applicable for LLC-isolation between VMs or even enclaves within a process, if those map to a security domain. Our focus is the shared LLC, as L1 and L2 caches are typically private. If L1 or L2 are shared, BCE assumes they are way-partitioned [16] across domains; this is practically feasible as typically L1 or L2 caches are shared by no more than 2–4 threads at a time.

Security Guarantees. BCE promises strict isolation between LLC partitions of different security domains. The invariant it guarantees is: *the state of the LLC-partition of one domain is solely determined by its own LLC accesses and not influenced by any other domain's LLC accesses.* BCE achieves this by guiding addresses of different domains to disjoint LLC sets via its flexible LLC-indexing, duplicating lines of read-only shared addresses across domains, and disallowing read-write sharing of addresses across domains. As a result, all LLC operations of a domain only affect the LLC-state of its own domain and cannot affect other domains:

- LLC read or write accesses only have hits on lines within the LLC sets allocated to the domain; cache-flushes only impact cache lines within a domain and do not affect any duplicate lines for the same address in other domains; coherence-related invalidations are only allowed within the same domain – this prevents any cross-domain shared-memory attacks.

- Replacement state updates and victim selections within a single set are only performed among the cache lines of a single domain – this prevents any cross-domain conflict-based attacks.

- LLC-partition sizes get allocated per domain by privileged software. The partition sizes are allocated at the domain creation time and are independent of any secret-dependent accesses of co-running domains – this prevents occupancy-based attacks.

The BCE substrate itself does not introduce any new side channels. While the metadata tables in BCE that maintain cluster mappings (CIM tables) are shared across domains, the set-index computation using these tables has constant latency irrespective of the number of domains in use or their allocation sizes. Additionally, malicious processes cannot escape the LLC-partition sandbox enforced by BCE, by accessing or modifying these mappings as they are maintained in micro-architectural structures and inaccessible to software. This ensures BCE is itself secure and free from any new attacks.

5.5 Evaluation Results

In this section, we evaluate the performance and costs of BCE and compare it against prior LLC-partitioning schemes DAWG [16] (that partitions along ways) and Page-Coloring [15] (that partitions along sets), and a non-secure baseline LLC without partitioning.

5.5.1 Methodology

The system modeled in our study is shown in Table 5.1. We use a 16-core system with 32MB 16-way shared L3 cache. For performance evaluations, we use a trace-driven simulator running program execution traces (of length 1 billion instructions) generated using Intel Pintool [99], drawn from a representative slice of a program using Simpoints [98]. We evaluate 56 workloads, including 22 SPEC-CPU2017 [119] and 6 GAP [120] (graph algorithms *bc*, *cc*, *pr* with *twitter* and *web* datasets) benchmarks (16-core rate-mode), and 28

mixed workloads (each has 16 randomly chosen SPEC or GAP benchmarks). Our baseline is a non-secure shared LLC. For BCE, we use a 3 cycle overhead for LLC lookup, based on the latency estimated in Section 5.3.2 Section 5.3.3.

Table 5.1: Baseline System Configuration

Processor and Private Caches	
Core	16-cores, In-order Execution, 3GHz
L1, L2-Cache / core	L1-32KB, L2-256KB, 8-way, 64B line size
Last-Level Cache and Main-Memory	
LLC (shared)	32MB, 16-way, 64B line size, Non-Inclusive SRRIP Repl [63], 24 cycle latency
DRAM	45 ns latency

5.5.2 Impact on Cache Misses

Table 5.2 shows the impact of partitioning on the LLC-misses per thousand instructions (MPKI) averaged across the different workload suites for BCE, Page-Coloring, and DAWG, compared to a non-secure LLC without partitioning. Each of the 16 benchmarks in a workload runs on a separate core and only the LLC is shared. With the partitioning schemes, each benchmark gets an equal-sized LLC partition. All partitioning schemes incur a higher number of misses than the non-secure baseline due to the capacity restrictions imposed.

Table 5.2: Average LLC MPKI for Non-Secure, DAWG, Page-Coloring, and BCE.

Workloads	**Non-Secure**	**DAWG**	**Page-Coloring**	**BCE**
Spec-22	8.42	8.73	8.44	8.44
Gap-6	41.64	42.61	41.63	41.63
Mix-28	27.56	29.82	28.53	28.59
All-56	**21.55**	**22.91**	**22.04**	**22.07**

On average, partitioning the LLC with BCE increases LLC MPKI by 2.4% similar to the increase with Page-Coloring, compared to non-secure LLC. This is because Page-Coloring and BCE allocate similar-sized set-partitions under an equi-partitioning policy.

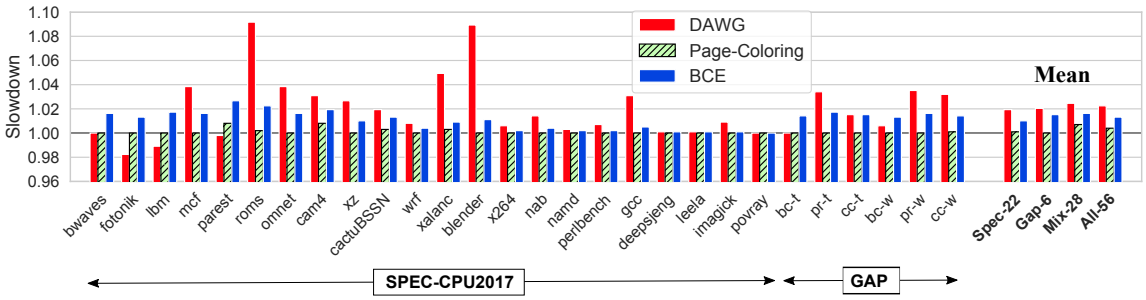


Figure 5.11: Slowdown of BCE compared with DAWG [16] and Page-Coloring [15]. All numbers are normalized to a Non-Secure baseline without partitioning. Across 56 workloads (sorted high to low by MPKI in SPEC and GAP suites), BCE has an average slowdown of 1.3%, while DAWG and Page-Coloring have slowdowns of 2.2% and 0.4%.

DAWG has 6% higher MPKI than the non-secure design and this increase is more than double that of BCE. This is due to increased conflict-misses, as with equal partitions, DAWG allocates each core with a 2MB direct-mapped partition. Among high MPKI workloads, *roms* is the worst affected with DAWG, incurring a 20% increase in LLC MPKI.

5.5.3 Impact on Performance

Figure 5.11 compares the slowdown of BCE, Page-Coloring and DAWG (normalized to non-secure LLC). Across 56 workloads, BCE has an average slowdown of 1.3%. The main drivers of the slowdown in BCE are the increase in both the LLC-misses and the LLC-access latency. The LLC-misses increase by less than 3% due to the cache space restrictions with partitioning, while the cache latency increases by 3-cycles. In comparison, Page-Coloring incurs a slowdown of 0.4%, as it incurs similar misses as BCE, but does not increase the latency for LLC-accesses. However, note that Page-Coloring requires memory allocation in the same ratio as LLC allocations, which can lead to memory pressure. Ideally, LLC and DRAM allocations should be independent like in BCE.

DAWG incurs a higher slowdown of 2.2%, mainly due to its higher LLC conflict-misses, which outweighs the fact that it has an identical access latency as a non-secure LLC. The higher conflict-misses cause DAWG to incur a worst-case slowdown of 8% slowdown for *roms*, with multiple workloads suffering slowdowns of 4 - 8%. In contrast, BCE incurs

a worst-case slowdown of 2.5% for *parest* and 1 - 2% for other high MPKI workloads, which are sensitive to the increase in LLC latency in BCE.

5.5.4 Sensitivity to Increase in LLC Latency

We use a latency overhead of 3 cycles for BCE set-index computation. Figure 5.12 shows the slowdown of BCE (normalized to the non-secure baseline) as the latency overhead is varied from 1 cycle to 6 cycles. As the latency of BCE increases, the slowdown increases from 0.8% (for 1-cycle latency) to 2.3% (for 6-cycle latency). In comparison, page-coloring, which has a 0-cycle additional latency, incurs a 0.4% slowdown. Thus, our 3-cycle index calculation (with 1.3% slowdown) adds less than 1% extra slowdown on average due to the added lookup LLC latency, which is negligible.

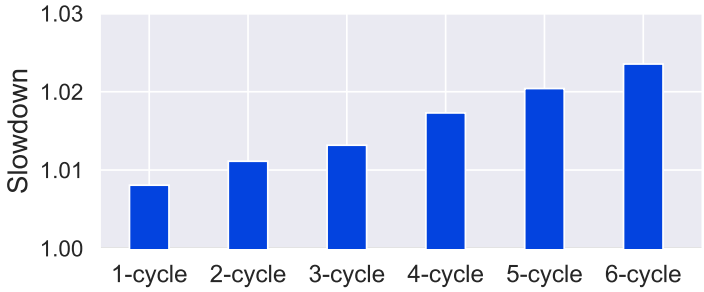


Figure 5.12: BCE slowdown as the latency overhead of set-index computation varies (default: 3 cycles)

5.5.5 Sensitivity of Performance to LLC Size

As LLC size increases from 8MB to 64MB, slowdown of BCE compared to non-secure LLC varies from 1.1% to 1.8%. At larger sizes, application working sets start to fit in. Therefore, at higher LLC hit rates, the impact of the increased access latency and misses is more pronounced; yet, BCE sustains a slowdown <2%. Similarly, Page-Coloring has slowdown varying between 0.2% to 0.9%; slowdown for DAWG varies between 2.2% to 2.7% with increasing cache size as its higher conflict-misses result in higher slowdowns.

5.5.6 Benefits of BCE's Fine-Grained Allocations

With DAWG, the cache partitioning scheme is constrained to making coarse-grained allocations at way-granularity. For a 16-core 16-way, 32MB LLC system, DAWG results in inflexible LLC allocations where each core gets 1-way 2MB partition regardless of its requirement. This is quite limiting if one core needs LLC space and the other 15 cores do not benefit from the allotted LLC space. As BCE has fine-grained allocations (at the granularity of 64KB), it allows resource allocation policies the flexibility to allocate few LLC clusters to programs not requiring much LLC space and more clusters to programs that require and benefit from LLC allocations.

Figure 5.13 shows such a scenario where *roms* is running with 15 copies of *PageRank* (*pr*). *roms* needs cache space, whereas *pr* does not benefit much from it. However, DAWG-Eq is still forced to equally allocate 2MB LLC (1-way) to *roms* and each copy of *pr*. With BCE, the LLC space allocated per *pr* copy can be 2MB if distributed equally among all 16 cores (BCE-Eq), or smaller allocations of 1.5MB, 1MB or 0.5MB per copy of *pr* are possible (with allocation policies: BCE-1, BCE-2, BCE-3) allowing *roms* LLC allocations to go up to 9.5MB, 17MB, 24.5MB respectively; with such BCE allocations, relative IPC of *roms* versus DAWG increases by up to 40%, while IPC of *pr* is unaffected (<2% change).

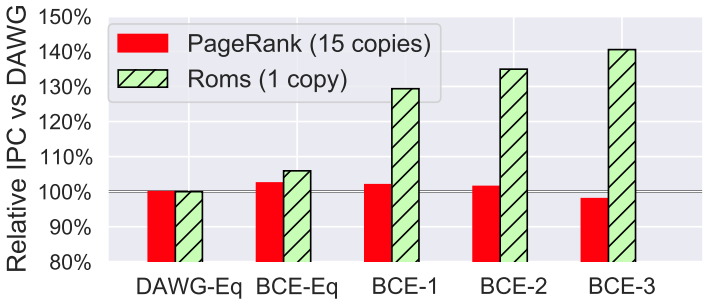


Figure 5.13: Flexibility of BCE vs DAWG. BCE allows smaller allocations to cache-insensitive *PageRank* while cache-friendly *roms* can have larger allocations, unlike DAWG with equal allocations of 1-way (2MB) for each of the 16 cores.

5.5.7 Storage Overheads

Implementing BCE requires additional structures to support the flexible indexing. Table 5.3 summarizes the storage overheads for the newly added structures with BCE. The index-computation module in BCE includes the Load-Balancing-Hash requiring 5 RBM hashes and three tables (Num-Clusters-Table, Domain-Base-Table, and Cluster-Location-Table), each with 512 entries. The newly added structures require a storage overhead of 2KB to support 512 clusters. To support 256 clusters, these structures would need an overhead of 1KB, whereas for 1024 clusters they would need 4.4KB.

Table 5.3: Storage Overheads for BCE Structures

New Structure	**Bits/Entry**	**NumEntries**	**Storage**
Random-Binary-Matrix Hash (RBM)	24×9	5	135B
Num-Clusters-Table (NCT)	9	512	576B
Domain-Base-Table (DBT)	10	512	640B
Cluster-Location-Table (CLT).	10	512	640B
Total Storage for New Structures			**2 KB**

BCE needs the tag to be increased by 9-bits to allow line address of a cacheline to be fully regenerated for writebacks. While conventional LLCs generate this by concatenating the tag with 15-bit set-index, BCE only has a 6-bit cluster-offset that it can use for this purpose. Hence, the tag is increased by 9-bits to allow concatenation of the tag and cluster-offset to regenerate the line address. This increases the LLC SRAM area by 1.8%.

