CACHE-BASED SIDE CHANNELS: MODERN ATTACKS AND DEFENSES

BY

MENGJIA YAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Assistant Professor Christopher W. Fletcher
Professor Darko Marinov
Professor Joel Emer, Massachusetts Institute of Technology/NVIDIA
Professor Ruby B. Lee, Princeton University
Assistant Professor Adam Morrison, Tel Aviv University

ABSTRACT

Security and trustworthiness are key considerations in designing modern processor hardware. It has been shown that, among various data leakage attacks, *side channel attacks* are one of the most effective and stealthy ones. In a side channel attack, an attacker can steal encryption keys, monitor keystrokes, or reveal a user's personal information by leveraging the information derived from the side effects of a program's execution. These side effects include timing information, micro-architecture states, power consumption, electromagnetic leaks and even sound.

This thesis studies the important type of micro-architecture side channel attacks that exploit the shared cache hierarchies. Recently, we have witnessed ever more effective cache-based side channel attack techniques and the serious security threats posed by these attacks. It is urgent for computer architects to redesign processors and fix these vulnerabilities promptly and effectively.

We address the cache-based side channel security problems in two ways. First, as modern caches are temporally and spatially shared across different security domains, the shared cache hierarchy offers a broad attack surface. It provides attackers a number of ways to interfere with a victim's execution and cache behavior which, in turn, significantly increase side channel vulnerabilities. We study the role of cache interference in different cache attacks and propose effective solutions to mitigate shared cache attacks by limiting malicious interference.

According to our analysis, in a multi-level cache hierarchy, creating "*inclusion victims*" is the key in a successful attack, since they give an attacker visibility into a victim's private cache. Based on this important observation, we present a secure hierarchy-aware cache replacement policy (*SHARP*) to defeat cache attacks on inclusive cache hierarchies by eliminating inclusion victims. In addition, we show that inclusion victims also exist in non-inclusive cache hierarchies, and that the non-inclusive property is insufficient to stave off cache-based side channel attacks. We design the first two conflict-based cache attacks targeting the directory structure in a non-inclusive cache hierarchy. We prove that the *directory* structure is actually the unified attack surface for all types of cache hierarchies, including inclusive, non-inclusive and exclusive ones. To address this problem, we present the first scalable secure directory design (*SecDir*) to eliminate inclusion victims by restructuring the directory organization.

Second, cache-based side channel attacks play an important role in *transient* execution attacks, leading to arbitrary information leakage and the violation of memory isolation.

Specifically, in transient execution attacks, speculative execution induces the execution of instructions in incorrect paths. Such instructions potentially access secrets, leaving side effects in the cache hierarchy before being squashed. We study how to effectively defend against transient execution attacks on cache hierarchies by hiding the side effects of transient load instructions. We call our scheme "Invisible Speculation" (*InvisiSpec*). It is the *first* robust hardware defense mechanism against transient cache-based side channel attacks for multiprocessors.

For my family.

ACKNOWLEDGMENTS

This Ph.D. journey is a precious chapter in my life. I am really happy that, after the six years, I have become a more mature researcher and a more persistent person, and I am still passionate about research and life as before. This Ph.D. journey is full of challenges. I could not make it here without the support and help from many people.

First and foremost, I am grateful to my advisor Josep Torrellas. Josep kindly accepted me as a member in the i-acoma research group, when I was naive about research and had a very weak technical background. I learned almost everything about research from him. Recently, I even found myself sometimes thinking in his way and speaking in his tone. Besides all the guidance I got from him, I really appreciate that he was extremely patient with me. He always patiently listened to my ideas, even the premature and incorrect ones, and constantly encouraged me to speak up during discussions. Especially in the first three years, when I made slow progress and felt upset, Josep provided a supportive environment for me to develop skills and build confidence. Learning how to do research can take a very long time. If Josep had not been that nice to me, and if he had not allowed me to take time, I might have given up. Thanks Josep for all the support.

Second, I would like to thank Chris Fletcher, my mentor and closest collaborator. My research productivity suddenly boosted up after Chris joined the University of Illinois at Urbana-Champaign (UIUC). It was great fun to work with Chris on all the projects. Many times when I came up with a small idea and went to discuss with Chris, he could see strong merits in the idea and provide insightful feedback to transform the idea into a revolutionary one. He is quick-minded, super positive and full of energy. More importantly, he knows the magic of turning research into an enjoyable thing. Thanks Chris for showing me this new aspect of research.

Besides, I am fortunate to have Adam Morrison, Joel Emer, Ruby Lee and Darko Marinov serve on my thesis committee. All of them played important roles in my Ph.D. journey. Adam has been a very close collaborator and has provided a lot of valuable feedback to my work. He was the first person who pointed out the consistency problem in an InvisiSpec-style defense solution. Joel is my role model. His Gandalf-style research philosophy and noble character have influenced me ever since I worked with him. Ruby's pioneering works on cache-based side channels have led me to this research problem and inspired multiple of my works. I had a lot of fun working with Darko, and I was always impressed by his humour and unique style of mentoring. If possible, I wish I could keep working and learning from them in the future.

Next, I would like to thank my colleagues in the i-acoma group. It was great fun to spend the six years with these nice people, namely Bhargava Gopireddy, Dimitrios Skarlatos, Yasser Shalabi, Thomas Shull, Jiho Choi, Wooil Kim, Raghavendra Pothukuchi, Antonio Garcia, Zhangxiaowen Gong, Azin Heidarshenas, Apostolos Kokolis and Serif Yesil. Together, we fought for paper deadlines and "cried" for paper rejections. I'm grateful to them for being my colleagues and friends. I would like to especially thank Yasser, Bhargava and Tom for helping me submit my first MICRO paper. In the last several hours before the MICRO submission deadline in 2016, I could not do much since I got a serious headache. They were nice enough to offer help, working on my paper until the last minute to fix as many problems as possible. The submission could not have happened without their generous help. Thanks Yasser, Bhargava and Tom. I hope the young students and future students will also be able to leverage the support and continue the great work in the i-acoma group.

I also want to express my appreciation to a lot of people outside of the i-acoma group who have helped me to complete this thesis. I would like to thank Fangfei Liu, who impressed me with her passion about side-channel attacks and defenses when we first met. A brief conversation with her triggered my initial interest in this problem. I thank Neil Wen and Read Sprabery for being co-authors of my papers. They are amazing and I learned a lot about system and software engineering from them. I thank Professor Michael Bailey for offering the best security course (CS461) in the department. The course showed me the fascinating world of security problems. Also, I would like to thank Nima Honarmand for his help in the very first project that I participated in. Nima's quest for perfection turned out to be the best training for me on conducting academic experiments and data collection. Special thanks to my dear friends, Tarun Prabhu, Bilge Acun, Pinlei Chen, Yunhui Long and Ying Chen, who helped me survive the extremely cold winters in Champaign.

Finally, I would like to thank my boyfriend Xuhao and my parents. Xuhao and I have been long-distance with twelve-hour time difference for the past four years, but he made me feel that he was always around me. He encouraged me to think bigger and motivated me to work harder. I might have delayed my graduation without his encouragement. Also, I would like to thank my parents for their unconditional love and support. Words cannot express my appreciation for what they have done for me since my birth. I know that they are always proud of me no matter what I achieve.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Security and trustworthiness are key considerations in designing modern processor hardware. It has been shown that, among various data leakage attacks, *side channel attacks* are one of the most effective and stealthy ones. The potential threats of side channel attacks were discussed by Butler W. Lampson in the early 1970s [1]. However, the problem did not get high attention in the computer architecture community until the late 2000s, when the first cache-based side channel attack successfully broke the AES encryption algorithm [2]. In the beginning of 2018, these attacks suddenly became an urgent threat after the disclosure of the first two transient execution attacks, i.e., Spectre [3] and Meltdown [4]. These attacks can completely break the fundamental software security mechanism: memory isolation. Important secret information can be leaked in these attacks, such as encryption keys in a cryptography algorithm, passwords stored in a password manager or browser, personal photos, emails, and personal health information.

To the best of our knowledge, there does not exist a strict definition for side channel attacks. Generally speaking, in computer security, a side channel attack leaks information from the *side effects* of a victim program's execution on a computer system. These side effects include timing information, micro-architecture states, power consumption, electromagnetic leaks, and even sound. For instance, in a keyboard acoustic side channel attack, an adversary learns what a victim is typing based on the sound produced by keystrokes.

This thesis is concerned with the important type of side channel attack that exploits the shared cache hierarchies. This attack is called *cache-based side channel attack* (or *cache attack* for short). Among all the side channels, caches offer one of the most broad and problematic attack surfaces. In a cache-based side channel attack [5–7], an attacker obtains secret information from a victim based on the interaction between victim's execution and cache states. More specifically, most attacks exploit the difference in the access times of cache hits and misses. It is extremely challenging to eliminate cache side channels efficiently, mainly because caches are essential to processor performance, and timing difference is an intrinsic property of cache structures [8].

1.1 THE PROBLEM: CACHE-BASED SIDE CHANNEL ATTACKS

There exist many variations of cache-based side channel attacks. To better understand what exactly a cache attack is, we start with an intuitive example, and then summarize the common attack procedure.

Algorithm 1.1: Victim code in a cache-based side channel attack example.

```
  Input: secret $s$, public array $parray$
1 Function victim($s$):
2     load $parray[s]$
3 End Function
```

In our example, the attacker and the victim are in different security domains. The attacker and the victim share a public data array *parray*, but only the victim can access the secret value s. The size of each element in *parray* is the same as the size of a cache block. Algorithm 1.1 lists the snippet of code executed by the victim. The victim uses the secret value s as an index to access the public array *parray*. Assume that the cache is initially empty and all the elements in *parray* are located in main memory. When the victim executes the memory access instruction (line 2), the processor inserts the cache line addressed by $parray[s]$ into the cache. As a result, the cache state is changed by the victim in a secret-dependent way.

The attacker in this example tries to steal the secret value s using a cache-based side channel attack. The attacker accesses each element in *parray* and measures the access latency. Since there exists a big timing difference between a cache hit and a cache miss, the attacker can use the access latency to infer the cache state. Specifically, the attacker will get a short access latency when accessing $parray[s]$, and a long latency when accessing other elements. Therefore, the index that exhibits the shortest access latency is the secret value s.

The above example shows that a cache-based side channel attack consists of two procedures: 1) a victim program performs secret-dependent memory accesses, which change the state of the cache; and 2) an attacker program extracts the secret by detecting the state of the cache. All the cache attacks follow the two procedures, while the concrete actions in the two procedures can vary a lot. For example, the secret-dependent access performed by the victim can be an instruction in a correct execution path, or it can be an instruction in an incorrect execution path that is speculatively executed and has to be squashed later. The latter case happens in a transient execution attack. In terms of the detection strategies, an attacker could detect the cache state based on the latency of a single cache access, the latency of multiple accesses, or the execution time of the victim program. Moreover, it has been shown that attackers can detect not only cache occupancy state, but also coherence state [9] and replacement information state [10].

1.2 CHALLENGES IN DEFENDING AGAINST CACHE ATTACKS

Given the variety of cache-based side channel attacks, it is necessary to understand the root cause of the attacks and the main challenges in designing effective defense solutions.

Micro-architecture side channels are due to processor optimizations that create detectable side effects. Processor hardware has traditionally been aggressively optimized for performance and energy efficiency, without taking security into consideration. Furthermore, many optimizations are made transparent to the software for programmability purposes. These optimizations have made processors fast and efficient, but they have also introduced many exploitable side effects, such as timing variations. An obvious example is caches, which are fast on-chip storage structures. They were introduced to improve performance and naturally created faster paths to access data. However, the faster access paths that we leverage for performance are exactly the exploitable side effects used in most cache-based side channel attacks.

We now consider the critical challenges in defeating cache-based side channel attacks. From a hardware design perspective, modern caches are spatially and temporally shared by different programs, regardless of whether they are in the same security domain or not. Thus, shared caches have become a broad attack surface and provide attackers with a number of ways to interfere with the victim's execution.

Given the recent expansion of cloud computing and web applications, it has never been easier than today for attackers to achieve co-location with victims in the same machine. On one hand, popular cloud platforms, such as Amazon EC2 [11], Google Cloud [12], and Microsoft Azure [13], provide convenient ways for users to access computing resources. Many companies are using cloud services to deploy their applications, given the configuration flexibility and reduced maintenance cost involved. Even though cloud computing uses virtual machines [14] and containers [15] to provide software-level isolation, the cache hardware is largely unaware of security domains. On the other hand, modern web applications are becoming dynamic and interactive, hosting programs written in high-level languages. It has become easy for potentially malicious applications to run on users' private desktops or mobile devices, as they merely require the user to click a link to access a website. Even though these applications are guarded by software isolation techniques and execute within sandboxes, the domain information is not communicated to the cache management modules in hardware. In short, both trends give attackers good chances to co-locate in the same computing machine as victim applications [16, 17], and achieve temporal and spatial sharing of cache hierarchies.

Obviously, a potential solution to close this channel is to disable cache sharing. Unfortunately, trivially disabling sharing can cause system resource underutilization and serious performance overhead. Disabling spatial sharing seriously hurts system throughput, as the maximum number of parallel virtual processes is limited by the physical resources. Disabling temporal sharing can significantly increase latency, as cache states have to be reset upon every switch of security domains.

From a software development perspective, a potential solution to close the channel is to make cache access traces oblivious of any secret. For example, a vulnerable program can be transformed by adding redundant memory accesses, so that cache states are changed in the same way irrespective of the value of the secret. Even though this approach generally suffers from serious performance overhead, it is effective and it has been used to fix vulnerabilities in several important cryptography algorithms. However, recent transient execution attacks have called such software defense solutions into question.

In transient execution attacks, attackers exploit hardware speculation to cause the execution of instructions in incorrect paths. These instructions potentially access secrets, leaving side effects on the cache hierarchy before being squashed. These instructions are outside of the scope of what programmers and compilers can reason about. Without controlling which memory access instructions are executed by the processor, software-only defenses become ineffective.

In summary, we consider that the challenge of defending against cache-based side channel attacks is two-fold. From the hardware design perspective, caches offer a broad attack surface mainly due to the temporal and spatial sharing of cache hierarchies across different security domains. From the software development perspective, software defense solutions become ineffective because speculative execution can cause execution to proceed in ways that were not intended by the programmer or compiler.

1.3 THESIS CONTRIBUTIONS AND ORGANIZATION

Given the aforementioned challenge in defending against cache-based side channel attacks, this thesis proposes *secure hardware processor designs to effectively and efficiently address the side channel vulnerabilities of modern cache hierarchies.*

This thesis makes two main contributions. First, we study the role of cache interference in different cache attacks and propose effective solutions to mitigate shared cache attacks by limiting malicious interference. Caches offer a broad attack surface, mainly because modern cache hierarchies are temporally and spatially shared by different security domains. The shared cache hierarchy provides attackers a number of ways to interfere with a victim's execution and cache behavior.

Based on our analysis, in a multi-level cache hierarchy, evicting lines from the private cache of the victim (i.e., creating "*inclusion victims*") is the key in a successful attack. This is because inclusion victims give an attacker visibility into a victim's private cache and provide useful information. Based on this important observation, we present a secure hierarchy-aware cache replacement policy (*SHARP*), which defeats cache attacks on inclusive cache hierarchies by eliminating inclusion victims. In addition, we show that inclusion victims also exist in non-inclusive cache hierarchies, and that the non-inclusive property is insufficient to stave off cache-based side channel attacks. We design the first two conflict-based cache attacks targeting the directory structure in a non-inclusive cache hierarchy. We prove that the *directory* structure is actually a unified attack surface for all types of cache hierarchies, including inclusive, non-inclusive and exclusive ones. To address this problem, we present the first scalable secure directory (*SecDir*) design to eliminate inclusion victims by restructuring the directory organization.

Second, we study how to effectively defend against *transient* (i.e., speculative) execution attacks in the cache hierarchies. Cache-based side channel attacks play an important role in transient execution attacks, leading to arbitrary information leakage and the violation of memory isolation. Specifically, in transient execution attacks, speculative execution causes the execution of instructions in incorrect paths. We present "Invisible Speculation" (*InvisiSpec*) to hide the side effects of transient load instructions. It is the *first* robust hardware defense mechanism against transient cache-based side channel attacks for multiprocessors.

The thesis is organized as follows.

Chapter 2 – Background and Related Work This chapter starts by providing a brief background on modern micro-architecture features and cache hierarchy designs. We then describe a generalized attack schema of cache-based side channel attacks. Based on key procedures in the attack schema, we go through a deep dive of two taxonomies. Finally, we review related work on countermeasures against cache attacks.

Chapter 3 – SHARP This chapter presents a secure hierarchy-aware cache replacement policy (*SHARP*) that enables an efficient and practical defense mechanism against cache-based side channel attacks for inclusive cache hierarchies. We observe that when a cross-core cache attack happens, the attacker evicts the victim's line from the shared cache, and also from the private cache of the victim process, creating an "*inclusion victim*". Creating inclusion victims is the key for the attacker to gain visibility into the victim's private cache. Consequently, to disable cache attacks, we propose to alter the line replacement algorithm of the shared cache to prevent a process from creating inclusion victims in the caches of cores running other processes. SHARP is an efficient defense mechanism, which requires minor hardware modifications and induces negligible average performance degradation. The work is published in ISCA 2017 [18].

Chapter 4 – Directory Attacks This chapter presents the first two directory-based side channel attacks on non-inclusive cache hierarchies. As modern caches move away from inclusive cache hierarchies to non-inclusive ones, previous cache attack strategies have been called into doubt, mainly due to the assumption that attackers should be unable to create "inclusion victims". However, we find that, in a non-inclusive cache, inclusion victims can be created via directory conflicts. We present *directory attacks* and prove that the directory can be used to bootstrap conflict-based cache attacks on any cache hierarchy, including inclusive, non-inclusive and exclusive ones. We demonstrate the directory attacks by extracting key bits during RSA operations in GnuPG on a state-of-the-art non-inclusive Intel Skylake-X server. The work appears in SP 2019 [19].

Chapter 5 – SecDir This chapter presents the first design of a scalable secure directory (*SecDir*). In the previous chapter, we showed the need to redesign directories for security. However, in an environment with many cores, it is hard or expensive to block directory interference. To address this problem, we propose SecDir, a secure directory design, to eliminate inclusion victims by restructuring the directory organization. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory areas used in a victim-cache manner called Victim Directories (VDs). The partitioned nature of VDs prevents directory interference across cores, defeating directory side channel attacks. We show that SecDir has a negligible performance overhead and is area efficient. The work is published in ISCA 2019 [20].

Chapter 6 – InvisiSpec This chapter presents the first robust hardware defense mechanism against transient cache-based side channel attacks for multiprocessors. It has been shown that hardware speculation offers a major surface for micro-architectural side channel attacks. Unfortunately, defending against speculative execution attacks is challenging. We propose InvisiSpec, a novel strategy to defend against transient cache attacks in multiprocessors by making speculation invisible in the data cache hierarchy. In InvisiSpec, unsafe speculative loads read data into a speculative buffer, without modifying the cache hierarchy. When the loads become safe, InvisiSpec makes them visible to the rest of the system. We show that InvisiSpec has modest performance overhead and is able to defend against future transient execution attacks where any speculative load can pose a threat. This work is published in MICRO 2018 [21].

Chapter 7 – Conclusion This chapter summarizes the results of the thesis and discusses future research directions.

Appendix A – Cache Telepathy This appendix presents another example of cache-based side channel attacks that target Deep Neural Networks (DNNs). A DNN's architecture (i.e., its hyper-parameters) broadly determines the DNN's accuracy and performance, and is often confidential. We propose Cache Telepathy, an efficient mechanism to help obtain a DNN's architecture using the cache side channel. We evaluate our attack by attacking VGG and ResNet DNNs. We show that our attack is effective in helping obtain the architectures by very substantially reducing the search space of the target DNN architectures. This work appears in USS 2020 [22].

CHAPTER 2: BACKGROUND AND RELATED WORK

2.1 MODERN PROCESSOR AND CACHE ORGANIZATION

2.1.1 Memory Hierarchy and Basic Cache Structures

Modern high-performance processors contain multiple levels of caches that store data and instructions for fast access. The cache structures closer to the core, such as the L1, are the fastest, and are called higher-level caches. The ones farther away from the core and closer to main memory are slower, and are called lower-level caches. High-performance processors typically feature two levels of private caches (L1 and L2), followed by a shared L3 cache—also referred to as LLC for last-level cache.

The L1 cache is designed to be small (e.g., 32-64KB) and to respond very fast, typically within a few cycles. The L2 cache is slightly bigger (e.g., 256KB-1MB) and takes around 10-20 cycles. Finally, the LLC is designed to be large (e.g., several to tens of MBs) and has a latency of 40-60 cycles. The LLC latency is still much lower than the main memory access latency, which is on the order of 200-300 cycles.

A cache consists of the *data array*, which stores the data or code, and the *tag array*, which stores the high-order bits of the addresses of the data or code. The cache is organized in a number of *cache lines*, each one of size B bytes. The cache is typically set-associative, with S sets and W ways. A cache line occupies one way of a cache set. The set in which a cache line belongs is determined by its address bits. A memory address is shown in Figure 2.1.

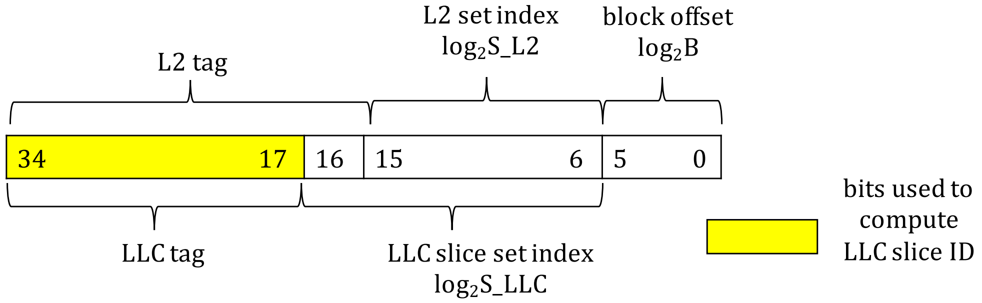
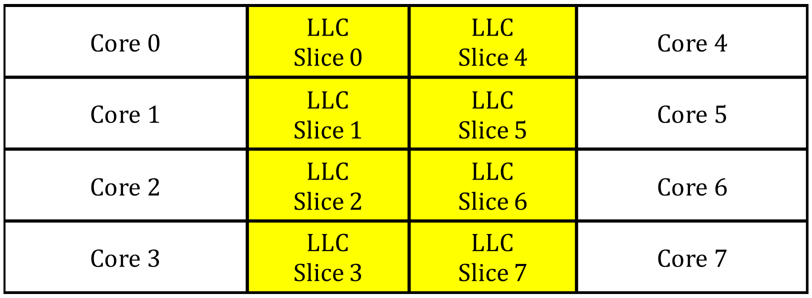


Figure 2.1: Example of a memory address broken down into tag, index, and block offset bits. The actual bit field sizes correspond to the L2 and the LLC slice of the Intel Skylake-X system. We refer to the LLC slice set index as the LLC set index in this thesis.

The lower $\log_2 B$ bits indicate the *block offset* within a cache line. The next $\log_2 S$ bits form the index of the set that the cache line belongs to. The remaining bits of the address form the *tag*. The tags of all the lines present in the cache are stored in the tag array. When a load or store request is issued by the core, the tag array of the L1 cache is checked to find out if the data is present in the cache. If it is a hit, the data is sent to the core. If it is a miss, the request is sent to the L2 cache. Similarly, if the request misses in L2 it is further sent to the LLC and then to main memory. Note that, generally, lower levels of the cache hierarchy have more sets than higher levels. In that case, cache lines that map to different LLC sets may map to the same L2 set, due to the pigeonhole principle.

2.1.2 Multi-core Cache Organization

The LLC in a modern multi-core is usually organized into as many slices (partitions) as the number of cores. Such an organization, shown in Figure 2.2, is helpful to keep the design modular and scalable. Each slice has an associativity of W_{slice} and contains S_{slice} sets. S_{slice} is $1/N$ the total number of sets in the LLC, where N is the number of cores.



Core 0	LLC Slice 0	LLC Slice 4	Core 4
Core 1	LLC Slice 1	LLC Slice 5	Core 5
Core 2	LLC Slice 2	LLC Slice 6	Core 6
Core 3	LLC Slice 3	LLC Slice 7	Core 7

Figure 2.2: Example of a sliced LLC design with 8 cores.

Processors often use an undocumented hash function to compute the slice ID to which a particular line address maps to. The hash function is designed to distribute the memory lines uniformly across all the slices. In the absence of knowledge about the hash function used, a given cache line can be present in any of the slices. Therefore, from an attacker's perspective, the effective associativity of the LLC is $N \times W_{\text{slice}}$. The hash function used in Intel's Sandybridge processor has been reconstructed in prior work [23], and found to be an xor of selected address bits. The slice hash function for the Skylake-X is more complex.

We now discuss two important cache design choices, and the tradeoffs behind them.

Inclusiveness The LLC can be either *inclusive, exclusive,* or *non-inclusive* of the private caches. In an inclusive LLC, the cache lines in private L2 caches are also present in the LLC, whereas in an exclusive LLC, a cache line is never present in both the private L2 caches and in the LLC. Finally, in a non-inclusive LLC, a cache line in the private L2 caches may or may not be present in the LLC.

The inclusive design wastes chip area and power due to the replication of data. Typically, as the number of cores increases, the LLC size must increase, and hence the average LLC access latency increases [24, 25]. This suggests the use of large L2s, which minimize the number of LLC accesses and, therefore, improve performance. However, increasing the L2 size results in a higher waste of chip area in inclusive designs, due to the replication of data. The replication of data can be as high as the L2 capacity times the number of cores. Therefore, non-inclusive cache hierarchies have recently become more common. For example, the most recent server processors by Intel use non-inclusive caches [26, 27]. AMD has always used non-inclusive L3s in their processors [28].

Cache Coherence and Directories When multiple cores read from or write to the same cache line, the caches should be kept coherent to prevent the use of stale data. Therefore, each cache line is assigned a state to indicate whether it is shared, modified, invalid, etc. A few state bits are required to keep track of this per-line state in hardware in the cache tag array or directory.

Two types of hardware protocols are used to maintain cache coherence—*snoop-based* and *directory-based*. The snoop-based protocols rely on a centralized bus to order and broadcast the different messages and requests. As the number of cores is increased, the centralized bus quickly proves to be a bottleneck. Therefore, most modern processors use a directory-based protocol, which uses point-to-point communication. In a directory-based protocol, a *directory* structure is used to keep track of which cores contain a copy of a given line in their caches, and whether the line is dirty or clean in those caches.

In an inclusive LLC design, the directory information can be conveniently co-located with the tag array of the LLC slice. Since the LLC is inclusive of all the private caches, the directory state of all the cache lines in any private cache is present in such a directory. The hardware can obtain the list of sharer cores of a particular line by simply checking the line's directory entry in the LLC. There is no need to query all the cores. However, the directory in a non-inclusive cache hierarchy design is more complicated.

2.1.3 Out-of-order and Speculative Execution

Dynamically-scheduled processors [29] execute data-independent instructions in parallel, out of program order, and thereby exploit instruction-level parallelism [30] to improve performance. Instructions are *issued* (enter the scheduling system) in program order, *complete* (execute and produce their results) possibly out of program order, and finally *retire* (irrevocably modify the architected system state) in program order. In-order retirement is implemented by queueing instructions in program order in a reorder buffer (ROB) [31], and removing a completed instruction from the ROB only once it reaches the ROB head, i.e., after all prior instructions have retired.

Speculative execution is the execution of an instruction before its validity can be made certain. If, later, the instruction turns out to be valid, it is eventually retired, and its speculative execution has improved performance. Otherwise, the instruction will be squashed and the processor's state rolled back to the state before the instruction. (At the same time, all of the subsequent instructions in the ROB also get squashed.) Strictly speaking, an instruction executes speculatively in an out-of-order processor if it executes before it reaches the head of the ROB.

There are multiple reasons why a speculatively-executed instruction may end up being squashed. One reason is that a preceding branch resolves and turns out to be mispredicted. Another reason is that, when a store or a load resolves the address that it will access, the address matches that of a later load that has already obtained its data from that address. Another reason is memory consistency violations. Finally, other reasons include various exceptions and the actual squash of earlier instructions in the ROB.

Memory Consistency and Its Connection to Speculation A memory consistency model (or memory model) specifies the order in which memory operations are performed by a core, and are observed by other cores, in a shared-memory system [32]. When a store retires, its data is deposited into the write buffer. From there, when the memory consistency model allows, the data is merged into the cache. We say that the store is *performed* when the data is merged into the cache, and becomes observable by all the other cores. Loads can read from memory before they reach the ROB head. We say that the load is *performed* when it receives data. Loads can read from memory and be performed out of order—i.e., before earlier loads and stores in the ROB are. Out-of-order loads can lead to memory consistency violations, which the core recovers from using the squash and rollback mechanism of speculative execution [33].

Total Store Order (TSO) [34,35] is the memory model of the x86 architecture. TSO forbids all observable load and store reorderings except store→load reordering, which is when a load bypasses an earlier store to a different address. Implementations prevent observable load→load reordering by ensuring that the value a load reads when it is performed remains valid when the load retires. This guarantee is maintained by squashing a load that has performed, but not yet retired, if the core receives a cache invalidation request for (or suffers a cache eviction of) the line read by the load. Store→store reordering is prevented by using a FIFO write buffer, ensuring that stores perform in program order. If desired, store→load reordering can be prevented by separating the store and the load with a fence instruction, which does not complete until all prior accesses are performed. Atomic instructions have fence semantics.

Release Consistency (RC) [36] allows any reordering, except across synchronization instructions. Loads and stores may not be reordered with a prior acquire or with a subsequent release. Therefore, RC implementations squash performed loads upon receiving an invalidation of their cache line only if there is a prior non-retired acquire, and have a non-FIFO write buffer.

2.2 CACHE-BASED SIDE CHANNEL ATTACKS

Cache-based side channel attacks (*cache attacks* for short) are serious threats to secure computing, and have been demonstrated on a variety of platforms, from mobile devices [37] and desktop computers [5, 38] to server deployments [6, 39, 40]. Side channel attacks bypass software isolation mechanisms and are difficult to detect. They can be used to steal coarse-grained information such as when a user is typing [37] down to much more fine-grained information such as a user's behavior on the web [41], and even RSA [6,42] and AES [5,40,43–45] encryption keys. Jakub Szefer [7] presented an in-depth analysis of various micro-architectural side channels and a comprehensive survey of existing defense proposals before the year of 2016.

In this section, we start with a classical cache attack example on the RSA encryption algorithm [42]. We then discuss a generalized attack schema, which is applicable to any cache attack. Based on key procedures in the attack schema, we go through a deep dive of two taxonomies, which define the defense scopes of this thesis.

2.2.1 An Example of Cache Attacks

A common type of cache-based side channel attack involves a victim process and an attacker process. It usually consists of an offline phase and an online phase. In the offline phase, the attacker identifies *target addresses*, which are addresses whose access patterns can leak secret information about the victim's program. Target addresses can be identified by analyzing the victim's program manually or with automatic tools [46, 47]. In the online phase, the attacker observes the victim's access patterns on target addresses and deduce the value of secret information.

Algorithm 2.1: Square-and-multiply exponentiation.

```
   Input  : base $b$, modulo $m$, exponent $e = (e_{n-1}...e_0)_2$
   Output: $b^e$ mod $m$
1  $r = 1$
2  for  $i = n - 1$ downto $0$ do
3      $r = sqr(r)$
4      $r = mod(r, m)$
5      if $e_i == 1$ then
6          $r = mul(r, b)$
7          $r = mod(r, m)$
8      end
9  end
10 return $r$
```

Algorithm 2.1 shows the Square-and-Multiply algorithm [42] from GnuPG version 1.4.13, which is vulnerable to side channel attacks. In the process of computing its output, the algorithm iterates over exponent bits from high to low. For each bit, it performs a `sqr` and `mod` operation. Then, if the exponent bit is "1", the algorithm performs a `mul` and a `mod` operation that are otherwise skipped. Effective target addresses are the entry points of the `sqr` function in Line 3 (which tells that the iteration is executed) and of the `mul` function in Line 6 (which tells that the bit is "1"). By observing access pattern on target addresses, the attacker can recover all the bits in the exponent.

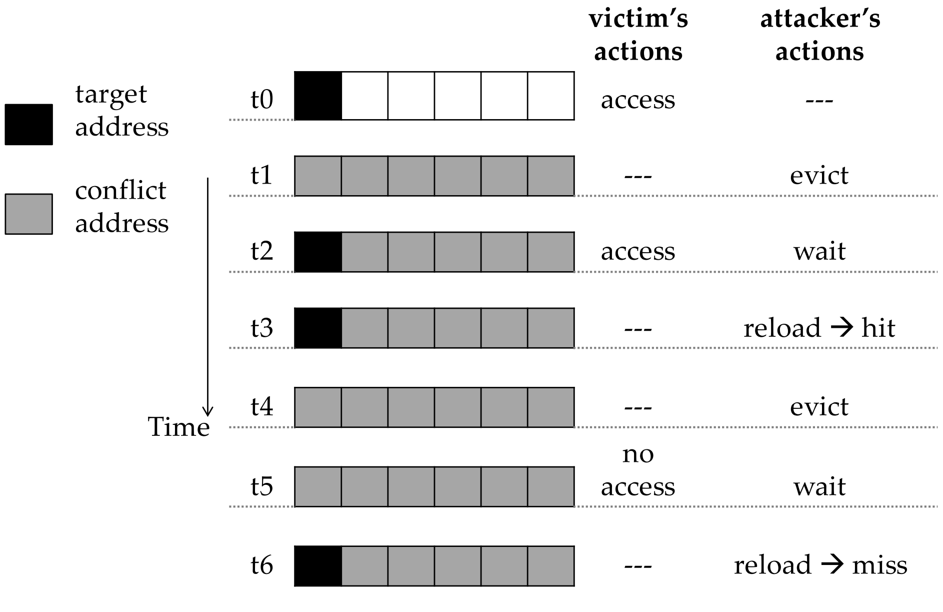


Figure 2.3: Evict+reload attack example.

There exist many different ways for an attacker to observe a victim's cache access patterns. We show an evict+reload [46] attack as an example. In this attack, the attacker and victim processes share the target addresses — possibly because they use a shared library or due to page deduplication [48]. The attack consists of multiple attack cycles, and each attack cycle involves three steps: evict, wait and reload. Figure 2.3 shows a timeline of the state of the 6 cache lines in a set of a 6-way set-associative cache.

At time *t0*, the victim loads a line with the target address into the cache (black square). At time *t1*, the attacker accesses 6 different addresses which are mapped to the same cache set. The 6 cache lines are brought into the cache, fill the set (gray squares), and evict the target line from the cache. Next, the attacker idles and waits a designated amount of time to allow the victim to potentially access the target address. In this example, at time *t2*, the victim does perform an access and brings the target address back to the cache. At time *t3*, the attacker reloads the target address and measures its access latency. The attacker gets a cache hit, as the victim accessed the target address during the waiting interval.

The next attack cycle repeats the evict, wait, and reload steps (times *t4*, *t5*, and *t6*). This time, the victim does not access the target address and the attacker records a cache miss. The latency of the reload access is longer than before. The example shows that the attacker can figure out whether the victim has accessed the target address during the wait interval based on the access latency of the reload operation. By repeating the attack cycle for multiple times, the attacker can collect a trace of the victim's accesses.

2.2.2 Generalized Attack Schema

The example above can be generalized as an attack schema[1] shown in Figure 2.4. The schema provides a good abstraction to understand the common procedures that are involved in any cache-based side channel attack.

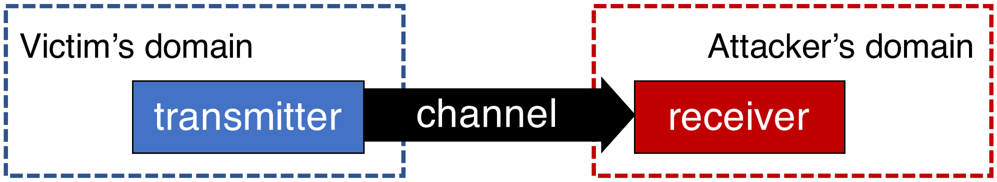


Figure 2.4: Cache attack schema involves two procedures: 1) the transmitter changes cache states in a secret-dependent way; and 2) the receiver detects the cache states and extracts the secret.

[1] The attack schema was first proposed by Kiriansky et al. [10]. The schema can be used to describe side channels other than cache states, such as branch predictors and TLBs.

As shown in Figure 2.4 the attacker application and the victim application exist in different security domains. Memory isolation techniques at the software layer guarantee that the attacker cannot directly access private information inside the victim's domain. However, information can be leaked via the side effects of the victim's execution on the computing system, including the cache hierarchy.

The generalized side channel attack schema involves two procedures. First, a victim program performs secret-dependent memory accesses, which change the states of the cache. We call these memory accesses *transmitters*. The transmitters modify the cache states based on secret information, forming detectable side channels that can be observed outside of the victim's domain. Second, an attacker program extracts the secret by detecting the states of the cache. The cache states that can be leveraged by attackers include cache occupancy state, coherence state [9] and replacement information state [10].

2.2.3 Classifications of Cache Attacks

We now discuss two taxonomies of cache-based side channel attacks. The taxonomies help define the defense scopes for later chapters.

Active and Passive Cache Attacks

We classify cache-based side channel attacks into *active* and *passive* cache attacks (Table 2.1), based on whether the attack interferes with the channel states before the transmitters execute. In an active cache attack, the attacker leverages the spatial and temporal sharing of cache hierarchies to bring the cache into a desired state before the execution of transmitters, so that it is able to control the way that transmitters interact with the cache. However, a passive attack does not involve such an interfere operation. It has been shown that active attacks are more effective towards a wider range of victim applications than passive ones.

We consider why the interfere operation is important. As discussed in Section 2.2.2, a critical procedure in a successful cache attack is that the transmitters make detectable changes to the cache state. However, many times, without the attacker's interference on the cache state, the transmitters do not cause observable changes. For example, assume that a transmitter tries to access a target cache line to bring it into the cache, but the target line was already present in the cache before the access happens. In this case, the transmitter's execution will not change the cache occupancy state, and thus cannot create a detectable side channel. Active attacks address this problem by interfering with the cache state. For example, in the evict+reload (Section 2.2.1) attack, every time before the transmitter executes, the attacker resets the cache state by evicting the target line from the cache. In this way, the transmitter will cause a cache miss, and the processor will insert the target line to the cache, leading to detectable changes to the cache occupancy state.

	Evict Strategies	Attacks
Active	Conflict-based	prime+probe [2], evict+reload [46], evict+time [38] alias-driven attack [49], evict+prefetch [50]
	Flush-based	flush+reload [48], flush+flush [51], invalidate+transfer [28], flush+prefetch [50]
Passive	–	cache collision attack [45]

Table 2.1: Classification of cache-based side channel attacks based on whether the attack interferes with the channel states before the transmitters execute.

Generally, an attack cycle in an active attack follows three steps: interfere, wait and analyze.

1) *Interfere*: The attacker brings cache into a desired state so that the execution of transmitters can cause detectable side effects. A common approach is to evict target lines from the level of cache in which they are resident.

   According to the approach used in the *Interfere* step, we can further classify attack strategies into conflict-based and flush-based (Table 2.1). When using conflict-based strategies, the attacker creates cache conflicts to evict cache lines containing target addresses. Specifically, it accesses multiple addresses that map to the same cache set as a target address. Often, these addresses are called *conflict addresses.* When using flush-based strategies, the attacker can access the target addresses — e.g., when the target addresses are in shared libraries. The attacker simply executes `clflush` instructions to evict the target addresses from the cache [52]. `clflush` guarantees that the addresses are written back to memory and invalidated from the cache.

2) *Wait*: The attacker waits a time period during which the victim may access the target address. The *waiting interval* of the *Wait* step is carefully configured [48]. It should be precisely long enough for the victim to access a target address exactly once before the Analyze step. If the interval is too long, the attacker gets only one observation for multiple accesses to the target address by the victim. If the interval is too short, the chances of overlapping the Interfere or Analyze step with the victim's target address access increases. In both cases, the accuracy of the attack decreases.

3) *Analyze*: the attacker determines whether the target address was accessed in the Wait step. There are several ways to accomplish this goal, including measuring the access time of either the target or conflict addresses (prime+probe [2, 6, 40] and flush+reload [41,48]), measuring the execution time of the victim program (evict+time [5, 38]), or reading values in main memory to see if the writebacks of cache lines have occurred (alias-driven attack [49]).

Different from an active attacker, a passive attacker does not interfere with the cache state when the victim executes. It simply monitors the victim's cache accesses or execution time, which may be affected by reuses or conflicts of cache lines within the victim itself (i.e., *self-reuse* and *self-conflict*). An example is the cache-collision attack [45].

The defense mechanisms presented in Chapter 3 and Chapter 5 target active cache attacks. The attacks presented in Chapter 4 are conflict-based cache attacks.

Non-transient and Transient Cache Attacks

Spectre [3], Meltdown [4] and follow-up attacks [21, 53–58] based on speculative execution, or more precisely *transient* [59] execution, have significantly increased hardware vulnerabilities. Depending on whether the transmitter in a cache attack is a transient instruction, i.e., a speculative instruction bound to squash, we can classify cache-based side channel attacks into *transient* and *non-transient* cache attacks.

In a non-transient cache attack, the transmitter execution is intended by the programmer. However, in a transient cache attack, the transmitter is executed due to incorrect speculation and is not intended by the programmer. Such instructions are out the scope of what compilers or programmers can reason about.

Theoretically, memory trace oblivious programming [60] is an effective mitigation towards non-transient cache attacks. Specifically, this technique changes the vulnerable code by adding redundant memory accesses, so that the transmitters will not cause secret-dependent cache state changes. However, the transmitters used in transient cache attacks can not be controlled by the software. Thus, traditional software defense solutions become ineffective in transient cache attacks.

The defense mechanism in Chapter 6 targets transient cache attacks.

2.3 COUNTERMEASURES AGAINST CACHE-BASED SIDE CHANNEL ATTACKS

In this section, we review related work on defending against side channel attacks on shared cache hierarchies, followed by a discussion on detection techniques. We classify defense mechanisms into two categories, isolation-based and randomization-based.

Isolation-based Defenses Isolation-based defense mechanisms rely on cache partitioning techniques to block unintended cache interference. There are two types of cache-partitioning techniques depending on the total number of partitions required.

The first type partitions the cache into as many regions as the number of security domains. Static way-partition [61] provides isolation by statically assigning certain cache ways to each security domain. Unfortunately, this approach can introduce serious performance overhead, since the cache cannot be dynamically shared. Moreover, when the number of cores is higher than the cache associativity, some cores cannot get a cache partition, resulting in serious under-utilization of core resources. Note that it is very common to run a high number of processes from different security domains on modern servers, which calls for scalable defense solutions.

DAWG [10], SecDCP [62] and NoMo cache [63] are dynamic way-partitioning techniques. These mechanisms can dynamically adjust the number of ways for each security domain to avoid cache under-utilization. However, when the number of security domains is higher than the cache associativity, they are forced to re-assign cache ways from one domain to another. These re-assignment operations can leak cache occupancy information from the victim to the attacker.

The second type of partitioning technique, used by CATalyst [64] and StealthMem [65], partitions the cache into two regions, i.e., a security-sensitive region and a non-secure shared region. The first region is reserved for security-sensitive data accesses. Cache interference within the security region is blocked via page coloring. The non-secure region can be dynamically shared by other applications. Unfortunately, both approaches require programmers to provide information about security-sensitive data or instruction accesses. Such information is not easy to obtain for many applications.

Randomization-based Defenses There are several mitigation techniques that rely on the randomization of the address mapping logic or system timing components. Newcache [66, 67], CEASER [68] and RPcache [69] randomize the mapping of addresses to cache sets to prevent the attacker from evicting target lines from caches. For example, CEASER dynamically remaps cache lines, so that the attacker cannot find an effective group of addresses that are mapped to the same set as the target address. However, these techniques can only reduce the bandwidth of cache attacks, instead of eliminating them. The attacker can still perform the evict operation when it accesses enough lines across a large number of cache sets. Liu et al. [70] proposed the Random Fill cache architecture for the L1 cache to defeat the cache-collision attack. However, this approach may suffer from substantial performance degradation if applied to the much larger last-level cache.

TimeWarp [71] and FuzzyTime [72] disrupt timing measurements by adding noise to the system clock. They can protect against attacks which measure cache access latency and execution time, but they are unable to prevent alias-driven attacks [49]. Furthermore, they hurt benign programs that require a high-precision clock.

Techniques to Detect Cache Attacks Several approaches have been proposed to detect cache-based side channel attacks. Chiappetta et al. [73] detect side channels based on the correlation of last-level cache accesses between victim and attacker processes. HexPADS [74] detects side channel attacks by the frequent cache misses caused by the attacker process. These heuristic approaches are not robust, and tend to suffer high false positives and false negatives. CC-Hunter [75] and ReplayConfusion [76] can effectively detect cache-based covert channel attacks. The main challenge in designing detection mechanisms is to achieve low false positives and low false negatives together. Consider an advanced attacker who mimics the behavior of benign applications and communicates secrets using an extremely low bandwidth. Most of the existing detection mechanisms are likely to miss such attacks.

2.4 COUNTERMEASURES AGAINST TRANSIENT EXECUTION ATTACKS

Wang et al. [77] pointed out that control speculation can enable new covert and side channels. Later, in January 2018, the disclosure of Spectre [3] and Meltdown [4] highlighted the broad attack surfaces offered by speculative execution in modern processors. Thereafter, transient execution attacks and defenses have become an active research problem. A systematization of transient execution attacks and defenses is presented by Canella et al. [59]. We review software-based defense solutions, followed by hardware-based ones.

Software-based Solutions Existing software-based solutions use two strategies to mitigate transient execution attacks. First, most software-based solutions block information leakage by limiting potentially malicious speculative execution. The immediate mitigations after the disclosure of Spectre and Meltdown were to place serializing instructions such as `lfence` after sensitive branches [78,79], or replace branches with non-speculative `cmovs`. Alternative approaches to limit speculative execution include introducing extra data-dependencies or control-dependencies between instructions, such as masking index bits before array look-ups or using return trampolines [80]. These solutions have performance, usability, or completeness issues. To reduce the performance overhead, oo7 [81] uses taint analysis to identify the branches that can be controlled by attackers, and only blocks the speculative execution of those branches.

Second, other software-based solutions aim to block interference between different security domains by providing better isolation. KAISER [82] proposes a practical kernel address isolation mechanism. It creates a shadow page table which only contains user-space mappings. When a process is running in user mode, the shadow page table is used, and thus, the user mode cannot obtain any information about kernel-space mappings. While KAISER claims to be able to close all hardware side channels on kernel address information, follow-on work [83] demonstrates that information leakage is still possible via a small set of kernel-space mappings in the shadow table, which are essential to handle system calls and interrupts. As another example, the Chrome web browser and V8 used to rely on language-enforced isolation, where untrusted code is sandboxed and executed in the same process as trusted code. Mcilroy et al. proposed to shift this isolation mechanism to process-based isolation [84].

Hardware-based Solutions Existing hardware defense mechanisms can also be classified into two categories. The first category focuses on limiting the side effects of transient instructions. InvisiSpec [21] and SafeSpec [85] prevent transient instructions from changing cache states, and use a shadow structure to hold speculative state for caches and TLBs. However, SafeSpec does not handle cache coherence or memory consistency model issues and, therefore, cannot support multi-threaded workloads. A relatively more aggressive approach is CleanupSpec [86], which allows speculative load instructions to change cache states and "undo" the changes upon misspeculations. These approaches introduce modest performance overhead, but require extensive changes to both the core and the memory subsystem.

Instead of introducing extensive hardware changes, a straightforward solution is to delay the execution of speculative instructions. However, conservatively delaying all speculative instructions can incur serious performance overhead. Several mechanisms have been proposed to address this performance problem by selectively delaying instructions. Conditional Speculation [87] and Delay-on-Miss [88] delay speculative load instructions only if they miss the L1 cache. Context-Sensitive Fencing [89] and ConTExT [90] use traditional taint tracking in cache and memory, and prevent user-marked secrets from being used by transient instructions. NDA [91] and SpecShield [92] restrict propagation of any data to speculative instructions that could form a side channel. They propose multiple design variants with different performance-security tradeoffs. STT [93] proposes a novel form of taint tracking mechanism to identify transiently-accessed data and prevent such data from reaching side channels. In addition, STT provides a comprehensive study of all variants of side channels on speculative microarchitectures.

The second category aims to provide isolation of various hardware resources assuming that the attacker and the victim are in different security domains. Cache partitioning techniques [10,62,64] all fall in this category. MI6 [94] extends isolation to more realistic memory hierarchies, which have MSHRs and page walks.

CHAPTER 3: SHARP

This chapter presents Secure Hierarchy-Aware cache Replacement Policy (SHARP), an efficient and practical defense mechanism against cache-based side channel attacks for inclusive cache hierarchies. The design of SHARP is based on an important observation: in a cross-core cache-based side channel attack, creating "inclusion victims" is the key for the attacker to gain visibility into the victim's private cache. Consequently, we alter the line replacement algorithm of the shared cache to prevent a process from creating inclusion victims in the caches of cores running other processes. SHARP requires minor hardware modifications and induces negligible average performance degradation.

Shared caches offer a broad attack surface for side channel attacks, mainly due to the spatial and temporal sharing of cache hierarchies across different security domains. This chapter studies the most effective type of cache attack, namely, active cache-based side channel attacks (Section 2.2.3) in a cross-core setup, where attacker and victim processes execute on different cores, sharing the L2 or L3 cache of an inclusive cache hierarchy. The reason for the attack's effectiveness is that it leverages widely-used commodity hardware, and is relatively easy to set up. Our goal is to defeat cross-core active cache attacks by blocking cache interference.

We make the following observation: in an inclusive cache hierarchy, it is easy for the attacker to evict the target line from the private cache of the core running the victim process. This is because, if the line is in the shared cache, it is also practically always in the private cache of the core running the victim process. The reason is the tight timing requirements to mount a successful attack. Then, when the attacker evicts the line from the shared cache, the line is also automatically evicted from the private cache of the core running the victim process, creating an *inclusion victim*.

Some authors have studied the impact of inclusion victims on performance (e.g., [95, 96]). However, in most designs, the cache replacement algorithm in the shared cache only uses information on shared cache hits and misses, and is oblivious of hits in the private caches.

The main proposal of this chapter is to alter the shared cache replacement algorithm to prevent a process from creating inclusion victims in the caches of cores running other processes. The attacker is unable to evict the target address from the shared cache and, hence, cannot glimpse any information on the victim's access patterns. Note that cache attacks do not always use load instructions to force the eviction of a victim's target addresses from the cache; sometimes they use an instruction called `clflush`. Hence, our proposal also involves a slightly modified `clflush` instruction to thwart these attacks.

We call our proposal SHARP (Secure Hierarchy-Aware cache Replacement Policy). SHARP requires minimal hardware modifications. It works for all existing applications without requiring any code modifications. Finally, it induces negligible average performance degradation.

To validate SHARP, we implement it in a cycle-level full-system simulator and test it against real-world attacks. SHARP effectively protects against these attacks. In addition, we run many workloads from SPEC and PARSEC applications to evaluate SHARP's impact on performance.

The contributions of this chapter are:

- The insight that, to effectively prevent cache attacks in an inclusive cache hierarchy, we can alter the shared cache replacement algorithm to prevent a process from inducing inclusion victims on other processes.
- The design of SHARP, which consists of a new cache line replacement scheme and a slightly modified `clflush` instruction.
- A simulation-based evaluation of SHARP that shows that it is effective against real-world attacks, and induces negligible average performance degradation.

3.1 ATTACK ANALYSIS

As discussed in Section 2.2.3, an active cache-based side channel attack always involves an "interfere" step. In this section, we analyze the two interfere strategies.

3.1.1 Conflict-based Attacks

We find that all successful conflict-based attacks share two traits: (1) they generate inclusion victims in the private cache of the core running the victim thread, and (2) they exploit modern cache line replacement policies that do not properly defend against malicious creation of inclusion victims.

Consider the first trait. As discussed in Section 2.2.3, an attack cycle in an active cache attack follows three steps: interfere, wait and analyze. Existing conflict-based attacks generate inclusion victims in the private cache of the victim process' core. This is because the duration of an attack cycle between consecutive interfere steps is usually very short — on the order of several thousand cycles. Attacks use such short cycles to reduce the noise in the analyze step. As a result, if the victim accesses the target addresses during the wait step, then such addresses will typically remain in the victim's private cache by the next interfere step. Hence, when the attacker performs the interfere step, it generates inclusion victims in the private cache of the victim's core.

The second trait concerns the fact that deployed cache line replacement algorithms and deployed algorithms for inserting referenced lines in the replacement priority list, do not take into consideration the possible creation of inclusion victims. This makes commercial systems vulnerable to conflict-based attacks.

Recent proposals (e.g., [97–100]) take into account the requesting core ID when deciding what line in the set to replace, or what priority in the replacement list to assign to the referenced line. However, they do it to improve resource allocation or to enhance performance, and do not try to eliminate inclusion victims. Only the TLA cache management proposal [96] uses some hints that try to minimize the probability of creating inclusion victims. However, since TLA is focused on performance, it does not guarantee the elimination of inclusion victims and, hence, cannot provide security guarantees.

3.1.2 Flush-based Attacks

The x86 `clflush` instruction invalidates a specific address from all levels of the cache hierarchy [52]. The invalidation is broadcasted throughout the cache coherence domain. If, at any cache, the line is dirty, it is written to memory before invalidation. In user space, `clflush` is used to handle memory inconsistencies such as in memory-mapped I/O and self-modifying codes. In kernel space, `clflush` is used for memory management, e.g., to flush from the caches all the lines belonging to a page that is being swapped out.

Flush-based attacks rely on the `clflush` instruction to evict target addresses from the cache. Entirely disabling the use of such instruction is impractical due to both legacy issues and valid use cases. However, we make a key observation about the legitimate uses of `clflush`: in user mode, `clflush` is only really needed in uses that update memory locations. Specifically, it is needed to handle the case when the value of a location in caches is more up-to-date than the value of the same location in main memory. In such cases, `clflush` brings the memory to the right state.

We argue that there is no need to use `clflush` in user mode for pages that are read-only or executable, such as those that contain shared library code. Allowing the use of `clflush` in these pages only makes the system vulnerable to flush-based attacks.

3.2 SHARP DESIGN

Secure Hierarchy-Aware cache Replacement Policy (SHARP) is composed of a new cache replacement scheme to protect against conflict-based cache attacks, and a slightly modified `clflush` instruction to protect against flush-based cache attacks. In the following, we discuss SHARP's two components, and then give some examples of defenses.

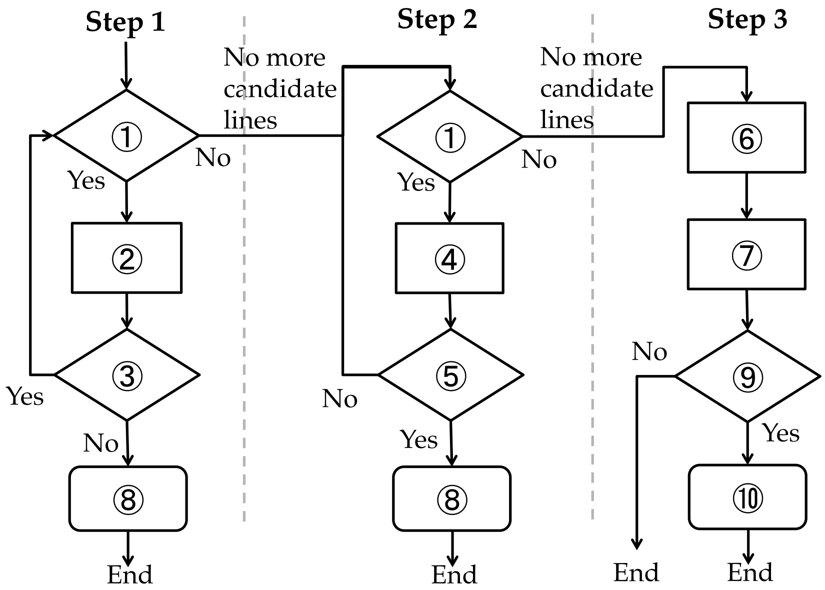
3.2.1 Protecting Against Conflict-based Attacks

To protect against conflict-based attacks, SHARP's main idea is to alter a shared cache's replacement algorithm to minimize the number of inclusion victims that a process induces on other processes. The goal is to prevent an attacker process from replacing shared-cache lines from the victim process that would create inclusion victims in the private caches of the victim process' core. The result is that the attacker cannot create a conflict-based cache attack.

Assume that a requesting process R (potentially an attacker) wants to load a line into a set of the shared cache that is full. The hardware has to find a victim line to be evicted. The high level operation of the SHARP replacement algorithm is shown in Figure 3.1. It has three steps. In Step 1, SHARP considers each line of the set at a time (①), in the order based on its replacement priority. For each line, it checks if the line is in any private cache (②)–(③). As soon as a line is found that is not in any private cache, it is used as the replacement victim (⑧). Victimizing this line will not create any inclusion victim. If no such line is found, the algorithm goes to Step 2.

In Step 2, SHARP considers again each line of the set at a time (①), in the order based on its replacement priority. For each line, it checks if the line is present only in the private cache of R (④)–(⑤). As soon as one such line is found, it is used as the replacement victim (⑧). Evicting this line will at worst create an inclusion victim in R. No other process will be affected. If no such line is found, the algorithm goes to Step 3.

In Step 3, SHARP increments a per-core local alarm event counter (⑥) and selects a random line as the replacement victim (⑦). In this case, SHARP may create a replacement victim in a process that is being attacked. For this reason, when the alarm event counter of any core reaches a threshold (⑨), a processor interrupt is triggered (⑩). The operating system is thus notified that there is suspicious activity currently in the system. Any relatively low value of the threshold suffices, as a real attacker will produce many alarms to be able to obtain any substantial information. In the worst case, a smart attacker, who is aware of the threshold used by SHARP, can bypass the protection of SHARP by lowering the attack



① More lines to consider?
②④ Obtain information on the presence of the line in private caches
③ Is the line in any private cache?
⑤ Is the line present only in the requester's private cache?
⑥ Increment the alarm counter
⑦ Evict a random line
⑧ Evict the selected line
⑨ Is alarm counter > threshold?
⑩ Generate interrupt

Figure 3.1: SHARP replacement algorithm.

bandwidth. We discuss potential attack strategies and the maximum attack bandwidth in Section 3.3.

From this discussion, we see that SHARP has very general applicability, requires no code modification (unlike [64, 65]), and does not partition the cache among processes (unlike [62, 101]). It allows multiple processes to dynamically share the entire shared cache, while transparently protecting against conflict-based side channel attacks.

SHARP requires hardware modifications to implement its replacement policy. Specifically, SHARP must be aware of what lines within the shared cache are present in the private caches. Such information is needed in operations ② and ④ of Figure 3.1. In the subsequent subsections, we present three different ways of procuring this information.

Using Core Valid Bits In SHARP, each line in the shared cache is augmented with a bitmap with as many bits as cores. The bit for core i is set if the line is present in core i's private cache. These are the *Presence* bits used in directory-based protocols [102]. For example, in Intel, they are used in multicores since the Nehalem microarchitecture [103], where they are called Core Valid Bits (CVB).

In this first design, SHARP simply leverages these bits to determine the information needed in operations ② and ④ of Figure 3.1. Note, however, that these bits carry *conservative* information. This means that if bit i is set, core i *may* have the line in its private cache, while if bit i is clear, core i is guaranteed not to have the line in its private cache. Such conservatism stems from silent evictions of non-dirty lines from private caches; these evictions do not update the CVB bits. As a result, the CVB bits will still show the evicting core as having a copy of the line in its private cache. Overall, this conservatism will cause Steps 2 and 3 in Figure 3.1 to be executed more often than in a precise scheme. However, correctness is not compromised.

Using Queries A shortcoming of the previous design is that it often ends-up assuming that shared cache lines are present in more private caches than they really are. As a result, a process may unnecessarily fail to find a victim in Step 1 and end-up victimizing its own lines in Step 2, or unnecessarily fail to find a victim in Step 2 and end-up raising an exception.

To solve this problem, this second SHARP design extends the first one with core queries. Specifically, Step 1 in Figure 3.1 proceeds as usual; it often finds a victim. In Step 2, however, as each line is examined in order based on its replacement priority, the SHARP hardware queries the private caches of the cores that have the CVB bit set for the line, to confirm that the bit is indeed up to date.

The CVBs of the line are refreshed with the outcome of the query. With the refresh, the CVBs may show that, in reality, the line is in no private cache, or only in the private cache of the requesting processor. In this case, the line is victimized and the replacement algorithm terminates; there is no need to examine the other lines.

As a line's CVBs are refreshed, the line is considered to be accessed, and is placed in its corresponding position in the replacement priority. This is done to ensure that such a line is not considered and refreshed again in the very near future.

This design generally delivers higher performance than the first one. The reason is that the queries of private caches refresh the CVB bits, obtaining a more accurate state of the system for the future. Note that the queries are typically hidden under the latency of the memory access that triggered them in the first place.

Similar query-based schemes have been proposed in the past. They have been used to reduce inclusion victims with the aim of improving performance [96].

Using Core Valid Bits and Queries A limitation of the previous design is that it does not scale well. For multicores with many cores, the latency of the queries may not be hidden by the cache miss latency. Moreover, the traffic induced by the queries may slow down other network requests. Consequently, we present a third SHARP design that reduces the number of queries.

Specifically, in Step 2 of Figure 3.1, SHARP only sends queries for the first *N* lines examined. For the remaining lines in the set, SHARP uses the CVBs as in the first scheme. As usual, Step 2 finishes as soon a victim line is found. There are no other changes relative to the second design.

3.2.2 Protecting Against Flush-based Attacks

As discussed in Section 3.1, there is no need to use `clflush` in user mode for pages that are read-only or executable. Hence, in user mode, SHARP only allows `clflush` to be performed on pages with write permissions.

With this restriction, sharing library code between processes and supporting page deduplication do not open up vulnerabilities to flush-based attacks. Specifically, if attacker and victim process share library code and the attacker invokes `clflush`, the attacker will suffer an exception because the addresses are execution-only. Hence, the victim process will not suffer inclusion victims in its cache. Similarly, if attacker and victim share a deduplicated page and the attacker invokes `clflush`, since the page is marked copy-on-write, the OS will trigger a page copy. All subsequent `clflush`es by the attacker will operate on the attacker's own copy. As before, the attack will be ineffective.

SHARP allows `clflush` to execute unmodified in kernel mode, as it is necessary for memory management.

3.2.3 Examples of Defenses

We give two examples to show how SHARP can successfully defend against conflict-based attacks. In the examples, victim and attacker share a target address, and the attacker uses the evict+reload attack (Section 2.2.1). Private caches are 4-way set-associative, and the shared one is 8-way. We consider first a single-threaded attacker and then a multi-threaded attacker.

Attack Using a Single-threaded Attacker Figure 3.2 shows the cache hierarchy, where the victim runs on Core *0* and the attacker on Core *1*. In Figure 3.2(a), the victim has loaded the target address, and the attacker has loaded four lines with conflict addresses. In Figure 3.2(b), the attacker loads four more lines with conflict addresses. Since the corresponding set in the shared cache only had three empty lines, one of the existing lines has to be evicted.

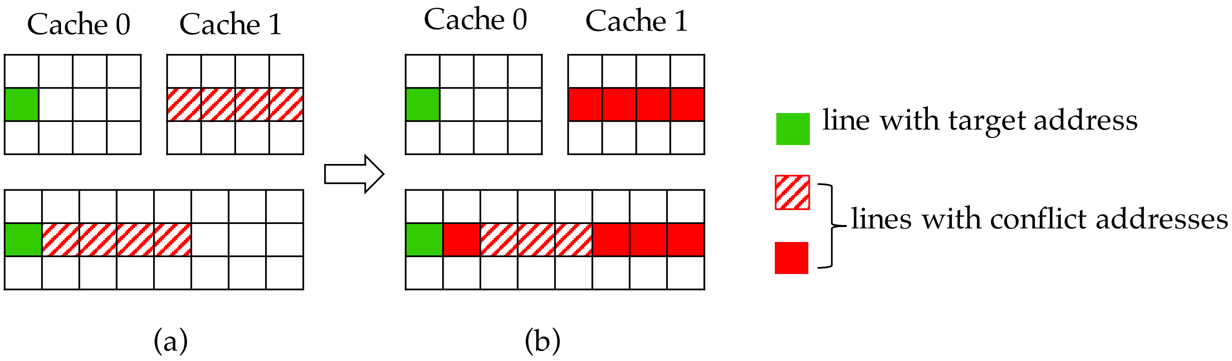


Figure 3.2: SHARP defending against a single-threaded attack.

SHARP forces the eviction of one of the old lines of the attacker — not the one with the target address.

Attack Using a Multi-threaded Attacker Figure 3.3 shows a cache hierarchy with four private caches, where the victim runs on Core *0* and three attacker threads run on Cores *1*, *2*, and *3*. In the figure, the victim has loaded the target address, and the attacker threads tried to evict it. Attacker *1* loaded four conflicting lines, and Attacker *2* three conflicting lines. If Attacker *2* now loads another conflicting line, it will only victimize one of its own lines. The same is true for Attacker *1*. SHARP is protecting the target address.

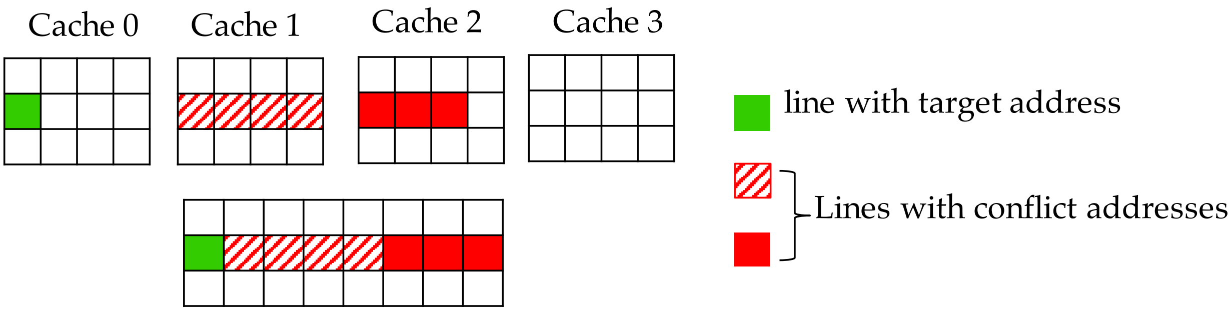


Figure 3.3: SHARP defending against a multi-threaded attack.

To have a chance to evict the target address, a third Attacker thread (Attacker *3*) needs to load a conflicting line. However, such access will only evict a *random* line in the set, and it will increment the alarm counter. Additional lines loaded by Attacker *3* will only victimize Attacker *3*'s line. To be able to cause multiple random replacements, the Attacker threads must be highly coordinated to ensure that, for each round of attack, at least one Attacker thread does not occupy any line in the corresponding shared cache set.

3.3 DISCUSSION

3.3.1 Handling Passive Attacks

SHARP only targets active cache-based side channel attacks. To put SHARP in perspective, we examine how it handles passive attacks.

Initial Access Vulnerability There is one type of passive cache-based side channel attack where the attacker simply wants to know whether the program execution loaded a given target address. For example, the attacker repeatedly loads the target address and evicts it, while timing the latency of the load. After the victim loads the target address, the load by the attacker is fast because it hits in the cache. We call this vulnerability the Initial Access vulnerability. The SHARP designs that we have presented are not able to thwart it. This is because, in these attacks, the attacker does not need to evict the lines that have been accessed by the victim.

The Initial Access vulnerability can be thwarted by adopting the preloading techniques proposed in [64, 104]. They involve loading all the security-sensitive addresses into the cache, so that the attacker cannot know which address the victim really needs to access. Such loading can be done with plain loads or with prefetches.

Exploiting Self-Conflicts Since the private cache used by the victim process has limited size and associativity, it is possible that a target address gets evicted by the victim's own accesses due to lack of space. We call this eviction a cache self-conflict. After the eviction, the SHARP designs that use queries may detect that the line is no longer in the private cache and pick it as a replacement victim in the shared cache. Hence, it is possible for an attacker to exploit victim's self-conflicts in the private cache to bypass SHARP's protection and mount a cache-based side channel attack.

In practice, if the victim's working set fits in the private caches, mounting such an attack is very difficult. The reason is that the attacker has no control on the way that lines evict each other in the private cache of the victim. In addition, the attacker can at best find out when a line with the target address was *evicted*, but not when the target address was last *accessed* before the eviction.

However, if an application's working set is much larger than the private cache size, self-conflicts can be exploited to leak useful information [105]. A potential solution is to modify the victim program to hide secret-dependent accesses inside the private cache, because SHARP can help hide the victim's private cache states.

Exploiting Self-Reuses Another type of passive cache attack exploits data self-reuses within the victim. For example, in a cache-collision attack [45], when the victim accesses a sequence of addresses, a "cache collision" happens if two addresses belong to the same cache line. Depending on the number of collisions, the victim may take different times to execute. SHARP cannot handle this attack. Potential solutions either require rewriting the program or using the Random Fill Cache [70].

3.3.2 Alarm Threshold and Attack Bandwidth

SHARP's alarm mechanism aims to detect side channel attacks that bypassed the first and second steps in the SHARP replacement algorithm. The alarm is incremented every time an attacker generates an inclusion victim in another core's private cache. As discussed in Section 3.2, inclusion victims may also be caused by benign applications under SHARP. Thus, we introduce an alarm threshold to filter false positives. Similar to other detection mechanisms, using a threshold is effective in limiting the bandwidth of side channel attacks, but cannot eliminate the channel. Kumar et al. [106] also pointed out the threshold dilemma. If the threshold is too low, the detection mechanism misclassifies many benign applications as attacks and suffers many false positives. If the threshold is too high, attackers can easily reduce the attack bandwidth to trigger a small enough number of alarms and avoid being detected.

We consider an implementation of SHARP which allows T alarms to be triggered for every W cycles. Assume that every time an attacker creates an inclusion victim, the target line is evicted from the victim's private cache. In this case, the attacker can achieve the following maximum bandwidth:

$$Bandwidth_{MAX} = \frac{T}{W} \tag{3.1}$$

Considering that SHARP uses a random replacement policy in Step 3, the attacker actually has to trigger multiple inclusion victims to make sure the target line is successfully evicted. If the LLC associativity is W_{LLC}, when using the random replacement policy, the probability of selecting the victim's target address to evict is $\frac{1}{W_{LLC}}$. Thus, the average bandwidth that can be achieved by a real attacker is:

$$Bandwidth_{AVG} \leq \frac{T}{W \times W_{LLC}} \tag{3.2}$$

Given a 2 GHz core with a 16-way LLC, $T = 2000$, and $W = 10^9$, which keeps false positives to a very low value (Section 3.4), the maximum attack bandwidth allowed by

SHARP is $4kbps$ and the average bandwidth is $0.25kbps$. This is much lower than the bandwidth of a practical LLC side channel attack [6], whose effective bandwidth is 200 to $1000kbps$ — triggering an inclusion victim every 2,000-10,000 cycles. Note that the above attack bandwidth can be achieved only if the attacker bypasses the first two steps in SHARP. It requires the attacker to use more than one thread as discussed in Section 3.2.3. Moreover, the timing of the requests from the multiple attacker threads needs to be finely coordinated, to ensure that, at every round of attack, at least one attacker thread does not occupy any line in the shared cache set.

3.3.3 Hardware Needs and Performance Impact

SHARP has modest hardware requirements. As per Section 3.2.1, it needs presence bits in the shared cache (i.e., the CVBs) and cache queries. The CVBs are already present in Intel multicores to support cache coherence, and can be reused. In directory-based multiprocessors that use limited-pointer directories, SHARP can be modified to also reuse the hardware.

To support queries, SHARP adds two additional messages to the coherence protocol, namely a query request and a query reply. The cache controller needs corresponding states to handle the two new messages. Such modification has also been used by Intel researchers to improve cache management [96].

SHARP induces negligible average performance degradation. This is because, unlike schemes that explicitly partition the shared cache among threads (e.g., [62, 101]), SHARP allows multiple threads to dynamically share a cache flexibly. In addition, the queries are performed in the background, in parallel to servicing a cache miss from memory. In practice, the great majority of the replacements that use queries are satisfied with the first query.

It can be argued that, in some cases, SHARP will cause a thread to be stuck with a single way of the shared cache, and repeatedly victimize its own private cache lines. This may be the case with the victim thread in Figure 3.3. While such case is possible, it is rare. Recall that the lines in a set in the private cache can map to multiple sets in the bigger, shared cache (say around 8 or so). The pathological case happens when many referenced lines across all cores map to the same set in both private and shared caches, and the shared-cache associativity is not enough. While possible, this case is rare, only temporary, and only affects the relevant cache set.

3.4 EVALUATION

3.4.1 Experimental Setup

To evaluate SHARP, we modify the MARSS [107] cycle-level full-system simulator. We model a multicore with 2, 4, 8, or 16 cores. Each core is 4-issue and out-of-order, and has private L1 and L2 caches. All cores share a multi-banked L3 cache, where the attacks take place. The chip architecture is similar to the Intel Nehalem [103]. The simulator runs a 64-bit version of Ubuntu 10.4. Table 3.1 shows the parameters of the simulated architecture. Unless otherwise indicated, caches use the pseudo-LRU replacement policy.

Parameter	**Value**
Multicore	2–16 cores at 2.5GHz
Core	4-issue, out-of-order, 128-entry ROB
Private L1 I-Cache/D-Cache	32KB each, 64B line, 4-way, Access latency: 1 cycle
Private L2 Cache	256KB, 64B line, 8-way, Access latency: 5 cycles after L1
Query from L3 to L2	3 cycle network latency each way
Shared L3 Cache	2MB bank per core, 64B line, 16 way, Access latency: 10 cycles after L2
Coherence Protocol	MESI
DRAM	Access latency: 50ns after L3

Table 3.1: Parameters of the simulated architecture.

We evaluate the 7 configurations of Table 3.2, which have different L3 line replacement policies: *baseline* uses the conventional pseudo-LRU policy; *cvb*, *query*, and *SHARPX* use the SHARP designs of Section 3.2.1. *SHARPX* includes 4 configurations (*SHARP[1-4]*), which vary based on the maximum number N of queries emitted. Recall that, for a given set, a query needs to be fully completed before a second one can be initiated.

Config.	**Line Replacement Policy in L3**
baseline	Pseudo-LRU replacement.
cvb	Use CVBs in both Step 1 and 2.
query	CVBs in Step 1 & queries in Step 2.
SHARPX	CVBs in Step 1. In Step 2, limit the max number of queries to X, where $X = 1, 2, 3$, or 4.

Table 3.2: Simulated LLC line replacement configurations.

3.4.2 Proof-of-Concept Defense Analysis

In this section, we evaluate the effectiveness of SHARP against two real cache-based side channel attacks. We implement the attacks using *evict+reload*, which consists of the attacker evicting the target address and then accessing it. If, in between, the victim has accessed the target address, the attacker's reload access hits in the cache; otherwise, the attacker's reload access misses.

To achieve page sharing between attacker and victim, the attacker *mmaps* the victim's executable file or shared library into the attacker's virtual address space. To select conflict addresses, the attacker first accesses the system files (i.e., */proc/$pid/pagemap* on Linux) to identify the physical addresses of target addresses. It then selects 16 addresses that map to the same L3 set as each of the target addresses to form an eviction set. When performing the evict operation, the attacker accesses the 16 addresses twice to ensure that the target address is replaced. When doing the reload operation, the attacker accesses the target address and measures the access time using *rdtsc*. Based on the time measured, it determines if it is an L3 hit. If so, it knows that the address has been accessed by the victim.

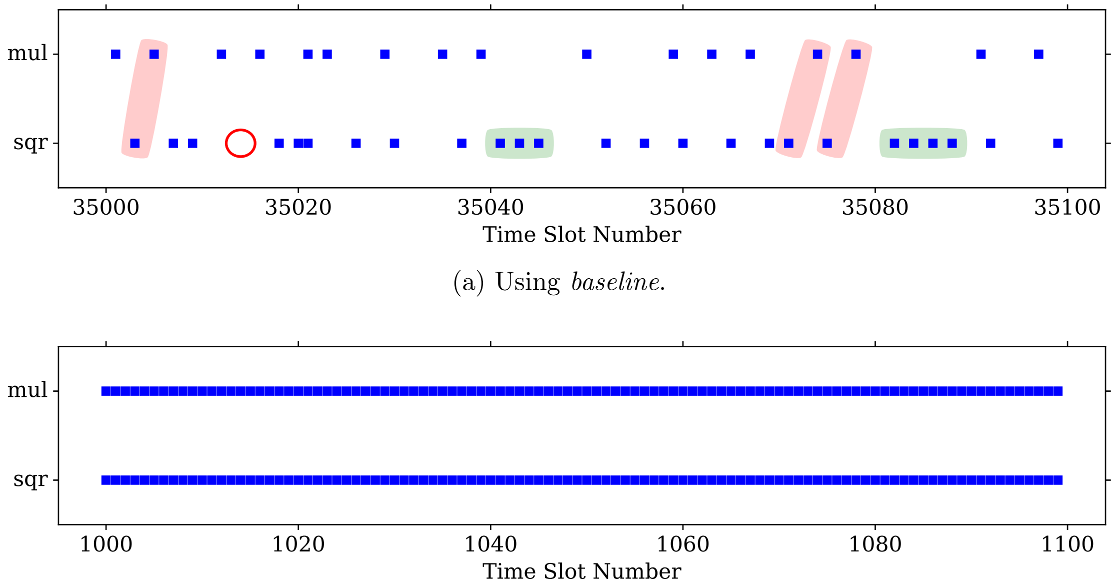
We measure the L3 hit and miss time for our architecture. We find that, on average, an L3 hit takes 48 cycles, and an L3 miss 170 cycles. Hence, we use 100 cycles as a threshold to decide if it is a hit or a miss.