5.6 Related Work

We first discuss cache-partitioning solutions for security and performance, and then other cache side-channel defenses.

5.6.1 Cache-Partitioning for Security

Page-Coloring [15, 21, 22] was an early OS/VMM-based approach for LLC partitioning that guides sensitive program memory to specific pages to ensure the usage of isolated LLC sets. But this requires memory allocations in the same ratio as LLC allocations, and is incompatible with large pages; this can result in high overheads on memory or cache-constrained systems. **MI6** [14] extends page-coloring to allow support of large pages with static changes to the set-indexing; however the LLC allocations are still in the same ratio as DRAM allocations causing under-utilization of LLC or DRAM; additionally, this design only supports up to 64 domains. In contrast, BCE allows independent LLC and DRAM allocations providing the flexibility needed by real-world applications such as graph workloads, and is scalable to hundreds of domains.

DAWG [16] is the state-of-the-art way-partitioning defense providing isolation from cross-domain hits and evictions. Other prior solutions such as PLCache [17], and CATalyst [20] prevent cross-domain evictions and not hits, and thus are vulnerable to hit-based replacement policy attacks. **NoMo cache** [18] and **SecDCP** [19] allow increasing way-allocations via a security-performance trade-off (NoMo) or by allowing one-way information leakage (SecDCP). All such way-partitioning-based solutions only support as many domains/partitions as the cache associativity (typically limited to 16-32 for LLCs). Our work provides equivalent security as DAWG, the state-of-the-art, while supporting up to 512 flexible partitions and with better performance.

Jumanji [121] proposes LLC partitioning at the granularity of LLC-Banks and allocating different LLC-banks to different VMs to prevent cross-VM cache side channels. However, it does not provide any security between distrusting processes within a VM, as it uses non-secure utility-based way-partitioning of the LLC between processes of a VM which is vulnerable to cross-process Flush+Reload [8] and cache-occupancy [3, 47] attacks. Moreover, it cannot support LLC isolation between enclaves of a single process as it

caches the LLC-bank mappings in a per-core "VTB" cache to enable fast indexing, which is vulnerable to new conflict-based attacks in this setting. In comparison, BCE is *more secure*, as it can isolate the cache state between any two trust-domains: enclaves within a process, processes within a VM, or VMs themselves. Moreover, BCE's 2-level indexing is carefully designed to be secure (constant-time) yet fast, unlike Jumanji's LLC-bank mappings that need to be cached leaving them vulnerable. Lastly, BCE can support hundreds of trust-domains and is *more scalable* than Jumanji, which only supports few tens of trust-domains (as many as LLC-Banks) and requires flushing LLC-Banks on domain-switches to support more trust-domains causing a high slowdown.

SHARP [7] modifies the replacement policy to prefer same-domain evictions on cache installs which do not leak information unlike cross-domain evictions. However, cross-domain evictions are still needed and performed when same-domain lines are unavailable in an indexed set, allowing conflict-based attacks to continue. Additionally, SHARP's restrictions on flush instructions prevent Flush+Reload attacks, but not Thrash+Reload attacks [43] (without flushes). BCE's principled cache isolation mitigates all such attacks.

5.6.2 Cache-Partitioning for Performance

Several studies have explored improving the cache performance by efficiently partitioning the LLC between the applications on different cores.

Utility-Based Cache Partitioning (UCP)[122] monitors the cache utility for each application and decides the number of ways to dedicate to each application. While this may be beneficial for performance, it leaves the cache vulnerable to occupancy-based attacks. Furthermore, the UCP scheme suffers from the scalability issues of way-partitioning. **K-Part** [123] and **PriSM** [124] overcome the granularity restrictions of way-partitioning by allowing multiple applications to co-reside in a single way; however, these schemes are insecure as applications can still evict each other's lines.

Prior studies [125] [126] have also used **Dynamic Page-Coloring** to dynamically partition the cache space. However, these schemes have the same short-coming as Page-Coloring in that memory and cache allocations are coupled. **Jigsaw** [127] uses page-mapping algorithms to share the capacity and reduce the latency of a banked cache. This design does not take security into account (the allocation and placement decisions based on utility monitoring are vulnerable to occupancy-based attacks). Finally, all cache partitioning schemes described in this sub-section are vulnerable to shared memory attacks.

5.6.3 Alternative Cache Side-Channel Defenses

Defenses like **CEASER** [9], **CEASER-S** [10], **RPCache** [17], and **NewCache** [84] adopt randomization of cache locations for defending against conflict-based side-channel attacks. **ScatterCache** [11] and **MIRAGE** [31] use randomization along with duplication of lines across domains to also make shared-memory based attacks harder. However, recent works [13, 12] have shown some of these randomization-based defenses are vulnerable to newer conflict-based attacks and none of these prevent cache-occupancy based attacks.

HybCache [82] prevents conflict-based attacks by providing applications fully-associative cache regions and prevents shared-memory-attacks by duplicating shared lines across applications. However, it allows dynamic sharing of cache space between multiple processes and is thus vulnerable to cache-occupancy attacks [3]. Also, authors of HybCache note that *"applying it to LLC or larger caches can be expensive"* [82]: a fully-associative lookup over thousands of LLC lines can be slower than a DRAM access and is impractical.

Attack-detection techniques proposing profiling attacks using performance counters [73, 74] or dedicated hardware [68] suffer from either false positives or false negatives.

In contrast, BCE provides secure isolation between domains and security against all three classes of cache-attacks (conflict, shared-memory, and occupancy-based attacks).

5.7 Key Contributions and Impact of this Research

5.7.1 Key Contributions

BCE provides a top-down redesign of the processor caches and interfaces to provide a principled defense against all cache side-channel attacks exploiting cache state. Overall, this work makes the following key contributions:

1. BCE enables a flexible and scalable cache isolation substrate. Unlike prior solutions, BCE supports hundreds of domains, and fine-grained allocations which can be flexibly managed independent of memory allocations. These benefits come while keeping performance loss within 1% of prior page-coloring solutions.
2. BCE's indirection module *(CIM)* enables flexibility in the placement of the logical clusters of a domain at arbitrary locations while the *Load Balancing Hash (LBH)* distributes lines of a domain uniformly among an arbitrary number of clusters. Together, they incur a negligible storage overhead of less than 2% of SRAM.
3. BCE's fine-grain custom-sized set-partitions can provide up to 40% performance improvement compared to prior way-partitioning solutions like DAWG.

5.7.2 Potential Impact

BCE primitives can potentially be used to extend the capabilities of software sandboxing solutions with cache isolation. Current software sandboxing solutions, like Google's NaCl or Chrome Site Isolation, provide isolation for software components (or web-page content sources) belonging to different trust domains – however, these distrusting software still contend for micro-architectural resources like caches resulting in privacy breaches. BCE's primitives can allow these distrusting software to map to isolated cache partitions, thus additionally providing them protection from cache side channels. This can enable better security for the Chrome Browser against even cache occupancy based attacks, which were recently shown to allow user web-browsing activity fingerprinting.

CHAPTER 6

CLEANUPSPEC - SECURING CACHES AGAINST TRANSIENT LEAKAGE

This thesis chapter focuses on preventing transient information leakage through caches. Randomization or partitioning defenses discussed in earlier chapters primarily focus on cross-process information leakage through shared caches. Such solutions cannot prevent information leakage within a process via transient cache accesses, *i.e.*, exploitation of the cache covert channel in transient execution attacks like Meltdown and Spectre. The focus of this chapter is to extend secure cache designs to prevent caches from being exploited in transient execution attacks, also known as speculation-based attacks.

6.1 Introduction

Speculation-based attacks like Spectre [5], Meltdown [6], Foreshadow [53], etc., have caused considerable concern in the computing industry given that they affect practically every processor-manufacturer and allow a software-based adversary to arbitrarily bypass software and hardware-enforced isolation without detection. These attacks are hard to detect or mitigate in software as they exploit micro-architectural vulnerabilities that are invisible to software. Therefore, commercially deployed software and micro-code patches are limited in being able to mitigate only specific attack variants (e.g. KAISER [128], Retpoline [129], IBRS [130]). As new attack variants continue to be discovered [52, 54, 55], there is a pressing need for broader hardware solutions [131].

Unfortunately, mitigating these attacks in hardware alone with minimal overheads is challenging, as these attacks leverage processor speculation and its side-effects on caches to obtain and leak secrets. Both speculation and caching being the cornerstones of high-performance processors, naively disabling them to mitigate the attacks would cause intolerable slowdown. While recent hardware designs [132, 133] have emerged that mitigate

Spectre attacks with low overheads by identifying potentially "unsafe" load patterns in the attacks and delaying them, they may not protect against other attacks that do not exhibit such patterns. In this context, there is a need for low-overhead mitigation of current and future speculation-based attacks exploiting caches, in particular data-caches[1].

Generalization of the Problem. These attacks can be generalized as having three key components, as observed by recent works [16, 34]: (a) *getting access to a secret* during speculative execution, (b) *speculatively transmitting the secret*, by making a secret dependent modification to the cache state (preserved even after a mis-speculation), and (c) *inferring the secret on the correct path* of execution, using side-channel attacks like Flush+Reload [8]. Stopping any of these three components is sufficient to defeat these attacks. In this work[2], we focus on preventing the third component, i.e. the secret inference on the correct path.

When a mis-speculation is detected, the processor invokes a pipeline flush to ensure that all of the data in the pipeline stages gets invalidated. However, a pipeline flush does not affect the content of the cache and any state change caused by the speculative instructions to the cache is retained. It is possible to close this cache side-channel by ensuring that either (a) no state change to the cache is caused by a speculative instruction or (b) the changes caused by the speculative instructions are *undone* when the mis-speculation is detected. We call the former approach a *Redo* approach to safe speculation (as the data may be read twice, once speculatively and a second time to change the cache state). The latter is an *Undo* approach (as the state change is performed and later undone on mis-speculation).

Challenges with Prior Work. A recent proposal, InvisiSpec [34], represents a *Redo-based* approach to safe speculation. To prevent speculatively executed (transient) instructions from making cache changes, for each data load, InvisiSpec first performs a load that

[1] I-Cache, TLB, or Branch Predictor can also be used to leak information, but delaying [134] or buffering [34, 135] transient changes to these structures can prevent such leaks. Port-contention [136] is out-of-scope due to its orthogonal nature, like prior works [16, 34].

[2] This chapter was published as a paper "*CleanupSpec: An Undo Approach to Safe Speculation*", appearing in the proceedings of MICRO 2019 [35]

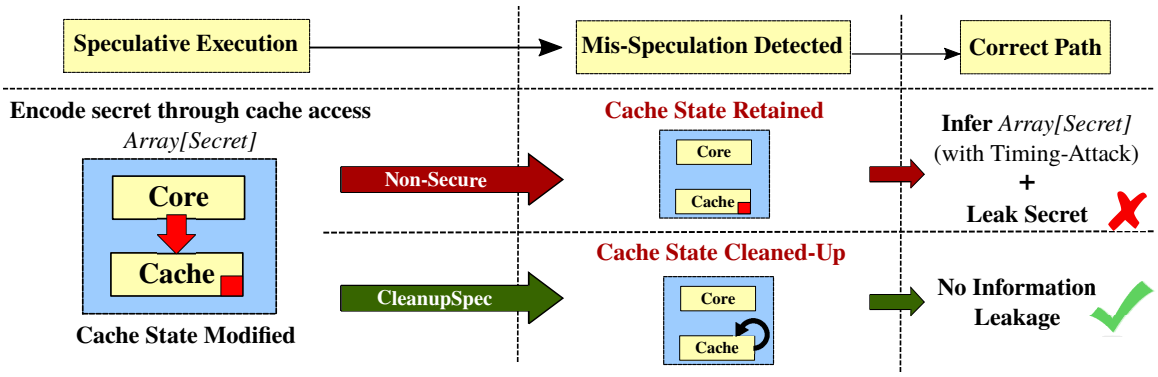


Figure 6.1: Speculation-based attacks leak secrets using transient instructions, by modifying state of a cache line whose address is based on the secret value. Currently, modifications are retained on the correct path and inferred using timing attacks, to leak the secret. CleanupSpec rolls back changes or ensures changes that remain are randomized, preventing information leakage on the correct path.

fetches the data without making any changes to the cache state and then a second load that changes the cache state (once it is deemed safe). Given that most loads are correctly speculated, InvisiSpec incurs the cost of double accesses to the cache hierarchy for most loads. Additionally, InvisiSpec requires the second access to be performed on the critical path before load-retirement, to verify that no memory consistency violations occurred during the *invisible* phase of the load, when the directory did not know the core had the data. As a result, InvisiSpec causes considerable slowdown (initial estimates show 67% on average)[3].

Key Idea of CleanupSpec. In this work, we observe that an *Undo-based* approach is better suited to mitigate these attacks with lower overheads. Just as the processor state is flushed when a mis-speculation is detected and any illegal instructions are removed, if the illegal changes to the cache made by transient instructions were also rolled-back at the same time, then any information leakage can be prevented. Such an approach would satisfy correctly speculated loads (which are the common case) with a single cache access. Additionally, overheads to roll back cache state would only be incurred in the uncommon-case of a mis-speculation. With this insight, we propose *CleanupSpec*, a low-cost hardware mechanism to prevent speculation-based attacks exploiting the data caches.