In the following attacks, we launch the victim process on one core and the attacker on another. We show results for the *SHARP4* configuration; the other configurations work equally well in terms of defense.

Defending against Attacks on GnuPG Our first attack example targets GnuPG, a free implementation of the OpenPGP standard. The modular exponentiation in GnuPG version 1.4.13 uses a simple Square-and-Multiply algorithm [42]. The calculation is shown in Algorithm 2.1, and is described in Section 2.2.1.

In this attack, the attacker divides the time into fixed time slots of 5,000 cycles, and monitors for 10,000 time slots. In each time slot, it evicts and reloads two target addresses: the entry points of the `sqr` and `mul` functions (Algorithm 2.1). Figure 3.4 shows the result of 100 time slots for the *baseline* and *SHARP4* configurations. In the figure, a dot represents a cache hit in the reload of the corresponding target address.

In *baseline*, when a hit occurs, it is because the victim has accessed the target address during the interval between evict and reload. In Figure 3.4(a), we see the pattern of victim accesses to `sqr` and `mul`. When a `sqr` hit is followed by a `mul` hit, the value of the bit in the exponent vector is 1. The figure highlights three examples of this case with a shaded vertical



(a) Using *baseline*.

(b) Using *SHARP4*.

Figure 3.4: Cache hits on the target addresses by the attacker process in GnuPG.

pattern. When a `sqr` hit is not immediately followed by a `mul` hit, the value of the bit in the exponent vector is 0. The figure highlights two examples of multiple `sqr` hits in a row with a shaded horizontal pattern. In some cases, the timing is such that the evict follows the victim's access. In that case, the reload may miss an access. The figure highlights one such example with a circle. Even with some such misses, the attacker can successfully attain most of the bits in the exponent vector, which is enough for the attack to succeed.

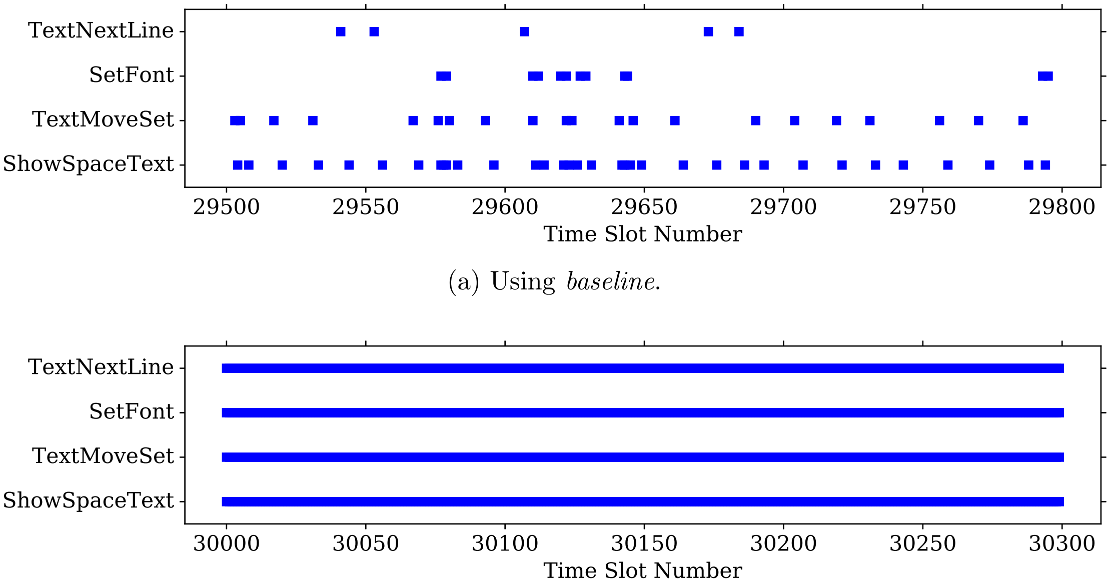
Consider now *SHARP4*. The first time that the victim calls `sqr` and `mul`, the target addresses are loaded into the shared cache and into the victim's private cache. Then, *SHARP4* prevents the attacker from evicting the target addresses from the shared cache. As a result, every single reload by the attacker will hit in the cache. The result, as shown in Figure 3.4(b), is that the attacker in unable to glean any information from the attack.

Defending against Attacks on Poppler Our second attack example targets Poppler, a PDF rendering library that is widely used in software such as Evince and LibreOffice. We select *pdftops* as the victim program. Pdftops converts a PDF file into a PostScript file. The execution of pdftops is very dependent on the input PDF file. Hornby et al. [47] design an attack that probes the entry points of four functions in pdftops that allow the attacker to distinguish different input PDF files with high fidelity:

- `Gfx::opShowSpaceText(Object*, int)`
- `Gfx::opTextMoveSet(Object*, int)`

- `Gfx::opSetFont(Object*, int)`
- `Gfx::opTextNextLine(Object*, int)`

Their attack consists of three stages: training, attack, and identification. In the training stage, they collect the probing address sequence for different input PDF files multiple times, to obtain the unique signature for each file. In the attack stage, the attacker records the target address sequence of the victim as it executes pdftops with an input PDF file. In the identification stage, the attacker computes the Levenshtein distance[1] [108] between the victim's access sequence and all of the training access sequences. The training sequence with the smallest Levenshtein distance to the victim's is assumed to correspond to the input file that the victim used. By using this approach, they can reliably identify 127 PDF files on a real machine.



(a) Using *baseline*.

(b) Using *SHARP4*.

Figure 3.5: Cache hits on the target addresses by the attacker process in Poppler.

In our attack, the attacker monitors the entry points of those functions using evict+reload. The attacker divides time into fixed time slots of 10,000 cycles each. During each time slot, it evicts and reloads the 4 target addresses. Figure 3.5 shows the reload hits in 300 slots. In *baseline* (Figure 3.5(a)), we can clearly monitor the execution of the 4 functions over time. Since each input PDF file results in a different execution order and frequency for these functions, the pattern can be used as a signature to uniquely identify the input file. In

[1] Levenshtein distance is the smallest number of basic edits (single-character insertions, deletions, and replacements) needed to bring one string to the other.

SHARP4 (Figure 3.5(b)), the reloads always hit, which makes it impossible to distinguish different input files by their probed cache behavior.

3.4.3 Performance Evaluation of Single-threaded Application Mixes

In this section and next section, we evaluate the performance impact of SHARP using both mixes of single-threaded applications (SPECInt2006 and SPECFP2006 [109]), and multi-threaded applications (PARSEC [110]).

We start by evaluating mixes of 2 SPEC applications at a time, using 2 cores with a total L3 size of 4MB. To choose the mixes, we use the same approach as Jaleel et al. [96]. We group the applications into three categories according to their cache behavior [111]: SW (small working set), MW (medium working set), and LW (large working set). SW applications, such as *sjeng, povray, h264ref, dealII*, and *perlbench*, fit into the L2 private caches. MW applications, such as *astar, bzip2, calculix*, and *gobmk*, fit into the L3. Finally, LW applications, such as *mcf* and *libquantum*, have a footprint larger than the L3. We choose a set of mixes similar to Jaleel et al. [96], which the authors suggest are representative of all mixes of the SPEC applications.

We use the *reference* input size for all applications. In each experiment, we start two applications and pin them to separate cores. We skip the first 10 billion instructions in each application; then, we simulate for 1 billion cycles. We measure statistics for each application.

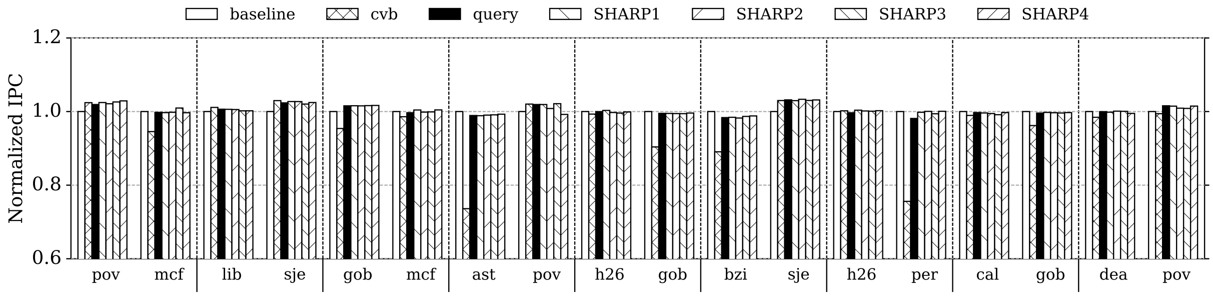


Figure 3.6: Normalized IPC of SPEC application mixes with different replacement policies on 2 cores.

Figure 3.6 shows the IPC of each application in each of the 9 mixes considered. For a given application, the figure shows bars for *baseline*, *cvb*, *query*, and *SHARP[1,4]*, which are all normalized to *baseline*. In the figure, higher bars are better. Figure 3.7 shows the L3 misses per kilo instruction (MPKI). It is organized as Figure 3.6 and, as before, bars are normalized to *baseline*.

From Figure 3.6, we see that the performance of the applications in *query* and *SHARP[1,4]*

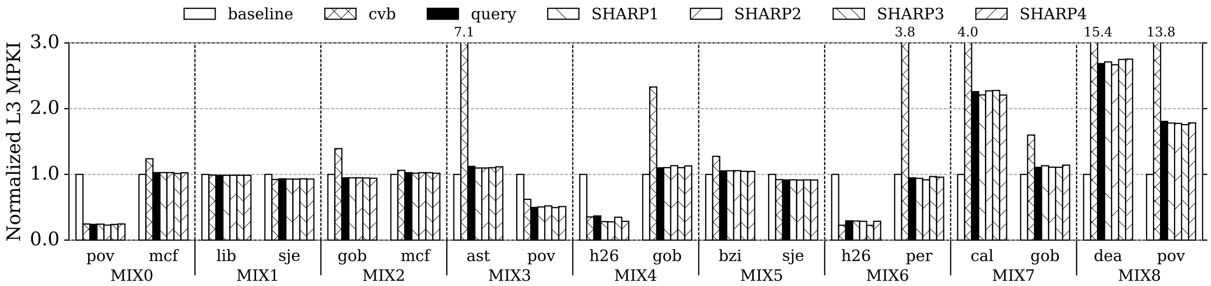


Figure 3.7: Normalized L3 MPKI of SPEC application mixes with different replacement policies on 2 cores.

is generally similar to that in *baseline*. Hence, SHARP has a negligible performance impact. In addition, the reason why *query* and *SHARP[1,4]* all behave similarly is that, in the large majority of cases, the first query successfully identifies a line to evict.

The figure also shows that *cvb* is not competitive. For applications such as *astar* in MIX3 and *perlbench* in MIX6, *cvb* reduces the IPC substantially. The reason is that the imprecision in the *CVBs* causes suboptimal line replacement. In particular, threads end up victimizing themselves. In some of these applications, the relative MPKI increases substantially (Figure 3.7). Note, however, that these are normalized MPKI values. In these SW and MW applications, while the bar changes may seem large, they correspond to small changes in absolute MPKI. For example, the MPKI of *dealII* in MIX8 is 0.025 in *baseline*, and it increases by 15x to a still modest value of 0.392 in *cvb*.

Some of the mixes expose the effects of SHARP more than others. For example, the mixes that contain an SW and an LW application are especially revealing (e.g., MIX0). In these workloads, *SHARP[1,4]* helps the SW application retain some L3 ways for itself — rather that allowing the LW application to hog all the L3 ways as in *baseline*. As a result, the SW application increases its IPC and reduces its MPKI (*povray* in MIX0). At the same time, the LW application does not change its IPC or MPKI much (*mcf* in MIX0). The reason is that the LW application already had a large MPKI, and the small increase in misses has little effect.

3.4.4 Performance Evaluation of Multi-threaded Applications

We now evaluate PARSEC applications running on 4 cores with a total L3 size of 8MB. The applications' input size is *simmedium*, except for *facesim*, which uses *simsmall*. The applications run for the whole region of interest, with at least 4 threads, and with threads pinned to cores. For these applications, we report total execution time, rather than average

IPC, as the performance metric. This is because these applications have synchronization and, therefore, may execute spinloops.

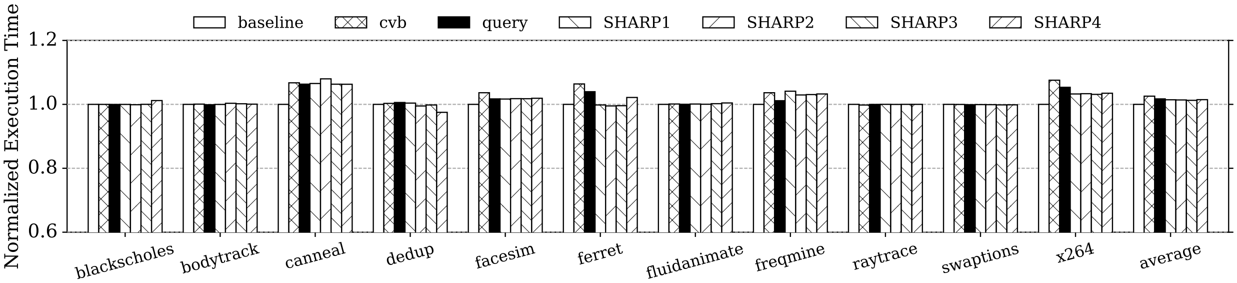


Figure 3.8: Normalized Execution Time of PARSEC applications with different replacement policies on 4 cores.

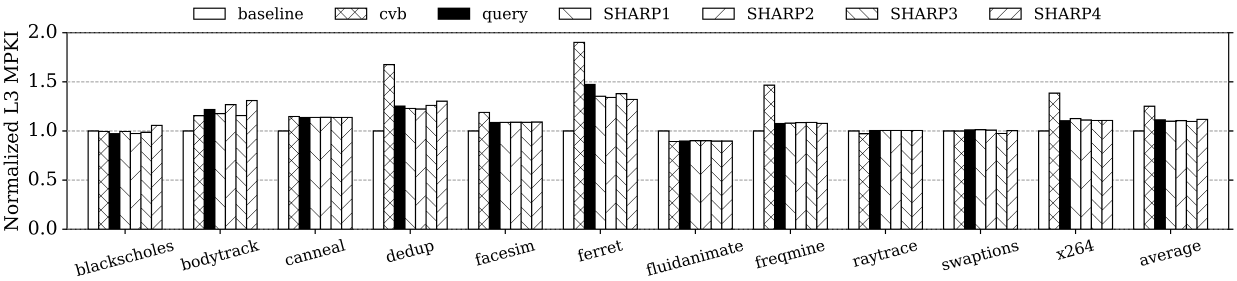


Figure 3.9: Normalized L3 MPKI of PARSEC applications with different replacement policies on 4 cores.

Figures 3.8 and 3.9 show the normalized execution time and L3 MPKI, respectively, for each application and for the average. These figures are organized as Figures 3.6 and 3.7. In Figure 3.8, lower is better.

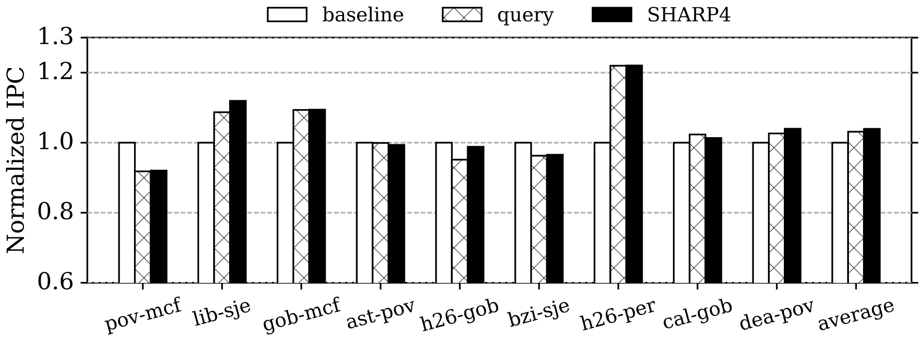
In this environment, threads share data and, therefore, a given cache line can be in multiple private caches. As a result, a thread may be unable to evict data that it has brought into the shared cache because another thread has reused the data and is caching it in its own private cache.

This property may have good or bad effects on the MPKI (Figure 3.9). For example, in *ferret*, the inability of a thread to evict shared data causes cache thrashing and a higher MPKI. On the other hand, in *fluidanimate*, the MPKI decreases slightly.

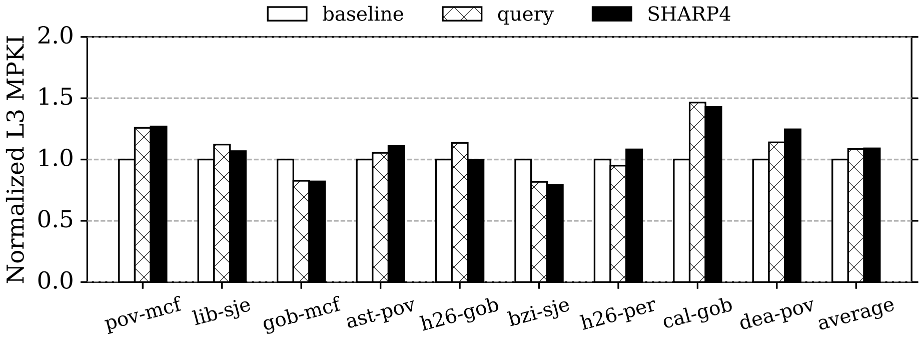
If we look at the execution time (Figure 3.8), we see that *query* and *SHARP[1,4]* have similar performance as *baseline*. The only difference is a modest slowdown of 6% in *canneal*. This application has a large working set, and the new replacement policy ends up creating more misses which, in turn, slow down the application. Overall, however, the impact of *query* and *SHARP[1,4]* on the execution time is negligible. *cvb* is slightly worse. The reason is that the imprecision of the CVBs causes a higher MPKI and a slightly slower execution.

3.4.5 Performance Evaluation on Scalability

To assess the scalability of SHARP, we run SPEC application mixes and PARSEC applications with larger core counts. Specifically, the SPEC application mixes run on 8 cores with a total L3 size of 16MB. The pairs of applications in each mix are the same as in Section 3.4.3, except that we run 4 instances of each of the two applications. We use the *reference* input sets, and collect statistics for 1 billion cycles after all applications have reached the region of interest.



(a) Normalized IPC.



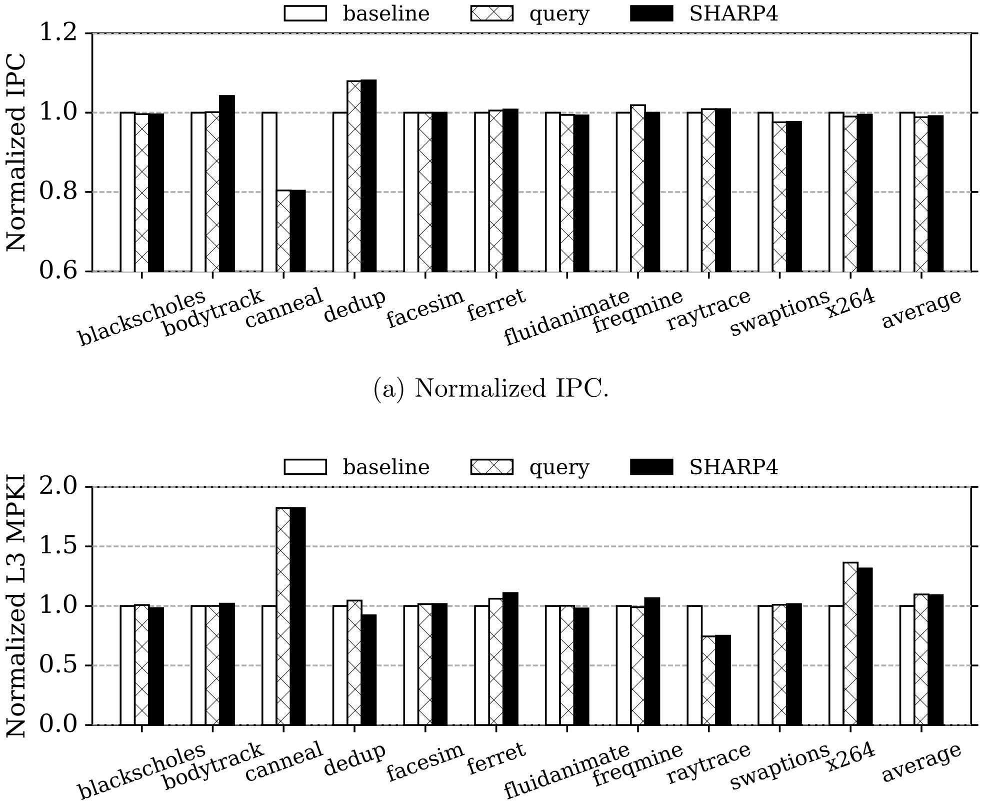
(b) Normalized L3 MPKI.

Figure 3.10: Normalized IPC and L3 MPKI of SPEC application mixes with different replacement policies on 8 cores.

Figures 3.10(a) and 3.10(b) show the normalized IPC and L3 MPKI, respectively, for the SPEC mixes. For each mix, we show the average of the 8 applications and, due to space limitations, only the *baseline*, *query*, and *SHARP*4 configurations. The figures also have bars for the average of all mixes.

The changes in the line replacement algorithm affect what data is kept in the L3 caches. Our SHARP designs try to avoid creating replacement victims in the private caches of other cores. As shown in Figure 3.10(a), sometimes this causes the average IPC of the applications to decrease (*povray+mcf*) and sometimes to increase (*h264ref+perlbench*). However, on average for all the mixes, the IPC under *SHARP*4 and *query* is about 3–4% higher than *baseline*. Overall, therefore, we conclude that SHARP has a negligible average performance impact.

As shown Figure 3.10(b), the L3 MPKI also goes up and down depending on the application mix, but the average impact is small. Note that, for a given application mix, it is possible that *SHARP*4 increases both the average IPC and the average MPKI. For example, in *libquantum+sjeng*, the average IPC goes up because *sjeng*'s IPC increases more than *libquantum*'s decreases. At the same time, *libquantum*'s average MPKI goes up more than *sjeng*'s goes down.



(a) Normalized IPC.

(b) Normalized L3 MPKI.

Figure 3.11: Normalized IPC and L3 MPKI of PARSEC applications with different replacement policies on 16 cores.

We also run PARSEC applications on 16 cores with a total L3 size of 32MB. The applications' input size is *simlarge*. Given the long simulation time, we report statistics for the first 1 billion cycles in the region of interest. Consequently, we report performance as IPC rather than execution time. Figures 3.11(a) and 3.11(b) show the normalized IPC and L3 MPKI, respectively. The figures are organized as Figures 3.10(a) and 3.10(b).

In Figure 3.11(a), we see that most of the applications have similar IPCs for *baseline*, *query*, and *SHARP*4. The one application where SHARP hurts IPC is *canneal*. This application has a very large working set. In *baseline*, the L3 MPKI of *canneal* is 5.18, compared to an MPKI lower than 1 for the other applications. In addition, there is fine-grained synchronization between threads. Data that is brought into the cache by one thread is used by other threads. This causes SHARP to avoid evicting such lines. The result is higher MPKI (Figure 3.11(b)) and lower IPC. This behavior is consistent with the one displayed for 4-core runs (Figure 3.8). On average across all the applications, however, *query* and *SHARP*4 have negligible impact on IPC and (to a lesser extent) on MPKI.

3.4.6 Alarm Analysis

Recall that when SHARP needs to evict a line from the shared cache, it looks for a victim that is not in any private cache or only in the private cache of the requester. If SHARP cannot find such a victim, it increments an alarm counter in the requesting core and evicts a random line in the set. For normal applications, the number of alarm increments is low. In an attack, however, the number of alarm increments will be very high. We now profile benign applications to find the appropriate threshold.

Table 3.3 shows the maximum alarm count observed per 1 billion cycles in any core while running benign workloads. Specifically, we run the 8-threaded SPEC mixes and 16-threaded PARSEC applications of Section 3.4.5, and try the *cvb*, *query*, *SHARP*1, *SHARP*2, *SHARP*3, and *SHARP*4 configurations. The last row of the table shows the maximum number across applications for each configuration.

From the table, we see that *cvb* can trigger many alarms in multiple workloads. These high numbers are due to the lack of precision of this replacement policy, where the CVBs can be stale. Consequently, *cvb* is not recommended.

With the other policies, the number of alarms decreases substantially. This is because the policies refresh CVBs. For most workloads, the number of alarms is less than 100 per 1 billion cycles. A few SPEC mixes, such as *libquantum+sjeng* and *gobmk+mcf* reach several thousand alarms. These alarms occur because four instances of memory-intensive applications (*libquantum* and *mcf*) cause contention on many cache sets, and force evictions of cache lines belonging to their companion computation-intensive applications (*sjeng* and

Applications.	*cvb*	*query*	$SHARP1$	$SHARP2$	$SHARP3$	$SHARP4$
pov-mcf	285238	89	7285	171	121	84
lib-sje	1618715	1460	4747	2033	1820	1403
gob-mcf	549976	687	10001	1160	1426	1045
ast-pov	22701	19	1774	137	36	7
h26-gob	511	0	16	2	0	0
bzi-sje	38669	7	177	9	7	2
h26-per	60536	1	974	184	6	2
cal-gob	132169	0	37	25	33	1
dea-pov	3	0	0	1	0	0
blackscholes	0	0	0	0	0	0
bodytrack	0	0	0	0	0	0
canneal	153165	37	1192	39	43	37
dedup	145079	13	410	32	18	36
facesim	46409	12	97	32	16	1
ferret	91443	6	2097	102	15	9
fluidanimate	25643	2	556	144	26	3
freqmine	0	0	0	0	0	0
raytrace	10013	1	85	5	1	1
swaptions	0	0	0	0	0	0
x264	35897	2	423	10	5	14
MAX	1618715	1460	10001	2033	1820	1403

Table 3.3: Alarms per 1 billion cycles in benign workloads.

gobmk). SHARP is unable to find a line that only exists in the requester's private cache, and the alarm counter is incremented.

PARSEC applications with very little shared data, such as *blacksholes* and *swaptions*, have no alarms. This is because SHARP can always find a line that exists only in requester's private cache. Applications with a larger amount of sharing between threads (*ferret* and *canneal*) have a relatively higher number of alarms. Even in such cases, however, the number of alarms is orders of magnitude lower than when an attack takes place.

Looking at the last row of the table, we see that $SHARP4$, $SHARP3$, and *query* have less than 2,000 alarms per 1 billion cycles in the worst case. Of them, $SHARP4$ is the best design. Hence, we recommend to use $SHARP4$ and use a threshold of 2,000 alarm events in 1 billion cycles before triggering an interrupt.

3.5 CONCLUSION

To combat the security threat of active cross-core cache-based side channel attacks, this chapter made three contributions. First, it made an observation for an environment with an inclusive cache hierarchy: when the attacker evicts the target address from the shared cache, the address will also be evicted from the private cache of the victim process, creating an inclusion victim. Consequently, to disable cache attacks, the attacker should be prevented from triggering inclusion victims in other caches.

Next, the chapter used this insight to defend against active cache-based side channel attacks with SHARP. SHARP is composed of two parts. It introduces a new line replacement algorithm in the shared cache that prevents a process from creating inclusion victims in the caches of cores running other processes. It also slightly modifies the `clflush` instruction to protect against flush-based cache attacks. SHARP is highly effective, needs only minimal hardware modifications, and requires no code modifications.

Finally, this chapter evaluated SHARP on a cycle-level full-system simulator and tested it against two real-world attacks. SHARP effectively protected against these attacks. In addition, SHARP introduced negligible average performance degradation to workloads with SPEC and PARSEC applications.

CHAPTER 4: DIRECTORY ATTACKS

This chapter presents the first two directory-based side channel attacks on non-inclusive cache hierarchies. As modern caches move away from inclusive cache hierarchies to non-inclusive ones, previous cache attack strategies have been called into doubt, mainly due to the assumption that attackers should be unable to create "inclusion victims". However, we find that, in a non-inclusive cache, inclusion victims can be created via directory conflicts. We present directory attacks and prove that the directory can be used to bootstrap conflict-based cache attacks on any cache hierarchy, including inclusive, non-inclusive and exclusive ones.

Despite the past successes of cache-based side channel attacks, the viability of shared cache attacks has been called into question on modern systems due to recent trends in processor design. To start with, many prior attacks [28, 39, 46, 48] can be mitigated out of the gate, as virtualized environments are now advised to disable shared virtual memory between VMs [112].

Without sharing virtual memory with a victim, the adversary must carefully consider the *cache hardware architecture* in mounting a successful attack. This is where problems arise. First, modern cache hierarchies are becoming *non-inclusive or exclusive*. Prior LLC attacks without shared virtual memory (e.g., [6]) rely on LLCs being *inclusive*, as this gives adversaries the ability to evict cache lines that are resident in the victim's private caches. Non-inclusive cache behavior is significantly more complicated than that of inclusive caches (Section 4.1). Second, modern LLCs are physically partitioned into multiple *slices*. Sliced LLCs notoriously complicate attacks, as the mapping between victim cache line address and cache slice is typically proprietary. Taken together, these challenges cause current LLC attacks to fail on modern systems (e.g., the Intel Skylake-X [27]).

Modern systems are moving to non-inclusive cache hierarchies due to the redundant storage that inclusive designs entail. Indeed, AMD servers have always used exclusive LLCs [28], and Intel servers are now moving to this design [27]. We expect the trend of non-inclusive caches to continue, as the cost of inclusive caches grows with core count.

In this chapter, we design a novel cross-core cache attack that surmounts all of the above challenges. Specifically, our attack does not require the victim and adversary to share cores or virtual memory, and succeeds on state-of-the-art *sliced non-inclusive caches*, such as those in Skylake-X [27]. Our key insight is that in a machine with non-inclusive cache hierarchies, we can still attack the directory structure. Directories are an essential part of modern cache hierarchies, as they maintain tracking information for each cache line resident in the cache hierarchy.[1] Since the directory must track *all* cache lines, and not just cache lines in the LLC, it offers an attack surface similar to that of an inclusive cache. Indeed, our work suggests that conflict-based LLC attacks (on inclusive, non-inclusive or exclusive cache hierarchies) should target directories, not caches, as directories are a homogeneous resource across these different cache hierarchy designs.

To summarize, this chapter makes the following contributions:

- We develop an algorithm to find groups of cache lines that completely fill a given set of a given slice in a non-inclusive LLC (called an *Eviction Set*). This modernizes prior work on Eviction Set creation, which only works for sliced inclusive LLCs.
- Using our Eviction Sets, we reverse engineer the directory structure in Skylake-X, and identify vulnerabilities in directory design that can be leveraged by cache-based side channel attacks.
- Based on our insights into the directory, we present two attacks. The first is a Prime+Probe attack on sliced non-inclusive LLCs. Our attack does not require the victim and adversary to share cores or virtual memory. The second attack is a novel, high-bandwidth Evict+Reload attack that uses multi-threaded adversaries to bypass non-inclusive cache replacement policies.
- We use our two attacks to attack square-and-multiply RSA on the modern Intel Skylake-X server processor. Both of these attacks are firsts: although prior work implemented an Evict+Reload attack on non-inclusive LLCs, it cannot attack RSA due to its low-bandwidth. Finally, we construct efficient covert channels for sliced non-inclusive LLCs.

4.1 THE CHALLENGE OF NON-INCLUSIVE CACHES

Previous cross-core cache side channel attacks only work for inclusive cache hierarchies (e.g., [6, 18]). In a non-inclusive cache hierarchy, attackers must overcome the two main challenges that we describe next. In this discussion, we assume that the `clflush` instruction is disabled [52] and that shared memory between attacker and victim has been disabled [112].

[1] This should not be confused with the "directory protocol" used in multi-socket attacks that assume shared virtual memory between the adversary and victim [28].

4.1.1 Lack of Visibility into the Victim's Private Cache

In a non-inclusive cache hierarchy, an attacker running on a core seemingly cannot evict an address from another core's private cache — i.e., it cannot create an *inclusion victim* in the second core's cache. To see why, consider Figure 4.1, which shows a shared LLC and two private caches. The attacker runs on Cache 1 and the victim on Cache 0. The target line is shown in a light shade in Cache 0.

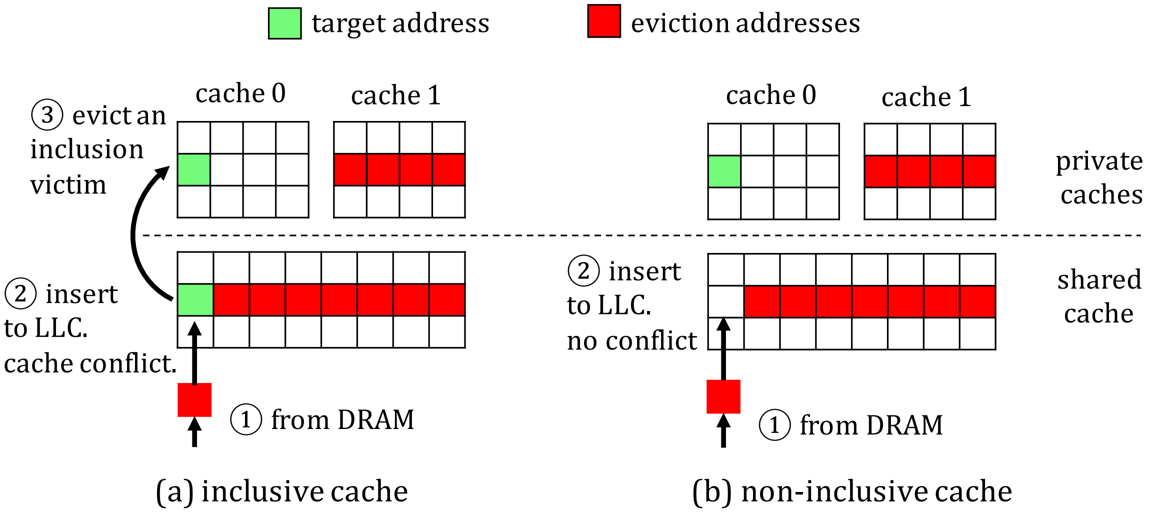


Figure 4.1: Attempting to evict a target line from the victim's private cache in inclusive (a) and non-inclusive (b) cache hierarchies.

Figure 4.1(a) shows an inclusive hierarchy. An LLC set contains lines from the attacker (in a dark shade) plus the target line from the victim (in a light shade). The attacker references an additional line that maps to the same LLC set. That line will evict the target line from the LLC, and because of inclusivity, also from private Cache 0, creating an inclusion victim. The ability to create these inclusion victims on another cache is what enables cross-core attacks.

Figure 4.1(b) shows a non-inclusive hierarchy. In this case, the target line is in the victim's cache, and not in the LLC. Consequently, when the attacker references an additional line that maps to the same LLC set, there is no invalidation sent to Cache 0. The attacker has no way to create inclusion victims in the victim's cache.

4.1.2 Eviction Set Construction is Hard

An *Eviction Set (EV)* is a collection of addresses that are all mapped to a specific cache set of a specific cache slice, and that are able to evict the current contents of the whole set in that slice. In a slice with W_{slice} ways, an eviction set must contain at least W_{slice} addresses to occupy all the ways and evict the complete contents of the set. We refer to *Eviction Addresses* as the addresses in an Eviction Set.

In a later section, we will show that we perform Prime+Probe and Evict+Reload attacks in non-inclusive cache hierarchies using an Eviction Set (EV). However, the algorithm used to create an EV in inclusive cache hierarchies [6] does not work for non-inclusive hierarchies. Creating an EV in non-inclusive hierarchies is harder. The reason is that it is less obvious what memory accesses are required to reliably evict the target line, which is currently in the private cache, from the entire cache hierarchy.

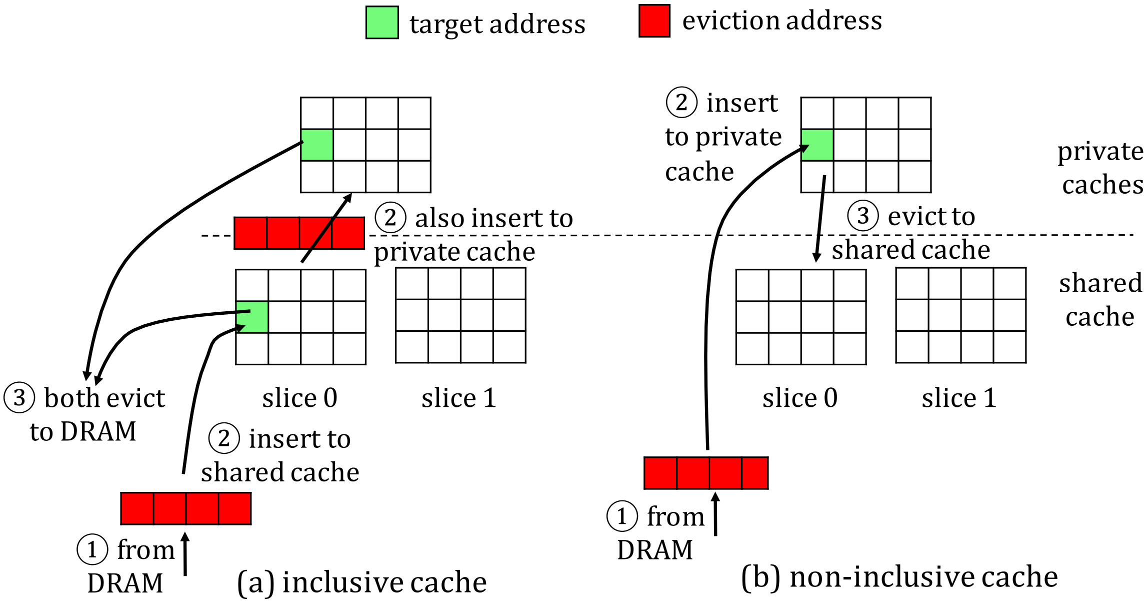


Figure 4.2: Attempting to evict a target line in inclusive (a) and non-inclusive (b) cache hierarchies. Victim and attacker run on the same core.

To see why, consider Figure 4.2, which shows a private cache and two slices of the shared LLC. Victim and attacker run on the same core. Figure 4.2(a) shows an inclusive hierarchy. The target line is in the private cache and in one slice of the LLC. To evict the target from the cache hierarchy, the attacker only needs to reference enough lines to fill the relevant set in the corresponding slice of the LLC. This is because, as these lines fill the set, they will also fill the set in the private cache, and evict the target line from it. This is the EV, shown in a dark shade. The order and number of accesses to each of the lines in the EV required to evict the target address is determined by the replacement algorithm used in the LLC slice.

Figure 4.2(b) shows a non-inclusive hierarchy. In this case, the target line is only in the private cache. As the core accesses the same dark cache lines as in Figure 4.2(a), the lines go first to the private cache, bypassing the LLC. The replacement algorithm used in the private cache will determine when the target line is evicted from the private cache into the LLC. When the target is evicted, it will go to one of the LLC slices, depending on the mapping of addresses to slices. Then, the core needs to evict enough lines into that LLC slice to create enough conflicts to evict the target line from the slice.

Overall, the order and number of accesses to each of the lines in the EV required to evict the target address is determined by multiple factors, including the replacement algorithm used in the private cache, the mapping of the line addresses to LLC slices, and the replacement algorithm used in the LLC slice. In the inclusive case, only the replacement algorithm in the LLC affects evictions.

We note that, in non-inclusive cache hierarchies, the replacement algorithms in the private caches and LLC slices can be quite sophisticated. What specific line is chosen as a replacement victim depends not only on the number and order of accesses to the lines, but also on the coherence state of the lines in the cache as well. Specifically, we have empirically observed that the replacement algorithm in the LLC slices tries to minimize the eviction of lines that are present in multiple private caches. This particular heuristic affects the ability to create an effective EV for Evict+Reload attacks, where lines are shared between attacker and victim.

We address these two challenges in two novel ways. To handle the difficulty of creating EVs, we propose a novel way to create EVs for non-inclusive caches in Section 4.2. Using EVs and other techniques, we reverse engineer the Intel Skylake-X directory structure in Section 4.3. This process reveals key insights into directory entry replacement policies and inclusivity properties. In particular, we derive conditions for when an attacker is able to use the directory to create inclusion victims in the private caches of a non-inclusive cache hierarchy. Based on our reverse engineering results, and the new EV construction methodology, we design effective "Prime+Probe" and "Evict+Reload" attacks in non-inclusive caches hierarchies in Section 4.4.

4.2 CONSTRUCTING EVICTION SETS

In this section, we present an EV construction algorithm for non-inclusive caches. Recall that an EV is a collection of memory addresses that fully occupy a specific cache set of a specific LLC slice. This is a core primitive that we use to reverse engineer the directory (Section 4.3) and later complete our attacks (Section 4.4).

Liu et al. [6] proposed an EV construction algorithm for sliced inclusive caches. However, it does not work for non-inclusive caches. We develop a new implementation for `check_conflict`, an important subroutine in the EV construction algorithm, that works on non-inclusive caches. We present evaluation results to demonstrate the effectiveness of our algorithm in Section 4.5.1.

4.2.1 The Eviction Set Construction Algorithm on Inclusive Caches

We first describe the EV construction algorithm, and then discuss the important role of the `check_conflict` function.

Algorithm 4.1: Constructing an eviction set.

```
Input  : candidate set $CS$
Output: $EV$
Function $find\_EV(CS)$:
    $EV = \{\}$
    $test\_addr$ = get a random addr from $CS$
    $CS' = CS - test\_addr$
    // make sure there are enough addresses to conflict with $test\_addr$
    if $check\_conflict(test\_addr, CS')$==$false$ then
        return fail
    end
    for $each\ addr\ in\ CS'$ do
        if $check\_conflict(test\_addr, CS' - addr)$==$false$ then
            // if conflict disappears, we know $addr$ contributes to
            // the conflict, and it should be in the EV
            insert $addr$ to $EV$
        else
            $CS' = CS' - addr$
        end
    end
    for $each\ addr\ in\ CS$ do
        if $check\_conflict(test\_addr, EV)$==$true$ then
            insert $addr$ to $EV$
        end
    end
    return $EV$
end
```

The complete EV construction algorithm that we used for non-inclusive caches is a slightly modified version of the algorithm proposed by Liu et al. [6], as shown in Algorithm 4.1. The `find_EV(Collection CS)` function takes a collection of addresses, which we call a Candidate

Set (CS) as input, and outputs an eviction set (EV) for one slice as output. For the algorithm to work, it is required that all the addresses in CS have the same LLC set index bits, and CS contains more than W_{slice} addresses for each slice. Such CS can be easily obtained by using a large number of addresses. To find EVs for all the slices within CS, we need to run the function the same number of times as the number of slices.

The function initializes EV as an empty set and selects a random address $test\_addr$ in CS (line 2-3). It then tries to construct an EV containing all the addresses which are mapped to the same slice and set as $test\_addr$ from CS. First, it creates a new set CS' by removing $test\_addr$ from CS (line 4), and then performs a sanity check to make sure CS' contains enough addresses to evict $test\_addr$ out of LLC using `check_conflict` (line 5-7).

The loop (line 8-14) performs the bulk of the work, checking whether an address is mapped to the same slice as $test\_addr$. Since CS' conflicts with $test\_addr$, if removing an address $addr$ causes the conflict disappear, we know that $addr$ contributes to the conflict, and $addr$ should be added to EV (line 9-10). Such addresses are kept in CS'. Addresses which are not strictly necessary to cause conflicts with $test\_addr$ are removed from CS' (line 12), and CS' should still conflict with $test\_addr$ after the remove operations. After the loop, we obtain a minimal EV with exactly W_{slice} number of addresses.

It is possible that there are more than W_{slice} addresses from the same slice as $test\_addr$ which have been conservatively removed in the loop. We use an extra loop (line 15-19) to find these addresses, by iteratively checking each address in the original CS to determine whether it conflicts with the obtained EV.

4.2.2 The Role of check_conflict in EV Construction Algorithm

The EV construction algorithm extensively uses a function [2] that we call

$$check\_conflict(Address\ x,\ Collection\ U) \tag{4.1}$$

This function checks if the addresses in Collection U conflict with x in the LLC. The function should return $true$ if U contains W_{slice} or more addresses, which are mapped to the same slice and set as x. The function should return $false$ otherwise. The EV construction algorithm works only if this function has very low false positive and false negative rates.

To see why these requirements are important, consider how `check_conflict` is used in the EV construction algorithm. The high-level idea is to use the function to test whether removing an address from a set can cause LLC conflicts to disappear. Starting with a

[2] This function is `probe` in [6].

collection U known to conflict with x in an LLC slice, one removes an address y from the collection U and obtains a new collection $U' = U - y$. If the conflict with x disappears when checking against U', then we know that y must contribute to the conflict. In such a case, y is considered as an eviction address for x. Clearly, the operation needs low false positive and false negative rates to precisely observe the disappearance of conflicts.

4.2.3 New check_conflict Function

We discuss why the `check_conflict` function designed by Liu et al. [6] has a high false negative rate when applied naïvely to non-inclusive caches. We then show how the function can be modified to work in non-inclusive caches. In the following discussion, we assume that all the addresses in U have the same LLC set index bits as x.

Algorithm 4.2: Baseline check_conflict for inclusive caches.

```
Function check_conflict (x, U):
    access $x$
    for each addr in U do
        access addr
    end
    $t$ = measure time of accessing $x$
    return $t \geq LLC\_miss\_threshold$
end
```

Baseline `check_conflict` [6] In Algorithm 4.2, the base function first accesses the target address x, ensuring that the line is cached. It then accesses all the addresses in U. If a later access to line x takes a short time, it means that the line is still cached. Otherwise, it means that the line has been evicted out of the cache due to cache conflicts caused by U. Thus, the access latency can be used to determine whether U contains enough addresses to evict x.

When applied to non-inclusive caches, this function has a high false negative rate. Specifically, when U contains enough addresses that, if they all were in the LLC, they would evict x, the function is supposed to return *true*. However, it may return $false$. To see how this false negative happens, consider a minimal U, which has exactly $\mathrm{W}_{\mathrm{slice}}$ addresses mapped to the same LLC slice as x. On non-inclusive caches, when accessing U, some of these $\mathrm{W}_{\mathrm{slice}}$ lines may remain in L2 and never be evicted from the L2 into the LLC. Hence, these addresses do not have a chance to conflict with x in the LLC, and x is not evicted, resulting in a false negative. Moreover, since the replacement algorithm of L2 is neither LRU nor pseudo-LRU, simply accessing U multiple times does not guarantee a small false negative rate, as we validate in Section 4.5.1.

Naïve New `check_conflict` To reduce the false negative rate, we need to flush all the lines in U from the L2 to the LLC. It would be convenient if we had a special instruction to do so, but such an instruction does not exist in x86. Hence, we leverage L2 conflicts to achieve the flush effect.

We create an extra collection of addresses, called *L2_occupy_set*, which contains W_{L2} addresses mapped to the same L2 set as U. When accessed, *L2_occupy_set* forces all lines in U to be evicted to the LLC. Our modified `check_conflict` function is shown in Algorithm 4.3. After accessing x and all the addresses in U as in the base function (line 2-5), the addresses in *L2_occupy_set* are accessed (line 6-8). In this way, every line in U gets evicted to the LLC slice where x is, and we can significantly reduce the false negative rate.

Algorithm 4.3: New check_conflict for non-inclusive caches.

```
1  Function check_conflict (x, U):
2      access $x$
3      for each addr in $U$ do
4          access addr
5      end
6      for each addr in L2_occupy_set do // this evicts $U$ from L2 to LLC
7          access addr
8      end
9      $t$ = measure time of accessing $x$
10     return $t \geq LLC\_miss\_threshold$
11 end
```

However, this naïve approach has a high false positive rate. A false positive can occur when U does not contain enough addresses to evict x from the LLC slice, but with the help of some addresses in *L2_occupy_set* that end up getting evicted to the LLC, they evict x from the LLC. In this case, the function is supposed to return $false$, but it returns *true*.

Reliable New `check_conflict` In order to reduce the false positive rate in the naïve new `check_conflict` function, we need to make sure accesses to *L2_occupy_set* do not interfere with the conflicts between U and x in the LLC. We can achieve this by leveraging the one-to-many set mapping relationship between L2s and LLCs.

For a reliable design, we select *L2_occupy_set* such that its addresses are mapped to the same L2 set as addresses in U, but to a *different* LLC set than used by addresses in U (and x). As mentioned before, upper level caches like the L2 contain fewer cache sets than lower level caches like the LLC. For example, in Skylake-X, the L2 has 1024 sets, while an LLC slice has 2048 sets. Correspondingly, the L2 uses 10 bits (bits 6-15) from the physical address as the set index, while the LLC slice uses 11 bits (bits 6-16). Therefore, *L2_occupy_set* can be constructed by simply flipping bit 16 of $\mathrm{W_{L2}}$ addresses in U. Such addresses can be used to evict U from the L2 but do not conflict with U in the LLC.

In summary, we design a reliable `check_conflict` function with both low false positive rate and low false negative rate. This function can be used in the EV construction algorithm of Liu et al. [6] to construct an EV for non-inclusive caches. We evaluate the effectiveness of the function in Section 4.5.1. For independent interest, we use our EV creation routine to partially reverse engineer the Skylake-X slice hash function in Section 4.5.4.

4.3 REVERSE ENGINEERING THE DIRECTORY STRUCTURE

We leverage our EV creation function to verify the existence of the directory structure in an 8-core Intel Core i7-7820X processor, which uses the Intel Skylake-X series microarchitecture. We also provide detailed information about the directory's associativity, inclusivity, replacement policies (for both private and shared data), and interactions with the non-inclusive caches. These insights will be used for the attack in Section 4.4. Skylake-X is a server processor for cloud computing and datacenters. A comparison of the cache parameters in this processor with previous Skylake series processors is listed in Table 4.1.

	Skylake-S	Skylake-X/Skylake-SP
L1-I	32KB, 8-way	32KB, 8-way
L1-D	32KB, 8-way	32KB, 8-way
L2	256KB/core 16-way, inclusive	1MB/core 16-way, inclusive
LLC	2MB/core 16-way, inclusive	1.375MB/core 11-way, non-inclusive

Table 4.1: Cache structures in Skylake processors.

Relative to the older Skylake-S processor, the Skylake-X/SP LLC is non-inclusive and, correspondingly, Skylake-X/SP can support larger L2 caches relative to the LLC. The L2 in Skylake-X/SP grows to 1 MB per core, which is 4 times larger than before, while the LLC size shrinks from 2 MB per core to 1.375 MB per core. The associativity in the LLC slice is also reduced from 16-way to 11-way.

4.3.1 Timing Characteristics of Cache Access Latencies

We first conduct a detailed analysis of the timing characteristics of the cache access latencies on Skylake-X. This information can be used to infer the location of a specified cache line, and is useful in reverse engineering the directory structure.

For each cache location, we measure the access latency by using the `rdtsc` instruction to count the cycles for one access. We use the `lfence` instruction to make sure we get the timestamp counter after the memory access is complete as suggested in [113]. Thus, all the latencies presented include delays introduced by the execution of `lfence`.

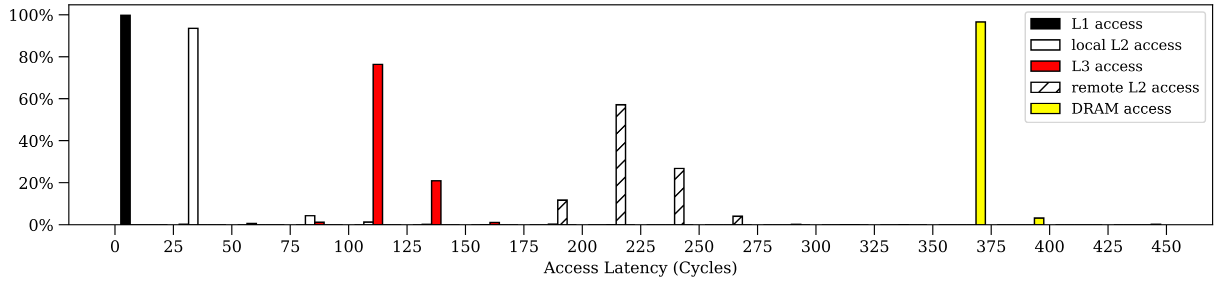


Figure 4.3: Latency of a cache line access when the line is in different locations in the Intel Skylake-X cache hierarchy.

Figure 4.3 shows the distribution of latencies to access lines in different cache layers. For each cache layer, we perform 1,000 accesses. The latency is for accessing a single cache line. From the figure, we see that L1 and local L2 access latencies are below 50 cycles. An LLC access takes around 100 cycles, and a DRAM access around 350 cycles.

A remote L2 access occurs when a thread accesses a line that is currently in another core's L2. From the figure, a remote L2 access takes around 200 cycles, which is shorter than the DRAM latency. We leverage the difference between the remote L2 latency and the DRAM latency in the "Evict+Reload" attack to infer the victim's accesses.

4.3.2 Existence of the Sliced Directory

Our first experiment is designed to verify the existence of a directory and its structure. In each round of the experiment, a single thread conducts the following three steps in order:

1. Access target cache line x.

2. Access a set of N eviction addresses. In a Reference setup, these are cache line addresses that have the same LLC set index bits as x, and can be mapped to different LLC slices. In a SameEV setup, these are cache line addresses that have the same LLC set index bits as x, and are mapped to the same LLC slice as x.
3. Access the target cache line x again while measuring the access latency.

Generally, step 2 is repeated multiple times (100 times) to avoid the noise due to cache replacement policy in both Reference and SameEV setups. The medium access latency over 1,000 measurements for step 3 is shown in Figure 4.4, as a function of the number of eviction addresses accessed. We validated that the experiment results are consistent for x mapped to different slices and sets.

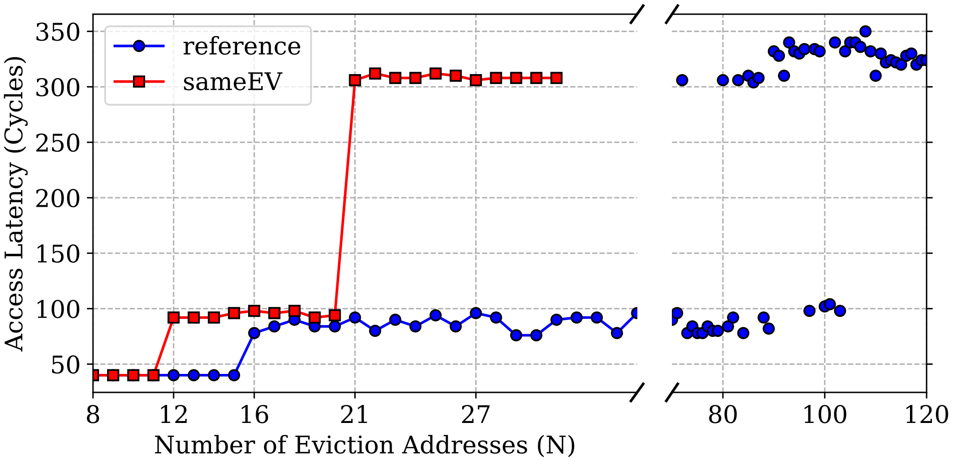


Figure 4.4: Target line access latency as a function of the number of eviction addresses, in experiments to verify the existence of the directory.

According to the timing characteristics in Figure 4.3, we know that latencies around 40, 100 and 300 cycles indicate that the target line is in local L2, LLC and DRAM, respectively. In the Reference configuration, if 16 or more lines are accessed in step 2, the target line is evicted from L2 to LLC. These evictions are caused by L2 conflicts because the L2 associativity (W_{L2}) is 16. Later, we start to observe that the target line is evicted to DRAM when more than 75 addresses are accessed in step 2. This number is less than 104 ($W_{\text{L2}} + N_{slice} \times W_{\text{slice}}$) because the hash function makes the addresses used in the experiment distribute unevenly across the different slices.

In the SameEV setup, we observe L2 misses when 12 cache lines are accessed in step 2, before reaching the L2 associativity. Moreover, the target line is evicted out of the LLC when 21 lines are accessed, even though the L2 cache and one LLC slice should be able to hold up to 27 lines ($W_{\text{L2}} + W_{\text{LLC}}$). The difference between 12 and 16 (L2 case), and between 21 and 27 (LLC case) indicates that there exists some bottleneck, other than the L2 and LLC slice associativity. This indicates the presence of some set-associative on-chip structure, where conflicts can cause L2 and LLC evictions. The structure's associativity seen for L2 lines is 12, and the associativity seen for L2 and LLC lines is 21.

In addition, we know that the structure is sliced and looked-up using the LLC slice hash function. Notice that addresses conflict in this structure only if they are from the same LLC slice, as in the SameEV configuration. Addresses from different LLC slices do not cause conflicts in this structure, as seen in the Reference configuration.

Finally, we can also reverse engineer the number of sets in each slice of the structure by testing which bits are used to determine the set index. We analyzed the EVs that we derived for this structure, and found that the addresses in a given EV always have the same value in bits 6-16. The addresses that belong to the same EV are mapped to the same set and slice, and should share the same set index bits and slice id. Since none of the bits 6-16 are used for slice hash function (see Section 4.5.4), we know that these bits are used as set index bits. Hence, the structure has 2048 sets, the same number of sets as an LLC slice.

To summarize, we have following findings in the Skylake-X non-inclusive cache hierarchy.

a) There exists a set-associative structure that operates alongside the non-inclusive caches. Contention on this structure can interfere with cache line states in both the L2 and LLC.

b) The structure is sliced and is looked up using the LLC slice hash function. Each slice has the same number of sets as an LLC slice.

c) The associativity of the structure for L2 lines is 12; the associativity of the structure for L2 and LLC lines is 21.

4.3.3 Inclusivity and Associativity for Private Cache Lines

We use the term *Private* cache line to refer to a line that has been accessed by a single core; we use the term *Shared* cache line to refer to a line that has been accessed by multiple cores. We observed that the non-inclusive LLC cache in Skylake-X behaves differently towards private and shared cache lines.

We conduct a two-thread experiment to reverse engineer the inclusivity of the set-associative structure that we found and the cache for private lines. The two threads are pinned to different cores.

1. Thread A accesses target line x.

2. Thread B accesses N eviction addresses. The addresses are selected for the Reference and SameEV setups as in the previous experiment.
3. Thread A accesses target line x again and measures the access latency.

The access latencies in step 3 are shown in Figure 4.5, as a function of the number of eviction addresses.

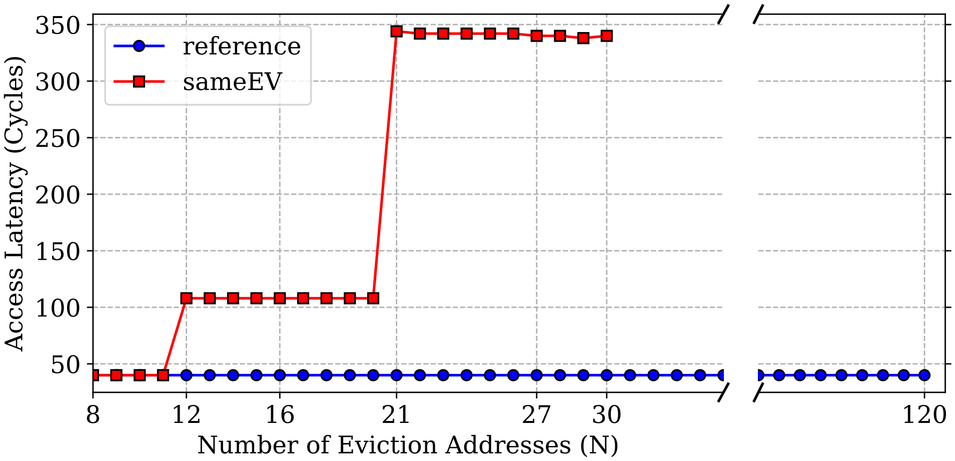


Figure 4.5: Target line access latency as a function of the number of eviction addresses in experiments to analyze the inclusive property for private lines.

In the Reference setup, the target line in thread A's private L2 is never evicted. Due to the non-inclusive property of the LLC, thread B is unable to interfere in thread A's private cache state, and hence loses any visibility into thread A's private cache accesses. However, in the SameEV setup, the target line is evicted from the L2 to LLC when 12 lines are accessed by thread B, and it is further evicted to DRAM when thread B accesses 21 lines.

This experiment shows that the structure is shared by all the cores, and that conflicts on this structure can interfere with L2 and LLC cache states. In particular, the SameEV configuration shows that we *can create inclusion victims across cores*, since lines in the L2 can be evicted due to the contention on the structure. We can safely conclude that the structure is inclusive to all the lines in the cache hierarchy, including L2 and LLC. This characteristic can be leveraged by an attacker to gain visibility into a victim's private cache and build Prime+Probe attacks. Moreover, the experiment also confirms the same associativity across cores as the last experiment.

We have additional findings as follows.

d) The structure is shared by all cores.

e) The structure is inclusive, and contention on the structure can cause inclusion victims across cores.

4.3.4 Inclusivity and Associativity for Shared Cache Lines

To reverse engineer the inclusivity and associativity of the structure for shared cache lines, we use 2 or 3 threads in different modes.

1. Thread A and B both access the target cache line x to ensure the line has been marked as shared by the processor.
2. In *1evictor* mode, thread B accesses N cache line eviction addresses; in the *2evictors* mode, thread B and C access N cache line eviction addresses to put those lines into the Shared state. In both modes, different eviction cache lines are selected for Reference and SameEV setups as in our previous experiments.
3. Thread A accesses the target line x again and measures the access latency.

From this discussion, in the 1evictor mode, only x is in the shared state; in the 2evictors mode, both x and the N eviction lines are in the shared state. Figure 4.6 shows the access latencies for step 3.

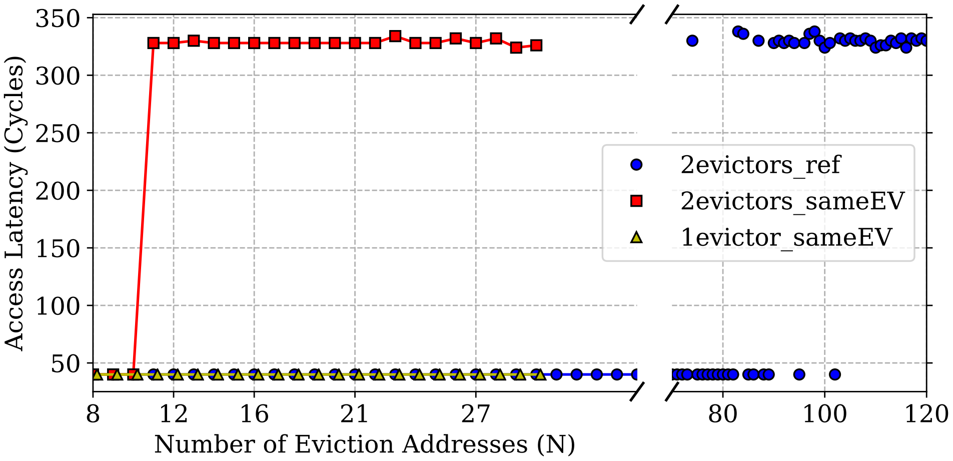


Figure 4.6: Target line access latency as a function of the number of eviction addresses in experiments to analyze the inclusive property for shared lines.

In the 1evictor_sameEV setup, the shared target line is never evicted out of thread A's private L2. However, we showed in Figure 4.5 that this pattern does cause remote L2 evictions of *private* lines. Comparing the two cases, we can infer that the cache coherence state–namely whether a line is shared or not–plays a role in the cache line replacement policy. The replacement policy prefers not to evict shared cache lines.

In the 2evictors_sameEV setup, threads B and C are able to evict the target line out of the LLC by accessing 11 shared lines, while in the 2evictors_ref setup, we begin to observe stable LLC misses when around 85 lines are accessed. The characteristics in 2evictors_sameEV indicates the associativity of the inclusive structure for shared cache lines is 11. Moreover, this experiment indicates how an attacker can use shared eviction lines to evict a shared target line out of the cache hierarchy, which we will leverage to build stable and efficient Evict+Reload attacks.

To summarize, this section presents following findings.

f) The cache replacement policy takes into account the coherence state and prefers not to evict cache lines which have been accessed by multiple cores.

g) The associativity of the inclusive structure for shared cache lines is 11.

4.3.5 Putting It All Together: the Directory Structure

We infer that the inclusive structure is a directory. Indeed, Intel has used a directory-based coherence protocol since Nehalem [103]. Supporting a directory-based protocol requires structures that store presence information for all the lines in the cache hierarchy. Thus the directory, if it exists, must be inclusive, like the structure we found. In the rest of the chapter, we will use the term directory to refer to the inclusive structure we found.

Overall Structure Figure 4.7 shows a possible structure of the directory in one LLC slice. From Section 4.3.2, we found that the directory is sliced, is looked-up using the LLC slice hash function, and has the same number of sets as the LLC. An LLC slice can co-locate with its directory, enabling concurrent LLC slice and directory look-up.

From Section 4.3.4, each directory slice has 21 ways in total (denoted $W_{\text{dir}} = 21$) for all the lines in the cache, including the L2 and LLC. From Section 4.3.3, there are maximally 12 ways can be used for lines present in the L2 but not in the LLC. We call the directory for these lines the *Extended Directory* (ED). We denote the ED associativity as $W_{\text{ED}} = 12$. From the public documentation in Table 4.1, we know that the LLC slice and its directory (which we call the *Traditional Directory* is 11-way set associative. We denote such associativity as $W_{\text{TD}} = W_{\text{slice}} = 11$. One might expect $W_{\text{dir}} = W_{\text{ED}} + W_{\text{TD}}$, but $21 < 12 + 11$. From this mismatch, we infer that 2 ways in each directory slice are dynamically shared between the

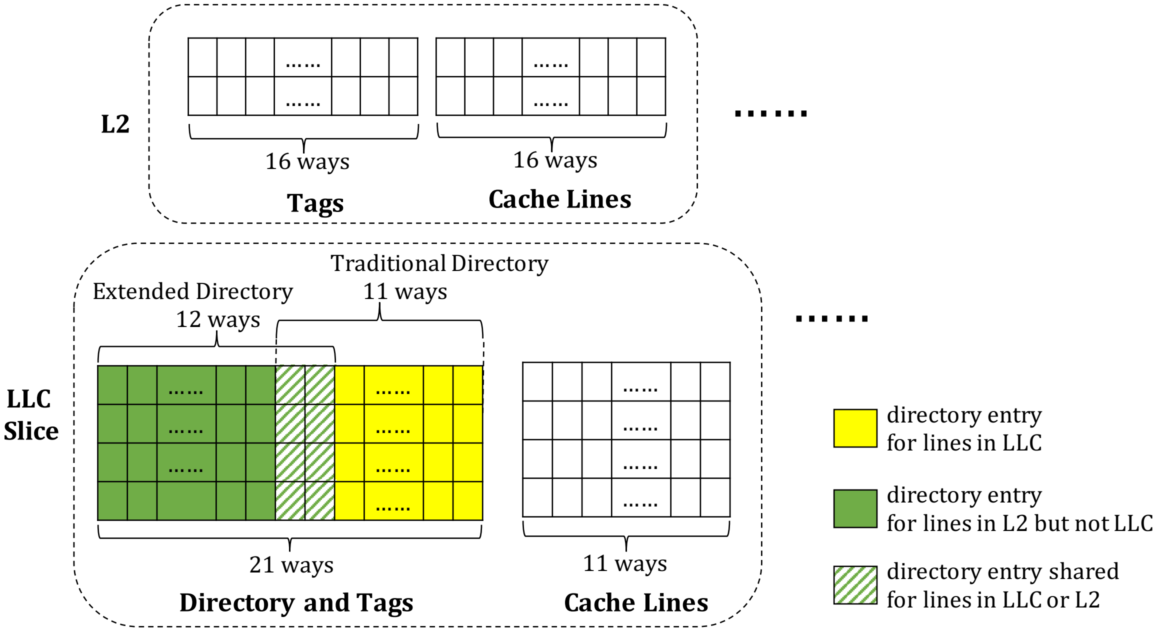


Figure 4.7: Reverse engineered directory structure.

traditional directory and the extended directory. How these ways are shared is determined by the replacement policy, which prefers to hold lines that are in state shared.

Migration between Directories The migration between the ED and the traditional directory operates as follows. An ED conflict causes a directory entry to be migrated from the ED to the traditional directory, and the corresponding cache line to be evicted from an L2 to the LLC. A conflict in the traditional directory causes a directory entry to be removed from the whole directory slice, which causes the corresponding cache line to be evicted out of the entire cache hierarchy. For private lines, 21 addresses are needed to cause a traditional directory conflict. From Section 4.3.3, the private lines will take up the ED and traditional directory, after which we see conflicts. For shared lines, only 11 addresses are needed to cause conflicts. We found that shared lines, after being accessed a sufficient number of times, allocate a data entry in the LLC and migrate their directory entry from the ED to the traditional directory. In this case, the line lives in multiple L2s and in the LLC at the same time.[3] This is likely a performance optimization as LLC hits are faster than remote L2 hits (Figure 4.3). Thus, heavily shared lines should be accessible from the LLC.

This directory structure matches our reverse engineered results. Even though the actual

[3] This is consistent with the cache being non-inclusive. Non-inclusive means that the cache may be inclusive for certain lines in certain circumstances.

implementation may use a slightly different design, our interpretation of the structure is helpful in understanding the directory and cache interactions, and in designing the attacks.

4.3.6 The Root Cause of the Vulnerability

The directory in non-inclusive caches is *inclusive*, since it needs to keep information for *all* the cache lines that are present in the cache hierarchy. This inclusivity can be leveraged to build cache-based attacks. An attacker can create conflicts in the directory to force cache line evictions from a victim's private cache, and create inclusion victims.

Considering the usage of directories, the directory capacity ($W_{\text{dir}} \times S_{\text{dir}}$) should be large enough to hold information for all the cache lines. In this case, how can it be possible to cause a directory conflict before causing cache conflicts? This section answers this question by analyzing the root cause of the vulnerability that we exploit.

The root cause is that *the directory associativity is smaller than the sum of the associativities of the caches that the directory is supposed to support.* More specifically, a directory conflict can occur before a cache conflict if any of the following conditions is true, where ED indicates the directory entries used by L2 cache lines.

$$\begin{aligned} & W_{\text{ED}} < W_{\text{L2}} \times N_{L2} \\ \textit{or} \quad & W_{\text{dir}} < W_{\text{L2}} \times N_{L2} + W_{\text{slice}} \end{aligned} \tag{4.2}$$

where N_{L2} is the number of L2s in the system (usually equal to the number of cores).

We believe that, for performance reasons, these conditions should be common. First, as the number of cores on chip increases, architects want to avoid centralized directory designs and, therefore, create directory slices. At the same time, architects try to limit the associativity of a directory slice to minimize look-up latency, energy consumption, and area. As a result, it is unlikely that each directory slice will be as associative as the sum of the associativities of all the L2 caches. For example, an 8-core Intel Skylake-X processor with 16-way L2s would require each directory slice to have a 128-way ED to avoid ED conflicts before L2 conflicts. This is expensive.

We found that the above conditions do not hold in some AMD processors. Consequently, our attack does not work on these AMD processors. We also found that memory and coherence operations on some AMD machines are slower than on Intel machines. This may suggest that these AMD machines do not use sliced directories. Section 4.5.7 describes the experiments we performed.

4.4 ATTACK STRATEGIES

Leveraging the EV construction algorithm in Section 4.2 and our reverse engineering of the directory structure in Section 4.3, we can now build effective side channel attacks in non-inclusive cache hierarchies. In this section, we first present our Prime+Probe attack targeting the ED. We then show a multi-threaded Evict+Reload attack to achieve fine-grained monitoring granularity. Finally, for completeness, we provide a brief discussion on Flush+Reload attacks.

4.4.1 Prime+Probe

To the best of our knowledge, this is the first Prime+Probe attack on non-inclusive caches. We first present our customized cross-core attack on Skylake-X, and then discuss how to generalize the attack to other vulnerable platforms.

On Intel Skylake-X, an attacker can leverage the inclusivity of the ED to gain visibility into a victim's private cache state. Before the attack, the attacker uses the EV construction algorithm of Section 4.2 to construct an EV that is mapped to the same LLC slice and set as the target address. Since the ED is both sliced and looked-up using the LLC slice hash function, it follows that the EV is mapped to the same ED set and slice as the target address. Thus, the EV can be used in the prime and probe steps.

Our cross-core Prime+Probe attack follows the same steps as a general Prime+Probe attack. In the prime step, the attacker accesses W_{ED} EV lines to occupy all the ways within the ED set, and evict the target line from the victim's L2 to the LLC. During the wait interval, if the victim accesses the target address, it causes an ED conflict and one of the EV addresses will be evicted from the attacker's private L2 cache to the LLC. In the probe step, when the attacker accesses the EV addresses again, it will observe the LLC access. Alternatively, if the victim does not access the target line in the wait interval, the attacker will observe only L2 hits in the probe step. After the probe step, the ED set is fully occupied by EV addresses, and can be used as the prime step for the next attack iteration.

Attack Granularity The attack granularity is determined by the time per attack iteration, which is composed of the wait time and the probe time. The more efficient the probe operation is, the finer granularity an attack can achieve.

In our ED-based Prime+Probe attack, the probe time is very short. The attacker only needs to distinguish between local L2 latency and LLC latency, which is shorter than the probe time in inclusive cache attacks, where the attacker needs to distinguish between LLC latency and DRAM latency.

Generalizing the Attack The attack above is customized for Intel Skylake-X. We now discuss how to generalize the attack to other vulnerable platforms which satisfy the conditions discussed in Section 4.3.6.

First, a characteristic of the Skylake-X is that the ED associativity is not higher than the L2 associativity ($W_{\text{ED}} \leq W_{\text{L2}}$), which allows us to trigger ED conflicts using a single attacker thread. If this condition is not satisfied, we can still mount a Prime+Probe attack with *multiple* attacker threads, running on different cores, as long as $W_{\text{ED}} \leq W_{\text{L2}} \times (N_{L2} - 1)$. For example, consider the case where $W_{\text{ED}} = W_{\text{L2}} \times (N_{L2} - 1)$. The attacker can use $(N_{L2} - 1)$ threads running on all the cores except for the victim's core, where each thread accesses W_{L2} addresses to occupy the ED set.

Second, the directory in Skylake-X uses the same hash function as the LLC. Therefore, we can directly use the EVs constructed for LLC slices to create directory conflicts. If the sliced ED uses a different hash function, the attack should still work but will need a new EV construction algorithm for the directory.

4.4.2 Evict+Reload

On non-inclusive caches, an attacker could leverage the directory's inclusivity to build Evict+Reload attacks using a similar approach as in Prime+Probe. However, the evict operation in Evict+Reload is more challenging than the prime operation, since the target line is shared by the attacker and the victim. As we showed in Section 4.3.4, the cache replacement policy takes into account the coherency state—namely that the target line is *shared*—and prefers not to evict the directory entries for shared lines.

We propose a novel multi-threaded Evict+Reload attack that can achieve fine-grained monitoring granularity by taking advantage of the characteristics of the replacement policy. The attack involves two tricks, namely, to upgrade the eviction addresses to a higher replacement priority, and to downgrade the target address to a lower replacement priority.

The attacker consists of three threads: a main thread which executes the evict and reload operations, and two helper threads to assist evicting the shared target line. The two helper threads share all the eviction addresses, and thus are able to switch the eviction addresses to the *shared* coherence state, similar to the 2evictors_sameEV setup in Section 4.3.4. This brings eviction addresses to the same replacement priority as the target address in the directory. In addition, the main attacker thread evicts the target address from its private cache to the LLC by creating L2 conflicts, which makes the target address non-shared.

Throughout the entire attack, the helper threads run in the background, continuously accessing the addresses in the EV in order to keep these addresses in their caches. In the eviction step, the main attacker thread introduces conflicts in its L2 cache to evict the target line from its L2 to the LLC. The helper threads then evict the target line (which they do not share) from the LLC to DRAM, by accessing the shared EV lines. If the victim accesses the target line during the wait interval, it will bring the line into its L2. Then, in the reload step, the main attacker will see a remote L2 access latency. Otherwise, the attacker will observe a DRAM access latency.

4.4.3 Flush+Reload

A Flush+Reload attack on non-inclusive caches follows the same procedure as the one on inclusive caches. This process has been referred to as Invalidate+Transfer [28] on AMD's non-inclusive caches. We evaluate this attack on Intel's platform for completeness, though it is not necessary to demonstrate our new attack channel on directories.

The attacker uses the `clflush` instruction to evict the target address from all levels of the cache hierarchy to DRAM. If, during the wait interval, the victim accesses the target address, the line will be brought into the victim's local L2. In the measurement phase, the attacker reloads the target address and measures the access latency. If the victim had accessed the line, the attacker will see a remote L2 access latency; otherwise, it will observe a DRAM access latency.