[3]While our estimates are close to InvisiSpec results in [34], its authors reached out to intimate us about an updated implementation with lower overheads (refer Section 6.6.5).

As shown in Figure 6.1, these attacks use transient instructions to extract a secret value to the correct-path. By encoding the secret value as the address of an array access ($Array[secret]$), these attacks install a secret-dependent array line in the cache. This line is retained in the cache even after a mis-speculation is detected. As a result, the secret is accessible on the correct-path with cache timing attacks [8] that infer which line was installed by the transient instruction.

Key Components. To roll back the cache changes made by transient instructions on a mis-speculation, the first step is to invalidate the cache line installs by the transient instructions. Unfortunately, invalidation alone is not sufficient [137] as information is leaked even through the cache lines evicted when these lines were installed. For example, an adversary could use an attack like Prime+Probe [38] to pre-load sets with its lines and observe which set experienced evictions during line installs by the transient instructions. Thus, the rollback of cache changes by transient instructions is feasible only if we prevent information leakage through evictions, without the overheads of potentially buffering evictions.

Recent works on cache randomization [9, 10, 11] make evictions benign for larger caches (L2 or LLC) with low overheads. By randomizing the cache-indexing, such proposals remove any discernible relation between co-resident lines in a set. Thus, observing an eviction leaks no information about the installed line causing it. Leveraging this, CleanupSpec uses randomization for L2/LLC (and for the directory [112]), and consequently only needs to roll back evictions from L1 data caches (as existing proposals [84, 83] for randomizing the L1 cache have pitfalls due to its VIPT design - see Section 6.2.4).

To enable L1 data cache rollback, CleanupSpec executes transient loads while tracking their cache side-effects (L1/ L2/ LLC installs and L1 eviction-victim line address). On a mis-speculation, to rollback the changes due to squashed loads, CleanupSpec invalidates the line in the L1 cache if it was installed speculatively and fetches the evicted line from L2 cache to restore the original line. For L2/LLC changes, CleanupSpec merely invalidates the copy there if it was installed speculatively (as L2/LLC evictions leak no information).

Rollbacks and invalidations are enabled with the help of side-effect tracking metadata in the load-store queue and L1/L2 MSHR (requiring <1KB of extra storage per core). These cleanup operations are only needed when a mis-speculation occurs and has squashed transient loads that missed in the L1 data cache and installed new lines. Given the high branch-prediction rates and L1-cache hit rates in typical applications, these operations are uncommon and cause limited slowdown. Additionally, the perceived cache state after these operations is as if the transient loads did not access the cache, so no information leaks on the correct path. Additionally, the transiently installed lines are also protected from access by other threads within the transient window, so no information leaks even in this window.

In addition to installs and evictions, CleanupSpec also prevents transient changes to replacement state and coherence state from leaking information. To protect the replacement state, it uses Random Replacement policy for the L1 data cache (any replacement policy can be safely used for a randomized L2 or LLC). For coherence state changes, CleanupSpec delays loads that cause such changes (M/E to S) until they are unsquashable on the correct path. As such transitions are infrequent compared to regular loads (<3% of loads cause such transitions – see Section 6.3.5), this adds negligible slowdown.

Key Contributions. Overall, this work makes the following contributions:

1. We propose ***CleanupSpec, an Undo-based Approach*** to mitigate speculation-based cache attacks, with lower overheads compared to prior approaches.
2. **Mechanisms for Safe Speculative Cache Accesses:**
   (a) **Roll back** changes to L1 efficiently.
   (b) **Invalidate** changes to the Randomized L2/LLC.
   (c) **Randomize** replacement policy for L1.
   (d) **Delay** changes to the coherence state in the remote cache.

Key Results. We model CleanupSpec in Gem5 [138], under a threat model where any mis-speculated load could leak information (like prior work [34]). As a proof-of-concept,

we demonstrate that it mitigates the Spectre V1 PoC. We also evaluate the performance of CleanupSpec over 19 SPEC-2006 workloads. Compared to a non-secure baseline, CleanupSpec incurs an average slowdown of 5.1%, that is much less than Redo-based approaches like InvisiSpec for a similar threat model. Moreover, CleanupSpec only requires <1KB of storage per core and simple logic, for tracking and restoring transient cache state changes.

6.2 Background and Motivation

6.2.1 Threat Model

Modern processors speculatively execute instructions out-of-order to avoid stalls due to control and data dependencies and achieve high performance. This can result in the execution of *transient instructions*, i.e. speculatively executed instructions on wrong execution paths. We deem all transient instructions potentially "unsafe" until they retire, to protect against current and future attacks like InvisiSpec [34].

Adversarial Capability: We assume attacker-executed transient instructions have arbitrary access to secrets. The secret may be accessed from the memory or a register, or computed, transiently. We assume the secret is transmitted to the correct path only using cache side channels, as a majority of attacks [6, 5, 53, 55] use them given their high bandwidth. We only consider attacks exploiting the data-cache hierarchy, including private data caches (e.g. L1-D cache) and shared caches (L2/LLC), as other structures like instruction caches, TLB, etc. can be protected with a low overhead with prior works [34, 134]. The adversary may transiently access the cache and modify its state through installs [8], evictions [38], updates to replacement [16] and coherence [69] state, and infer changes on the correct path through timing differences on cache accesses.

Out of Scope: We do not consider speculation-based attacks using AVX as a covert-channel [43], since they are easily mitigated by disabling the speculative power-up of AVX units. We also assume speculative hardware prefetching is disabled for caches (similar to [34]), preventing information leakage through training of the prefetcher on transient loads. Similarly, we assume a close-page row-buffer policy for the memory controller, to prevent covert-channels like DRAMA [139]. We further consider out-of-scope side-channels due to SMT port-contention [136], network [140] or DRAM [141] contention, EM radiation or power, given their orthogonal nature.

6.2.2 Speculation-Based Attacks

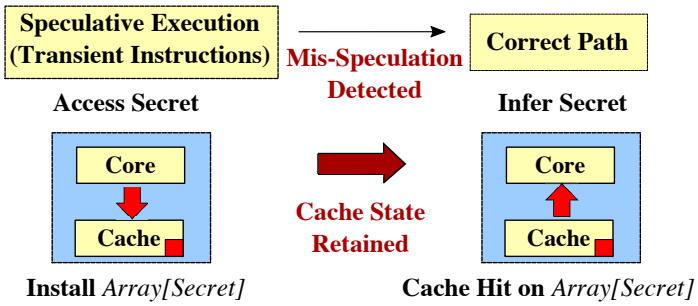


Figure 6.2: Recipe for speculation-based attacks, using the cache as a transmission channel for leaking secrets.

Attacks like Spectre [5], Meltdown [6], etc., leverage transient instructions to access secrets, bypassing any software checks. When a mis-speculation is detected by the processor, these transient instructions are squashed. Hence, these attacks attempt to exfiltrate the secret to the instructions on the correct path of execution before mis-speculation is detected – most commonly using the Flush+Reload [8] attack on the cache. As shown in Figure 6.2, these attacks transiently access an array entry, whose index is computed using the secret value (the array is completely evicted from the cache previously, using `clflush`). This installs array entry $Array[Secret]$ into the cache. As the cache state is preserved even after a mis-speculation, the attacker infers the secret on the correct path by accessing each array entry and observing which one has a cache hit.

Meltdown and Spectre (and other variants [52, 53, 54, 55]) use this recipe to leak secrets accessed transiently from memory. Future attacks could use this recipe to leak secrets stored in registers, or results of malicious computations performed transiently. In fact, a recent malware called exSpectre [50] computes some of its malicious payload speculatively to evade reverse engineering attempts, and extracts the results of these computations through a cache covert channel.

While software [128, 129] and microcode patches [142] mitigate the original attacks, their adoption is hindered [143] due to high slowdown [144]. Recent hardware mitigations [133, 132] limit the slowdown by detecting and delaying only certain *unsafe* load patterns in Spectre, arising from transient access to secrets in memory; but they fail against attacks without such patterns (e.g. leaking secrets from registers or computations). Thus, there is a need to prevent *any* transient load from leaking information through cache state changes (preferably without OS/SW support unlike other proposals [16, 14]).

6.2.3 InvisiSpec: A Prior Redo-Based Mitigation

A recent design InvisiSpec [34] developed a way to tolerate all such attacks in hardware. As transient instructions modify the cache state to transmit secrets, InvisiSpec disallows any changes to the cache state during speculative execution. For speculatively issued loads, InvisiSpec executes an *Invisible* load that makes no changes to the cache hierarchy and only brings the data to the core, as shown in Figure 6.3. Once the speculation is determined as correct and the load is ready for commit at the head of the ROB, it *Redo*es the load – executing a second load to update the cache. Thus, a mis-speculated instruction that is squashed leaves no trace in the cache, preventing information leakage on the correct path.

While this approach prevents mis-speculated loads modifying the cache, most loads are issued speculatively and correctly speculated in the common case – requiring a second *Update* load for cache update. This increase in loads can impact performance.

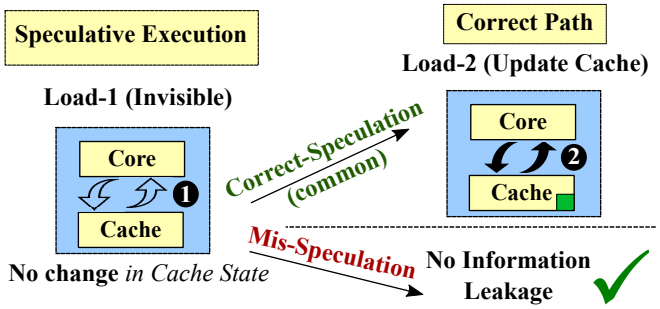


Figure 6.3: InvisiSpec adopts a redo-based approach to prevent mis-speculated loads from modifying the cache – speculative loads are invisible to the cache; On the correct path, the load is repeated to update the cache.

Performance Problem in InvisiSpec

For common-case correctly speculated loads, InvisiSpec requires the second load to be performed on the critical path at commit-time, before retiring the instruction. This is because of the need to ensure no memory consistency violations occurred in the period between the invisible load and commit, due to a modification of the data by a different core. As the invisible load does not update the ownership of the line in the directory, the core is unable to receive invalidation updates until the second load updates the cache and directory state.

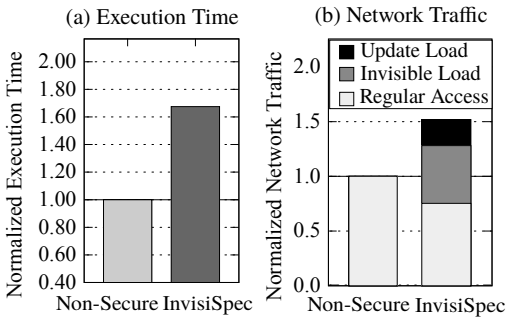


Figure 6.4: (a) Execution Time and (b) Network Traffic for InvisiSpec normalized to Non-Secure Baseline (initial estimates)

Figure 6.4 illustrates the performance and network traffic for InvisiSpec normalized to a Non-Secure baseline, using 19 SPEC-CPU2006 workloads. As per initial estimates[4], InvisiSpec suffers an average slowdown of 67.5% compared to Non-Secure, that is accom-

[4]We report our results using the InvisiSpec public code-base (commit: 39cfb85) [145] as initial estimates, as the authors intimated us shortly before the camera-ready deadline that they have an updated implementation with lower overheads (refer Section 6.6.5).

panied by a 51% increase in network traffic. Approximately half of the additional traffic is driven by the extra accesses to update the cache at commit-time (Update-Loads in Figure 6.4(b)) and check for potential consistency violations. As these accesses fall on the critical path, the resulting stalls cause slowdown. Additionally, almost half the network traffic is due to speculative loads, indicating a majority of loads are issued speculatively.

6.2.4 Undo Approach: Benefits and Challenges

Given the high branch prediction accuracy in modern processors, a large fraction of the loads issued would typically be correctly speculated. Therefore, rather than adopting a *redo*-based approach that requires performing a load twice for correctly-speculated loads, an *undo*-based approach is preferable from a performance perspective.

An undo-based approach would allow all loads to modify the cache speculatively, as shown in Figure 6.5. In case a mis-speculation is detected, then corrective action could be activated that cleans up the cache state changes made by the illegal transient loads to ensure that no information is leaked on the correct path. Such an undo-based approach would be beneficial for performance, as any overhead would be incurred only in the uncommon case of a mis-speculation, while the correctly speculated loads execute without any change. Moreover, some of the cleanup overhead would be hidden by the pipeline drain latency that is incurred in any case.