4.5 EVALUATION

4.5.1 Effectiveness of the check_conflict Function

We evaluate the effectiveness of the `check_conflict` function by measuring the false positive rates and the false negative rates. We consider three designs, the baseline function proposed by Liu et al. (no_flushL2), and the two modified functions discussed in Section 4.2, i.e. flushL2_naive and flushL2_reliable.

We obtained 8 EVs and confirmed their correctness by checking their conflicting behaviors as in Section 4.3. All the addresses in the 8 EVs have the same LLC set index bits, and each EV is mapped to a different LLC slice. To measure the false positive rate, we select an address x and set the argument U of `check_conflict` to be a collection of addresses with 10 addresses ($< W_{\text{slice}}$) from the same EV as x, and 5 addresses from each of the other EVs. The function should return $false$. We then count the number of times when the function mistakenly returns *true*. To measure the false negative rate, an extra address from the same EV as x is added to the collection U, so that U contains 11 eviction addresses ($= W_{\text{slice}}$). The function should return *true*. Then, we count the number of times when the function mistakenly returns *false*. In each of the three `check_conflict` implementations, the eviction operation (line 3-5 of Algorithm 4.2, line 3-8 in Algorithm 4.3) is repeated multiple times. Figure 4.8 shows how the false positive rate and the false negative rate change with the number of eviction operations performed.

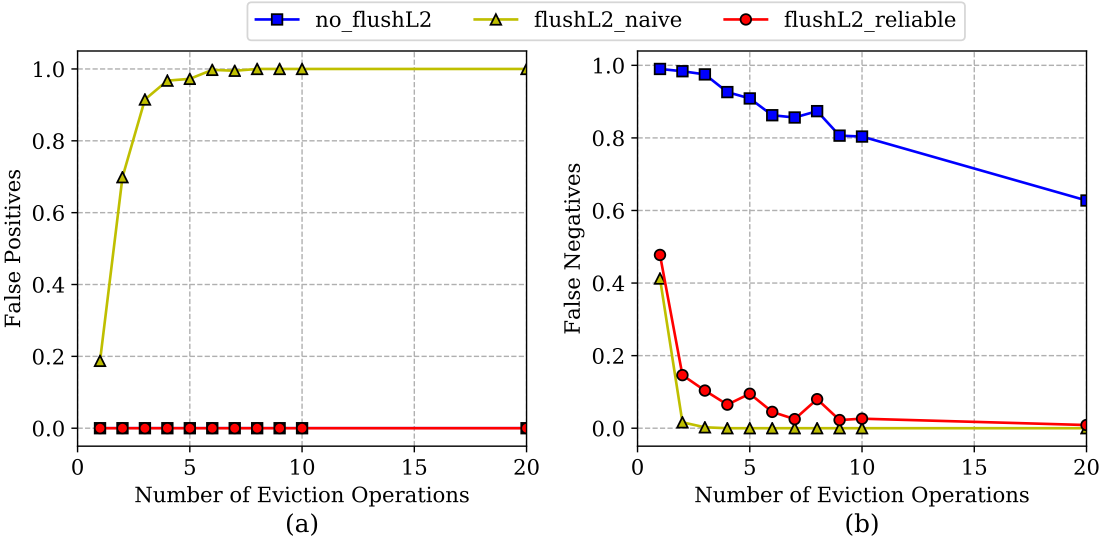


Figure 4.8: Comparing effectiveness of different check_conflict functions.

In Figure 4.8(a), both no_flushL2 and flushL2_reliable have no false positives. flushL2_naive has a much higher false positive rate due to the extra conflicts introduced by the *L2_occupy_set*. In Figure 4.8(b), both flushL2_naive and flushL2_reliable can achieve very low false negative rate when eviction operations are repeated around 10 times. The false negative rate of the no_flushL2 approach stays high even though the evictions are performed 20 times. In conclusion, our reliable flushL2 approach in `check_conflict` function is effective and can achieve both low false negative rate and false positive rate.

4.5.2 Extended Directory Timing Characteristics

As we leverage ED conflicts to construct our Prime+Probe attack, it is very important to understand their timing impact on cache access latencies, as shown in Figure 4.9. The figure shows the access latency of a number of addresses from the same EV. In the "no_EDconf" case, we simply measure the latency of EV accesses. In the "1_EDconf" case, between two measurements, we use a different thread on another core to issue one access to the same ED set to cause one ED conflict. Thus, the latency in "no_EDconf" is the expected probe latency with no victim accesses during wait intervals, while the "1_EDconf" latency corresponds to the expected probe latency when victim accesses the target line.

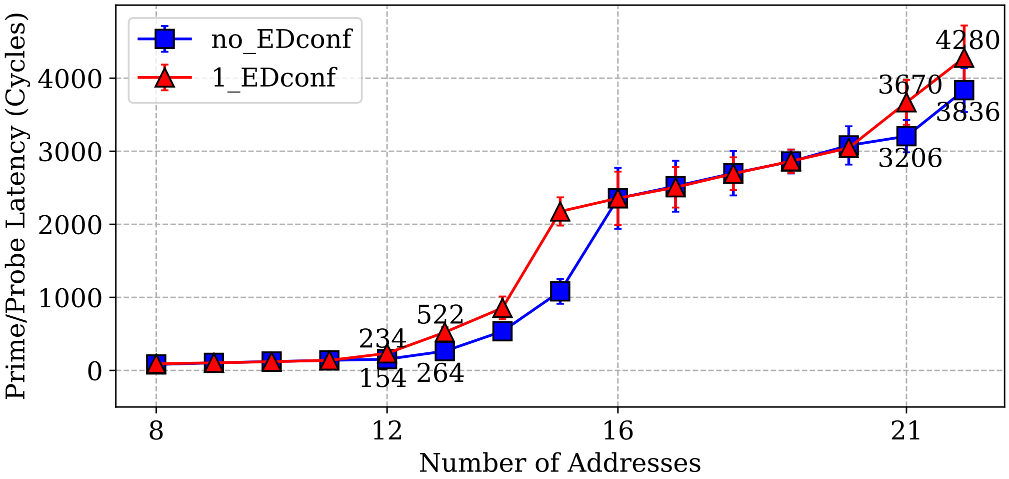


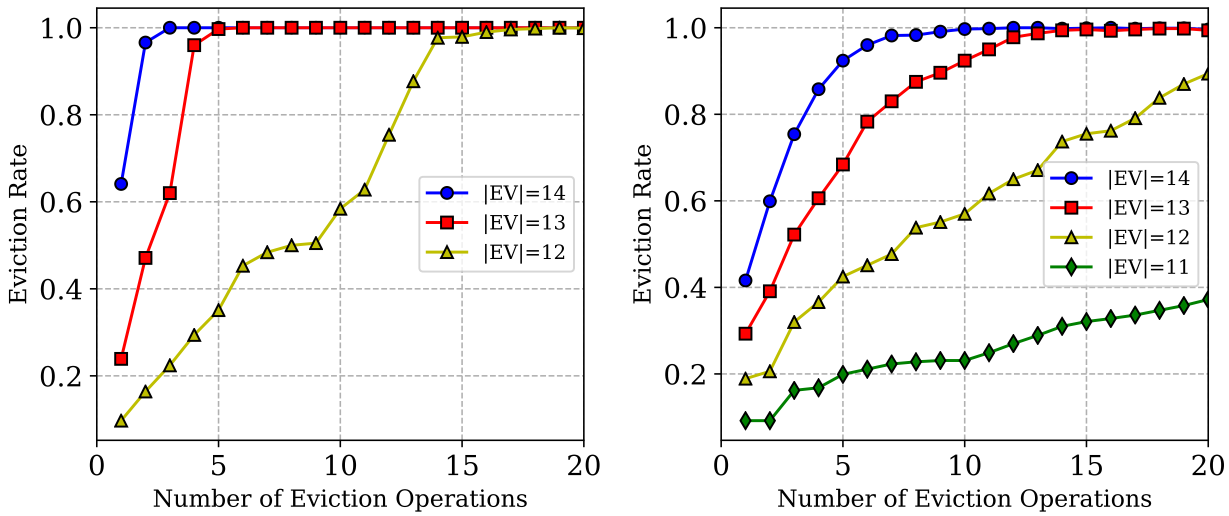
Figure 4.9: Prime and probe latency.

A high-resolution low-noise Prime+Probe attack requires the probe operation to be efficient and clearly distinguishable. From Figure 4.9, W_{ED} (12) is the optimal number of probe addresses we should use in Prime+Probe. First, the impact of ED conflicts is large and clearly observable. The timing difference between no ED conflicts and a single ED conflict is around 80 cycles. Second, accessing 12 addresses takes very short time, around 230 cycles with a ED conflict. With such efficient prime/probe operation, we can do fine-grained monitoring. It is also feasible to use 13-15 addresses, but it is not optimal due to the longer access latency and larger variance. Note that the variance in Figure 4.9 is measured in a clean environment, there is more noise when running the attacker code with the victim code.

4.5.3 Directory Replacement Policy Analysis

As discussed before, the directory uses a complex replacement policy. We analyze how the replacement policy affects the effectiveness of eviction operations on a private and a shared cache line in Figure 4.10. This is an important factor an attacker needs to consider in designing efficient cache attacks.

Figure 4.10(a) shows the eviction rate of evicting a private cache line from a remote L2 to the LLC by creating ED conflicts. To repeat the eviction operation, we simply re-access each



(a) Evict a private line

(a) Evict a shared line

Figure 4.10: Analysis of directory replacement policy.

address in the EV in the same order. When using 12 EV addresses, the eviction rate reaches 100% after accessing the EV for 14 times, while the eviction rate increases much faster when we increase the size of the EV. For example, accessing 13 EV addresses for 5 times can ensure eviction. Figure 4.10(b) shows the eviction rate of evicting a shared cache line from a remote L2 to DRAM by creating directory conflicts with 2 eviction threads. It turns out when using 14 EV addresses, it requires repeating the eviction operation 9 times to ensure complete eviction. This indicates the necessity to downgrade the target line replacement priority to achieve fine-grained attack granularity, as we discussed in Section 4.4.2.

In summary, due to the complexity of the directory replacement policy, we find it difficult to come up with an efficient eviction strategy. A possible approach would be to try all the combinations of EV sizes and access orders as in [37]. Nevertheless, in this chapter, we show that our attacks can tolerate this imperfect eviction rate.

4.5.4 Skylake-X Slice Hash Function

Based on our EV construction results, we are able to reverse engineer part of the slice hash function in the Intel Skylake-X processor. We found that the slice hash function is not a simple XOR operation of selected physical address bits. This design is significantly different from the one in previous Intel processors such as SandyBridge and IvyBridge. Considering that all of the previous works on reverse-engineering slice hash functions [114, 115] rely on the use of a simple XOR hash function, our results identify the need for more advanced reverse-engineering approaches.

We briefly discuss how to get the partial hash function. We select 128 addresses with the same LLC set index bits to form a Candidate Set (CS). Bits 6-16 of these addresses are set to the same value, while bits 17-23 are varied. The goal is to reverse engineer how bits 17-23 affect the output of the slice hash function.

First, we run `find_EV` on the 128-address CS and obtain 8 EVs. Each EV is mapped to one cache slice. Second, we try to figure out the slice id for each EV. Since the Skylake-X processor uses a mesh network-on-chip to connect L2s and LLC slices [27], a local LLC slice access takes shorter time than a remote slice access. We check the access latency of each EV from each core. We then get the id of the core from which the access latency is the lowest, and assign the core id to the EV. Finally, we use the Quine-McCluskey solver [116] to get the simplified boolean functions from the input bits to the slice id as below. In the following, o_i is the ith bit in the slice id, and b_i is the ith bit in the physical address.

$$\begin{aligned}
o_2 =& b'_{23}b'_{19} + b'_{22}b'_{19} + b_{23}b_{22}b_{19} \\
o_1 =& (b_{23} + b_{22})(b_{20} \oplus b_{19} \oplus b_{18} \oplus b_{17}) + b'_{23}b'_{22}(b_{20} \oplus b_{19} \oplus b_{18} \oplus b_{17})' \\
o_0 =& b'_{22}(b_{19} \oplus b_{18}) + b_{22}(b_{23} \oplus b_{21} \oplus b_{19} \oplus b_{18})' \\
& \quad where \quad \{b_{63}...b_{24}\} = 0x810
\end{aligned} \tag{4.3}$$

These functions can not be further reduced to a simple XOR function. According to our observations, some of the higher bits (bits 24-63) also affect the hash function, which we have not fully reverse engineered.

4.5.5 Covert Channel on Extended Directories

We demonstrate a covert channel between two different processors that utilizes the extended directory between two different processes. One process serves as the sender and the other as the receiver. The sender and receiver run on separate cores, and each utilizes 7 addresses that are mapped to the same LLC slice. Together there are 14 addresses, which are enough to cause measurable ED conflicts.

Since we have not reverse engineered the slice hash function, the sender and the receiver cannot directly negotiate which slice to use. Before communication, the receiver scans all slices to find the one the sender is using. The sender transmits a bit “0” by idling for 5000 cycles, and keeps accessing the 7 addresses for 5000 cycles to transmit a bit “1”. The receiver decodes the two states by taking latency samples every 1000 cycles. On our 3.6GHz machine, it takes 5000 cycles to transmit one bit, thus the bandwidth is $0.2Mbit/s$. With a better

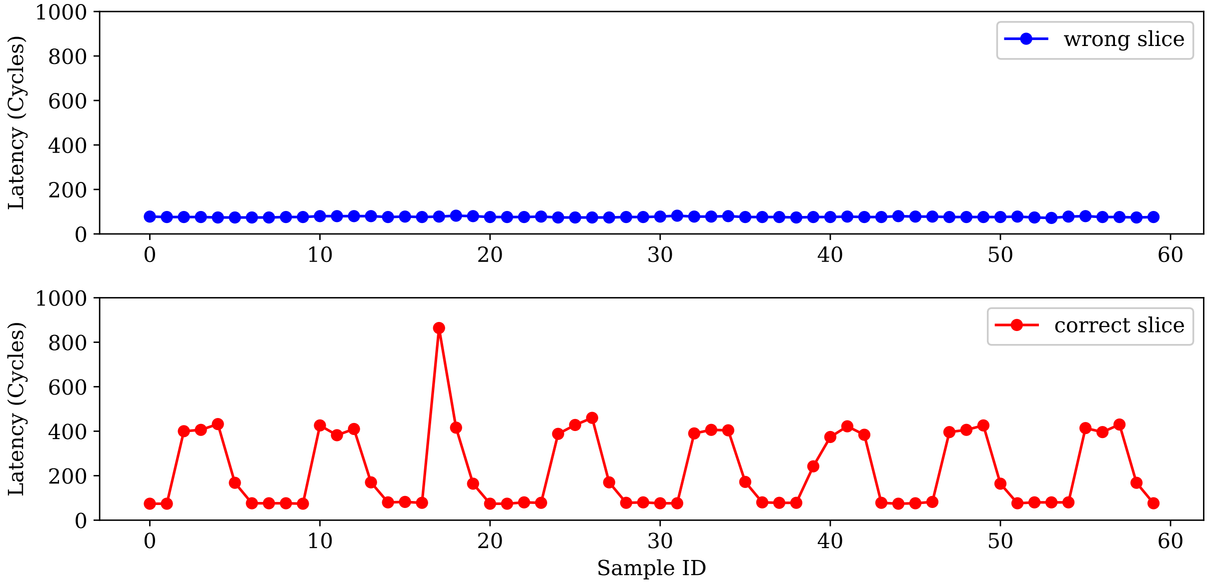


Figure 4.11: The upper plot shows receiver's access latencies on a slice not being used for the covert channel, while the lower one shows the one used in the covert channel. Sender transmits sequence "101010...".

protocol than we are using, the bandwidth can be further improved.

Figure 4.11 shows the results of our reliable covert communication channel. The upper plot shows the latencies that the receiver observes when accessing the wrong slice. All the latencies are low, as they correspond to L1 cache hits. In the lower plot, it is clear to see the sender's message of "101010". The receiver observes a $\sim$ 400 cycle latency due to ED conflicts when decoding a "1" bit, which is easily differentiated from the $\sim$ 80 cycle L1 hits for a "0" bit.

4.5.6 Side Channel Attacks on Directories

We evaluate the effectiveness of our side channel attacks on the square-and-multiple exponentiation vulnerability in GnuPG 1.4.13. The implementation is similar to the one presented in Algorithm 2.1 in Section 2.2.1. As discussed before, a victim's accesses on function `sqr` and `mul` can leak the value of exponent. In GnuPG, these two functions are implemented recursively, thus the target address identifying each function will be accessed multiple times throughout the execution of either operation. We show how this algorithm remains vulnerable on non-inclusive caches by attacking it with Prime+Probe, Evict+Reload and Flush+Reload attacks.

Flush+Reload We evaluate a cross-core Flush+Reload attack on this new platform for completeness. The victim and the attacker run on separate cores. The flush and reload operations are used on the addresses located at the entry of the `sqr` and `mul` functions. We use a wait time of 2000 cycles between the flush and reload.

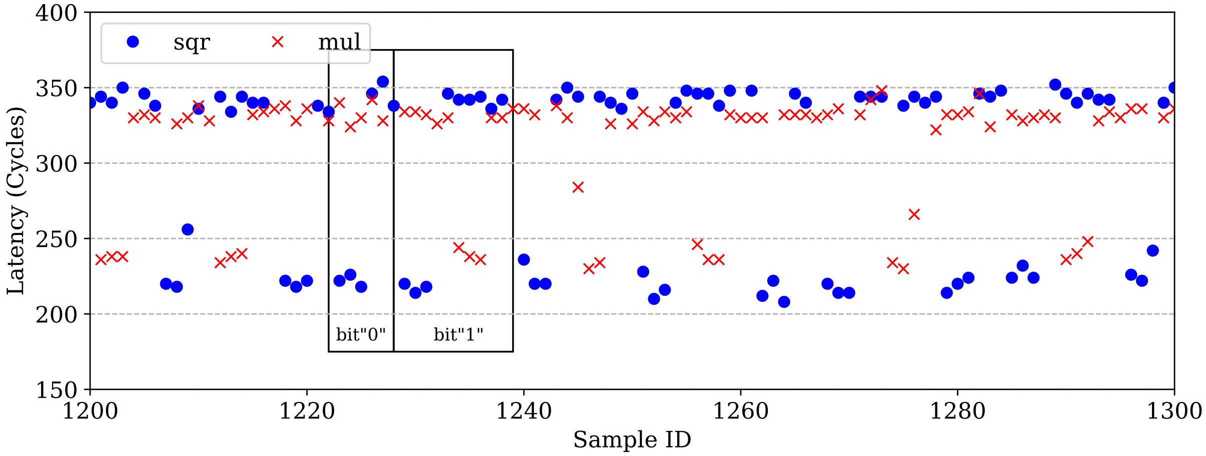


Figure 4.12: Access latencies measured in the reload operation in Flush+Reload. A sequence of "1001110101" can be deduced as part of the exponent.

Figure 4.12 shows the time measurement of the reload operation for 100 samples. A low latency reload operation, less than 250 cycles, indicates the victim has accessed the target address during the wait interval. A high latency, around 350 cycles, means the victim has not accessed the target address. According to the algorithm, an access on `sqr` followed by an access on `mul` indicates a bit "1", and two consecutive accesses on `sqr` without `mul` accesses in the between indicate a bit "0". From Figure 4.12, we can see that each `sqr` operation completes after 3 samples, or about 6000 cycles. Leveraging this information, the attacker is able to deduce part of the exponent as "1001110101".

In Flush+Reload, errors stem from times when the attacker's flush operation overlaps with victim accesses. Such occurrences cause lost bits.

Prime+Probe In our Prime+Probe attacks, we use 12 probe addresses from an eviction set for the target address, and use 500 cycles as the attacker wait interval. [4] We are able to monitor with such small granularity due to the efficient probe operation on the ED. We only monitor one target address, i.e. the address located at the entry of `mul` function, which is good enough.

[4] We use 12 EV addresses instead of 13 addresses, because we can get more precise and clean measurements of accessing 12 addresses, even though we suffer some noise due to relatively low eviction rate.

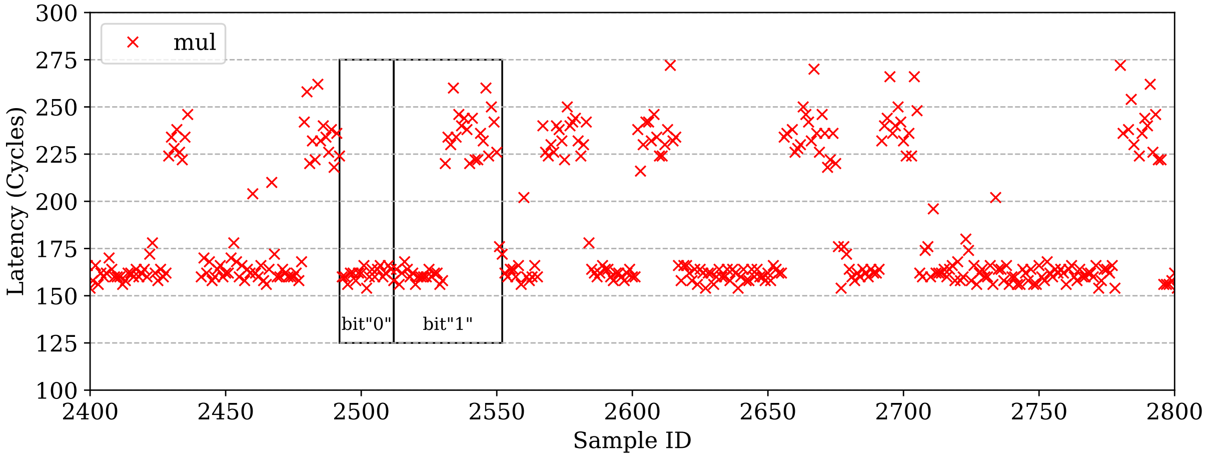


Figure 4.13: Access latencies measured in the probe operation in Prime+Probe. A sequence of “01010111011001” can be deduced as part of the exponent.

Figure 4.13 shows the access latencies measured in the probe operation as results of our Prime+Probe attack for 400 samples. If there is no victim access of the target address, the probe operation will see L2 hits for all the probe addresses without ED conflicts, taking around 160 cycles. Otherwise, if the victim accesses the target address, ED conflicts will be observed, resulting in long access latency, around 230 cycles. We do not track victim accesses on the `sqr` function; this the same approach taken in [6]. Instead, the number of `sqr` operations can be deduced from the length of the interval between two consecutive multiply operations. The attacker can deduce a sequence of “01010111011001” as part of the exponent from Figure 4.13.

In Prime+Probe attacks, most errors stem from the imperfect eviction rate, which leads to observing a multiply operation for more samples than it actually executed.

Evict+Reload Our novel Evict+Reload attack utilizes 1 attacker thread and 2 helper threads. The 2 helper threads access the same EV with 11 (W_{TD}) addresses mapped to the same LLC slice and set as the target address. The attacker thread accesses 16 (W_{L2}) addresses mapped to the same L2 set as the target line 6 times. We tested multiple eviction approaches, and found this method is highly reliable, and also very efficient, only taking around 1200 cycles. We monitor both the square and multiply operations and use 4000 cycles as the wait interval.

Figure 4.14 shows the access latencies measured in the reload step as the results for the Evict+Reload attack for 100 samples. The figure can be interpreted in the same way as the one for Flush+Reload, and the attacker can decode the part of the exponent as sequence

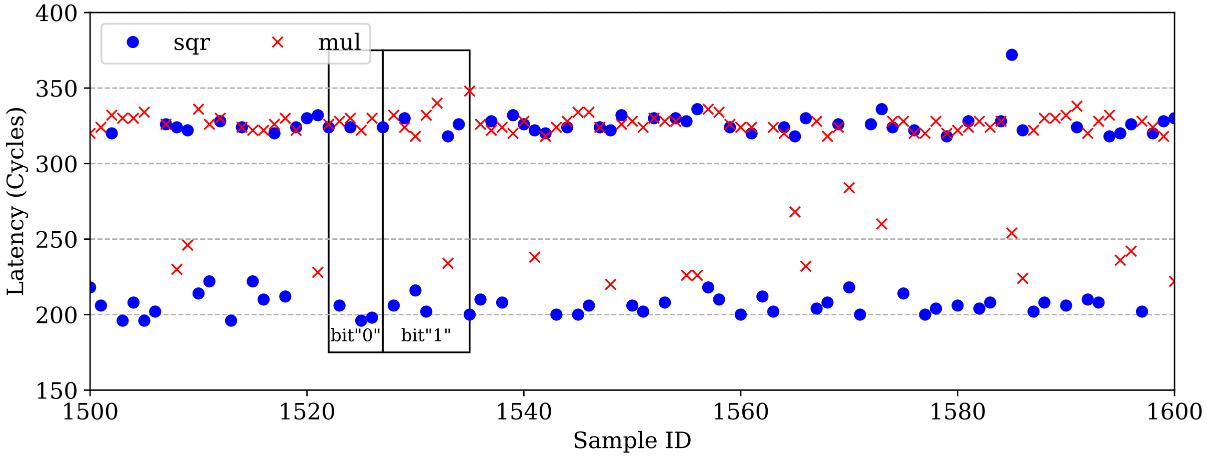


Figure 4.14: Access latencies measured in the reload operation in Evict+Reload. A sequence of “0101011110110101” can be deduced as part of the exponent.

“0101011110110101”.

Compared to Flush+Reload, the Evict+Reload attack on non-inclusive caches tends to suffer more errors. Since the evict operation takes longer than the flush operation, the probability that the evict step overlaps with the victim’s access is higher.

4.5.7 Attacking AMD Non-Inclusive Caches

We tried to reverse engineer the non-inclusive cache hierarchy in an 8-core AMD FX-8320 processor, which uses the AMD Piledriver microarchitecture. The cache parameters in this processor are listed in Table 4.2.

	AMD Piledriver	AMD Zen (4-core CCX)
L1-I	64KB/2cores, 2-way	64KB, 4-way
L1-D	16KB, 4-way	32KB, 8-way
L2	2MB/2cores, 16-way, inclusive	512KB, 8-way, inclusive
LLC	8MB/8cores 64-way, non-inclusive	2MB/core 16-way, non-inclusive

Table 4.2: Cache structures in AMD processors.

We found that the L2 caches in this processor are inclusive and shared by two cores. We verified that previous inclusive cache attacks work well, if the attacker and the victim are located on neighboring cores and share the same L2.

To see whether the non-inclusive LLC is vulnerable to cache attacks, we tried the reverse engineering experiments in Section 4.3 to detect the existence of directories. We did not observe extra conflicts besides cache conflicts. It is possible that the processor uses a snoopy-based cache coherence protocol [117], in which case there is no directory. It is also possible that the processor uses a centralized and high-associativity directory design, such that the directory associativity is at least as high as the total cache associativity. In this case, the directory for L2 lines needs to have $\geq$ 64 ways. Overall, the conditions in Section 4.3.6 do not hold.

We also evaluated our attack on an 8-core Ryzen 1700 processor, which uses the latest AMD Zen microarchitecture. The cache parameters are also listed in Table 4.2. The processor consists of 2 Core Complexes (CCX). A CCX is a module containing 4 cores, which can connect to other CCX modules via Infinity Fabric [118, 119]. We did not observe extra conflicts other than the cache conflicts on this processor either. Given the small number of cores on each die and low L2 associativity (8 on AMD CCX compared to 16 on Intel Skylake-X), we hypothesize that this processor either uses a snoopy-based protocol or a 32-way centralized directory for L2 lines.

Performance Implications for AMD Designs We measured the remote L2 access latency for the Piledriver processor, and found that it was about as long as a DRAM access. This time is significantly longer than the corresponding operation in the Intel Skylake-X (Figure 4.3). This observation backs up our claim that sliced directories are important structures in high performance processors. For the Ryzen processor, we have a similar result. Specifically, a cross-CCX access takes a similar amount of time as a DRAM access. The Skylake-X/Skylake-SP processors can support up to 28 cores. Since each CCX is only 4 cores, constructing a similarly provisioned Ryzen system can mean that most cross-core accesses turn into cross-CCX accesses.

4.6 CONCLUSION

In this chapter, we identified the directory as a unifying structure across different cache hierarchies on which to mount a conflict-based side channel attack. Based on this insight, we presented two attacks on non-inclusive cache hierarchies. The first one is a Prime+Probe attack. Our attack does not require the victim and adversary to share cores or virtual memory, and succeeds in state-of-the-art non-inclusive sliced caches such as those of Skylake-X [27]. The second attack is a novel, high-bandwidth Evict+Reload attack that uses a multi-threaded adversary to bypass non-inclusive cache replacement policies. We attacked square-and-multiply RSA on the modern Intel Skylake-X processor, using both of our attacks.

Moreover, we also conducted an extensive study to reverse engineer the directory structure of the Intel Skylake-X processor. Finally, we developed a new eviction set construction methodology to find groups of cache lines that completely fill a given set of a given slice in a non-inclusive LLC.

CHAPTER 5: SECDIR

This chapter presents the first design of a scalable secure directory (SecDir). Directories need to be redesigned for security. However, in an environment with many cores, it is hard or expensive to block directory interference. To address this problem, we propose SecDir, a secure directory design that eliminates inclusion victims by restructuring the directory organization. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory areas used in a victim-cache manner called Victim Directories (VDs). The partitioned nature of VDs prevents directory interference across cores, defeating directory side channel attacks. We show that SecDir has a negligible performance overhead and is area efficient.

The design of directories for cache coherence has been an active area of research for many years (e.g., [120–125]). Most of the research has focused on making the directories efficient and scalable to large core counts. As a result, commercial machines have incorporated directories (e.g., [126, 127]).

In Chapter 4, we have shown that directories are vulnerable to conflict-based side channel attacks. The insight is that every single line in the cache hierarchy has a corresponding directory entry. Since directories are themselves cache structures organized into sets and ways, an attacker can access data whose directory entries conflict in a directory set with entries from the victim. The eviction of victim directory entries automatically triggers the eviction of victim cache lines from the private caches of the victim — irrespective of whether the cache hierarchy is inclusive, non-inclusive, or exclusive. As the victim re-accesses its data, the attacker can indirectly observe the directory state changing, hence succeeding in their purpose. These observations strongly indicate that directories have a fundamental security problem. Hence, they need to be redesigned with security concerns in mind.

The key to a secure directory is to block interference between processes. Sadly, this is hard or expensive to do in an environment with many cores. For example, an apparent solution is to substantially increase the associativity of the directory structures. Unfortunately, it is unrealistic to aim for a directory associativity as high as the associativity of the private L2 cache times the number of cores — which is the total number of different live directory entries that could be mapped to one directory set. A second approach is to way-partition the directory. Each application is given some of the directory ways, to which it has uncontested use. This solution is similar to cache partitioning schemes [10, 62, 64]. Unfortunately, this approach is inflexible, low performing, and limited, since servers can have many more cores than directory ways.

In this chapter, our security goal is to prevent an attacker from creating inclusion victims in the victim's private caches through the eviction of the victim's directory entries. Note that we target an *active* attacker, rather than a *passive* one (Section 2.2.3). An active attacker is one that interferes with the victim's cache accesses by using directory conflicts, and exposes some of the victim's security-sensitive cache accesses. We consider that evictions of victim cache lines or victim directory entries from the victim's private caches or private directories due to self-conflicts (i.e., conflicts between victim cache lines or victim directory entries) do not leak information.

Ideally, a secure directory has to have several characteristics. First, it should set aside some directory area to support many isolated partitions inexpensively and scalably. Second, each partition should provide high associativity, so that a victim suffers few self-conflicts under the pathological environment of an attack. Finally, the directory should have little area overhead and provide fast look-ups.

We use these ideas to design a secure directory for a server multiprocessor. We call it *SecDir*. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory structures of high effective associativity called *Victim Directories* (VDs). The partitioned design of VDs prevents directory interference across cores, thus defeating directory side channel attacks. The VD of a core is distributed, and holds as many entries as lines in the private L2 cache of the core. To provide high effective associativity, a VD is organized as a *cuckoo* directory. Such a design also obscures victim self-conflict patterns from the attacker.

We model with simulations the directory of an Intel Skylake-X server [27] without and with SecDir. We run SPEC and PARSEC applications. Our results show that SecDir has negligible performance overhead. Furthermore, SecDir is area-efficient: it only needs 28.5KB more directory storage per core than the Skylake-X for an 8-core machine, while it uses less directory storage than the Skylake-X for 44 cores or more. Finally, a cuckoo VD organization eliminates substantial victim self-conflicts in a worst-case attack scenario.

5.1 ATTACK ANALYSIS

Following the discussion in Chapter 4, we review the root cause of directory attacks. Consider the parameters of the Skylake-X caches and simplified directories that we use in this chapter. In a given slice, the TD and ED have associativities of $W_{TD} = 11$ and $W_{ED} = 12$, respectively. Consequently, a given directory slice can hold at most 23 entries mapping to the same set. On the other hand, the associativity of an LLC slice is $W_{LLC} = W_{TD} = 11$, and that of an L2 cache is $W_{L2} = 16$. Further, the machine can have many cores — i.e., from $N = 8$ to 28.

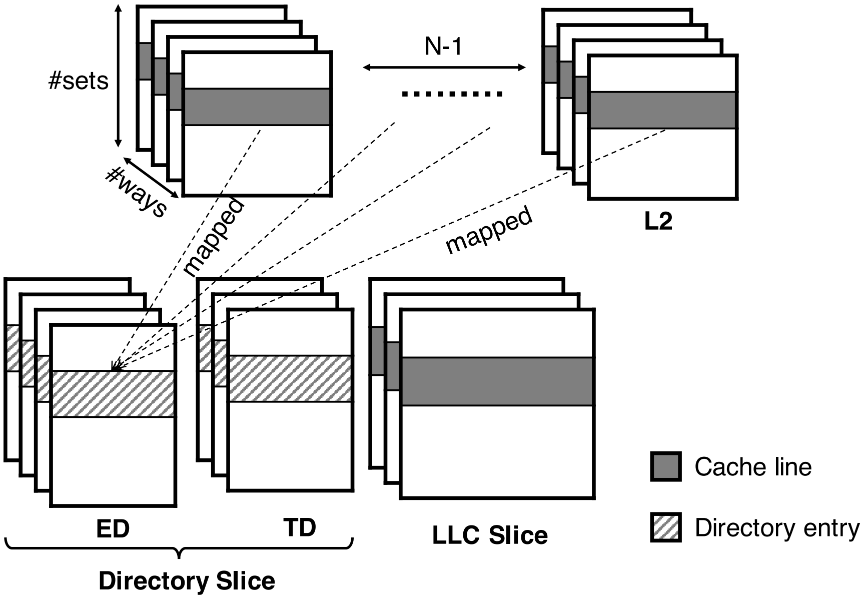


Figure 5.1: Attackers exploiting the limited associativity of a directory slice.

An attacker can use up to N-1 cores to bring enough lines into L2 caches and into one LLC slice to require more than 23 directory entries to be mapped into a single set of a directory slice (Figure 5.1). The result is that any directory entry in that set belonging to the victim process is evicted, automatically evicting the corresponding line from the victim's L2 cache, which is an inclusion victim. As the victim process later reloads the data, its directory entry is automatically reloaded, which allows the attacker to indirectly observe the victim's action.

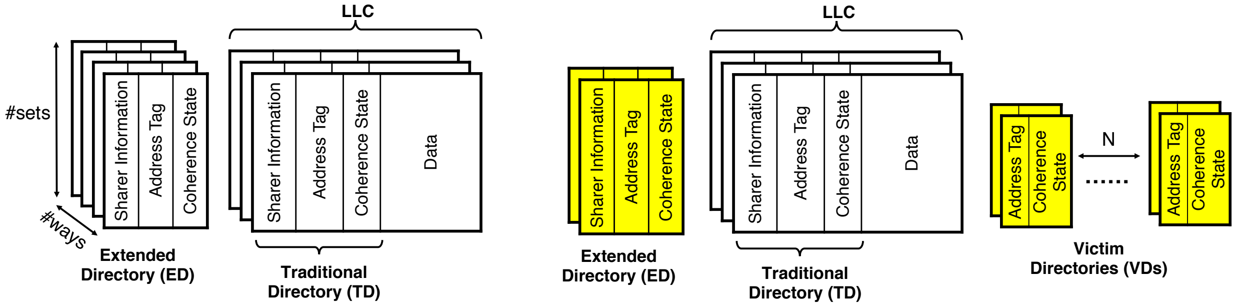
For a victim running on a core to be able to keep at least one entry in the directory, a directory slice would have to have an associativity $W_{TD} + W_{ED}$ such that

$$W_{TD} + W_{ED} > W_{L2} \times (N - 1) + W_{LLC} \tag{5.1}$$

which assumes that the attacker can use all the cores minus one. In a Skylake-X with 8 cores, this requires a directory slice with an associativity higher than 123. With more cores, the required associativity increases rapidly. Since this is an unreasonable associativity, current directories are easy targets of conflict-based side channel attacks.

5.2 SECDIR DESIGN OVERVIEW

The root cause of the directory vulnerability is the limited associativity of individual directory slices, given the number of cores in current servers. To defeat directory attacks, we need a new directory organization with three attributes. First, the directory should set aside some storage to support inexpensive and scalable per-core isolated directory partitions. Such support will provide victim isolation, and prevent the attacker from creating inclusion victims. Second, each partition should have high associativity, so that a victim suffers minimal self-conflicts under an attack. Finally, the directory should add little area overhead and provide fast look-ups. In this chapter, we propose a new directory organization with these three attributes called *SecDir*.



(a) Slice of the Intel Skylake-X directory.

(b) Slice of SecDir.

Figure 5.2: Slice of a conventional directory (a) and SecDir (b).

Figure 5.2(a) shows a slice of a conventional directory for non-inclusive cache hierarchies, such as that of Intel's Skylake-X, and how we change it into SecDir in Figure 5.2(b). The key idea of SecDir is to take a portion of the ED (e.g., some of its ways) and re-assign the storage to per-core private directory structures called the *Victim Directories* (VDs) (Figure 5.2(b)). We do not modify the TD because that would also require modifying the LLC.

A given core's VD in the slice shown is only *one bank* of the core's total VD. In other words, a core has a VD bank in every slice, each mapping a different set of addresses. This distributed VD of a core is sized such that it can hold as many directory entries as cache lines can be in a private L2 cache. This size will minimize self-conflicts in the VD in benign applications. Further, to provide high effective associativity, each VD bank uses *Cuckoo hashing* [125, 128]. Such a design also obscures any victim conflict patterns from the attacker. Finally, since, to a large extent, VD access hits should occur mostly during attacks, we simplify the hardware and access the VD only after ED and TD.

A slice has as many VDs as cores. Like the ED and TD, the VDs in a slice only contain directory entries for lines mapped to the local slice. A VD is set associative and, because it keeps information for a single core, it does not need sharer information bits. Such information is effectively encoded in the VD ID.