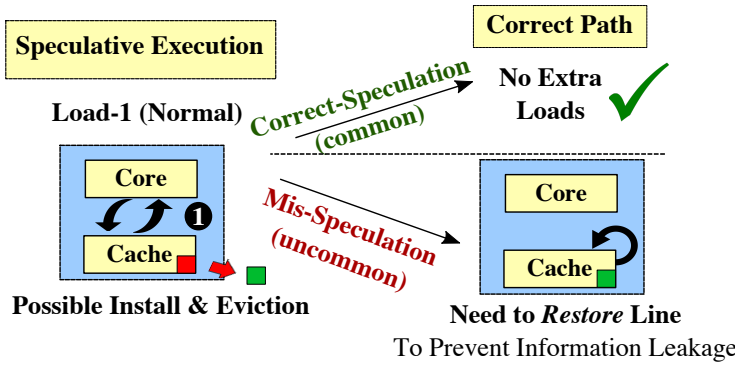


Figure 6.5: A low-overhead Undo-approach is viable as long as cache state cleanup on mis-speculation prevents information leakage on the correct path.

Naive Invalidation - Vulnerable to Prime+Probe

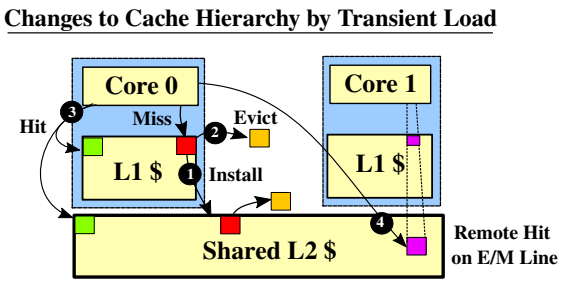
A naive design to undo cache changes by transient instructions could track lines installed in each level of the cache hierarchy by such instructions and invalidate them on a mis-speculation. This prevents an adversary from using Flush+Reload attack to infer transiently installed lines.

Unfortunately, lines installed transiently can evict other lines from the same set. Leveraging this, an adversary can use Prime+Probe [38] attack – pre-load sets with its lines and observe which lines get evicted due to installs. Thus, an adversary can infer cache sets that had a transient line install. To prevent such attacks, we need to not just remove the installed line, but restore the evicted line in its place.

Mitigating Prime+Probe for L2/LLC

Recently, CEASER [9] and Skewed-CEASER [10] proposed randomizing last-level caches with address encryption, to mitigate eviction-based attacks like Prime+Probe. With randomized cache indexing, these solutions map random lines to the same set. As a cache install evicts an unrelated line, evictions stop leaking information about installed lines.

Large caches like L2/LLC or Directories [112] can adopt such randomization without any changes to OS/SW. However, randomization is challenging for performance-sensitive L1 caches. Address encryption is not feasible as it would add 1-2 cycles of encryption latency to every access, doubling the L1-access latency, which would be detrimental to performance. Additionally, other proposals like NewCache [84] or RPCache [83] face challenges due to the preferred VIPT operation of L1 caches – they either need VIVT operation of the cache that has the problem of synonyms [146], or need PIPT operation that has slower access (TLB look-up is on the critical path). Thus, L1 caches are hard to randomize and vulnerable to eviction-based attacks, making an Undo-approach challenging.



Change by Transient Load			CleanupSpec Mitigation
❶	Install on Miss	L1/L2	Remove on Squash
❷	Eviction on Miss	L1	Restore on Squash
		L2	Randomize L2 LineAddr
❸	Replacement State Update on Hit	L2	
		L1	Random Replacement
❹	Coherence Downgrade (E/M -> S)		Delay Until Correct Path

Figure 6.6: Design overview. To prevent transient loads leaking information on the correct path, CleanupSpec *Removes, Restores, Randomizes*, or *Delays* their cache changes.

6.2.5 Goal of this Work

Our goal is to enable a secure, low-overhead implementation of an undo-approach to safe speculation by reusing existing structures. We develop our solution as a combination of restoration of evictions from L1 caches, invalidation and randomization for L2/LLC, random replacement for L1 cache to avoid state changes of replacement metadata, and delaying the state change due to coherence for speculative instructions.

6.3 Design of CleanupSpec

Our design philosophy with CleanupSpec is to optimize the design for the common case where loads are correctly speculated. To this end, CleanupSpec allows transient loads to speculatively access the cache and make changes as required. To enable security on a misspeculation, we study the changes a transient load could make to the data caches, and delay, reverse or randomize these changes.

We study the transient cache changes using a configuration shown in Figure 6.6, with two cores having private L1 data caches and sharing a common L2 cache. A load can incur a miss in the L1 or L2 cache, that evicts a victim and installs a new cache line in its place, which is retained on the correct path. Furthermore, if a dirty line is evicted from L1 cache as a result of the install, it would cause a write-back to L2, that could, in turn, cause an eviction from L2. A load hit can update the replacement state that can also

leak information [16], as it affects victim selection and can be used to engineer evictions. Similarly, a load to a line in Exclusive (E) or Modified (M) state owned by a different core would cause a coherence state downgrade for the line to Shared (S) – even this can leak information [69] on the correct path due to differences in access latencies to E/M and S lines. To prevent such changes by transient loads from leaking information, CleanupSpec needs to track, protect and reverse these changes on a mis-speculation, so that they are imperceptible to an adversary.

6.3.1 Overview of CleanupSpec Design

There are six main components to CleanupSpec that make the transient changes to the cache hierarchy benign.

1. **Address Randomization for L2 Cache – Prevent leaks from L2 Evictions and Replacement Policy**: Buffering evictions from the L2 cache can be expensive in storage and complexity. Moreover, tracking recursive evictions in the cache hierarchy due to writebacks causes further complications. Address randomization (e.g. CEASER [9]) randomizes the sets that spatially contiguous lines map to, making co-residents of a line in a set unpredictable. As a result, an eviction leaks no information about the address of the Install or L1-Writeback that caused it. Similarly, the replacement state of L2 becomes benign, as it can only be exploited to induce evictions of random-lines. We analyze the overheads of L2-Randomization in Section 6.3.2.

2. **Removing L1 or L2 Installs from the Cache:** To prevent a transiently installed line from causing hits on the correct path after a mis-speculation, we remove the line from the levels of the cache it got installed in by issuing an invalidation to only those cache levels. We enable this by tracking which levels a load caused an install, propagating this information with the load through its lifetime in the L1/L2-MSHR and the Load-Queue, till it is retired (more details in Section 6.3.3).

3. **Restoring L1 Evictions:** Without randomizing the L1 cache, we need to prevent evictions from leaking information. So, on a mis-speculation, while removing the installed line, we also restore the original line that was evicted. We achieve this by tracking the line address of the evicted line in the L1-MSHR on an install and propagating it with the load-data to the Load-Queue. After restoring the evicted lines, we achieve a L1 cache state such that the unsafe L1 installs and evictions never occurred (more details in Section 6.3.4).

4. **Random Replacement Policy for L1:** To prevent replacement state updates on L1 hits from leaking information, we use random replacement policy for the L1 cache. We find that this has a negligible impact on the overall system performance (see Section 6.3.2).

5. **Delaying Coherence Downgrades from M/E to S, till correct path:** Changes in coherence state are not only perceptible from the same core on the correct path, but also have a non-reversible impact on other cores. Thankfully, only transitions from M/E to S are perceptible due to differences in access latency (and relatively infrequent), hence we delay them till the correct path. We describe the changes required to the coherence protocol to achieve this in Section 6.3.5.

6. **Protecting lines in the window of speculation:** In the small window of time between a speculative line install and a load-squash, an access from another thread/core that has a hit on this speculatively installed line can leak information. Our design can detect such hits and service them with cache-miss latency (using dummy requests) to prevent information leakage. We describe this mechanism in Section 6.3.6.

6.3.2 Randomizing L2 Lines & L1 Replacement

We use address randomization for L2 cache, using an encrypted line address to index the cache (like CEASER [9]). As the encryptor only adds 2-cycles to access latency, this incurs

minimal overhead. While this prevents L2-evictions from leaking information, we observe it also prevents the L2 replacement state from being exploited. Prior work [16] exploited transient replacement state updates to influence victim selection and engineer evictions. However, as evictions are benign with randomized caches, even replacement state updates cannot be exploited and intelligent replacement policies can be freely used for the L2 cache. As randomizing L1 data caches is challenging, we use a random-replacement policy for L1. Thus, replacement state updates for L1 and L2 cache leak no information.

Table 6.1 shows the performance impact of randomization (L1 Random Replacement, and L2 Index Randomization), compared to a non-secure design using 19 SPEC-2006 benchmarks. While random replacement for L1 cache causes a minor increase in the miss rate, these misses are serviced from L2 with minimal overhead. While L2 randomization causes a minor increase in L2 misses, together they have negligible slowdown (<1%).

Table 6.1: Performance Impact of Randomization for L2 (2MB) and Random Replacement for L1 DCache (64KB) vs LRU-Baseline.

Configuration	**Slowdown**
L1-Rand Replacement	0.1%
L2-Randomization	0.4%
Both Together	0.8%

6.3.3 Removing L1 and L2 Installs

On a transient cache miss, data is speculatively installed in the cache. On a mis-speculation, to remove any trace of such installs, the line is removed by issuing an invalidation for it in those levels. To identify which levels of the cache had an install, we track the side-effects of every load on L1 and L2 cache in a Side-Effect Entry (SEFE, pronounced *safe*), by augmenting entries in the L1/L2-MSHR and Load Queue (LQ). As shown in Figure 6.7, the SEFE includes 1-bit (*isSpec*) indicating a speculative load and 1-bit per cache level *(L1-Fill*, *L2-Fill*) to indicate that the load caused an install at this cache level. The fields *L1-Fill* and *L2-Fill* are updated in L1/L2-MSHR entry when the load causes cache changes during

miss-handling, and the *LoadID* tracks the order in which these changes occur. Whereas, the shaded SEFE fields – *EpochID* (that uniquely identifies the phase of execution between 2 cleanups) and *isSpec*, are updated by the Load/Store unit when the load is issued. The SEFE is returned with the load to the core and retained in LQ, until the load is retired.

On a load squash, the LQ-Entry SEFE is referred to decide whether to send invalidation message to L1/L2 based on L1-Fill/L2-Fill values, or if it can be skipped – if the load was not issued or if there was a L1-Hit (L1-Fill=0 and L2-Fill=0).

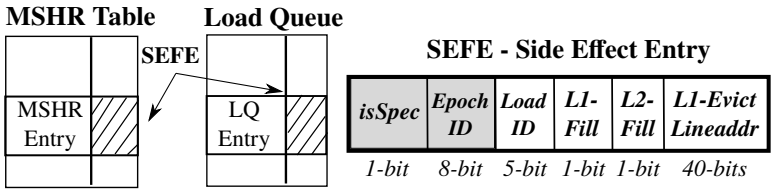


Figure 6.7: Side-Effect Entry (SEFE) tracks side effects of a load, in Load Queue and L1/L2-MSHR. Shaded SEFE fields are filled by Load/Store unit, unshaded by L1/L2.

If a load is yet to return to the LQ, and is in flight, then a cleanup request is sent to L1/L2-MSHR to squash pending loads and increment the current active EpochID. The core is stalled only till an acknowledgment from L1/L2-MSHR. Subsequently, loads are safely issued with an incremented EpochID, using a new MHSR entry and resulting in a fresh memory request. Squashed MSHR entries (with an outdated EpochID) waiting for inflight memory requests are preserved until the data is returned from memory, upon which both the data and the MSHR entry are safely dropped (cache changes like install and victim replacement are made only when a load returns and is for the current EpochID). EpochID has enough bits so that the incremented value does not match that of an inflight load.

6.3.4 Restoring Lines Evicted due to L1 Installs

Squashed loads that install a line in the L1 cache also leak information through evictions. To prevent this, in addition to invalidating the line, we restore the original line evicted from

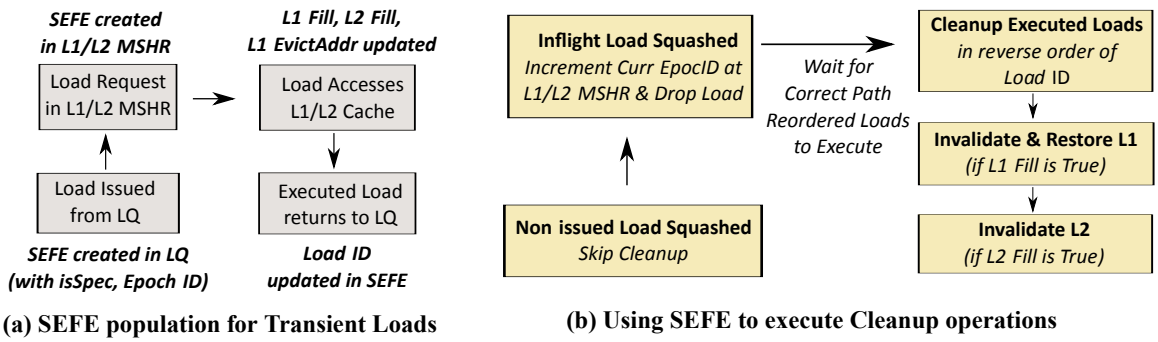


Figure 6.8: Flowchart for Two Phases of Execution. (a) Regular Operation where Side-Effect Entries (SEFE) are updated. (b) Cleanup on Squash – state in SEFE is used for determining and executing Cleanup operations.

the L1 cache. We enable this by recording the line address of the evicted victim in the L1-MSHR SEFE (*L1-Evict Lineaddr* in Figure 6.7). When the load is squashed, the evicted line is restored from the L2 by issuing a normal demand access for this address to L2, and installing it in place of the invalidated line. To ensure no illegal changes are introduced due to the restoration, we need to consider the following cases:

Avoiding Recursive Squash During Cleanup: Before operations like invalidate and restore begin, we wait for the retirement of all in-flight loads before the squashed loads in program order (potentially correct-path loads), reordered due to out-of-order execution. Thus, any squash at an earlier point in program order is detected before cleanup begins. Moreover, no new loads are issued while the cleanup requests are being issued and an acknowledgment is yet to be received, avoiding restoration of an incorrect transient state.

Squashing Re-ordered Loads: We ensure cleanup operations for loads are issued in the reverse of the order in which they executed, with cleanup of independent sets proceeding in parallel. Moreover, we squash inflight loads before the invalidation and restoration of executed loads. Ensuring that cleanup operations are correctly ordered is possible using the *Load-ID* field in the SEFE that tracks the order of loads returning from the L1 cache. Thus, removal/restoration effectively reverses time for the cache and no illegal state is retained.

Squashing Loads Re-ordered with Correct-Path Loads: Changes made to the cache by older non-squashed loads that execute after squashed loads due to re-ordering, need to be preserved and not reversed, as they would have occurred irrespective of the squashed load. Such execution patterns can be detected by the LQ-SEFE, and corresponding invalidations and restorations are skipped.

6.3.5 Delaying Coherence State Downgrades

Prior work [69] showed that the coherence state of a line can be used to leak information. This is due to the access latency difference between E/M and S lines: accesses to S lines in L2 can be satisfied directly from the L2, whereas E/M lines typically require servicing the line from a remote-L1. As allowing the change from E/M $\rightarrow$ S transiently can be a vulnerability, CleanupSpec delays such changes until the correct path. Note that transient changes like I $\rightarrow$ E or adding S-sharers (even in remote directories), that do not affect the state of a line cached by a remote core, are allowed and reversed.

Table 6.2: Coherence state transitions in a remote core, caused due to actions initiated by transient instruction.

Old State	New State	Transient Instruction	Mitigation
M,E	S	Load Shared Data	Retry on correct-path
M,E,S	I	`clflush`	Delay till correct-path

A transient load to shared data can cause a change from E/M state to S in a remote cache. To delay such loads till they are unsquashable, we add a new transaction called GetS-Safe that is used by default instead of GetS. GetS-Safe only succeeds in getting the data if a E/M$\rightarrow$S transition is not required. In case it fails, the core delays the load till it is unsquashable, and repeats the load with GetS only once it is safe to force the transition.

We expect these transitions to be infrequent as they usually occur on transfers of lock ownership in multi-threaded workloads. Such transfers are relatively uncommon, compared to accesses for actual work. We characterized this in 23 multi-threaded PARSEC [147]

and SPLASH-2 [148] benchmarks using *simlarge* dataset on a 4-core system with Sniper-sim [149]. As shown in Figure 6.9, loads to Remote-E/M lines make up just 2.4% of total loads on average. Thus, delaying such loads would have negligible slowdown, with 96.8% of loads to local lines in M/E/S state or Remote-S lines proceeding as is.

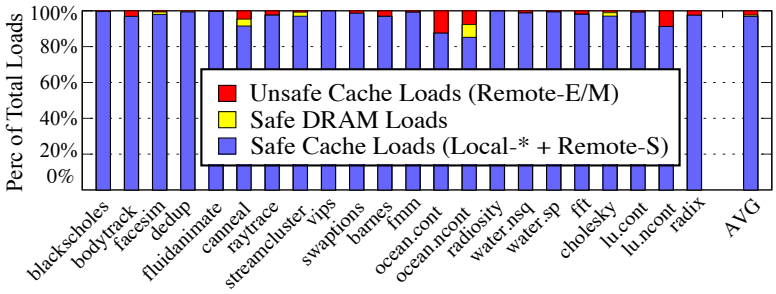


Figure 6.9: Breakup of Loads based on State of Line for multi-threaded PARSEC and SPLASH2 benchmarks.

A transiently executed `clflush` can also be used to leak information, as it can invalidate all copies of a cacheline, even in adversary-controlled remote caches. We delay such instructions, until they are unsquashable on the correct path. As such instructions are ordered with respect to stores, and normally used after stores, they typically execute on the correct path anyway and see no added delays.

6.3.6 Protecting Installs in Speculation Window

In the small window of time between the speculative install of a line into L1/L2 cache, and removal of the load from the pipeline through a squash, a cross-core or cross-thread SMT adversary might access the line, hoping to infer the speculative install with a cache hit. To prevent such leaks within the speculation window, we ensure the first access to a line in this window from a different thread/core than the one installing it incurs a dummy cache miss. In this case, the cache issues an explicit request to the backing store (L2/main-memory), waits till it completes and only then returns the data, leaking no information.

We characterized the speculation window for SPEC workloads and observed that >98% loads commit/squash within 200 cycles of being issued, with >99.5% finishing within 600

cycles. Within this speculation window, detecting cross-thread L1-cache accesses for a line is possible with SEFE metadata in LQ/L1-MSHR. To detect cross-core accesses to L2-cache within this window, we keep the SEFE in L2-MSHR active for 200 cycles after a line install (MSHR entries are accordingly provisioned). For loads that continue to be speculated beyond 200 cycles, the core sends a message to the cache to extend the window for such a load every 200 cycles. These messages constitute <2% cache traffic.

Once detected, accesses from a different thread to speculatively installed data can be served as cache misses. This is only needed in uncommon scenarios in benign programs and incurs negligible slowdown, as repeated cross-program accesses to recently installed lines within the small window are unlikely. In multi-threaded programs, such behavior triggers coherence-downgrades that we showed to be uncommon (<3% loads in Figure 6.9).

An adversary may also try to infer speculative evictions in this window. For a cross-core adversary, our L2 cache randomization prevents L2 evictions from leaking information (Section 6.3.2). For SMT adversary, a mitigation like a way-partitioned L1 cache (e.g. NoMo [18]), that is anyway required for preventing non-speculative L1 cache side-channel attacks, also prevents speculative L1 evictions being observed. Our evaluations show that partitioning L1-ways between 2 threads incurs <2% slowdown (concurring with [18]).

6.4 Security Analysis

We study three adversary models: *SameThread* - where a transient cache access is initiated and observed from the same thread, *CrossCore* - where the initiator and observer are on different cores and *SMT* - where they are in different threads running simultaneously on the same core. In each scenario, we assume the transient change is initiated with a speculative read and observed by the adversary using: speculative reads, or non-speculative reads or writes. We assume the transient change (or lack thereof) is inferred based on latency difference of a cache hit or miss, or a latency difference on a coherence upgrade or downgrade operation on cached lines.

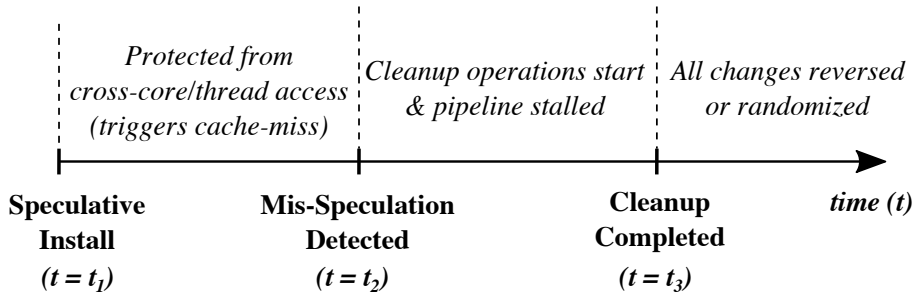


Figure 6.10: High level idea of security with CleanupSpec.

For security, we argue that CleanupSpec satisfies three properties:

P1 **Before detection of mis-speculation** *(t1 to t2)*, transient cache changes are protected and not observable.

P2 **During cleanup operations** *(t2 to t3)*, the process of restoration of cache state does not practically leak information.

P3 **After cleanup** *(t3 onwards)*, all transient changes are restored or made imperceptible, so no changes are observable later.

(a) Security until detection of mis-speculation: CrossCore and SMT adversaries are prevented from observing a hit on a line transiently installed in L1 or L2 cache by the SEFE entries in MSHR, which trigger a dummy cache miss in such scenarios (refer Section 6.3.6). Similarly, a CrossCore adversary cannot infer any information from L2 evictions due to L2-randomization. SMT adversaries are prevented from observing transient L1-evictions, with L1-cache way partitioning across threads (required anyway for preventing non-speculative L1 cache side-channel attacks under SMT). These hit/miss attacks are impractical with SameThread adversary, as observing a hit or miss on a transiently installed/evicted L1-cacheline requires hoisting wrong-path loads over older correct-path loads. But, measuring their timing with any fidelity requires using timers with serializing instructions (like `cpuid` or memory fences) [150] that would prevent any such hoisting, precluding this attack.

To leak information, CrossCore adversary may also initiate coherence upgrades (using writes) or downgrades (using non-speculative reads) on transiently installed/evicted addresses. To prevent latency differences in such scenarios, CleanupSpec imposes two constraints: (1) writes need to be constant-time and equal to the worst-case of downgrading S lines in every core. (2) reads that trigger downgrades on a transiently installed line in a remote L1-cache, need to be serviced directly from the L2-cache if the transient install did not cause a L2-Fill (otherwise from the main-memory) and not from the remote-L1 cache. Note that this is possible because the L2 copy never becomes stale transiently as RFOs are issued non-speculatively to prevent Spectre-Prime[72] attacks. If the adversary attempts downgrade on a line transiently evicted from remote L1 cache, a dummy delay can be added to the adversary's read, to emulate service from the remote cache. These constraints can prevent latency differences for coherence operations while incurring negligible performance loss – as write-latency is not on the critical path of program execution, and read-caused-downgrades are infrequent in typical programs (as shown in Figure 6.9). Note that CleanupSpec prevents speculative initiation of a coherence downgrade (E $\rightarrow$ S), as it is naively observable by a CrossCore adversary (as described in Section 6.3.5).

(b) Security during cleanup operations: On detection of mis-speculation, all non-squashed loads that are inflight are completed before cleanup operations begin and no new instructions are fetched until cleanup finishes. This prevents SameThread adversary from observing any contention during cleanup operations. While this increases the stall time after a mis-speculation, this causes the actual time taken for cleanup operations to make up only a small portion of the stall (as shown in FigureFigure 6.14) and hence hard to exploit for the SameThread adversary. This is also because the corresponding restoration cache accesses are pipelined and serviced from the inclusive L2 (which is the common configuration in modern processors like Intel Skylake-X or AMD Ryzen-2). Moreover, the adversary cannot transiently evict the line to be restored from L2-cache to increase the time for cleanup operations, as that would need thousands of accesses due to the randomized-

mapping (that is impractical to perform transiently). Future work can explore making the cleanup operations incur a constant-time stall to make this theoretically impossible to exploit. For CrossCore or SMT adversary, ensuring security during cleanup operations is along the same lines as Section 4(a).

(c) Security after cleanup on mis-speculation: After cleanup, all lines are invalidated from the cache levels where they were transiently installed, and the evicted lines are restored for L1 and randomized for L2. So no new cache hits or misses are observable due to the transient changes, in any adversary model. Moreover, attempting to even infer if a single L2 eviction occurred is impractical, as filling the randomized L2 deterministically (required to initialize this attack) is not feasible – the randomized and changing mapping would result in self-evictions during the attempt to fill the cache. Finally, the coherence state of the restored/invalidated lines is updated based on presence in other private caches, such that it is independent of the transient access by the victim, and it leaks no information.

6.5 Experimental Methodology

6.5.1 Simulation Framework

We evaluate our design using Gem5 [138], a cycle-accurate simulator that models the out-of-order processor including the wrong-path execution effects in the core and the caches. We simulate a single-core system in System-call Emulation (SE) mode in Gem5. For the multi-core characterization in Section 6.3.5, we used Sniper [149] that simulates multi-threaded binaries using a Pin front-end (due to errors with multi-threaded workloads on InvisiSpec's infrastructure). For InvisiSpec evaluation, we use their public code-base (commit: 39cfb85 [145]) and evaluate the *InvisiSpec-Future* mode that treats all speculative loads as "unsafe" (same as our threat model).

6.5.2 Workloads

We use 19 workloads from SPEC-CPU2006 [151] with the *reference* dataset. We forward the execution by 10 billion instructions and simulate 500 million instructions. Table 6.3 shows the important workload characteristics. For the multi-threaded workload characterization in Section 6.3.5, we used 23 workloads from PARSEC [147] and SPLASH2 [148] benchmarks with the *sim-large* dataset, and collect statistics for the entire region of interest.