The VD of a core loads an entry when the local ED plus TD have a conflict and need to evict the directory entry for a line that lives in that core's L2 cache. The VD evicts an entry in two cases. The first one is when the entry suffers a conflict; in this case, if the entry belonged to a dirty line, the line is written back from the core's L2 to memory. The second case is when the corresponding data line is evicted from the core's L2 into the LLC; in this case, the directory entry is moved from the VD to the TD. Full details of the VD operation are presented in Section 5.3.

The VD is accessed after the ED/TD declare a miss. Hence, the VD complements the ED/TD: the ED/TD provide fast directory lookup, while the VD blocks interference of directory entries used by different cores in an attack. Overall, SecDir has the attributes discussed above:

Provides Isolation Inexpensively and Scalably In a slice, there are as many VDs as cores, each owned by a core. Hence, directory entries used by different cores are isolated, and cannot interfere with each other. A given core has one VD bank in each slice. We size the banks so that, together, all the banks for a core across slices can accommodate as many entries as lines fit in an L2 cache. This helps minimize self-conflicts in the distributed VD of a core running a victim program, as we can assume that the lines referenced by a benign victim program are largely uniformly distributed across the different slices.

Note that the VD design is *scalable* with the number of cores in the machine: irrespective of the number of cores in the machine, the size of the distributed VD for a core is practically constant. As more cores are added, the size of a VD bank in each slice decreases, but the number of slices and, therefore, the number of VD banks, increases.

Provides High Associativity A slice of SecDir has a high effective associativity. It has the associativity of the ED plus TD (available to all processes) plus the associativity of the private VD bank augmented with cuckoo hashing (Section 5.4). Under benign conditions, the VD is unlikely to be highly utilized. Under attack conditions, the victim can utilize its core's VD banks across all the slices to isolate the directory entries corresponding to its L2 lines.

Uses Low Area SecDir reassigns some storage from ED to VD. An important insight in SecDir is that the ED needs substantial storage to keep sharer information, while a core-private directory structure like VD does not need such information (Figure 5.2(b)). Hence, SecDir takes ED tags, which include sharer information, and converts them to VD tags, which do not. The VD is area efficient.

Importantly, the overhead of the sharer information in an ED entry tends to increase with the number of cores in the machine (e.g., more presence bits). Hence, as the number of cores increases, we can add more VD entries per core, as the VDs reuse more sharer information bits.

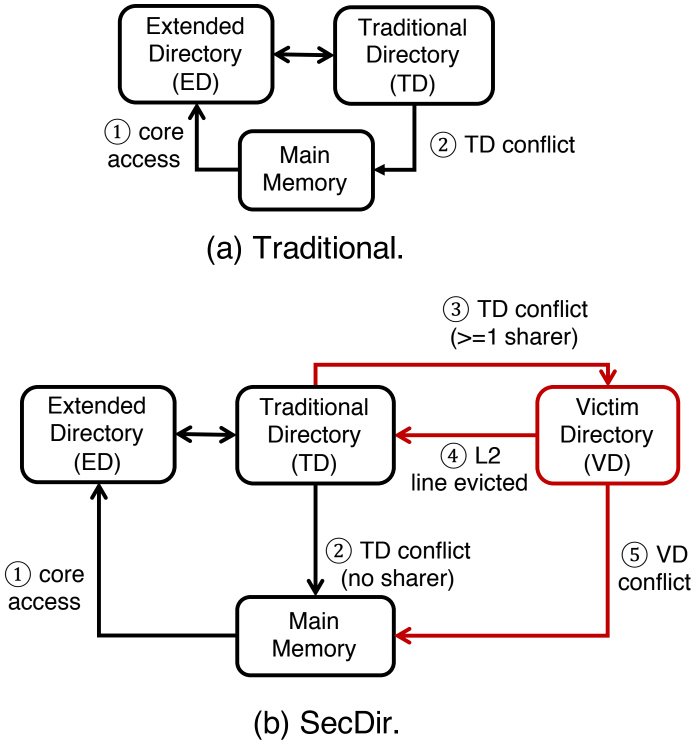
This fact produces a surprising effect. Suppose that we redesign Skylake-X's directory into SecDir's, using the following guidelines: the number of entries in an ED slice, and the number of entries in a core's VD machine-wide is *each* equal to the number of lines in a private L2 cache. In this case, SecDir only needs 28.5KB more directory storage per slice than the Skylake-X directory for an 8-core machine, and it uses *less* directory storage than the Skylake-X for 44 cores or more. We present more details in Section 5.6.

Delivers Efficient Directory Lookup Under ordinary, attack-free conditions, most of the directory hits are satisfied by the TD or ED. When TD and ED miss and a VD bank is accessed, the VD typically misses. At that point, a main memory access is initiated. Compared to a main memory access latency, a VD access latency is very small. However, it is still important for the VD accesses to be efficient. Consequently, as we will see, VDs have an *Empty Bit* (EB), which helps to avoid unnecessary VD accesses. The EB saves substantial energy and some latency (Section 5.4.2).

Note that a given directory entry can be, at the same time, in multiple VD banks in the slice (i.e., banks of different cores). This is a security requirement, as we will see. Hence, on a VD access, we sometimes need to search all the banks. Searching multiple banks does not increase the VD access latency, as the different banks are independent (Section 5.4.1).

5.3 SECDIR DIRECTORY OPERATION

Any line in the cache hierarchy has to have a corresponding directory entry. In this section, we describe how the SecDir directory operates. First, however, we recall how a traditional directory operates. For simplicity, the following discussion assumes a MESI cache-coherence protocol. SecDir can work with any protocol — e.g., our evaluation in Section 5.7.1 uses a MOESI protocol.



(a) Traditional.

(b) SecDir.

Figure 5.3: Operation of a traditional (a) and SecDir (b) directory.

5.3.1 Traditional Directory Operation

Figure 5.3(a) shows the operation of a traditional directory in a non-inclusive cache hierarchy. When a core accesses a line that does not exist in the cache hierarchy, the line is directly brought from main memory into the core's private cache. No data is inserted into the LLC. At the same time, a directory entry for the line is inserted in the ED (①). As discussed in Section 4.3.5, a directory entry can migrate from ED to TD due to a conflict in an ED set, or due to a cache line eviction from an L2; a directory entry can migrate from TD to ED due to a core writing to a line that has a directory entry in the TD.

When a TD conflict occurs, the conflicting TD entry cannot be moved to the ED (where it could have possibly come from), as it could cause a conflict deadlock going back and forth. Instead, the conflicting TD entry is discarded, which automatically causes all copies of the corresponding cache line to be evicted from the cache hierarchy (②). This is the transition that an attacker uses to create an inclusion victim in the victim's private cache. Specifically, the attacker first forces the eviction of a victim directory entry from ED, and then from TD.

5.3.2 SecDir Operation

Figure 5.3(b) shows the operation of SecDir. As in the traditional directory, when a line is fetched from main memory to a core's private cache, its directory information is stored in the ED (①). Further, the directory state can migrate between ED and TD. The migration follows the operations in the traditional directory, except for one limitation in the Skylake processor, which will be discussed shortly in Section 5.3.3. This limitation is not the root cause of directory attacks.

The main difference between SecDir and traditional directory occurs on a TD conflict. Depending on the sharer information in the conflicting directory entry, two different transitions may occur. First, if the directory entry shows that there are no sharers of the cache line (i.e., the cache line is only in the LLC), then the conflicting TD entry is discarded and, if the cache line is dirty, the cache line is written back to main memory (②). Note that it is secure to allow attackers to evict victim lines in such a way. The reason is that, as we will see, for a victim line to be only in LLC, it means that the victim process has evicted the line from its private L2 due to a self-conflict. We do not consider a victim's self-conflicts in its private caches as part of the attack model, since they are not caused by an active attacker.

The second case is when the conflicting TD entry shows that there are one or more caches with a copy of the cache line. Hence, evicting the cache line from the L2s would create a vulnerability like in conventional directories (Section 5.3.1). Consequently, in this case, the state in the conflicting TD entry is migrated to VD (③). Specifically, for each of the sharers of the line, as specified in the TD entry, SecDir creates an entry in the corresponding local VD bank. To be secure, every single sharer needs to have a VD entry, because every sharer needs to retain the line in its L2. Fortunately, this operation is local to the directory, does not generate cache coherence transactions, and has no impact on L2 cache states. In particular, the victim process is unaffected: it continues to access the cache line out of its L1/L2 caches, completely unaware that the directory entry has moved from the TD to the VD. Further, after the entry is inserted in the VD, it cannot be tampered with by the actions of other cores, thanks to the partitioned nature of the VD.

A directory entry in the VD can be moved out of the VD in two cases. The first one is when, due to a conflict in an L2 cache, a cache line is evicted from an L2. In this case, SecDir consolidates all the VD entries for the line (which result from multiple cores sharing the line) into a single TD entry (④). It also inserts the line into the LLC, so that future accesses to this cache line get it from the LLC.

Note that SecDir has to consolidate entries from as many VD banks as cores share the line. Hence, this operation requires searching all the VD banks in the slice and, for each match found, removing the entry from the VD bank and setting a bit in the presence bit vector in the new directory entry in the TD. Fortunately, an L2 line eviction is not on the critical path of serving cache accesses, and this search overhead can be hidden. Further, Section 5.4 shows how to optimize this operation. Finally, note that this transition does not create a vulnerability, since it is created by a victim self-conflict in the victim's L2. The resulting self-eviction is not part of our attack model because it is not caused by an active attacker.

The second case when SecDir moves an entry out of the VD is a conflict in the VD. The conflicting entry cannot be moved back to TD (where it came from) because it could cause a conflict deadlock. Instead, SecDir discards the entry, and invalidates the corresponding cache line from the corresponding L2 — writing it back to main memory if dirty (⑤). Any other copies of the line in other L2s, together with their VD entries, are undisturbed.

This operation is secure according to our model because, as the VD is partitioned, such VD conflicts can only be self-conflicts among directory entries owned by the same core. An active attacker attempting a cross-core cache attack has no way to directly enforce such VD self-conflicts. Also, recall that we size the distributed VD for a core across all the slices to be similar to the size of the core's L2. In this case, even in the worst case when an attacker forces all of the victim's directory entries into the victim's VD, the victim will likely still be able to retain most of its L2 lines.

The SecDir transitions are summarized in Table 5.1.

Transition	VD Access Type	Coherence Transaction	Security
②: TD → Memory	—	If cache line in Dirty state in LLC, write it back to memory; invalidate line from LLC	No leakage
③: TD → VD	Insert the directory entry into the VDs of all the sharers	—	No leakage
④: VD → TD	Search all VD banks to remove any matching directory entry	Write back the cache line to the LLC	Leak only L2 self-conflicts (safe)
⑤: VD → Memory	Remove the conflicting directory entry from the VD bank	Write back the corresponding cache line from the core's L2 to memory if in Dirty state; invalidate the line from that L2	Leak only VD self-conflicts (safe)

Table 5.1: Summary of SecDir transitions.

5.3.3 Fixing A Limitation in Skylake-X

Among the attacks discussed in Chapter 4 on the non-inclusive caches of Intel's Skylake-X processor, one of them does not exploit the limited associativity of the overall directory structure — which, as discussed in Section 5.1, is the root cause of directory attacks. Instead, our prime+probe attack exploits a limitation in the implementation of the cache coherence states in Skylake-X's cache hierarchy. In this section, we discuss this limitation and suggest a simple method to fix it. Such a fix has been incorporated in our SecDir implementation.

Consider a victim with the target cache line in its private cache in the Exclusive coherence state, and the line's directory entry in the ED. The attacker creates conflicts in the ED, evicting the target line's directory entry from ED to TD. In Skylake-X, each TD entry must be associated with data — i.e., it must have a corresponding cache line in the LLC. Consequently, as the directory entry moves from ED to TD, the target line is copied to LLC. Unfortunately, the line cannot remain, at the same time, in Exclusive state in the victim's private cache. Hence, it is invalidated from the victim's private cache. This invalidation causes an inclusion victim, which is leveraged by the attacker to complete the prime step. This is a limitation of Skylake-X's implementation, since this invalidation is unnecessary.

To fix this limitation, we suggest to allow TD entries to be associated with empty LLC lines. In the example presented, as the directory entry is moved from ED to TD, we propose to keep the LLC entry empty, and retain the Exclusive cache line in the private cache. In this way, ED conflicts do not cause L2 evictions. After adopting this mechanism, the only way to create an L2 eviction is to exploit the limited associativity of the combined TD plus ED directory structure.

5.4 VICTIM DIRECTORY DESIGN

This section discusses how to access the VD, and VD's features for security and efficiency.

5.4.1 Accessing the Victim Directory

The VD is accessed differently than the ED and TD. Let us assume that the ED and TD have associativities equal to W_{ED} and W_{TD}, respectively, and that they have the same number of sets. As shown in Figure 5.4(a), the ED/TD are accessed as a conventional cache structure of $W_{ED}+W_{TD}$ associativity. For simplicity, the figure neglects the Coherence State bits of each directory line. When an address tag match is found, the sharer information is read.

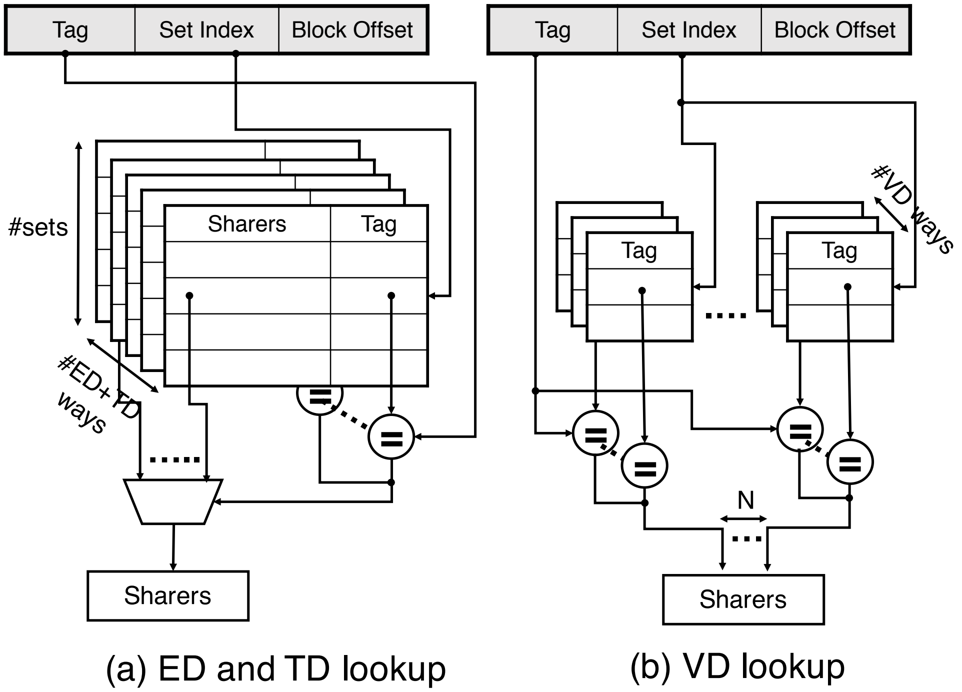


Figure 5.4: Accessing directories. For simplicity, we neglect the Coherence State bits.

In a VD bank, there is no sharer information field. An access only provides a single bit (hit or miss). A directory entry can be in multiple VD banks. Hence, as shown in Figure 5.4(b), SecDir needs to potentially search all the local VD banks. Each VD bank may match, in which case it contributes with a set bit to a presence bit vector. The vector gives the sharing information for the cache line.

Given the size of the ED and TD, most accesses under attack-free conditions hit in the ED/TD. However, we still want the VD access to be fast and, therefore, SecDir keeps the associativity of each VD bank (W_{VD}) modest. Note that searching all the local banks in parallel does not slow down the VD access, as the searches are independent. Unlike searches in an associative cache, this design does not need an additional multiplexer to select one of the banks, as it produces a bit vector.

There are three types of searches performed on the VDs of a slice. On a read request, SecDir only needs to find *one* VD bank with the matching address tag. The coherence protocol will access the L2 of the core that owns the bank, and retrieve the line. On a write, SecDir searches all the local VD banks to obtain the complete sharer bit vector. A VD entry for the writing core is allocated (if it does not exist already), and all the other matching entries are invalidated. Finally, if SecDir performs transition ④ in Table 5.1 (i.e., a cache line is evicted from a private cache), the VD operation involves finding all the matching entries in VD, generating a complete sharer bit vector, creating a TD entry, and invalidating all the matching entries in the VD.

In machines with many cores, SecDir can save hardware by performing the VD search operation in batches — e.g., by accessing and searching 8 VD banks at a time. This implementation saves hardware, but results in slower searches. In this case, on a read operation, SecDir calls off the search as soon as one matching entry is found.

5.4.2 Victim Directory Features

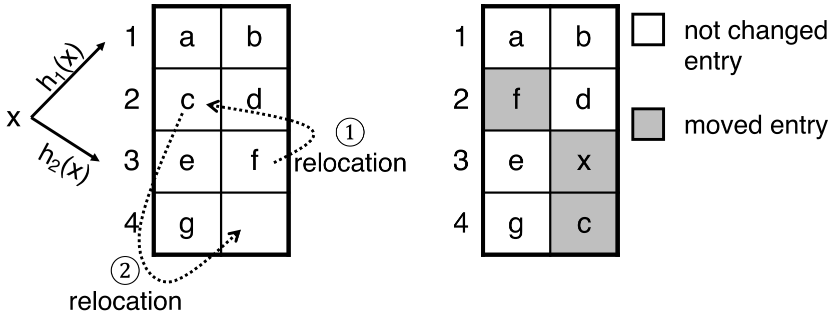
The VD has two features that are helpful in two different scenarios: one helps in pathological directory conflicts caused by attackers, and the other in executions without attacks.

VD Bank Organization as a Cuckoo Directory In a directory attack, the attacker tries to cache in the private caches of multiple cores many lines that map to a single set of a single directory slice. In the worst case, the attacker completely fills the set of both ED and TD in the slice, and SecDir has to move all the victim directory entries in that set to the victim's VD bank.

While the attacker concentrates its accesses on lines that map to specific directory sets and slices, a benign victim application generally distributes its directory entries across directory sets and slices evenly. Consequently, in our design, we size the distributed VD for a core across all the slices so that it holds as many entries as the number of lines that fit in an L2 cache. This design should allow the VD to retain many of the directory entries needed by the victim. However, the victim may still suffer self-conflicts in the VD. To minimize the number of self-conflicts in the VD, SecDir organizes each VD bank as a *Cuckoo Directory* [125,128].

A cuckoo directory is an organization that increases the occupancy of a directory by using multiple hash functions to insert an entry in the directory. The result is a higher effective associativity and, hence, fewer evictions.

A cuckoo directory admits multiple organizations, some more sophisticated than others. In SecDir, we use a very simple design, to show the potential of the scheme. Figure 5.5 shows an example of how to insert an item in a two-way set-associative directory that uses SecDir's design. When inserting a new item `x`, assume that the two hash functions $h_1(x)$ and $h_2(x)$ select sets 1 and 3 (Figure 5.5(a)). Since both sets are full, one of them is selected and one entry in that set is evicted to provide space for `x`. Assume that item `f` in set 3 is selected for eviction. After its eviction, `f` is re-inserted in a line of its alternative set — Set 2 in the figure (①). Since there is no space in Set 2, one item needs to be evicted and relocated. Assume that it is item `c`, which is moved from Set 2 to 4 (②). Since there is space



(a) Before inserting item x (b) After item x inserted

Figure 5.5: Example of cuckoo directory insertion operation.

in Set 4, c is inserted and there is no additional eviction. Figure 5.5(b) shows the resulting directory state. Shaded entries are those that have been moved. In the general case, the relocation procedure is repeated until it either finds an empty slot in the directory, or until a maximum number of relocations (*NumRelocations*) is reached. In the later case, an item is kicked out of the directory structure, which is not likely to be from the same cache set as the item that was first inserted. This fact confuses the attacker.

On a VD bank look-up, the two hash functions can return at most one hit. To confirm a VD bank miss, both functions have to miss. Note that the cuckoo organization requires one extra bit per VD entry, to indicate which hash function was used (*Cuckoo* bit).

The cuckoo operation is useful during an attack, as it increases the occupancy of victim VD banks and reduces victim self-conflicts in VD. Moreover, even if the victim suffers self-conflicts in its VD banks, the randomization of the conflicts obscures the conflict patterns from the attacker. This increases the victim's resistance against even a passive attacker.

Early Detection of VD Misses In an execution without attacks, the VD is likely to be highly underutilized. In this case, it is desirable to quickly detect when a VD bank access is guaranteed to miss, and save energy by skipping the access. SecDir supports this operation by adding an *Empty Bit* (EB) to each set of each VD bank. The EB bit is wired to the NOR of the Valid bits of all the entries in the set of the VD bank. Hence, if an EB bit is set, it means that the corresponding set in the VD bank is empty.

With this support, VD bank queries that search for a certain directory entry proceed as follows. First, the hardware accesses the EB arrays of the VD banks with the correct set index bits. If an EB array returns a logic one, the hardware skips the ordinary access to the VD bank array; otherwise, it proceeds with the access. This design saves energy. Moreover, since accessing the EB arrays is faster than accessing the VD bank arrays, this design also saves some access latency.

The EB hardware can be organized in different ways. One implementation has a separate EB array per VD bank. Another combines the EB bits of all the VD banks into a single EB array of width N.

5.5 DICUSSION: VD TIMING CONSIDERATIONS

The fact that the VD is accessed after the ED/TD are accessed raises the question of whether an attacker could exploit a timing side channel. Specifically, an attacker could push a victim's directory entries from the ED/TD to the VD, and then time the execution of the victim's program to find whether it takes longer.

If the victim is a single-threaded program, there is no such timing side channel. A victim's execution is oblivious to whether the directory entry is in ED/TD or VD. In either case, the corresponding cache line is in the victim's private cache. On accessing the line, the victim hits in its private cache, and does not access the directory.

The situation is different in a multi-threaded victim where two or more victim threads share writable data. Specifically, every time that one core writes a line and sends an invalidation to another core, the directory is accessed. Similarly, when a core accesses the line and obtains it from another core's cache, the directory is accessed. In both cases, accessing the directory in the VD rather than in the ED/TD makes the coherence transaction take a bit longer. For example, using the parameters of the system we evaluate in Section 5.7.1, accessing the VD extends by about 7 cycles a transaction that would otherwise typically take about 100 cycles.

While this timing side channel may be hard to exploit due to the non-determinism of cross-thread communication (i.e., each thread's accesses occur asynchronously to other threads'), we need to disable this side channel. One way to do so is by artificially slowing down a response from the ED/TD by the time it would take to additionally access the VD. A naive solution would apply such slowdown to every ED/TD-satisfied transaction. A more advanced solution would apply such slowdown only to ED/TD-satisfied transactions that involve invalidating or querying another core's cache. We leave the implementation and evaluation of this solution to future work.

5.6 A POSSIBLE DESIGN OF SECDIR

As an example of a possible SecDir design, we take the parameters of the Intel Skylake-X directory [19] and modify them to support SecDir. We are interested in comparing the Skylake-X and SecDir directories for the same total directory storage. For simplicity, in our analysis, we make a few assumptions on the cache coherence protocol and the encoding of the cache coherence states. Specifically, we use the MESI coherence protocol, and encode the sharer information in each directory entry as a "full-mapped" bit vector of N presence bits (where N is the number of cores in the machine) [120]. Using a full-mapped bit vector is reasonable for modest core counts. Also, we neglect any extra bits needed to encode transient cache coherence states. With these assumptions, the structures of Figure 5.2 only need the following Coherence State bits: TD entries need a Dirty and a Valid bit, while ED and VD entries only need a Valid bit.

The storage of the Skylake-X directory includes TD and ED; the storage of SecDir includes TD, a new ED, and VD. To size VD, we partition the original (i.e., Skylake-X's) ED into a new (i.e., SecDir's) ED and VD. Specifically, we take some ways off the original ED and give the storage to the VD. Hence, the original ED and new ED have the same number of sets but different number of ways. This is the simplest reorganization strategy. We use random replacement in ED and VD, and conservatively neglect the storage taken by any replacement algorithm bits in TD.

Table 5.2 lists the relevant parameters of Intel's Skylake-X to the best of our knowledge. They include physical address, L2 cache, TD, and ED parameters. The table also shows the SecDir parameters for ED and VD. Since the values of SecDir's parameters will change in our experiments, we refer to them as variables W_{ED}, W_{VD}, and S_{VD}. Recall that each entry in a VD bank has a Cuckoo bit, and each set in a VD bank has an Empty bit (EB).

The ED in Skylake-X has an associativity of 12, and the ED in SecDir has an associativity of only W_{ED}. We use the difference in ways (i.e., $12 - W_{ED}$ ways) to build the VD banks in a slice. Specifically, we consider SecDir designs where W_{ED} is 6, 7, 8, 9, or 10. For a given W_{ED} and core count, we design the VD as follows. We consider VD bank associativities (W_{VD}) ranging from 3 to 8. We choose the VD design with the highest directory entry count and a power-of-two number of sets (S_{VD}) that fits in the storage available.

Once we pick a VD design, we count the number of directory entries that such a design provides to a single core across all the slices. In case of an attack, these are the directory entries that the victim can use in an isolated manner. We compare this number to the number of lines in an L2 cache. The ratio between these two numbers is shown in Figure 5.6. Values above 1 mean that the per-core VD contains more directory entries than lines in L2.

Parameter	**Value**	**Parameter**	**Value**
Physical Address		**Extended Directory (ED) in SecDir**	
Line address	40 bits	# ways	W_{ED}
Line offset	6 bits	# sets	2048
L2 Cache		**Victim Directory (VD) in SecDir**	
# ways	16	# VD banks/slice	N
# sets	1024	# ways per VD bank	W_{VD}
Traditional Directory (TD)		# sets per VD bank	S_{VD}
# ways	11	Cuckoo bit (per entry)	1 bit
# sets	2048	Empty bit (per set)	1 bit
Address tag	29 bits		
Extended Directory (ED)			
# ways	12		
# sets	2048		
Address tag	29 bits		

Table 5.2: Parameters of the Intel Skylake-X directory and SecDir. The number of cores is represented by N.

In SecDir, we want to have at least as many directory entries in the per-core VD as lines in L2. The figure shows that, even allowing the ED to retain a large number of ways (i.e., W_{ED} out of the original 12), we quickly attain a per-core VD that has as many entries as lines in L2. This is because the VD, unlike the ED, does not store sharer information. Specifically, the Skylake-X ED has 12 ways and holds 1.5× as many entries as L2 lines. SecDir can keep 8 ways for the ED (ensuring that the ED holds as many entries as L2 lines), and reassign 4 ways to the per-core VD. At 44 cores or more, such per-core VD can also hold as many entries as L2 lines or more. If we only have 8 cores, which is the design we evaluate in Section 5.7.1, and we still want to keep 8 ways for the ED, the per-core VD needs extra storage to have as many entries as the L2. It can be easily computed that it only needs 28.5 Kbytes per slice to have as many entries as the L2. This is a very small overhead compared to the sizes of an L2 and an LLC slice.

5.7 EVALUATION

5.7.1 Experimental Setup

We evaluate SecDir and compare it to Skylake-X's directory using simulations with Gem5 [129]. The parameters of the architecture with SecDir are shown in Table 5.3, which augments the parameters in Table 5.2. We implement a directory-based MOESI cache coherence proto-

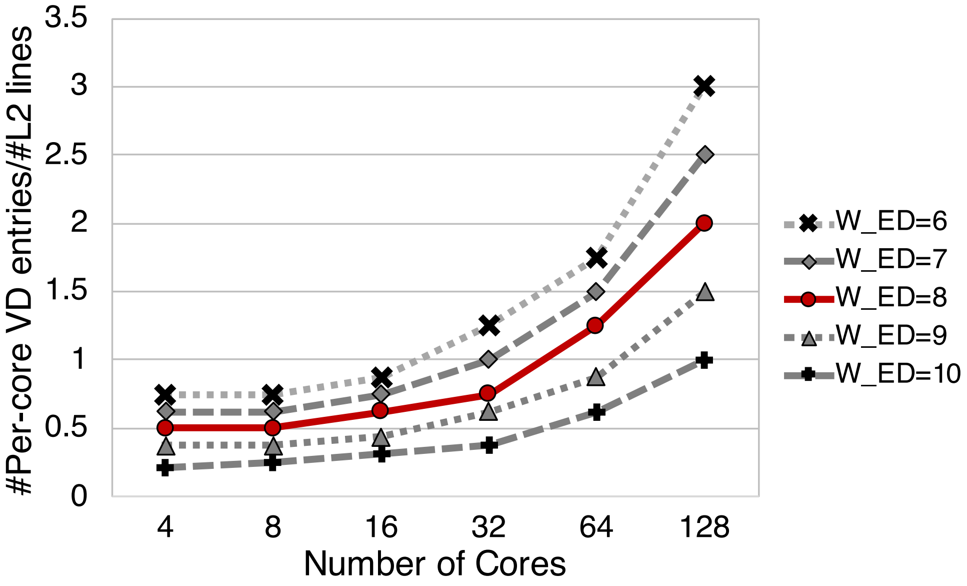


Figure 5.6: Comparing the number of per-core VD entries machine-wide to the number of lines in an L2 cache. Values above 1 mean that the per-core VD has more entries than lines in an L2. We use a SecDir design with the same directory storage as a Skylake-X.

col. Recall from Section 5.6 that both SecDir and Skylake-X use the same cache and TD configurations. As shown in the tables, the EDs of Skylake-X and SecDir in a slice have the same number of sets (2048) but different set-associativity, namely 12 and 8, respectively. This means that the ED of SecDir in a slice has as many entries as lines in L2. The VD is designed so that, per core across all the slices, it also has as many entries as lines in L2. As indicated in Section 5.6, with these parameters, SecDir needs 28.5KB more directory storage per slice than the Skylake-X directory for an 8-core machine. This is a very small overhead considering the sizes of the L2 and the LLC slice. We call the Skylake-X architecture *Baseline*.

In our cuckoo directory implementation, we use the skewing hash functions proposed by Seznec and Bodin [130] as our $h_1(x)$ and $h_2(x)$ functions. These functions distribute cache lines equally among sets, possess local and inter-bank dispersion properties, and can be easily implemented in hardware. We set the cuckoo NumRelocations to 8.

We evaluate SecDir and Skylake-X with 12 mixes of single-threaded SPEC applications [109] and 10 multi-threaded PARSEC applications [110]. We use the same approach as Jaleel et al. [96] to pick the mixes of SPEC applications. Specifically, we run 23 individual SPECInt2006 and SPECFP2006 applications on a single core and one slice of the non-inclusive LLC structure. These applications are classified into three categories, namely core cache fitting (CCF), last-level cache fitting ($LLCF$), and last-level cache thrashing ($LLCT$),

Parameter	**Value**
Architecture	8 cores at 2.0GHz using MOESI dir coherence
Core	8 issue, out-of-order, no SMT, 32 load queue entries, 32 store queue entries, 192 ROB entries
Private L1-I	32KB, 64B line, 4-way, 4 cycle round-trip (RT) latency
Private L1-D	32KB, 64B line, 8-way, 4 cycle RT latency
Private L2	1MB, 64B line, 16-way, 10 cycles RT latency
Shared L3 (per slice)	1.375MB, 64B line, 11-way, 30 cycles RT local latency, 50 cycles RT remote latency
Directory (per slice)	TD: 11-way, 2048 sets; ED: 8-way, 2048 sets; num VD banks: 8; VD bank: 4-way, 512 sets; NumRelocations: 8
Directory RT latency	To TD/ED: same as L3. To VD: over L3, add 2 cycles (EB access) and, if miss in EB, add 5 cycles (VD access)
Network	4×2 mesh, 128b link width
DRAM	RT latency: 50 ns after L3

Table 5.3: Parameters of the SecDir architecture.

according to their L2 and L3 miss rates. We consider the 6 possible combinations of two of these categories, and select 2 application mixes in each combination, as listed in Table 5.4.

Category	Name & Applications	Name & Applications
CCF, CCF	mix0: 4 gobmk + 4 sjeng	mix1: 4 hmmer + 4 gamess
LLCF, LLCF	mix2: 4 bzip2 + 4 omnetpp	mix3: 4 gromacs + 4 zeusmp
LLCT, LLCT	mix4: 4 libquantum + 4 lbm	mix5: 4 bwaves + 4 sphinx3
CCF, LLCF	mix6: 4 sjeng + 4 omnetpp	mix7: 4 h264ref + 4 zeusmp
CCF, LLCT	mix8: 4 gobmk + 4 libquantum	mix9: 4 namd + 4 bwaves
LLCF, LLCT	mix10: 4 omnetpp + 4 bwaves	mix11: 4 zeusmp + 4 lbm

Table 5.4: SPEC workload mixes.

We use the reference input size for the SPEC applications. When running these mixes on 8 cores, we run 4 copies of each application, and assign them to different cores. We skip the first 10 billion instructions, and report simulation results for 500 million cycles. For the PARSEC applications, we use the simmedium input size and report simulation results for the region-of-interest (ROI).

5.7.2 Security Evaluation

We evaluate the security properties of SecDir on the AES encryption algorithm [131]. Software implementations of AES usually use 4 look-up tables, called T-tables, to improve the performance of the cipher computation. The encryption process involves multiple rounds, and each round has a round key. To generate the result for a certain round, the algorithm uses the last round's result to look-up the T-tables, and then perform an XOR operation on the obtained value and the round key. This implementation is known to be vulnerable to conflict-based cache attacks [2], as the access patterns on the T-tables leak intermediate encryption results and round keys.

In a conflict-based cache attack, the attacker aims to observe the victim's access patterns on the T-tables. First, it evicts all the T-table entries from the cache hierarchy, and then tests which entries have been accessed by the victim in each round of encryption by analyzing the resulting cache states.

SecDir can effectively block such attacks. In SecDir, the ED and TD are shared by all the cores, but the VDs are per-core private. The most powerful adversary can take full control of the ED and TD, but is unable to interfere with the entries in the victim's VD. In order to emulate such a powerful attacker, we simulate SecDir without ED or TD, assuming that the attacker fully controls these two structures. This is the most pathological scenario that an attacker can create.

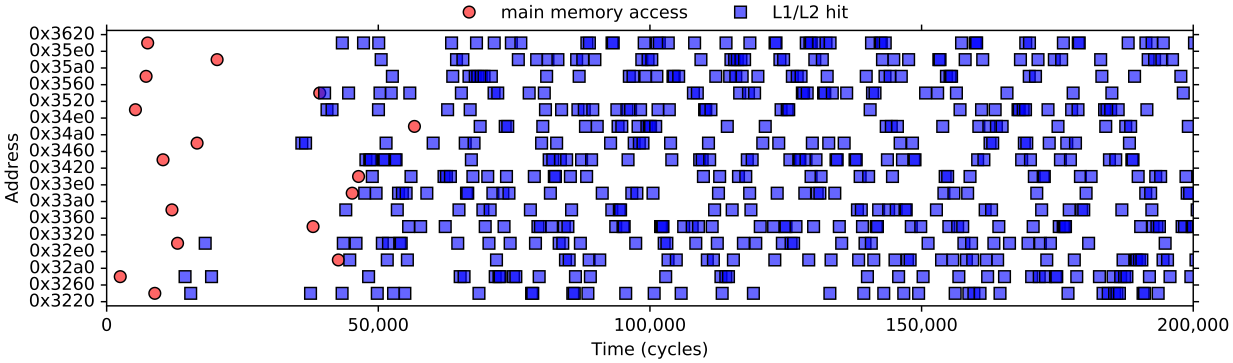


Figure 5.7: Trace of accesses to the T_0 table in AES encryption. The application runs on SecDir with VD but no ED or TD.

In this scenario, we run the AES encryption implementation from OpenSSL 0.9.8, and record the access patterns on the T_0 T-table. The table uses 16 memory lines. Figure 5.7 shows the addresses of the T_0 memory lines accessed as a function of time. Accesses are classified as: (i) main memory accesses or (ii) L1 or L2 hits. Note that, in this experiment, if a line is evicted from L2, its VD entry is evicted too because there is no TD. Consequently, since this is a single-threaded application, no L2 cache miss will hit in the VD; all L2 cache misses will access main memory.

From the figure, we see that the first access to each memory line of the table misses in the cache hierarchy and causes a main memory access. As the line is fetched from memory, its directory entry is inserted into VD, since there is no TD or ED. As seen in the figure, all of the victim's subsequent accesses to the T_0 table lines hit in the private L1/L2 caches. The attacker cannot observe these private cache hits, and cannot directly interfere with the entries in the victim's VD.

SecDir is effective at protecting other applications, such as the square-and-multiply operations in the RSA encryption algorithm. The data in the leaky region of RSA is much smaller than the T-table. Hence, it fits in the L2 cache and its directory entries fit in the VD. An attacker cannot evict the target lines from L2.

If the victim's security-sensitive data is large, the victim may suffer self-conflicts in its L2 and/or in its VD. In this case, the victim may leak information. However, recall that protecting against such leakage is not within the scope of SecDir, as such leakage is not created by an active attacker.

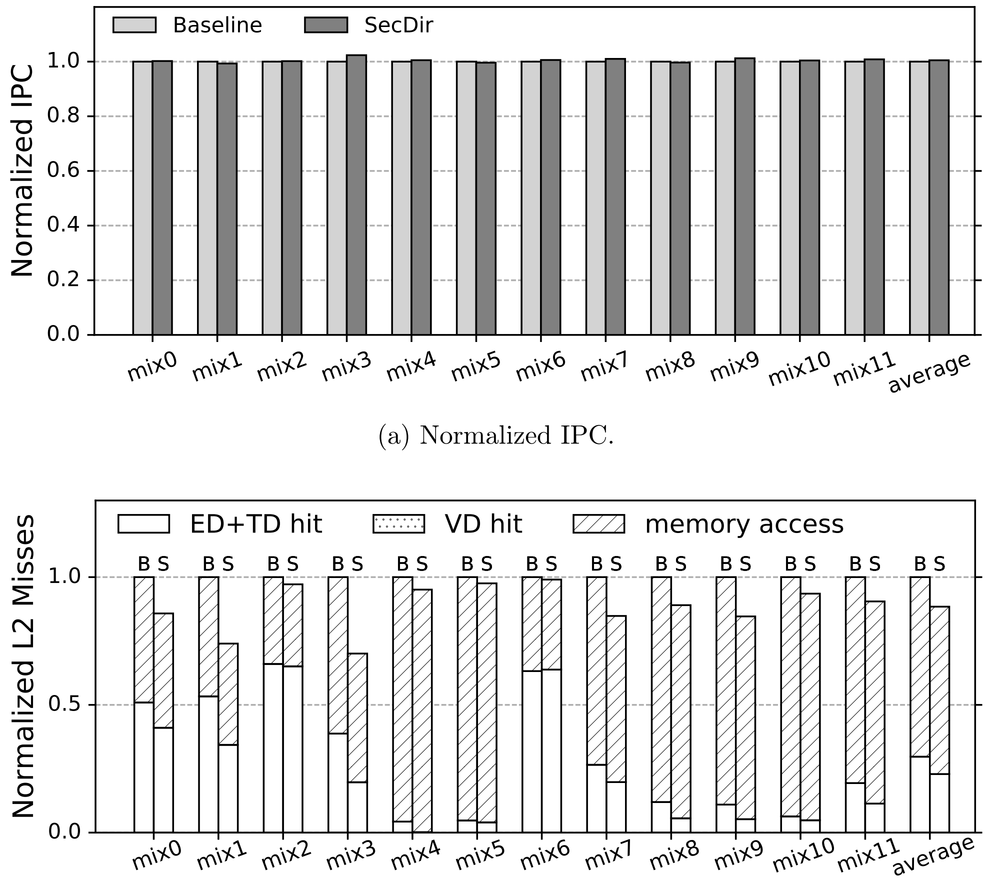
5.7.3 Evaluation of SPEC Application Mixes

Figure 5.8 evaluates SecDir executing SPEC mixes. Figure 5.8(A) shows the average instructions per cycle (IPC) of the SPEC mixes running on SecDir and Baseline. For each mix, the bars are normalized to Baseline. From the figure, we see that the IPC of the mixes changes little across architectures. The reason is that SecDir has a positive and a negative effect on the execution time, and both effects tend to cancel out.

SecDir's negative effect is that it slightly increases the latency of a main memory access. This is because a request that misses in ED and TD checks the VD on its way to main memory. As indicated in Table 5.3, checking the VD takes 2 cycles (if the EB array satisfies the request) or 2 plus 5 cycles (if it does not). Under attack-free conditions, the VD is not highly utilized, and most VD accesses miss. However, this does not mean that the VD is ineffective, since whatever entries the VD contains are very useful. Indeed, such entries enable the L2 to retain cache lines and, therefore, avoid an L2 miss and directory access in the first place.

SecDir's positive effect is that it reduces the number of directory entry conflicts and, therefore, the number of cache line inclusion victims. In Baseline, a TD entry conflict typically results in an L2 cache line eviction to DRAM. In SecDir, instead, the directory entry is moved to VD and the corresponding cache line remains in L2, avoiding an inclusion victim. Avoiding L2 inclusion victims results in fewer memory accesses and fewer ED/TD accesses. This effect improves performance.

To understand the positive effect, Figure 5.8(B) shows the number of L2 misses and breaks them into: (i) hits in the ED or TD, (ii) hits in the VD, and (iii) misses in the directory,



(a) Normalized IPC.

(b) L2 miss characterization. In the figure, *B* is Baseline, and *S* is SecDir.

Figure 5.8: Evaluation of the SPEC mixes.

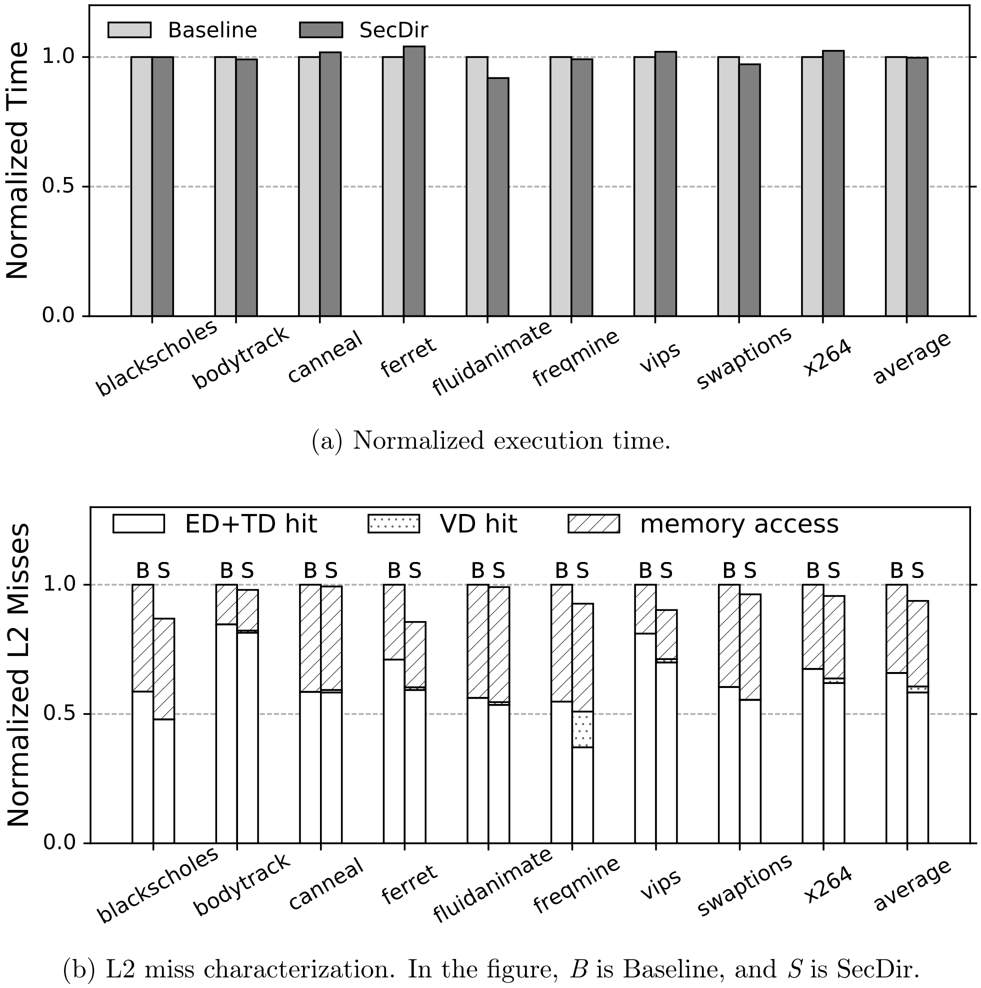
which cause main memory accesses. For each SPEC mix, the figure shows bars for Baseline (B) and SecDir (S), which are normalized to the former.

From the figure, we see that SecDir decreases the number of L2 misses in practically all the mixes. On average, the reduction is 11.4%. The reduction comes from reducing L2 inclusion victims. It results in both fewer memory accesses and fewer ED/TD hits.

The bars show that there are no VD hits. This is the normal behavior in single-threaded applications and, as indicated before, it does not mean that the VD is useless. On the contrary, for VD entries in use, there is no reason to access the VD: the L1/L2 caches contain the line and intercept any access to the line from the core. When the line is evicted from L2 and written back to the LLC, SecDir migrates the VD entry to TD (④ in Figure 5.3(b)). Hence, subsequent L2 misses will either hit in the TD or, if the entry is evicted from TD before the line is re-referenced, obtain the data from main memory. In none of the cases will the VD hit.

5.7.4 Evaluation of PARSEC Applications

Figure 5.9 shows the same data as Figure 5.8 for the PARSEC applications. Figure 5.9(A) shows the execution time on Baseline and on SecDir, normalized to the Baseline. Similar to Figure 5.8(A), SecDir has a very small impact on the performance of PARSEC applications. As before, this is a combination of both positive and negative effects.



(a) Normalized execution time.

(b) L2 miss characterization. In the figure, *B* is Baseline, and *S* is SecDir.

Figure 5.9: Evaluation of the PARSEC Applications.

Figure 5.9(B) shows the L2 miss count of PARSEC applications in Baseline and in SecDir, normalized to Baseline. As in Figure 5.8(B), the bars are broken down into: (i) hits in the ED or TD, (ii) hits in the VD, and (iii) main memory accesses. Like in SPEC mixes, SecDir reduces the L2 misses in most applications, causing a reduction in the number of ED/TD hits and, to a lesser extent, in the number of memory accesses. The average L2 miss reduction is 7%.

The figure shows that, on average, the number of VD hits is very small. However, in the *freqmine* application, nearly 14% of the L2 misses are intercepted by VD. This scenario is possible in multi-threaded applications. It occurs when the L2 of a core misses on a cache line present in another core's L2 and the line has its directory entry in the VD of the second core. We expect this case to occur in applications with high data sharing between cores. In this case, a fast VD would improve performance.

Overall, in both single-threaded and multi-threaded applications, SecDir provides an effective defense against directory based side channel attacks, while introducing a negligible performance cost in the absence of attacks.

5.7.5 Evaluating VD Features

Table 5.5 examines two features of the VD, namely, the Empty bit (EB) and the cuckoo organization. To assess the impact of the EB, we compare the number of VD bank look-ups performed when using the EB (EBVD) to the number of VD bank look-ups performed when not using the EB (NoEBVD). Recall that, without EB, we need to perform a look-up of the N VD banks in a slice every time that we miss in the ED/TD directory.

SPEC	EBVD /	CKVD /	PARSEC	EBVD /	CKVD /
Mix	NoEBVD	NoCKVD	Appl.	NoEBVD	NoCKVD
mix0	0.45	0.75	blackschol.	0.01	0.63
mix1	0.18	0.95	bodytrack	0.18	0.55
mix2	0.36	0.53	canneal	0.09	0.51
mix3	0.53	0.93	ferret	0.23	0.59
mix4	0.38	1.00	fluidanim.	0.08	0.46
mix5	0.44	0.84	freqmine	0.38	0.62
mix6	0.43	0.41	vips	0.24	0.83
mix7	0.49	0.81	swaptions	0.01	0.56
mix8	0.45	0.96	x264	0.34	0.59
mix9	0.52	0.90	Avg.	0.17	0.59
mix10	0.48	0.73			
mix11	0.47	1.03			
Avg.	0.43	0.82			

Table 5.5: Evaluating the Empty bit and cuckoo organization.

The columns in the table labeled EBVD/NoEBVD show the ratio between the two measures. Since the VD is under-utilized in executions without attacks, the EB effectively decreases the number of VD bank look-ups across all applications. On average, only 43% and 17% of the VD bank accesses are needed with EB in SPEC and PARSEC applications, respectively. For two PARSEC applications, namely blackscholes and swaptions, EB eliminates practically all of the VD look-ups, as the VD remains highly unused during execution.

We also evaluate the effectiveness of using the cuckoo organization in the VD under the worst possible attack conditions. Specifically, we assume the case where the adversary fully controls the TD and ED directories, and the victim can only use the VD. Consequently, we disable TD and ED, and the application can only use its VD. We compare the number of VD self-conflicts when the VDs are used as cuckoo directories (CKVD) and when they are used as plain directories (NoCKVD). CKVD uses two of the skewing hash functions proposed by Seznec and Bodin [130] (Section 5.7.1); NoCKVD simply uses one of them.

The columns in the table labeled CKVD/NoCKVD show the ratio between the two self-conflict counts. We see that the cuckoo organization often eliminates a substantial fraction of the self-conflicts. The impact on a given application depends on a variety of factors, including the application's access patterns and its working set. On average, with SecDir's simple cuckoo organization, 82% and 59% of the VD self-conflicts remain in SPEC and PARSEC applications, respectively. Note that there are two SPEC mixes (i.e., mix4 and mix11) for which our cuckoo design does not reduce the number of conflicts. As shown in Table 5.4, these mixes contain last-level cache thrashing (LLCT) applications. In these cases, the VD bank in one or more slices is full, and the cuckoo approach is unable to reduce self-conflicts. To reduce the self-conflicts in these mixes, we need to either increase the size or associativity of VD, or make the cuckoo implementation more sophisticated — e.g., by improving the hash functions used, or by increasing NumRelocations. We leave these avenues to future work. In any case, although our threat model does not target victim self-conflicts, we note that the cuckoo operation effectively reduces self-conflicts and obscures victim self-conflict patterns from the attacker.

5.7.6 Storage and Area Overhead

We compute the storage and area required by the directory of the baseline Skylake-X architecture and of SecDir. We use the parameters of Section 5.7.1. We compute storage in Kbytes and area in mm^2 as given by CACTI 7 [132] using 22 nm technology. Table 5.6 shows the per-slice results for TD, ED, and VD.

We see that SecDir requires 28.5 KB additional storage per LLC slice, which is 12.9% more than the baseline architecture. Also, the SecDir directory structures take 0.027 mm^2 more area, which is 16.2% more than the baseline. Overall, these are modest numbers. Also, this analysis is for a machine with 8 cores. As indicated in Section 5.6, SecDir uses less directory area than the baseline for 44 cores or more.

Baseline Structure	Storage (KB)	Area (mm^2)	SecDir Structure	Storage (KB)	Area (mm^2)
TD	107.25	0.080	TD	107.25	0.080
ED	114.00	0.087	ED	76.00	0.057
—	—	—	VD	66.50	0.057
Total	221.25	0.167	Total	249.75	0.194

Table 5.6: Storage and area used by the directory structures in a slice in the baseline Skylake-X and in SecDir.

5.8 CONCLUSION

This chapter presented *SecDir*, a secure directory to defeat directory side channel attacks. SecDir takes part of the storage used by a conventional directory and re-assigns it to per-core private directory areas used in a victim-cache manner called Victim Directories (VDs). To minimize victim self-conflicts in a VD during an attack, a VD is organized as a cuckoo directory. Such a design also obscures victim self-conflict patterns from the attacker. We modeled with simulations the directory of an Intel Skylake-X server and a modified design that supports SecDir. Our results showed that SecDir has a negligible performance impact. Furthermore, SecDir is area-efficient: it only needs 28.5KB more directory storage than the Skylake-X per slice for an 8-core machine, while it uses less storage than the Skylake-X for 44 cores or more. Finally, a cuckoo VD organization eliminated substantial victim self-conflicts in a worst-case attack scenario.

CHAPTER 6: INVISISPEC

This chapter presents the first robust hardware defense mechanism against transient cache-based side channel attacks for multiprocessors. It has been shown that hardware speculation offers a major surface for micro-architectural side channel attacks. Unfortunately, defending against transient execution attacks is challenging. To address this challenge, we propose Invisible Speculation (InvisiSpec). The idea is to make speculation invisible in the data cache hierarchy. In InvisiSpec, unsafe speculative loads read data into a speculative buffer, without modifying the cache hierarchy. When the loads become safe, InvisiSpec makes them visible to the rest of the system. We show that InvisiSpec has modest performance overhead, and it is able to defend against future transient execution attacks where any speculative load can pose a threat.

The recent disclosure of Spectre [3] and Meltdown [4] has opened a new chapter in hardware security pertaining to the dangers of speculative execution. Hardware speculation can cause execution to proceed in ways that were not intended by the programmer or compiler. Until recently, this was not thought to have security implications, as incorrect speculation is guaranteed to be squashed by the hardware. However, these attacks demonstrate that squashing incorrect speculative paths is insufficient for security.

Spectre and Meltdown monitor the micro-architectural footprint left by speculation, such as the state left by wrong-path speculative loads in the cache. This footprint enables micro-architectural covert or side channels, where an adversary can infer the speculative thread's actions. In Meltdown, a user-level attacker is able to execute a privileged memory load and leak the memory contents using a cache-based covert channel, all before the illegal memory load triggers an exception. In Spectre, the attacker poisons the branch predictor or branch target buffer to force the victim's code to speculate down specific paths of the attacker's choosing. Different types of attacks are possible, such as the victim code reading arbitrary memory locations by speculating that an array bounds check succeeds—allowing the attacker to glean information through a cache-based side channel. In this case, unlike in Meltdown, there is no exception-inducing load.

It can be argued that Meltdown can be fixed with a relatively modest implementation change: mark the data loaded by an exception-triggering load as unreadable. However, defending against Spectre and other forms of transient execution attacks that do not cause exceptions presents a two-fold challenge. First, potentially *any* instruction performing speculation can result in an attack. The reason is that speculations destined to be squashed inherently execute incorrect instructions (so-called *transient* instructions), outside the scope of what programmers and compilers reason about. Since program behavior is undefined, transient execution attacks will be able to manifest in many ways, similar to attacks exploiting lack of memory safety [133]. Second, speculative code that makes *any* change to micro-architectural state—e.g., cache occupancy [38] or cache coherence [9] state—can create side channels.

In this chapter, we propose *InvisiSpec*, a novel strategy to defend against hardware speculation attacks in multiprocessors, by making speculation invisible in the data cache hierarchy. The goal is to block micro-architectural side channels through the multiprocessor data cache hierarchy due to speculative loads—e.g., channels stemming from cache set occupancy, line replacement information, and cache coherence state. We wish to protect against not only Spectre-like attacks based on branch speculation; we also want to protect against futuristic attacks where any speculative load may pose a threat.

InvisiSpec's micro-architecture is based on two mechanisms. First, unsafe speculative loads read data into a new *Speculative Buffer* (SB) instead of into the caches, without modifying the cache hierarchy. Data in the SB do not observe cache coherence transactions, which may result in missing memory consistency violations. Our second mechanism addresses this problem. When a speculative load is finally safe, the InvisiSpec hardware makes it visible to the rest of the system by reissuing it to the memory system and loading the data into the caches. In this process, if InvisiSpec considers that the load could have violated memory consistency, InvisiSpec *validates* that the data that the load first read is correct—squashing the load if the data is not correct.

To summarize, this chapter makes the following contributions:

- We present a generalization of speculative execution attacks that goes beyond current Spectre attacks, where any speculative load may pose a threat.
- We present a micro-architecture that blocks side and covert channels during speculation in a multiprocessor data cache hierarchy.
- We simulate our design on 23 SPEC and 9 PARSEC workloads. Under TSO, using fences to defend against Spectre attacks slows down execution by 82% relative to a conventional, insecure processor; InvisiSpec reduces the execution slowdown to only 5%. Using fences to defend against futuristic attacks slows down execution by 231%; InvisiSpec reduces the slowdown to 17%.

6.1 UNDERSTANDING TRANSIENT EXECUTION ATTACKS COMPREHENSIVELY

6.1.1 An Example of Transient Execution Attacks

Transient execution attacks exploit the side effects of *transient* instructions, which are the instructions that are speculatively executed but are destined to be squashed. Algorithm 6.1 describes the Spectre Variant 1 attack, also known as array bounds bypassing attack.

Algorithm 6.1: Spectre Variant 1 attack.

```
1  uint8 $array1[array1\_size]$; // array that contains a secret
2  uint8 $array2[256 * 64]$;// cache line size is 64 bytes
   // victim code begins here:
3  Function victim($x$):
4      if $x < array1\_size$ then // source of transient instruction
5          $secret = array1[x]$;
6          $junk = array2[secret * 64]$; // transmitter
7      end
8  End
   // attacker code begins here:
9  Function attacker($x$):
10     train(); // train the branch predictor of line 4 to be taken
11     flush($array2$); // flush every cache line of array2
12     call victim($Y - \&array1$);// secret value is stored in address $Y$
13     scan($array2$); // access each cache line of array2
14 End
```

The victim code (Line 3-8) is part of a function (e.g., a system call or a library function) receiving an unsigned integer x from an untrusted source. If the given x is within the bounds of *array1*, the code uses x as the index to access *array1* and the returned value is later used as the index to access another array *array2*.

The attacker's goal is to read content from arbitrary addresses (address Y in this example) in the victim's address space. It uses the following steps. First, the attacker trains the branch predictor of the victim's branch (Line 4) to be taken. This can be achieved by triggering the victim function multiple times using a x smaller than the size of *array1*. It also prepares the cache states for the later attack steps by flushing all the cache lines of *array2* from the cache hierarchy.

Second, the attacker performs the exploitation. It gives the victim function a malicious $x = Y - \&array1$ to make $array1[x]$ points to address Y. The branch in Line 4 should guard the two loads in Lines 5 and 6, and prevent them from executing when x is out of the bounds of *array1*. However, due to speculative execution, the branch is predicted to be taken and the two loads will be speculatively executed before the branch is resolved. The first load (Line 5) accesses Y and brings the secret value into a register. The second load (Line 6) uses the secret value as an index to access *array2*, and brings a line into the cache, changing the cache state. We call the second load a *transmitter* (Section 2.2.2), whose access leaks the secret value. Note that both load instructions are transient instructions, as they will be squashed after the branch is resolved.

Finally, the attacker scans *array2* and measures the access latency to each cache line in the array (Line 13). Based on the access latency, the attacker can figure out which line has been accessed by the victim and deduce the value of the secret.

6.1.2 Sources of Transient Instructions

We discuss the events that can be the source of transient instructions leading to a security attack. Table 6.1 shows examples of such attacks and what event creates the transient instructions.

Attack	What Creates the Transient Instructions
Meltdown	Virtual memory exception
L1 Terminal Fault	
Lazy Floating Point	Exception reading a disabled or privileged register
Rogue System Register Read	
Spectre	Control-flow misprediction
Speculative Store Bypass	Address alias between a load and an earlier store
Futuristic	Various events, such as: • Exceptions • Control-flow mispredictions • Address alias between a load and an earlier store • Address alias between two loads • Memory consistency model violations • Interrupts

Table 6.1: Understanding transient execution attacks.

Exceptions. Several attacks exploit speculation past an exception-raising instruction. The processor squashes the execution when the exception-raising instruction reaches the head of the ROB, but by this time, dependent transmitting instructions can leak data. The Meltdown [4] and L1 Terminal Fault (L1TF) [53,134] attacks exploit virtual memory-related exceptions. Meltdown reads a kernel address mapped as inaccessible to user space in the page table. L1TF reads a virtual address whose page table entry (PTE) marks the physical page as not present, but the physical address is still loaded.

The Lazy Floating Point (FP) State Restore attack [135] reads an FP register after the OS has disabled FP operations, thereby reading the FP state of another process. The Rogue System Register Read [135] reads privileged system registers.

Control-flow misprediction. Spectre attacks [3] exploit control-flow speculation to load from an arbitrary memory address and leak its value. Variant 1 performs an out-of-bounds array read, exploiting a branch misprediction of the array bounds check [3]. Other variants direct the control flow to an instruction sequence that leaks arbitrary memory, either through indirect branch misprediction [3], return address misprediction [56], or an out-of-bounds array write that redirects the control flow (e.g., by overwriting a return address on the stack) [136].

Memory-dependence speculation. Speculative Store Bypass (SSB) attacks [55] exploit a speculative load that bypasses an earlier store whose address is unresolved, but will resolve to alias the load's address [137]. Before the load is squashed, it obtains stale data, which can be leaked through subsequent dependent instructions. For example, suppose that a JavaScript runtime or virtual machine zeroes out memory allocated to a user thread. Then, this attack can be used to peek at the prior contents of an allocated buffer. We discovered the SSB attack in the process of performing this work.

6.1.3 A Comprehensive Model of Transient Execution Attacks

Transient instructions can be created by many events. As long as an instruction can be squashed, it can create transient instructions, which can be used to mount a transient execution attack. Since we focus on cache-based covert and side channel attacks, we limit ourselves to load instructions.

We define a *Futuristic* attack as one that can exploit any speculative load. Table 6.1 shows examples of what can create transient instructions in a Futuristic attack: all types of exceptions, control-flow mispredictions, address alias between a load and an earlier store, address alias between two loads, memory consistency model violations, and even interrupts.

In the rest of the chapter, we present two versions of InvisiSpec: one that defends against Spectre attacks, and one that defends against Futuristic attacks.

6.2 INVISISPEC DESIGN

6.2.1 Overview of the Techniques

Unsafe Speculative Loads Strictly speaking, any load that initiates a read before it reaches the head of the ROB is a speculative load. In this work, we are interested in the (large) subset of speculative loads that can create a security vulnerability due to speculation. We call these loads *Unsafe Speculative Loads (USLs).* The set of speculative loads that are USLs depends on the attack model.

In the Spectre attack model, USLs are the speculative loads that follow an unresolved control-flow instruction. As soon as the control-flow instruction resolves, the USLs that follow it in the correct path of the branch transition to *safe loads*—although they remain speculative.

In our Futuristic attack model, USLs are all the *speculative loads that can be squashed by an earlier instruction.* A USL transitions to a *safe load* as soon as it becomes either (i) non-speculative because it reaches the head of the ROB or (ii) speculative non-squashable by any earlier instruction (or *speculative non-squashable* for short). *Speculative non-squashable* loads are loads that both (i) are not at the head of the ROB and (ii) are preceded in the ROB only by instructions that cannot be squashed by any of the squashing events in Table 6.1 for Futuristic. Note that in the table, one of the squashing events is interrupts. Hence, making a load safe includes delaying interrupts until the load reaches the head of the ROB.

To understand why these two conditions allow a USL to transition to a safe load consider the following. A load at the ROB head cannot itself be transient; it can be squashed (e.g., due to an exception), but it is a correct instruction. The same can be said about speculative non-squashable loads. This is because, while not at the ROB head, they cannot be squashed by any earlier instruction and, hence, can be considered a logical extension of the instruction at the ROB head.

Making USLs Invisible The idea behind InvisiSpec is to make USLs *invisible.* This means that a USL cannot modify the cache hierarchy in any way that is visible to other threads, including the coherence states. A USL loads the data into a special buffer that we call *Speculative Buffer* (SB), and not into the local caches. As indicated above, there is a point in time when the USL can transition to a safe load. At this point, called the *Visibility Point*, InvisiSpec takes an action that will make the USL *visible*—i.e., will make all the side effects of the USL in the memory hierarchy apparent to all other threads. InvisiSpec makes the USL visible by re-loading the data, this time storing the data in the local caches and changing the cache hierarchy states. The load may remain speculative.

Maintaining Memory Consistency The *Window of Suppressed Visibility* for a load is the time period between when the load is issued as a USL and when it makes itself visible. During this period, since a USL does not change any coherence state, the core may fail to receive invalidations directed to the line loaded by the USL. Therefore, violations of the memory consistency model by the load run the risk of going undetected. This is because such violations are ordinarily detected via incoming invalidations, and are solved by squashing the load. To solve this problem, InvisiSpec may have to perform a *validation* step when it re-loads the data at the load's visibility point.

It is possible that the line requested by the USL was already in the core's L1 cache when the USL was issued and the line loaded into the SB. In this case, the core may receive an invalidation for the line. However, the USL ignores such invalidation, as the USL is invisible, and any effects of this invalidation will be handled at the visibility point.

Validation or Exposure of a USL The operation of re-loading the data at the visibility point has two flavors: *Validation* and *Exposure*. Validation is the way to make visible a USL that would have been squashed during the Window of Suppressed Visibility (due to memory consistency considerations) if, during that window, the core had received an invalidation for the line loaded by the USL. A validation operation includes comparing the actual bytes used by the USL (as stored in the SB) to their most up-to-date value being loaded from the cache hierarchy. If they are not the same, the USL and all its successive instructions are squashed. This is required to satisfy the memory consistency model. This step is reminiscent of Cain and Lipasti's value-based memory ordering [138].

Validations can be expensive. A USL enduring a validation cannot retire until the transaction finishes—i.e., the line obtained from the cache hierarchy is loaded into the cache, the line's data is compared to the subset of it used in the SB, and a decision regarding squashing is made. Hence, if the USL is at the ROB head and the ROB is full, the pipeline stalls.

Thankfully, InvisiSpec identifies many USLs that could not have violated the memory consistency model during their Window of Suppressed Visibility, and allows them to become visible with a cheap *Exposure*. These are USLs that would *not* have been squashed by the memory consistency model during the Window of Suppressed Visibility if, during that window, the core had received an invalidation for the line loaded by the USL. In an exposure, the line returned by the cache hierarchy is simply stored in the caches without comparison. A USL enduring an exposure can retire as soon as the line request is sent to the cache hierarchy. Hence, the USL does not stall the pipeline.

To summarize, there are two ways to make a USL visible: validation and exposure. The memory consistency model determines which one is needed. Figure 6.1 shows the timeline of a USL with validation and with exposure.

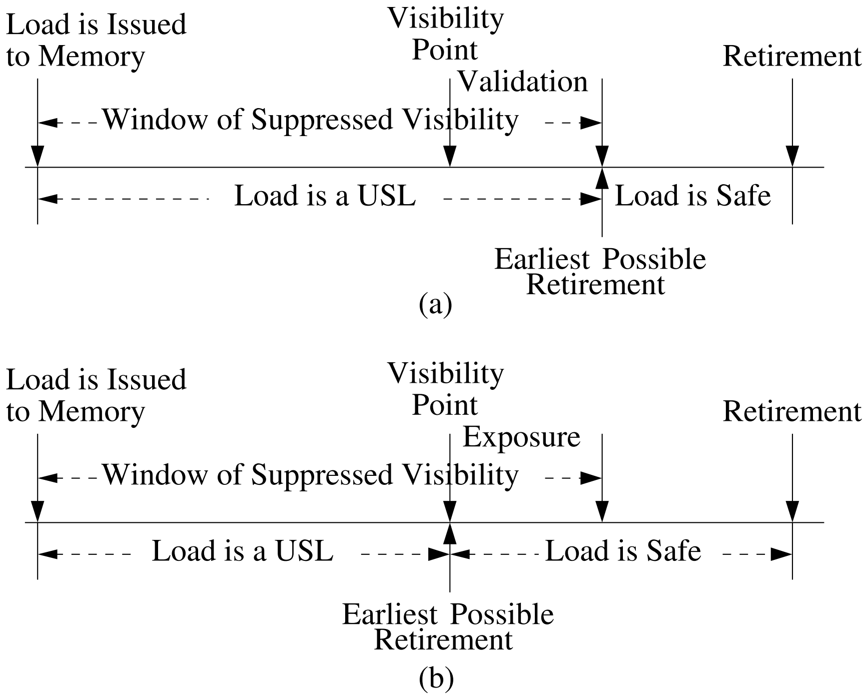


Figure 6.1: Timeline of a USL with validation (a) and exposure (b).

6.2.2 InvisiSpec Operation

A load in InvisiSpec has two steps. First, when it is issued to memory as a USL, it accesses the cache hierarchy and obtains the current version of the requested cache line. The line is only stored in the local SB, which is as close to the core as the L1 cache. USLs do not modify the cache coherence states, cache replacement algorithm states, or any other cache hierarchy state. No other thread, local or remote, can see any changes. However, the core uses the data returned by the USL to make progress. The SB stores lines rather than individual words to exploit spatial locality.

When the USL can be made visible, and always after it has received its requested cache line, the hardware triggers a validation or an exposure transaction. Such a transaction re-requests the line again, this time modifying the cache hierarchy, and bringing the line to the local caches. As detailed in Section 6.2.3, validation and exposure transactions operate differently and have different performance implications.

We consider two attack models, Spectre and Futuristic, and propose slightly different InvisiSpec designs to defend against each of these attacks. In our defense against the Spectre attack, a USL reaches its visibility point when all of its prior control-flow instructions resolve. At that point, the hardware issues a validation or an exposure transaction for the load depending on the memory consistency model and the load's position in the ROB (Section 6.2.3). If multiple USLs can issue validation or exposure transactions, the transactions have to start in program order, but can otherwise all overlap (Section 6.2.4).

In our defense against Futuristic attacks, a USL reaches its visibility point only when: (i) it is not speculative anymore because it is at the head of the ROB, or (ii) it is still speculative but cannot be squashed anymore. At that point, the hardware issues a validation or an exposure for the load depending on the memory consistency model and the load's position in the ROB (Section 6.2.3). If multiple USLs can issue validation or exposure transactions, the transactions have to be issued in program order. However, when a validation transaction is issued, no subsequent validation or exposure transaction can overlap with it; they all have to wait to be issued until the validation is complete (Section 6.2.4). On the other hand, when an exposure transaction is issued, all subsequent exposure transactions up to, and including, the next validation transaction can overlap with it (Section 6.2.4).

Overall, in the Spectre and Futuristic defense designs, pipeline stalls may occur when a validation transaction holds up the retirement of a load at the head of the ROB and the ROB is full. This is more likely in Futuristic than in Spectre.

We call these designs *InvisiSpec-Spectre* (or *IS-Spectre*) and *InvisiSpec-Future* (or *IS-Future*). They are shown in the first row of Table 6.2. For comparison, the second row shows how to defend against these same attacks using fence-based approaches—following current proposals to defend against Spectre [139]. We call these designs *Fence-Spectre* and *Fence-Future*. The former places a fence after every indirect or conditional branch; the latter places a fence before every load.

	Defense Against Spectre	Defense Against Futuristic
InvisiSpec Based	*InvisiSpec-Spectre*: 1) Perform invisible loads in the shadow of an unresolved BR; 2) Validate or expose the loads when the BR resolves	*InvisiSpec-Future*: 1) Perform an invisible load if the load can be squashed while speculative; 2) Validate or expose the load when it becomes either (i) non-speculative or (ii) un-squashable speculative
Fence Based	*Fence-Spectre*: Place a fence after every BR	*Fence-Future*: Place a fence before every load

Table 6.2: Summary of InvisiSpec designs (BR means indirect or conditional branch.)

Compared to the fence-based designs, InvisiSpec improves performance. Specifically, loads are speculatively executed as early as in conventional, insecure machines. One concern is that validation transactions for loads at the head of the ROB may stall the pipeline. However, we will show how to maximize the number of exposures (which cause no stall) at the expense of the number of validations. Finally, InvisiSpec does create more cache hierarchy traffic and contention to various cache ports. The hardware should be designed to handle them.

6.2.3 When to Use Validation and Exposure

The memory consistency model determines when to use a validation and when to use an exposure. Consider TSO first. In a high-performance TSO implementation, a speculative load that reads when there is no older load (or fence) in the ROB will not be squashed by a subsequent incoming invalidation to the line it read. Hence, such a USL can use an exposure when it becomes visible. On the other hand, a speculative load that reads when there is at least one older load (or fence) in the ROB will be squashed by an invalidation to the line it read. Hence, such a USL is required to use a validation.

Now consider RC. In this case, only speculative loads that read when there is at least one earlier fence in the ROB will be squashed by an invalidation to the line read. Hence, only those will be required to use a validation; the very large majority of loads can use exposures.

From this discussion, we see that the design with the highest chance of observing validations that stall the pipeline is IS-Future under TSO. To reduce the chance of these events, InvisiSpec implements two mechanisms. The first one enables some USLs that would use validations to use exposures instead. The second one identifies USLs that would use validations, and squashes them early if there is a high chance that their validations would fail. We consider these mechanisms next.

Transforming a Validation USL into an Exposure USL Assume that, under TSO, USL_1 initiates a read while there are earlier loads in the ROB. Ordinarily, InvisiSpec would mark USL_1 as needing a validation. However, assume that, at the time of issuing the read, all of the loads in the ROB earlier than USL_1 satisfy two conditions: 1) they have already obtained the data they requested—in particular, if they are USLs, the data they requested has already arrived at the SB and been passed to a register, and 2) if they needed to perform validations, they have completed them. In this case, USL_1 is not reordered relative to any of its earlier loads. As a result, TSO would not require squashing USL_1 on reception of an invalidation to the line it loaded. Therefore, USL_1 is marked as needing exposure, not validation.

To support this mechanism, we tag each USL with one bit called *Performed*. It is set when the data requested by the USL has been received in the SB and passed to the destination register. In addition, as we will see in Section 6.3.1, we tag each USL with two state bits to indicate whether the USL needs a validation or an exposure, and whether the validation or exposure has completed.

Early Squashing of USLs Needing Validations Assume that a core receives an invalidation for a line in its cache that also happens to be loaded into its SB by a USL marked as needing validation. Receiving an invalidation indicates that the line has been updated. Such update will typically cause the validation of the USL at the point of visibility to fail. Validation could only succeed if this invalidation was caused by false sharing, or if the net effect of all the updates to the line until the validation turned out to be silent (i.e., they restored the data to its initial value). Since these conditions are unlikely, InvisiSpec squashes such a USL on reception of the invalidation.

There is a second case of a USL with a high chance of validation failure. Assume that USL_1 needs validation and has data in the SB. Moreover, there is an earlier USL_2 to the same line (but to different words of the line) that also has its data in the SB and needs validation. When USL_2 performs the validation and brings the line to the core, InvisiSpec also compares the line to USL_1's data in the SB. If the data are different, it shows that USL_1 has read stale data. At that point, InvisiSpec conservatively squashes USL_1.

6.2.4 Overlapping Validations and Exposures

To improve performance, we seek to overlap validation and exposure transactions as much as possible. To ensure that overlaps are legal, we propose two requirements. The first one is related to correctness: during the overlapped execution of multiple validation and exposure transactions, the memory consistency model has to be enforced. This is the case even if some of the loads involved are squashed and restarted.

The second requirement only applies to the Futuristic attack model, and is related to the cache state: during the overlapped execution of multiple validation and exposure transactions, no squashed transaction can change the state of the caches. This is because the essence of our Futuristic model defense is to prevent squashed loads from changing the state of the caches. For the Spectre attack model, this requirement does not apply because validations and exposures are only issued for instructions in the correct path of a branch. While some of these instructions may get squashed (e.g., due to memory consistency violations) and change the state of the caches, these are correct program instructions and, hence, pose no threat under the Spectre attack model.

We propose sufficient conditions to ensure the two requirements. First, to enforce the correctness requirement, we require that the validation and exposure transactions of the different USLs start in program order. This condition, plus the fact that no validation or exposure for a USL can start until that USL has already received the response to its initial speculative request, ensures that the memory consistency model is enforced.

To enforce the second requirement for the Futuristic model, we require the following conditions. First, if a USL issues a validation transaction, no subsequent validation or exposure transaction can overlap with it. This is because, if the validation fails, it will squash all subsequent loads. Second, if a USL issues an exposure transaction, all subsequent exposure transactions up to, and including, the next validation transaction can overlap with it. This is allowed because exposures never cause squashes. In the Spectre model defense, validations and exposures can all overlap—since the cache state requirement does not apply.

Recall that validations and exposures bring cache lines. Hence, validations and/or exposures to the same cache line have to be totally ordered.

Proving Correctness of Enforcing Memory Consistency As discussed above, validation and exposure transactions have to be initiated in program order to guarantee that TSO memory consistency is maintained. To see why, recall that the only load-load reordering that leads to an observable violation of TSO is one in which two loads from processor P1, which in program order are $ld(y)$ first and $ld(x)$ second, interact with two stores from other processors, $st(x)$ and $st(y)$, such that they end up being globally ordered in a cycle as: $ld(x) \rightarrow st(x) \rightarrow st(y) \rightarrow ld(y)$ [140]. In other words, $ld(x)$ reads a value that gets overwritten by $st(x)$, $st(y)$ is globally ordered after $st(x)$, and $ld(y)$ reads the value stored by $st(y)$. We now prove that such global order is not possible.

We start by defining what it means for a load $ld(x)$ to be ordered between two writes to the same variable as in the expression $st(x) \rightarrow ld(x) \rightarrow st'(x)$. In InvisiSpec, the $ld(x)$ can be of one of three types: (i) a safe load, (ii) an unsafe speculative load (USL) that is followed by an exposure, or (iii) a USL that is followed by a validation. For type (i), the order expression above is clear. For type (ii), the $ld(x)$ in the expression is the USL access and not the exposure access, since the data returned by the exposure never reaches the pipeline. Finally, for type (iii), the $ld(x)$ in the expression is the validation access and not the USL access, since the data returned by the validation is the one that confirms the correctness of the transaction.