Table 6.3: Workload Characteristics

Workload	**Branch Mis-Prediction Rate**	**L1-Data Cache Miss Rate**
astar	12.4%	1.8%
gobmk	11.9%	1.0%
sjeng	11.3%	0.2%
bzip2	9.7%	2.0%
perl	7.7%	0.5%
povray	7.5%	0.2%
gromacs	6.8%	1.1%
h264	5.4%	0.5%
namd	4.2%	0.3%
sphinx3	4.1%	4.0%
wrf	2.2%	0.5%
hmmer	1.9%	0.2%
mcf	1.6%	2.5%
soplex	1.5%	5.9%
gcc	1.3%	0.1%
lbm	0.3%	11.0%
cactus	0.1%	0.9%
milc	0.0%	4.6%
libq	0.0%	10.4%

6.5.3 Configuration

We evaluate a system configured as in Table 6.4. A minor difference compared to prior work InvisiSpec, is that we use a Close-Page policy in the memory controller to prevent information leakage through DRAM row-buffer hits and misses. Additionally, we increase the L2 cache access latency by 2 cycles (from 8 cycles to 10 cycles) to incorporate the overheads of address randomization [9]. We evaluate a 2-level inclusive cache-hierarchy for simplicity, but our design philosophy is equally applicable to other configurations.

Table 6.4: System configuration (similar to InvisiSpec [34])

Architecture	1-core OOO, no SMT, 2GHz
Core	ROB-192 Entry, LQ / SQ-32 Entry Tournament Branch-Pred, BTB-4096 entry, RAS-16 entry
L1 I-Cache	32KB, 4-Way, 64B line 1-cycle RT Latency
L1 D-Cache	64KB, 8-Way, 64B line 1-cycle RT Latency
Shared L2-Cache (inclusive)	2MB/core, 16-Way, 64B line, 10-cycle RT (incl. 2-cycle addr-encryption latency)
Coherence	Directory-based MESI
DRAM	50ns RT after L2

6.6 Results

6.6.1 Proof-of-concept Defense

We test our Gem5-based design with Spectre Variant-1 PoC [152], that exploits branch prediction in victim code – `if(x < array1_bound) {array2[array1[x] * 512]}`. The attack attempts to bypass the array bounds-check by using benign values of `x` to prime the branch-predictor. Subsequently, using a `malicious_x` causes a speculative access to a secret memory location `array1[malicious_x]`, which is used to generate the array-index for an access to `array2`. On the correct path, the secret is inferred by testing which line of the `array2` gets a low-latency cache hit. Figure 6.11 shows the access latency for different `array2` indices during the secret-inference phase, averaged across 100 iterations.

In the baseline (NonSecure), the attacker perceives lower latency for `array2` entries installed during the benign training phase (corresponding to `array1[x]` values 1–5). For these locations, the average latency lies between the cache and memory access latency, because the attack fails to detect cache hits in some attack iterations. A low latency

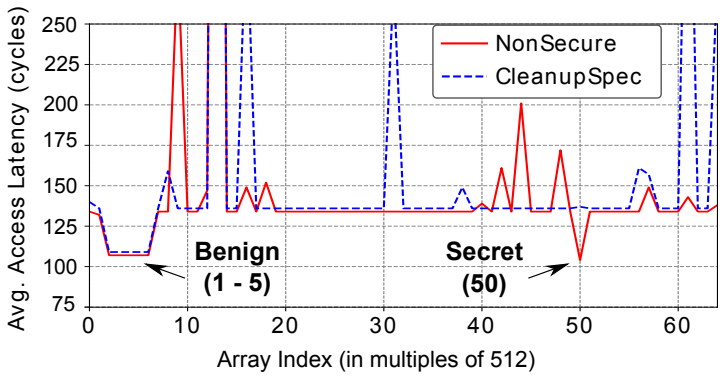


Figure 6.11: Average array access time for secret-inference phase of Spectre Variant-1. CleanupSpec has no latency difference for the secret index installed on the wrong path, while having identical behavior as non-secure for lines installed on the correct path.

is also observed for `array2[50 * 512]`, that leaks the secret value of 50 stored in `array1[malicious_x]`. On the other hand, CleanupSpec only has low latency for the benign accesses (installed on the correct path), and has no latency difference for the secret index (accessed on the wrong path). This is because, when a mis-speculation is detected after the wrong-path access, the cleanup operations restore the cache state and make the wrong-path accesses imperceptible. Thus, CleanupSpec prevents leakage of information.

6.6.2 Performance

Figure 6.12 shows the execution time of CleanupSpec, normalized to a non-secure baseline, for all 19 workloads and also the *Avg* or the geometric mean. On average, CleanupSpec incurs a slowdown of 5.1%. This is because CleanupSpec allows the loads to speculatively modify the cache and incurs no overheads for correctly speculated loads. As most workloads in Table 6.3 have branch mis-prediction rates of <10% and a majority of the loads are correctly speculated, CleanupSpec only incurs minimal slowdown. In Figure 6.12, we order the workloads h terms of branch mis-prediction rates (from highest to lowest) and observe that the workloads on the left-hand side with higher mis-prediction rates have the highest slowdowns (e.g. *astar (24%), bzip2 (11%)*). Whereas, workloads on the right side with the lowest mis-predictions have negligible slowdown (e.g *lbm, milc, libq*).

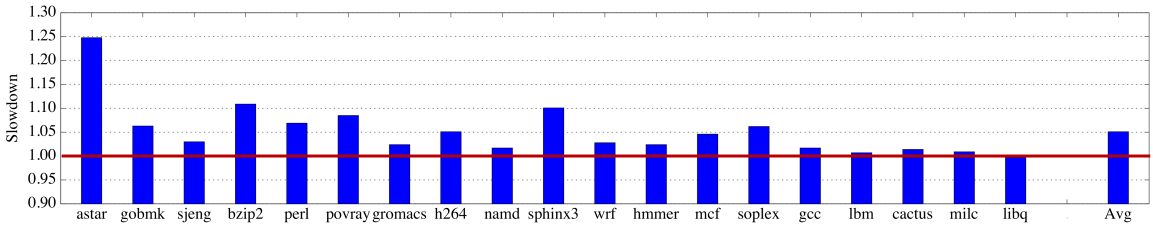


Figure 6.12: Execution time of CleanupSpec normalized to Non-Secure baseline. On average, CleanupSpec causes a slowdown of 5.1%.

Additionally, CleanupSpec incurs low overheads for workloads with high L1 data-cache hit rates. This is because CleanupSpec issues Restore or Invalidate requests only for squashed loads that install a *new* line on incurring a cache miss. The randomization-based approach (L1 rand replacement and L2 cache randomization) of CleanupSpec makes the hits benign, so no overhead is incurred for them. As the L1 data cache hit-rate is typically high (>95% for our workloads as shown in Table 6.3), many of the workloads on the left side of Figure 6.12 incur limited slowdowns (e.g. *gobmk*, *sjeng*), and the only workloads on the right of Figure 6.12 with slowdown (e.g. *sphinx3 (10%), soplex (7%))* have higher data cache miss-rates.

Thus, CleanupSpec provides low overheads in the common-case operation (high cache hit rate and accurate branch prediction) for most workloads. In the next section, we analyze the time required for the cleanup operations, to better understand the root cause of the overheads in the workloads with the maximum slowdown.

6.6.3 Main Cause of Slowdown - Cleanup Stalls

The slowdown in CleanupSpec is due to a core stall (no new instructions fetched), while a cleanup is in progress on a mis-prediction. This stall time depends on the frequency of squashes (Figure 6.13) and the stall time per squash (Figure 6.14). Note that it is necessary to first wait for inflight correct-path loads to complete, before starting cleanup operations (invalidation and restoration). This prevents interference of cleanup operations with inflight correct-path loads, that could leak information about locations undergoing cleanup, and

even prevents nested mis-predictions. On average, workloads have 20 squashes per 1000 instructions, with most of the stall time per squash (20 out of 25 cycles on avg.) spent waiting for inflight correct-path loads to retire. Only a small fraction (5 cycles) is needed for invalidation and restoration operations, on average.

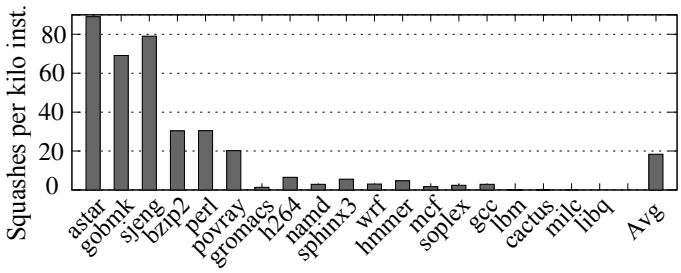


Figure 6.13: Squash frequency (per 1000 instructions). As squash frequency decreases (from left to right), the slowdown due to CleanupSpec also typically decreases.

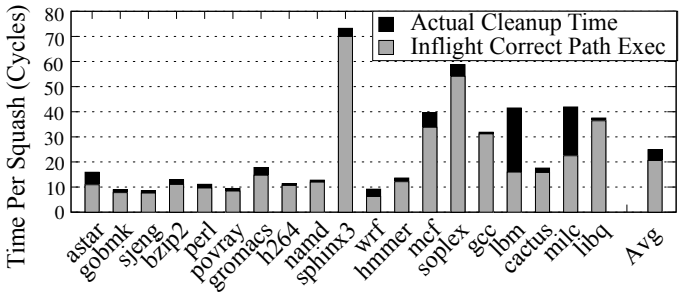


Figure 6.14: Stall time due to cleanup, per squash. Large fraction of the time is spent waiting for inflight correct-path instructions to complete, on every squash.

Workloads on the left (e.g. *astar*) have a short stall time per squash, but a high frequency of squashes due to high branch mis-prediction rates, and incur higher overheads. Workloads on the right have infrequent squashes, but those with high L1 data cache miss-rates (e.g. *mcf*, *sphinx3*, *soplex*) still have high stall time per squash while waiting for the correct path loads to execute, and consequently incur modest slowdowns (5%–10%). Outliers like *lbm* and *milc* need 20-25 cycles per squash to execute cleanup operations as they have a larger number of loads per squash that need cleanup (see Table 6.5), but they have no slowdown as squashes are uncommon.

Table 6.5: Cleanup statistics – Squash per kilo instruction (PKI), Loads/Squash, State of the load when squashed – Not issued (NI), L1-Hit (L1H), L2-Hit (L2H) or L2-Miss (L2M). Cleanup is needed only for Squashed Loads that are L2H or L2M.

Workload	**Squash PKI**	**Loads/ Squash**	**Squashed Loads (%)**			
			NI	L1H	L2H	L2M
astar	89.02	11.20	40	57	1.72	0.36
gobmk	69.10	4.27	52	45	0.57	0.43
sjeng	79.06	4.09	49	49	0.33	0.23
bzip2	30.37	8.01	48	50	1.00	0.11
perl	30.44	4.09	51	46	0.97	0.52
povray	20.20	5.71	50	48	0.13	0.07
gromacs	1.38	8.36	38	59	1.30	0.23
h264	6.53	6.46	57	41	0.36	0.18
namd	2.92	9.77	28	71	0.29	0.05
sphinx3	5.56	8.33	45	51	0.86	0.43
wrf	3.07	4.64	30	59	0.31	0.71
hmmer	4.68	15.09	41	58	0.33	0.07
mcf	1.68	17.51	68	28	0.22	0.87
soplex	2.42	11.49	29	67	0.57	0.75
gcc	2.90	4.59	60	38	0.16	0.12
lbm	0.08	24.51	52	39	0.36	3.63
cactus	0.10	13.26	37	60	0.36	0.42
milc	0.01	29.88	12	78	0.26	0.30
libq	0.00	1.37	70	23	0.00	0.36

6.6.4 Analysis of Loads Requiring Cleanup

Cleanup operations are only required for loads that were issued and had an L1-Miss. However, close to 50% of these loads on average are still in flight when a cleanup request is issued to the cache hierarchy, as shown in Figure 6.15. For such loads, it is sufficient to drop the pending request without any invalidation or restoration, as any changes to the cache like an install or eviction of a victim are only done when the request returns. We observe such low-overhead cleanups commonly for L2-misses, where branch mis-prediction is often detected before a memory request issues or before it completes.

6.6.5 Comparison with InvisiSpec

Table 6.6 compares the performance of CleanupSpec with InvisiSpec, all normalized to a non-secure baseline. Our initial estimates using the public InvisiSpec code-base showed

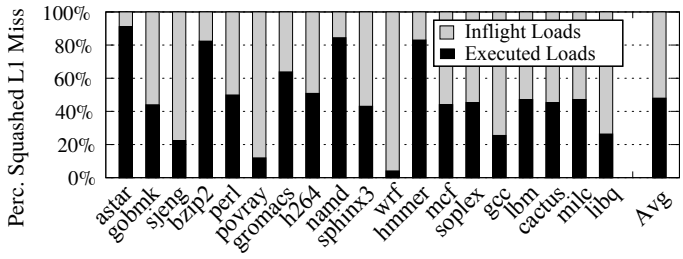


Figure 6.15: Breakup of Loads Cleaned-up (L1-Misses). For squashing Inflight Loads (50% L1-Misses), pending requests can be dropped without invalidation or restoration.

that it incurs an average slowdown of 67.5% across 19 SPEC benchmarks we used (close to the slowdown reported in [34]). Subsequently, the authors of InvisiSpec intimated us through an email (M. Yan, Personal Communication, August 19, 2019) that they had an updated implementation incurring only 15% slowdown. According to the authors, the difference was due to "a simulation bug that incorrectly delayed the propagation of speculatively accessed data to dependent instructions". The initial estimate incorrectly delayed the propagation till the visibility point of the speculative load, whereas the revised results propagate the data to dependents as soon it is received by the speculative load. Nevertheless, both InvisiSpec implementations incur extra accesses to update the cache for correctly speculated loads.

In comparison, CleanupSpec only incurs 5.1% slowdown, because it incurs no extra accesses for correctly speculated loads, which make up the common-case. Even on a misspeculation, no extra accesses are required for loads that were not issued, or loads that were issued and had L1-Hit – these make up >98% of squashed loads, as shown in Table 6.5. Only on squashed L1-misses, the overhead of extra cleanup accesses is incurred. Additionally, messages to extend the preservation of SEFE entries in the speculation window are only required for loads that do not commit/squash within 200 cycles (which is uncommon). This allows CleanupSpec to incur <2% extra accesses to the cache hierarchy and incur lower overheads.

Table 6.6: Overheads for CleanupSpec vs InvisiSpec, normalized to a Non-Secure baseline.

Configuration	**Slowdown**
InvisiSpec (initial estimates close to [34])	67.5%
InvisiSpec (revised results as per its authors)	15%
CleanupSpec	5.1%

6.6.6 Storage Overhead

We require extra storage for the Side-Effect Entries (SEFE), associated with each LQ entry at the core-side and with each L1/L2-MSHR entry at the cache-side, that track the side-effects of the loads as they execute. Each SEFE in L2-MSHR occupies 2-bytes (3 status bits, a 5-bit Load-ID, a 8-bit Epoch-ID), while each SEFE in L1-MSHR/LQ occupies 7-bytes (with an extra 40-bit L1 evicted line address), as shown in Figure 6.7. For a configuration with 32 LQ and 64 L1/L2-MSHR entries per core, our design incurs a overhead of <1KB per core, scaling linearly with LQ and L1/L2-MSHR entries.

6.7 Related Work

6.7.1 Types of Speculation-Based Attacks

Numerous speculation-attacks have been demonstrated that break intra-process, inter-process and trust-domain isolation in software. These differ based on the speculation-mechanism exploited: exception-based attacks (e.g. Meltdown [6], Foreshadow [53]) exploit race-condition between illegal data access and permission-check, whereas control or data speculation-based attacks (e.g. Spectre [5], SSB [153]) exploit speculation to bypass permission checks. Many subsequent variants of these attacks [52, 154, 155, 156, 72, 55, 157] have also emerged in recent years. CleanupSpec mitigates these attacks by breaking the cache channel used to transmit information. We invalidate transient installs after mis-speculation, preventing flush-based attacks [23, 8] and randomize the L2/Directory and restore L1 state to prevent eviction-based attacks [38, 95, 158, 112].

6.7.2 Software and Microcode Based Defenses

Software mitigations can prevent speculative access to secrets by unmapping them (*e.g.*, KAISER [128]) or by disabling speculation in unsafe contexts (e.g. Retpoline [129], Memory Fences [159], Intel's microcode patches [130]). Unfortunately, many of these mitigations require SW or OS changes [160] and are incompatible with legacy code. Recent studies [144] show that some commercially deployed SW mitigations can have up to 50% slowdown. In contrast, our defense has low overheads and requires no SW changes.

6.7.3 Hardware-Based Defenses

Redo-based Mitigations

InvisiSpec [34] represents a *Redo-based* approach to safe speculation, whereby the load instruction is done twice: first time to get the value and second time to change the cache state. **Safespec** [135] is a similar mitigation for single-core systems that buffers transient changes to caches, until speculation is resolved. In contrast, we propose a *Undo-based* approach to safe speculation that incurs minimal overheads and does not require buffering data, making it more practical for adoption.

Delay-based Mitigations

Context Sensitive Fencing [133] (CSF) and **Conditional Speculation** [132] (CS) mitigate Spectre variants in hardware by delaying potentially *unsafe* loads, until speculation is resolved and they are deemed *safe*. However, these mitigations are limited to attacks exploiting control/data speculation that leak secrets stored in memory. For example, CSF uses taint-tracking to identify and delay kernel loads dependent on user-space data, that might potentially access secrets. CS uses heuristics to filter a subset of L1-cache misses that could potentially leak data, and delays them. In comparison, CleanupSpec prevents information leakage through *any* speculative load, while incurring two-third of the slowdown of CS

and CSF. A contemporary work **Context** [161] leverages taint-tracking to determine the secret-dependent instructions in a program and modifies the processor to avoid speculation on such instructions, delaying their execution until they are non-speculative. CleanupSpec can be combined with CS, CSF, or Context, to selectively perform clean-up operations only when *unsafe* or *secret-dependent* loads execute, to achieve even lower slowdown.

Another contemporary work leverages **Value Prediction** [134] to continue performing computations, despite delayed cache accesses. However, such a proposal only benefits workloads with significant value locality and still incurs close to 10% slowdown on average. Other contemporary works like **NDA** [162] and **SpecShield** [163] delay any usage of speculatively accessed data until speculation resolves, resulting in slowdown upwards of 20%. In comparison, CleanupSpec incurs lesser slowdown since it allows speculative usage of data and only penalizes mis-speculated loads that are uncommon.

Partitioning-based Mitigations

DAWG [16] proposed a hardware software co-design to prevent information leakage through cache hits or misses on lines shared between a victim and spy, by way-partitioning the cache and duplicating such shared cache lines in each partition. However, this requires all software to be rewritten utilizing protection-domains, so that they may be mapped to separate cache partitions. Additionally, it hinders inter-process communication. In contrast, we provide a software transparent solution that also protects legacy applications.

MI6 [14] proposed a defense providing strong isolation guarantees for code inside enclaves. It leverages mechanisms like spatial isolation (set partitioned L2/LLC) and temporal isolation (flushing L1 on enclave entry/exit) to prevent speculation-based attacks using caches. However, it is limited by its requirement for a HW/SW infrastructure supporting enclaves, and faces scalability challenges due to its static cache set-partitioning between enclaves. In comparison, our design is more scalable as it allows efficient sharing of LLC (no partitioning), requires no software support, and has a lesser slowdown.

Defenses against non-speculative side-channel attacks

Other proposals mitigate cache side channels [18, 7, 164, 84, 83, 9, 10, 11, 76, 19] in a non-speculative setting – we build on these to prevent speculation-based attacks from exploiting cache covert channels for leaking data.

6.8 Potential Impact of this Research

CleanupSpec provides a practical approach to undo transient cache state changes to protect against transient execution attacks. As CleanupSpec incurs overheads only for wrong-path loads and only those that miss in the L1 cache, it only incurs a minimal slowdown of 5.1% and a minimal cost of <1 kilobyte per core storage overhead. As a result, it provides a low overhead solution to prevent transient cache state changes from leaking information.

In addition to its low overhead, CleanupSpec does not require significant changes to the memory system and consistency protocols (unlike prior works). Moreover, it reuses features like randomized caches that are likely to be adopted as a solution for non-speculative side-channels in future CPUs. As a result, CleanupSpec is ripe for future adoption as an extension of such features, to harden the memory system against transient execution attacks.

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Concluding Remarks

Processor caches enable fast access to data and are a major source of performance gains in modern processors. Unfortunately, cache side-channel attacks arising from sharing of processor caches allow malicious programs to leak sensitive information causing security and privacy breaches. Cache interactions between a victim and a malicious process have been shown to leak encryption keys, user-sensitive data such as browsing histories, and even confidential intellectual property such as machine-learning models.

Unfortunately, current strategies to defend against such attacks suffer from practical challenges. On the one hand, successive defenses that rely on cache randomization have been broken by adaptive attacks. On the other hand, several defenses that rely on cache partitioning have been found to have practical challenges like limited scalability and flexibility. This thesis addresses the gap in the understanding of the capabilities of current attacks and the lack of principled yet practical defenses. It develops faster attacks with fewer limitations than prior attacks, and uses this understanding to develop principled defenses capable of future-proof security, and practical defenses that are scalable and low-cost.

First, this thesis investigates the capabilities of attackers and in the process develops a new cache covert channel attack called Streamline, that is considerably faster than current state-of-the-art attacks. Moreover, it works without the requirement of flush instructions, which restrict prior fast attacks to the x86 ISA. Streamline is currently the fastest known cache covert channel attack that reaches bit-rates of more than 1MB/s, while being applicable to a broader class of processors than prior attacks.

Second, this thesis develops a principled and practical defense using cache randomization, called MIRAGE. MIRAGE provides the security of a fully-associative design and eliminates set-conflict based cache side channels for system lifetime while preserving practical set-associative lookups. Thus, it provides principled security at modest costs of 17-20% extra SRAM storage and 2% slowdown. While 5 randomized cache defenses were broken by adaptive attacks between 2018-2020, MIRAGE has remained unbroken so far.

Third, this thesis explores a practical cache partitioning defense to eliminate all potential cache side channels, with Bespoke Cache Enclaves. Bespoke provides 500+ simultaneous partitions of customizable sizes, surpassing practical limitations of prior works using way or set-partitioning, which only provide a small number of partitions or partitions that are inflexible in size. It provides such scalable cache isolation and side-channel resilience while incurring negligible storage overheads (2%) and slowdown (1%). Moreover, Bespoke's isolated cache allocations benefit not just security but also performance: the fine-grain customizable cache allocations can improve the performance of applications by up to 40%, compared to prior works.

Finally, this thesis develops a practical and low-cost technique to harden caches against new forms of side-channel leakage via transient execution attacks like Spectre and Meltdown. This work develops CleanupSpec, a low overhead approach to roll back or randomize speculative changes to the cache to prevent exploitation of caches in such attacks. This technique hardens the memory system against transient attacks with minimal performance impact (5% slowdown), at a modest storage overhead of 1 kilobyte per core.

In summary, this thesis presents a principled yet practical approach to architect secure processor caches. The solutions developed in this thesis enable security against side channels for future processors while retaining a majority of the performance and practicality benefits of shared caches. Future processors can deploy such practical techniques to support confidential computing in shared systems without prohibitive costs.

7.2 Future Works

In future works, this research provides opportunities for extensions in several directions:

1. **Cache partitioning in real-world applications like web browsers.** Web browsers like Chrome have features like site isolation [48] to insulate the data of websites from each other. This sandboxes data of different websites in independent processes and protects them from untrusted websites rendering malicious content. However, such websites continue to share cache resources in the processor and are vulnerable to data leakage via cache side channels. Cache partitioning is challenging to leverage for browsers as users tend to open tens of web-pages at a time, requiring tens to hundreds of cache partitions. Future works can explore how fine-grain partitions using BCE primitives can be leveraged to enable low-cost cache isolation in browsers.

2. **Low-Cost Randomization For Caches.** State-of-the-art cache designs like Mirage suffer from performance costs due to an increase in latency of cache access and storage costs due to the extra tags. Future research can reduce the cache access latency by exploring faster set-indexing functions using simpler hashes (e.g. reduced round encryptors) that may be computed in fewer cycles. If such hashes can provide sufficient randomization, even if an adversary learns the mapping, and attempts a conflict-based attack, such an attack can be detected as it requires set-associative evictions, which are otherwise improbable. Similarly, reducing the extra tags can also allow an adversary to learn the mappings more easily. However, attacks can still be detected if the set-associative evictions occur more frequently than the expected values as per the stochastic modeling of the cache. Future works can explore such optimizations for Mirage in combination with attack detection.

3. **Defending Against Contention on Cache Interconnects:** Current secure cache defenses prevent information leakage through the cache state either by randomizing or

by isolating it from a malicious program. However, new stateless attacks have recently emerged which exploit contention on the cache interconnect between cache slices. Attacks like LOTR [115] on the ring interconnect and MeshUp [116] on the mesh interconnect succeed despite existing cache randomization or partitioning based defenses, because the interconnect is shared between cache request and response packets of different processes even with these defenses. Future research can explore how these randomization and partitioning techniques can be extended to network-on-chips (NoC), with techniques like spatial and temporal partitioning of NoC link bandwidth and randomized packet routing algorithms for NoCs.

4. **Automated Methodologies for Security Evaluation.** In recent years, there have been a series of defenses [34, 35, 134, 165, 166] against transient execution attacks. Each defense provides slightly differing security properties and invariants to prevent transient execution attacks. However, there have also been a series of adaptive attacks [167, 168, 166] on these defenses, exploiting corner cases where leakage is still possible despite these defenses. Thus, there is a need to develop automated methodologies to analyze the security of existing defenses, to detect any adaptive attacks. One promising technique is to deploy fuzzing techniques [169] on Gem5 implementations of defenses, to detect corner cases where the security invariant is violated. Future works can explore new automated methodologies to mutate inputs in such fuzzers, get better coverage of execution patterns, and detect potential vulnerabilities in existing defenses, to help end the arms race in micro-architectural security.

REFERENCES

[1] McAfee, "New McAfee Report Estimates Global Cybercrime Losses to Exceed $1 Trillion," 2020, https://ir.mcafee.com/news-releases/news-release-details/new-mcafee-report-estimates-global-cybercrime-losses-exceed-1.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Security and Privacy (SP)*, 2015.

[3] A. Shusterman *et al.*, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[4] M. Yan, C. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn dnn architectures," in *USENIX Security*, 2020.

[5] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE Security and Privacy (SP)*, 2019.

[6] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

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

[8] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack.," in *USENIX Security*, 2014.

[9] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via dynamically encrypted address," in *MICRO*, 2018.

[10] ——, "New attacks and defense for encrypted-address cache," in *ISCA*, 2019.

[11] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *USENIX Security*, 2019.

[12] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE Security and Privacy (SP)*, 2020.

[13] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches," in *MICRO*, 2020.

[14] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *MICRO*, 2019.

[15] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *ACM workshop on Cloud computing security*, 2009.

[16] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *MICRO*, 2018.

[17] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007.

[18] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan. 2012.

[19] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure dynamic cache partitioning for efficient timing channel protection," in *DAC*, 2016.

[20] F. Liu *et al.*, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[21] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *DSN-W*, 2011.

[22] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *USENIX Security*, 2012.

[23] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: A fast and stealthy cache attack," in *DIMVA*, 2016.

[24] G. Saileshwar, C. Fletcher, and M. Qureshi, "Streamline: A Fast, Flushless Cache Covert-Channel Attack by Enabling Asynchronous Collusion," in *ASPLOS*, 2021.

[25] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization," in *NDSS*, 2020.

[26] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18, 2018.

[27] R. Bodduna, V. Ganesan, Patanjali, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE CAL*, 2020.

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

[29] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE Security and Privacy (SP)*, 2019.

[30] 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.

[31] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design," in *USENIX Security*, 2021.

[32] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[33] 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)*, IEEE, 2021, pp. 37–49.

[34] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 428–441.

[35] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[36] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[37] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[38] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *CT-RSA'06*, 2006.

[39] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *CCS*, 2009.

[40] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ abort: A timer-free high-precision l3 cache attack using Intel TSX," in *USENIX Security*, 2017.

[41] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proceedings of the Linux Symposium*, 2009.

[42] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015.

[43] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *ESORICS*, 2019.

[44] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security*, 2020.

[45] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *HPCA*, 2020.

[46] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *AsiaCCS*, 2020.

[47] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+ Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[48] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *USENIX Security*, 2019.

[49] M. Schwarz, M. Lipp, and D. Gruss, "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks," in *NDSS*, 2018.

[50] J. Wampler, I. Martiny, and E. Wustrow, "ExSpectre: Hiding Malware in Speculative Execution," in *NDSS*, 2019.

[51] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-cores cache covert channel," in *DIMVA*, 2015.

[52] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[53] J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[54] O. Weisse *et al.*, "Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution," Tech. Rep., 2018.

[55] C. Canella *et al.*, "A systematic evaluation of transient execution attacks and defenses," *arXiv preprint arXiv:1811.05441*, 2018.

[56] S. Van Schaik *et al.*, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 88–105.

[57] M. Schwarz *et al.*, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[58] C. Canella *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 769–784.

[59] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think," in *USENIX Security*, 2017, pp. 1075–1091.

[60] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 131–145.

[61] K. Viswanathan, *Disclosure of hardware prefetcher control on some Intel processors*, https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html, (accessed August 1, 2020).

[62] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security*, 2020.

[63] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ISCA*, 2010.

[64] C. King, *Ubuntu wiki: Stress-ng*, https://wiki.ubuntu.com/Kernel/Reference/stress-ng, (accessed January 15, 2021).

[65] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Security*, 2016, pp. 549–564.

[66] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *DIMVA*, 2017.

[67] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-speculative load-load reordering in TSO," in *ISCA*, 2017.

[68] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2014, pp. 216–228.

[69] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" In *HPCA*, 2018.

[70] C. Fletcher, M. Tiwari, M. Yan, M. El Hajj, S. Wei, and Y. Shalabi, *Covert channel attack tutorial at ISCA 2019*. https://github.com/yshalabi/covert-channel-tutorial, 2019.

[71] CPU-Monkey, *Comparing intel xeon e3-1270 v5 vs intel core i7-4790*, https://www.cpu-monkey.com/en/compare_cpu-intel_xeon_e3_1270_v5-601-vs-intel_core_i7_4790-355, (accessed January 15, 2021).

[72] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018.

[73] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[74] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *International Symposium on Engineering Secure Software and Systems*, Springer, 2016, pp. 138–154.

[75] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks.," *IACR Cryptology ePrint Archive*, vol. 2017, p. 564, 2017.

[76] M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: detecting cache-based covert channel attacks using record and replay," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[77] A. Fuchs and R. B. Lee, "Disruptive prefetching: Impact on side-channel attacks and cache designs," in *Proceedings of the 8th ACM International Systems and Storage Conference*, 2015, pp. 1–12.

[78] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[79] 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.

[80] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks," in *30th USENIX Security Symposium (USENIX Security 21)*, https://github.com/vusec/fpvi-scsb/blob/main/fpvi_firefox_exploit/fpvi.js, 2021.

[81] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic prime+ probe attacks and covert channels in scattercache," *arXiv:1908.03383*, 2019.

[82] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments," in *USENIX Security*, 2020.

[83] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007.

[84] ——, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008.

[85] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way cache: demand-based associativity via global replacement," in *ISCA*, 2005.

[86] J. H. Edmondson *et al.*, "Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor," *Digital Technical Journal*, vol. 7, no. 1, 1995.

[87] D. Weiss, J. J. Wuu, and V. Chin, "The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1523–1529, 2002.

[88] D. H. Albonesi, "An architectural and circuit-level approach to improving the energy efficiency of microprocessor memory structures," in *VLSI: Systems on a Chip*, Springer, 2000, pp. 192–205.

[89] A. Seznec, "A case for two-way skewed-associative caches," in *ISCA*, 1993.

[90] J. Borghoff *et al.*, "PRINCE–a low-latency block cipher for pervasive computing applications," in *ASIACRYPT*, 2012.

[91] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM journal on computing*, vol. 29, no. 1, pp. 180–200, 1999.

[92] B. Vöcking, "How asymmetry helps load balancing," *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 568–589, 2003.

[93] D. Lilja, "Measuring computer performance," in Cambridge University Press, 2000, pp. 228–229.

[94] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 353–364.

[95] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 368–379.

[96] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[97] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *European Symposium on Algorithms*, Springer, 2001, pp. 121–133.

[98] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X, 2002.

[99] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[100] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[101] J. Harttung, *PRINCE block cipher - VHDL implementation*, https://github.com/huljar/prince-vhdl.

[102] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *ISPD*, 2015.

[103] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000.

[104] R. Avanzi, "The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory

central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, pp. 4–44, 2017.

[105] WikiChip, *POWER9 - Microarchitectures - IBM*, https://en.wikichip.org/wiki/ibm/microarchitectures/power9.

[106] B. Goel and S. A. McKee, "A methodology for modeling dynamic and static power consumption for multicore processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2016, pp. 273–282.

[107] AnandTech, *Intel 9th Generation Power Consumption*, https://www.anandtech.com/show/13400/intel-9th-gen-core-i9-9900k-i7-9700k-i5-9600k-review/21.

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

[109] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*, IEEE, 2000, pp. 107–116.

[110] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, IEEE, 2011, pp. 169–180.

[111] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: A secure directory to defeat directory side-channel attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 332–345.

[112] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 888–904.

[113] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[114] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[115] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical," in *USENIX Security*, 2021.

[116] J. Wan, Y. Bi, Z. Zhou, and Z. Li, "MeshUp: Stateless Cache Side-channel Attack on CPU Mesh," in *IEEE Symposium on Security and Privacy (SP)*, 2022.

[117] J. T. Butler and T. Sasao, "Fast hardware computation of x mod z," in *IPDPS Workshops and PhD Forum*, 2011.

[118] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[119] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

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

[121] B. C. Schwedock and N. Beckmann, "Jumanji: The Case for Dynamic NUCA in the Datacenter," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 665–680.

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

[123] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *HPCA*, 2018.

[124] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, Portland, Oregon, 2012, pp. 428–439, ISBN: 9781450316422.

[125] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, IEEE, 2008, pp. 367–378.

[126] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: A dynamic cache partitioning system using page coloring," in *PACT*, 2014.

[127] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 213–224.

[128] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *International Symposium on Engineering Secure Software and Systems*, Springer, 2017, pp. 161–176.

[129] P. Turner, *Retpoline: a software construct for preventing branch-target-injection*, https://support.google.com/faqs/answer/7625886, (Accessed: December 1, 2018), 2018.

[130] Intel, *Speculative Execution Side Channel Mitigations*, https://software.intel.com/security - software - guidance / api - app / sites / default / files / 336996 - Speculative - Execution-Side-Channel-Mitigations.pdf, (Accessed: December 1, 2018), 2018.

[131] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," 2019.

[132] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks," in *High Performance Computer Architecture (HPCA), 2019 IEEE International Symposium on*, IEEE, 2019.

[133] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019.

[134] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the 46th International Symposium on Computer Architecture*, ACM, 2019, pp. 723–735.

[135] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the Design Automation Conference (DAC)*, 2019.

[136] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. Garcia, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 870–887.

[137] R. Lee, *Security aware microarchitecture design*, Keynote at the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Fukuoka, Japan, Oct. 2018.

[138] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[139] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks.," in *USENIX Security Symposium*, 2016, pp. 565–581.

[140] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, IEEE, 2012, pp. 142–151.

[141] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud.," in *USENIX Security symposium*, 2012, pp. 159–173.

[142] Intel, *Intel Analysis of Speculative Execution Side Channels*, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, (Accessed: December 1, 2018), 2018.

[143] Zdnet.com, *After big Linux performance hit, Spectre v2 patch needs curbs*, https://www.zdnet.com/article/linus-torvalds-after-big-linux-performance-hit-spectre-v2-patch-needs-curbs/, (Accessed: December 1, 2018), 2018.

[144] Phoronix, *Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower*, https://www.phoronix.com/scan.php?page=article&item=linux-420-bisect&num=1, (Accessed: December 1, 2018), 2018.

[145] M. Yan, *Invisispec 1.0*, https://github.com/mjyan0720/InvisiSpec-1.0/tree/39cfb858d4b2e404282b54094f0220b8098053f6, (Accessed: December 1, 2018), 2018.

[146] M. Cekleov and M. Dubois, "Virtual-address caches. part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5, 1997.

[147] C. Bienia, "Benchmarking modern multiprocessors," in *Ph.D. Thesis, Princeton University*, 2011.

[148] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH computer architecture news*, ACM, vol. 23, 1995, pp. 24–36.

[149] T. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov. 2011, pp. 1–12.

[150] G. Paoloni, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*, https://www.intel.com/content/dam/www/public/us/

en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, 2010.

[151] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[152] E. August, *Spectre example code on erikaugust github repository*, https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6, (Accessed: March 19, 2019), 2018.

[153] J. Horn, *Speculative Execution, Variant 4: Speculative Store Bypass*, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, (Accessed: December 1, 2018), 2018.

[154] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[155] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution.(2018)," *arXiv preprint arXiv:1802.09085*, 2018.

[156] S. Deng, W. Xiong, and J. Szefer, "Cache timing side-channel vulnerability checking with computation tree logic," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, ACM, 2018.

[157] ——, "Analysis of secure caches and timing-based side-channel attacks," *IACR Cryptology ePrint Archive*, vol. 2019, p. 167, 2019.

[158] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 38–55.

[159] I. Corporation. "Intel® 64 and IA-32 Architectures Software Developer's Manual." (Accessed: December 1, 2018). (2018).

[160] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *USENIX Security*, 2018.

[161] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre.," in *NDSS*, 2020.

[162] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikc, "Nda: Preventing speculative execution attacks at their source," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[163] K. Barber, L. Zhou, A. Bacha, Y. Zhang, and R. Teodorescu, "Isolating speculative data to prevent transient execution attacks," *IEEE Computer Architecture Letters*, 2019.

[164] M. Kayaalp *et al.*, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017.

[165] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[166] K. Loughlin *et al.*, "DOLMA: Securing Speculation with the Principle of Transient Non-Observability," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1397–1414.

[167] M. Behnia *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 1046–1060.

[168] M. Li, C. Miao, Y. Yang, and K. Bu, "Unxpec: Breaking undo-based safe speculation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2022, pp. 98–112.

[169] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 226–239.

VITA

Gururaj Saileshwar grew up in Mumbai, India. He completed a Bachelor of Technology in Electrical Engineering and a Master of Technology in Electrical Engineering with a specialization in Microelectronics, from the Indian Institute of Technology - Bombay (IIT-B), Mumbai, India, in 2014. Saileshwar completed his PhD in Electrical and Computer Engineering from the Georgia Institute of Technology, Atlanta, USA in 2022, where he was advised by Prof. Moinuddin Qureshi.

Saileshwar's research interests are broadly in the areas of computer architecture and system security, with a focus on improving the security, performance, and efficiency of future computing hardware and systems. His research has been published at several top computer architecture and security conferences. Saileshwar's PhD research has been supported in part by the Georgia Tech IISP Cybersecurity Fellowship and the Georgia Tech ECE Bourne Fellowship. His works have been awarded an IEEE Micro Top Picks (Honorable Mention) and an IEEE HOST 2022 Best PhD Dissertation Award.