To prove that the cycle $ld(x) \rightarrow st(x) \rightarrow st(y) \rightarrow ld(y)$ is impossible, we consider nine cases, namely the first load $ld(y) \in$ {safe, USL with exposure, USL with validation} and the second load $ld(x) \in \{$safe, USL with exposure, USL with validation$\}$. We consider each case in turn.

1. $ld(x)$ is safe. In this case, the first load $ld(y)$ can only be safe. Further, when both $ld(y)$ and $ld(x)$ are safe, the conventional architecture design ensures that the cycle cannot be formed.
2. $ld(x)$ is a USL with validation. This is impossible because if $ld(x)$'s validation did read a value that got overwritten by $st(x)$ before $ld(y)$ committed, the resulting invalidation would squash $ld(x)$.
3. $ld(x)$ is a USL with exposure and $ld(y)$ is safe. This case cannot create a cycle because the condition for $ld(x)$ to be a USL with exposure is that $ld(y)$'s read must have received its requested data before $ld(x)$ reads.
4. $ld(y)$ and $ld(x)$ are both USLs with exposures. This case cannot create a cycle for the same reason as Case 3.
5. $ld(x)$ is a USL with exposure and $ld(y)$ is a USL with validation. This case cannot create a cycle because one condition for $ld(x)$ to be a USL with exposure is that $ld(y)$'s validation is completed before $ld(x)$ reads.

Therefore, our conditions of transforming validations to exposures (Sections 6.2.3) and the condition of overlapping validations and exposures (this sections) are sufficient to maintain the TSO consistency model.

RC does not prevent load-load reordering unless the loads are separated by a fence or synchronization. Hence, the in-order validation/exposure initiation requirement can be relaxed. For simplicity, InvisiSpec enforces this ordering for RC as well.

6.2.5 Reusing the Data in the Speculative Buffer

To exploit the spatial locality of a core's speculative loads, a USL brings a full cache line into the SB, and sets an Address Mask that identifies which part of the line is passed to a register. If a second USL that is *later* in program order now requests data from the line that the first USL has already requested or even brought into the SB, the second USL does not issue a second cache hierarchy access. Instead, it waits until the first USL brings the line into the first USL's SB entry. Then, it copies the line into its own SB entry, and sets its Address Mask accordingly.

6.2.6 Reducing Main-Memory Accesses

A USL performs two transactions—one when it is first issued, and one when it validates or exposes. Now, suppose that a USL's first access misses in the last level cache (LLC) and accesses main memory. Then, it is likely that the USL's second access is also a long-latency access to main memory.

To improve performance, we design InvisiSpec to avoid this second main-memory access most of the time. Specifically, we add a per-core LLC Speculative Buffer (LLC-SB) next to the LLC. When a USL's first access reads the line from main memory, as the line is sent back to the requesting core's L1 SB, InvisiSpec stores a copy of it in the core's LLC-SB. Later, when the USL issues its validation or exposure, it will read the line from the core's LLC-SB, skipping the access to main memory.

If, in between the two accesses, a second core accesses the line with a validation/exposure or a safe access, InvisiSpec invalidates the line from the first core's LLC-SB. This conservatively ensures that the LLC-SB does not hold stale data. In this case, the validation or exposure transaction of the original USL will obtain the latest copy of the line from wherever it is in the cache hierarchy.

6.3 DETAILED INVISISPEC DESIGN

6.3.1 Speculative Buffer in L1

Speculative Buffer Design InvisiSpec places the SB close to the core to keep the access latency low. Our main goal in designing the SB is to keep its operation simple, rather then minimizing its area; more area-efficient designs can be developed. Hence, we design the SB with as many entries as the Load Queue (LQ), and a one-to-one mapping between the LQ and SB entries (Figure 6.2).

This design makes several operations easy to support. Given an LQ entry, InvisiSpec can quickly find its corresponding SB entry. Further, since the LQ can easily find if there are multiple accesses to the same address, InvisiSpec can also identify multiple SB entries for the same line. Importantly, it is trivial to (i) allocate an SB entry for the next load in program order, (ii) remove the SB entry for a retiring load at the ROB head, and (iii) remove the SB entries for a set of loads being squashed. These operations need simple moves of SB's Head and Tail pointers—which are the LQ's Head and Tail pointers.

An SB entry does not store any address. It stores the data of a cache line plus an Address Mask that indicates which bytes were read. Each LQ entry has some status bits:

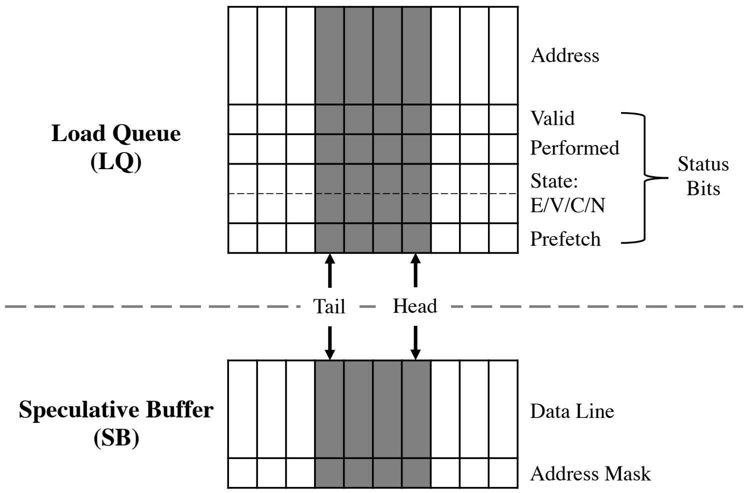


Figure 6.2: Speculative Buffer and its logical connection to the load queue.

Valid, Performed, State, and Prefetch. *Valid* records whether the entry is valid. *Performed* indicates whether the data requested by the USL has arrived and is stored in the SB entry. *State* are four bits that characterize the state of the load: "requiring an exposure when it becomes visible" (E), "requiring a validation when it becomes visible" (V), "exposure or validation has completed" (C), and "invisible speculation is not necessary for this load" (N). The latter is used when invisible speculation is not needed, and the access should go directly to the cache hierarchy. More than one of these bits can be set at a time. Recall that the *Performed* bit and the *State* bits are used to change a validation to a cheaper exposure (Section 6.2.3). Finally, *Prefetch* indicates whether this entry corresponds to a prefetch (Section 6.3.2).

Operation of the Load Queue and Speculative Buffer We describe the LQ and SB algorithms at key events.

- A load instruction is issued: The hardware allocates an LQ entry and an SB entry. The LQ entry's Valid bit is set.

- The address of a load is resolved: The load is ready to be sent to the cache hierarchy. If the load is safe according to the attack model, the State bits in the LQ entry are set to N and the load is issued to the cache hierarchy with a normal coherence transaction. The SB entry will be unused.

  Otherwise, the load is a USL, and the State is set to E or V, as dictated by the memory consistency model (Section 6.2.3). The USL is issued to the cache hierarchy with a new *Spec-GetS* transaction. This transaction requests the line in an invisible mode, without changing any state in any cache. The address mask in the SB entry is set. More details on how *Spec-GetS* changes the coherence protocol are given in Section 6.3.5.

Similar to conventional core designs, as a *Spec-GetS* request is issued, the hardware first tries to reuse any relevant local state. However, InvisiSpec reuses only state allocated by prior (in program order) instructions, to avoid creating new side channels (Section 6.4). Specifically, the LQ determines whether there is any prior load in the LQ that already requested the same line. If so, and if the line has been received by the SB, InvisiSpec obtains the line from the SB entry where it is, and copies it to the requesting USL's SB entry. If, instead, the line has not yet been received, the requesting USL modifies the prior load's MSHR waiting for the line, so that the line is also delivered to its own SB entry on arrival. If there is a pending request for the line by a later (in program order) load, the USL issues a *Spec-GetS* that allocates a new MSHR. Having multiple *Spec-GetS* in flight for the same address does not pose a problem because they do not change directory state (Section 6.3.5).

Finally, if there is a prior store in the ROB to the same address, the store data is forwarded to the USL's destination register, and to the USL's SB entry. The Address Mask of the SB entry is set. Then, a *Spec-GetS* transaction is still issued to the cache hierarchy, while recording that, when the line is received, it should not overwrite the bytes in the SB entry that are marked by the Address Mask. We choose this design to treat all SB entries equally: they should contain the data line that their USLs have read speculatively.

- The line requested by a USL arrives at the core: The line is copied to the SB entry of the requesting USL, and the requested bytes are passed to the USL's destination register—unless, as indicated above, the data has already been forwarded by a store. The Performed bit of the LQ entry is set. All these operations are performed for the potentially multiple USLs waiting for this line. These operations are performed no matter where the line comes from; in particular, it could come from the local L1 cache, which remains unchanged.

- A USL reaches the visibility point: If the USL's State in the LQ is V, InvisiSpec issues a validation; if it is E, InvisiSpec issues an exposure. These transactions can also reuse MSHRs of earlier requests (in program order) to the same line.

- The response to an exposure or validation arrives: The incoming line is saved in the local cache hierarchy as in regular transactions. If this was an exposure response, the requesting USL may or may not have retired. If it has not, InvisiSpec sets the USL's

State to C. If this was a validation response, the USL has not retired and needs to be validated. The validation proceeds by comparing the bytes that were read by the USL with the same bytes in the incoming cache line. If they match, the USL's State is set to C; otherwise, the USL is squashed—together with all the subsequent instructions. Squashing moves the LQ and SB tail pointer forward.

All these operations are performed for the potentially multiple USLs waiting for this line to become visible. At this point, InvisiSpec also can perform the early squash operation of Section 6.2.3.

Note that, in IS-Spectre, squashing a USL can lead to squashing subsequent USLs that have already become visible. Squashing such USLs does not cause any security problem, because they are safe according to IS-Spectre, and so their microarchitectural side-effects can be observed.

- An incoming cache line invalidation is received: In general, this event does not affect the lines in the SB; such lines are invisible to the cache coherence protocol. However, we implement the optimization of Section 6.2.3, where some USLs may be conservatively squashed. Specifically, the LQ is searched for any USL with the Performed bit set that has read from the line being invalidated, and that requires validation (i.e., its State is V). These USLs are conservatively squashed, together with their subsequent instructions. Of course, as in conventional cores, the incoming invalidation may affect other loads in the LQ that have brought lines into the caches.
- A cache line is evicted from the local L1: Since the lines in the SB are invisible to the cache coherence protocol, they are unaffected. As in conventional cores, the eviction may affect other loads in the LQ that have brought lines into the caches.

Primitive Operations Table 6.3 shows the primitive operations of the SB. Thanks to the SB design, all the operations are simple. Only the comparison of data in a validation is in the critical path.

6.3.2 Supporting Prefetching

InvisiSpec supports software prefetch instructions. Such instructions follow the same two steps as a USL. The first step is an "invisible" prefetch that brings a line to the SB without changing any cache hierarchy state, and allows subsequent USLs to access the data locally. The second one is an ordinary prefetch that brings the line to the cache when the prefetch

Operation	How It is Done	Complexity?	Critical Path?
Insert the data line requested by a USL	Index the SB with the same index as the LQ. Fill the entry	Low	No
Validate an SB entry	Use the Address Mask to compare the data in the SB entry to the incoming data	Low	Yes
Copy one SB entry to another	Read the data from one entry and write it to another	Low	No

Table 6.3: Primitive operations of the SB.

can be made visible. This second access is an exposure, since prefetches need not go through memory consistency checks.

To support software prefetches, InvisiSpec increases the size of a core's SB and, consequently, LQ. The new size of each of these structures is equal to the maximum number of loads and prefetches that can be supported by the core at any given time. InvisiSpec marks the prefetch entries in the LQ with a set *Prefetch* bit.

To be secure, InvisiSpec does not support speculative hardware prefetching. Only when a load or another instruction is made visible, can that instruction trigger a hardware prefetch.

6.3.3 Per-Core Speculative Buffer in the LLC

InvisiSpec adds a per-core LLC-SB next to the LLC. Its purpose is to store lines that USLs from the owner core have requested from main memory, and to provide the lines when InvisiSpec issues the validations or exposures for the same loads—hence avoiding a second access to main memory.

To understand the LLC-SB design, consider the case of a USL that issues a request that will miss in the LLC and access the LLC-SB. However, before the USL receives the data, the USL gets squashed, is re-issued, and re-sends the request to the LLC-SB. First, for security reasons, we do not want the second request to use data loaded in the LLC-SB by a prior, squashed request (Section 6.4). Hence, InvisiSpec forbids a USL from obtaining data from the LLC-SB; if a USL request misses in the LLC, the request bypasses the LLC-SB and accesses main memory. Second, it is possible that the two USL requests get reordered on their way to the LLC-SB, which would confuse the LLC-SB. Hence, we add an Epoch ID to each core, which is a counter that the hardware increments every time the core squashes instructions. When a core communicates with its LLC-SB, it includes its Epoch ID in the message. With this support, even if two USL requests from different epochs are reordered in transit, the LLC-SB will know that, given two requests with different IDs, the one with the higher ID is the correct one.

We propose a simple design of the per-core LLC-SB, which can be easily optimized. It is a circular buffer with as many entries as the LQ, and a one-to-one mapping between LQ and LLC-SB entries. Each LLC-SB entry stores the data of a line, its address, and the ID of the epoch when the line was loaded. USL requests and validation/exposure messages contain the address requested, the index of the LLC-SB where the data should be written to or read from, and the current Epoch ID. With this design, the algorithm works as follows:

- A USL request misses in the LLC: The request skips the LLC-SB and reads the line from memory. Before saving the line in the indexed LLC-SB entry, it checks the entry's Epoch ID. If it is higher than the request's own Epoch ID, the request is stale and is dropped. Otherwise, line, address, and Epoch ID are saved in the entry, and the line is sent to the core.
- A validation/exposure request misses in the LLC: The request checks the indexed LLC-SB entry. If the address and Epoch ID match, InvisiSpec returns the line to the core, therefore saving a main memory access. Otherwise, InvisiSpec accesses main memory and returns the data there to the core. In both cases, InvisiSpec writes the line into the LLC, and invalidates the line from the LLC-SBs of all the cores (including the requesting core). This step is required to purge future potentially-stale data from the LLC-SBs (Section 6.2.6). Invalidating the line from the LLC-SBs requires search, but is not in the critical path, as InvisiSpec does it in the background, as the line is being read from main memory and/or delivered to the requesting core.
- A safe load misses in the LLC: The request skips the LLC-SB and gets the line from main memory. In the shadow of the LLC miss, InvisiSpec invalidates the line from the LLC-SBs of all the cores, as in the previous case. The line is loaded into the LLC.

6.3.4 Disabling Interrupts

In IS-Future, the hardware can initiate a validation or exposure only when the USL becomes either (i) non-speculative because it reaches the ROB head or (ii) speculative non-squashable by any earlier instruction. If we wait until the load reaches the ROB head to start the validation, the pipeline may stall. Therefore, it is best to initiate the validation as soon as the load becomes speculative non-squashable. As indicated in Section 6.2.1, speculative non-squashable loads are loads that, while not at the ROB head, are preceded in the ROB only by instructions that cannot be squashed by any of the squashing events in Table 6.1 for Futuristic. As shown in the table, one of the squashing events is interrupts. Therefore, to make a load speculative non-squashable, interrupts need to be delayed from the time that the load would otherwise be speculative non-squashable, until the time that the load reaches the ROB head.

To satisfy this condition, InvisiSpec has the ability to automatically, transparently, and in hardware disable interrupts for very short periods of time. Specifically, given a USL, when the hardware notices that none of the instructions earlier than the USL in the ROB can be squashed by any of the squashing events in Table 6.1 for Futuristic except for interrupts, the hardware disables interrupts. The validation or exposure of the USL can then be initiatiated. As soon as the USL reaches the head of the ROB, interrupts are automatically enabled again. They remain enabled for at least a minimum period of time, to ensure that interrupts are not starved.

6.3.5 Other Implementation Aspects

Changes to the Coherence Protocol InvisiSpec adds a new *Spec-GetS* coherence transaction, which obtains a copy of the latest version of a cache line from the cache hierarchy without changing any cache or coherence states. For instance, in a directory protocol, a *Spec-GetS* obtains the line from the directory if the directory owns it; otherwise, the directory forwards the request to the owner, which sends a copy of the line to the requester. The directory does not order forwarded *Spec-GetS* requests with respect to other coherence transactions, to avoid making any state changes. For this reason, if a forwarded *Spec-GetS* arrives at a core after the core has lost ownership of the line, the *Spec-GetS* is bounced back to the requester, which retries. The requesting USL cannot starve as a result of such bounces, because eventually it either gets squashed or becomes safe. In the latter case, it then issues a standard coherence transaction.

Atomic Instructions Since an atomic instruction involves a write, InvisiSpec does not execute them speculatively. Execution is delayed as in current processors.

Securing the D-TLB To prevent a USL from observably changing the D-TLB state, InvisiSpec uses a simple approach. First, on a D-TLB miss, it delays serving it via a page table walk until the USL reaches the point of visibility. If the USL is squashed prior to that point, no page table walk is performed. Second, on a D-TLB hit, any observable TLB state changes such as updating D-TLB replacement state or access/dirty bits are delayed to the USL's point of visibility. A more sophisticated approach would involve using an SB structure like the one used for the caches.

No ABA Issues In an ABA scenario, a USL reads value A into the SB, and then the memory location changes to B and back to A prior to the USL's validation. An ABA scenario does not violate the memory model in an InvisiSpec validation.

6.4 SECURITY ANALYSIS

InvisiSpec's SB and LLC-SB do not create new side channels. To see why, we consider the ways in which a transient, destined to be squashed USL (i.e., the *transmitter*) could try to speed up or slow down the execution of a load that later retires (i.e., the *receiver*). We note that if this second load is also squashed, there is no security concern, since a squashed load does not have side effects, and hence does not pose a threat.

Speeding Up The transmitter could attempt to speed up the receiver's execution by accessing the same line as the receiver, so that the latter would get its data with low latency from the SB or LLC-SB entry allocated by the transmitter. For this to happen, the transmitter has to execute before the receiver. In the following, we show that this side channel cannot happen. To see why, consider the two possible cases:

a) *The transmitter comes before the receiver in program order:* In this case, the receiver has to be issued after the transmitter is actually squashed. Otherwise, the receiver would be squashed at the same time as the transmitter is. However, when the transmitter is squashed, its entries in the SB and LLC-SB become unusable by a later request: the SB entry's Valid bit is reset, and the LLC-SB entry's Epoch ID tag becomes stale, as the receiver gets a higher Epoch ID. As a result, the receiver cannot reuse any state left behind by the transmitter.

b) *The transmitter comes after the receiver in program order:* This is the case when, because of out-of-order execution, the transmitter has already requested the line (and maybe even loaded it into its own SB or LLC-SB entries) by the time the receiver requests the line. The receiver could be sped-up only if it could leverage the transmitter's earlier request or buffered data. However, InvisiSpec does not allow a load (USL or otherwise) to reuse any state (e.g., state in an SB entry or in an MHSR entry) allocated by a USL that is later in program order. Instead, the receiver issues its own request to the cache hierarchy (Section 6.2.1) and is unaffected by the transmitter. For the LLC-SB, the only state reuse allowed occurs when a validation/exposure for a load reuses the entry left by the *Spec-GetS* request of the same load.

Slowing Down The transmitter could attempt to slow down the receiver's execution by allocating all the entries in one of the buffers. But recall that the receiver must retire for the slow-down to be observable. Therefore, the receiver must come before the transmitter in program order. However, SB and LLC-SB entries are allocated at issue time, due to their correspondence to LQ entries. Therefore, allocation of SB or LLC-SB entries by the later transmitter cannot affect the allocation ability of the earlier receiver. Finally, contention on other resources, such as MSHRs or execution units, could slow down the receiver, but such side channels are considered out of scope in this work.

6.5 EVALUATION

6.5.1 Experimental Setup

To evaluate InvisiSpec, we modify the Gem5 [129] simulator, which is a cycle-level simulator with support for modeling the side effects of squashed instructions. [1]

We run individual SPECInt2006 and SPECFP2006 applications [109] on a single core, and multi-threaded PARSEC applications [110] on 8 cores. For SPEC, we use the *reference* input size and skip the first 10 billion instructions; then, we simulate for 1 billion instructions. For PARSEC we use the *simmedium* input size and simulate the whole region-of-interest (ROI). Table 6.4 shows the parameters of the simulated architecture. When running a SPEC application, we only enable one bank of the shared cache.

Parameter	**Value**
Architecture	1 core (SPEC) or 8 cores (PARSEC) at 2.0GHz
Core	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

Table 6.4: Parameters of the simulated architecture.

We model the 5 processor configurations shown in Table 6.5: *Base* is a conventional,

[1] The implementation has been open sourced at `https://github.com/mjyan0720/InvisiSpec-1.0`.

insecure processor, *Fe-Sp* inserts a fence after every indirect/conditional branch, *IS-Sp* is InvisiSpec-Spectre, *Fe-Fu* inserts a fence before every load, and *IS-Fu* is InvisiSpec-Future. We model both TSO and RC.

Names		Configurations
Base	UnsafeBaseline	Conventional, insecure baseline processor
Fe-Sp	Fence-Spectre	Insert a fence after every indirect/conditional branch
IS-Sp	InvisiSpec-Spectre	USL modifies only SB, and is made visible after all the preceding branches are resolved
Fe-Fu	Fence-Future	Insert a fence before every load instruction
IS-Fu	InvisiSpec-Future	USL modifies only SB, and is made visible when it is either non-speculative or spec non-squashable

Table 6.5: Simulated processor configurations.

In InvisiSpec-Future, a USL is ready to initiate a validation/exposure when the following is true for all the instructions before the USL in the ROB: (i) they cannot suffer exceptions anymore, (ii) there are no unresolved control-flow instructions, (iii) all stores have retired into the write buffer, (iv) all loads have either finished their validation or initiated their exposure transaction, and (v) all synchronization and fence instructions have completed. At that point, we temporarily disable interrupts and initiate the validation/exposure. These are slightly more conservative conditions than those listed in Table 6.1 for Futuristic.

6.5.2 Proof-of-Concept Defense Analysis

We evaluate the effectiveness of InvisiSpec-Spectre at defending against the attack in Section 6.1.1. We set the secret value stored at address Y to 84. After triggering the misprediction in the victim, the attacker scans array $array2$ and reports the access latency. Figure 6.3 shows the median access latency for each cache line measured by the attacker after 100 times.

From the figure, we see that under *Base*, the attacker can obtain the secret value. Only the access to the line corresponding to the secret value hits in the caches, and takes less than 40 cycles. All the other accesses go to main memory, and take over 150 cycles. However, with *IS-Sp*, the attack is successfully thwarted. All the accesses to all the lines go to main memory because loads in the mispredicted path of a branch do not change the cache state, and the SB does not leak information from squashed loads as discussed in Section 6.4.

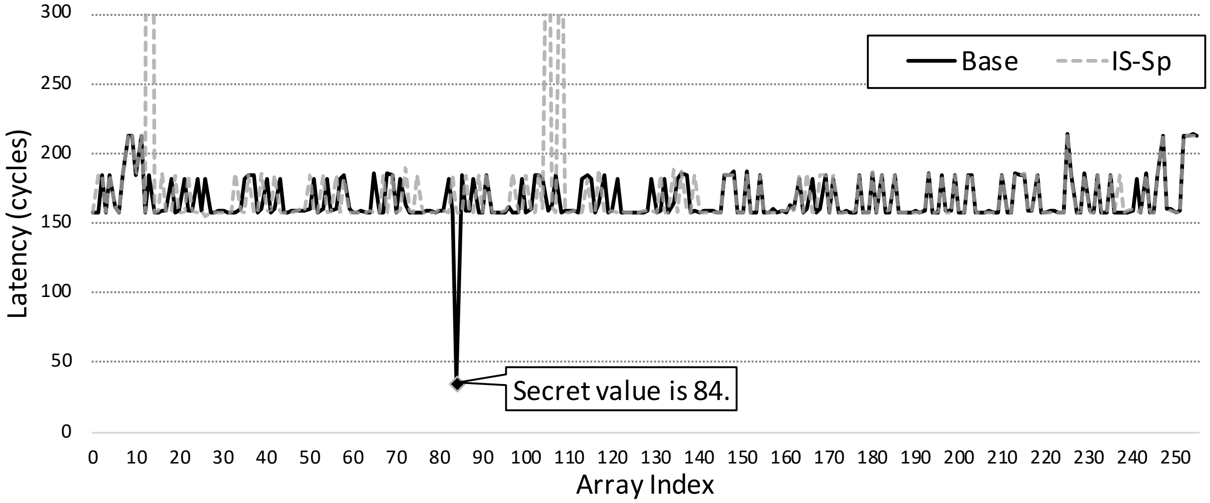


Figure 6.3: Access latency measured in the PoC attack.

6.5.3 Performance Evaluation of the SPEC Applications

Execution Time Figure 6.4 compares the execution time of the SPEC applications on the 5 processor configurations of Table 6.5. From left to right, we show data for each application under TSO, for the average application under TSO, and for the average under RC (*RC-Average*). Each set of bars is normalized to *Base*.

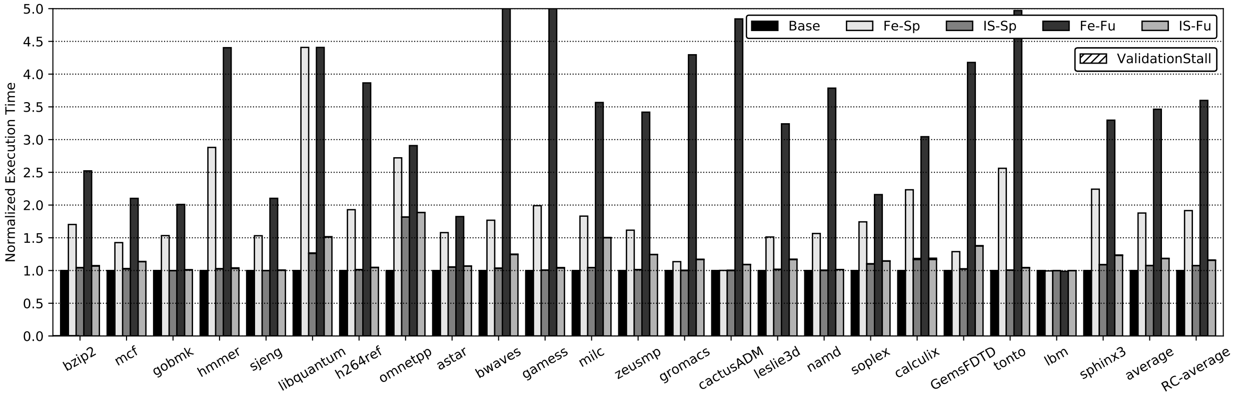


Figure 6.4: Normalized execution time of the SPEC applications.

If we focus on the fence-based solutions, we see that they have high overhead. Under TSO, the average execution time of *Fe-Sp* and *Fe-Fu* is 88% and 246% higher, respectively, than *Base*. The overhead of InvisiSpec is very small. Under TSO, the average execution time of *IS-Sp* and *IS-Fu* is 7.6% and 18.2% higher, respectively, than *Base*.

There are three main reasons for the slowdowns of *IS-Sp* and *IS-Fu*: validation stalls, TLB miss stalls, and contention due to two accesses per load. We consider each of them in turn.

For *IS-Sp* and *IS-Fu*, the figure shows the contribution of the stall caused by validation operations. However, such stalls are so small that they cannot be seen in the figure. The reason is that most of the validations hit in the L1 cache and are served quickly.

In *IS-Sp*, the application with the highest execution overhead is *omnetpp* (around 80%). The main reason is that *omnetpp* suffers many TLB misses. In *Base*, TLB misses are not delayed, as they are served speculatively. In contrast, in *IS-Sp* and *IS-Fu*, TLB misses are not served until the corresponding USL reaches its visibility point.

In some applications, a high rate of cache-missing USLs, many inducing two cache hierarchy accesses, causes contention in the cache hierarchy and slows down execution. Specifically, in *IS-Fu*, applications with a high number of L1 misses per kilo-instruction, such as *libquantum* and *GemsFDTD*, have some of the highest execution time increase (around 40-50%). These applications have about 30 L1 misses per kilo-instruction.

In RC, the trends are similar. The average execution time in *IS-Sp* and *IS-Fu* is 8.2% and 16.8% higher than in *Base*.

Network Traffic We record the network traffic, measured as the total number of bytes transmitted between caches, and between cache and main memory. Figure 6.5 compares the traffic in the 5 processor configurations of Table 6.5. The bars are organized as in Figure 6.4. For *IS-Sp* and *IS-Fu*, we show the fraction of the traffic induced by USLs (*SpecLoad*) and exposures/validations. The rest of the bar is due to non-speculative accesses.

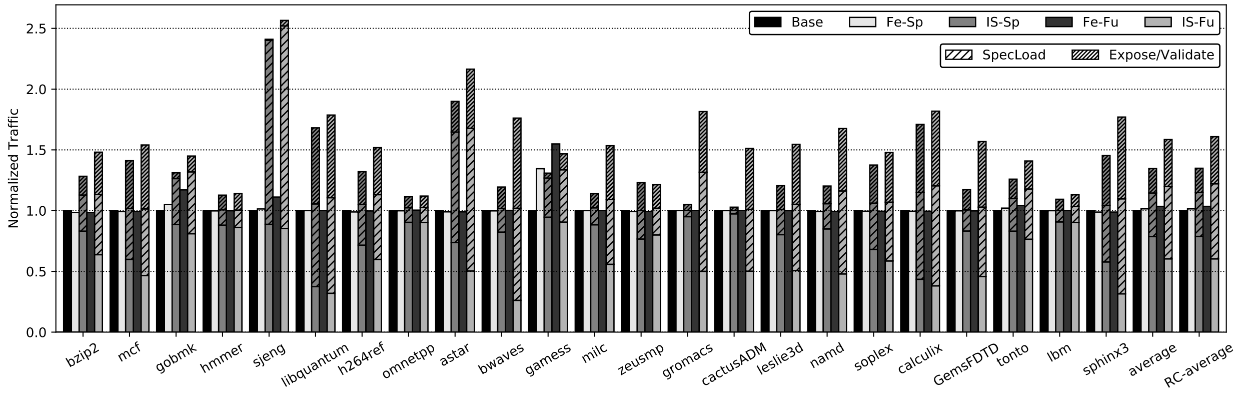


Figure 6.5: Normalized network traffic of the SPEC applications.

Under TSO, the average network traffic in *IS-Sp* and *IS-Fu* is 35% and 59% higher, respectively, than *Base*. The traffic increase is about the same under RC.

The higher traffic in *IS-Sp* and *IS-Fu* is due to the combination of the impact of the USLs (*SpecLoad*) and of the exposures/validations. In some applications such as *sjeng*, the traffic without exposures/validations is already substantially higher than *Base*. The reason is that these applications have a high branch misprediction rate, and a given load may be squashed multiple times in a row — if it is in the shadow of other mispredicted branches. In *IS-Sp* and *IS-Fu*, every time that a given USL is re-issued to the same address after its squash, it creates traffic; in *Base*, re-issuing a load to the same address after its squash creates no network traffic. This is the reason some applications have a large *SpecLoad* category.

In applications with a large *SpecLoad* category, the *Expose/Validate* category is usually small. The reason is that, by the time a USL is squashed for its last time, it may be reissued as a non-speculative load, and not require exposure/validation.

On average, the traffic in *Fe-Sp* and *Fe-Fu* is like in *Base*. Intuitively, it should be lower than *Base* because *Fe-Sp* and *Fe-Fu* do not execute speculative instructions. In reality, it can be shown that, while the data traffic is lower, the instruction traffic is higher, and the two effects cancel out. The reason is that our simulation system still *fetches* speculative instructions (since this paper only focuses on the data cache hierarchy), without executing them. Fences in the pipeline cause a larger volume of instructions fetched in mispredicted branch paths.

6.5.4 Performance Evaluation of the PARSEC Applications

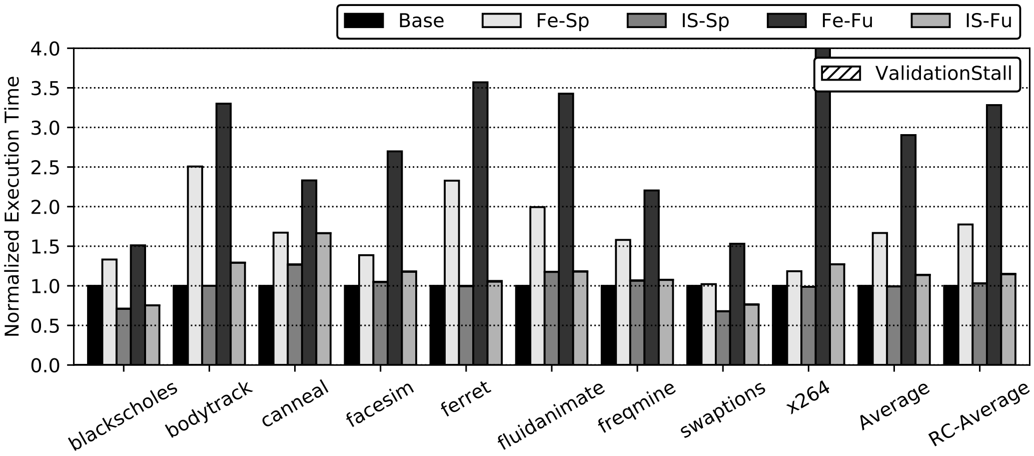


Figure 6.6: Normalized execution time of the PARSEC applications.

Execution Time Figure 6.6 shows the normalized execution time of the PARSEC applications on our 5 processor configurations. The figure is organized as in Figure 6.4. We see that, under TSO, *IS-Sp* decreases the average execution time over *Base* by 0.8%, and *IS-Fu* increases it by 13.7%. Under RC, *IS-Sp* and *IS-Fu* increase the average execution time over *Base* by 3.0% and 14.8%, respectively. These slowdowns are smaller than in the SPEC programs.

From Figure 6.6, we again see that validation stalls in *IS-Sp* and *IS-Fu* are negligible. The main reason for the slowdowns in most applications is the resource contention caused by two accesses per speculative load. However, among all of the applications, *blackscholes* and *swaptions* are faster under *IS-Sp* and *IS-Fu* than under *Base*. This effect occurs because of a reduction in the number of pipeline squashes due to the eviction of cache lines from the L1 cache. Specifically, in *Base*, we model existing processors, which conservatively squash any in-flight load upon receiving an invalidation message for the line read by the load, or upon evicting the line read by the load from the L1 cache. In contrast, our InvisiSpec design does not squash the pipeline on reception of an invalidation for the line read by a load marked as only needing *exposure*, or on eviction from the L1 of the line read by a load marked as only needing *exposure*.

This is a correct implementation that saves InvisiSpec many squashes. For example, *blackscholes* suffers from more than 2M squashes due to L1 evictions in *Base*, while few occur in *IS-Sp* and *IS-Fu*. *swaptions* has more than 5 times more squashes due to L1 evictions in *Base* than in *IS-Sp* and *IS-Fu*. Excluding *blackscholes* and *swaptions*, under TSO, *IS-Sp* and *IS-Fu* increase the average execution time of the applications by 8% and 24%, respectively, over *Base*.

We also see that *Fe-Sp* and *Fe-Fu* increase the average execution time over *Base* by a substantial 67% and 190%, respectively, under TSO. Higher numbers are seen under RC.

Network Traffic Figure 6.7 shows the normalized traffic of the PARSEC applications in our 5 configurations. The figure is organized as usual. The *IS-Sp* and *IS-Fu* bars are broken down into traffic induced by USLs (*SpecLoad*), by exposures/validations, and by non-speculative accesses (the rest of the bar).

On average, *Fe-Sp* and *Fe-Fu* have less traffic than *Base*. The main reason is that they do not execute speculative instructions. Under TSO, *IS-Sp* and *IS-Fu* increase the traffic by an average of 13% and 33%, respectively, over *Base*. Slightly higher numbers are observed under RC.

Generally, *IS-Sp* and *IS-Fu* have higher traffic than *Base*. However, in two applications—*blackscholes* and *swaptions*—the traffic is lower in the InvisiSpec-based configurations (and in the fence-based ones) than in *Base*. The reason is that, as mentioned before, these two applications have more pipeline squashes due to L1 evictions in *Base*. This leads to more memory accesses being re-executed, resulting in higher traffic.

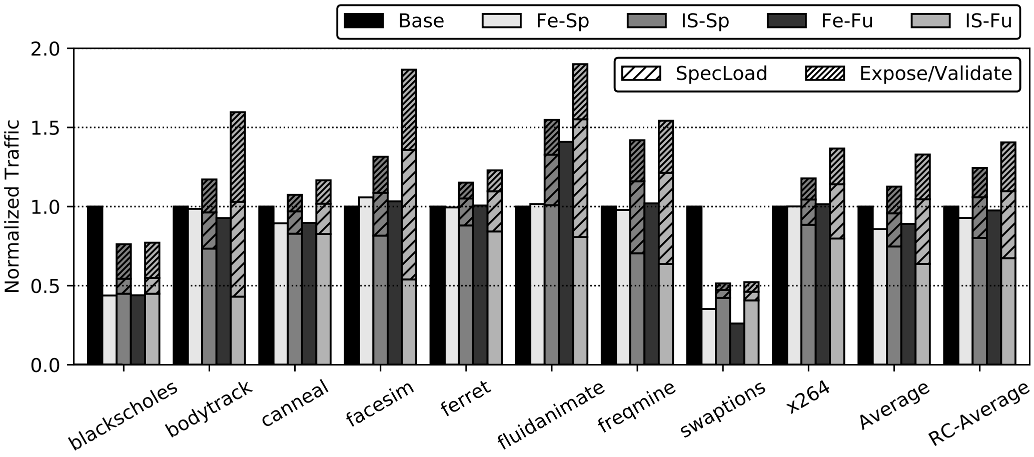


Figure 6.7: Normalized network traffic of the PARSEC applications.

6.5.5 Characterization of InvisiSpec's Operation

	Application Name	Exposures and Validations						L1-SB Hit Rate (%)		LLC-SB Hit Rate (%)	
		% Exposures		% L1 Hit Validations		% L1 Miss Validations					
		Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu
SPEC	sjeng	26.6	37.5	68.9	62.5	4.4	0.0	0.0	0.0	100.0	99.7
	libquantum	13.9	13.4	0.1	0.0	86.0	86.6	5.6	5.4	100.0	100.0
	omnetpp	43.9	44.9	47.0	46.9	9.1	8.2	4.8	4.4	100.0	100.0
	Average	14.7	15.7	71.3	73.3	14.0	11.0	0.6	1.9	99.9	99.8
PARSEC	bodytrack	4.2	3.3	87.3	90.5	8.5	6.2	0.5	0.8	99.7	99.4
	fluidanimate	6.4	6.7	92.9	92.4	0.7	0.9	0.1	3.5	100.0	99.9
	swaptions	4.5	5.6	88.0	92.3	7.5	2.1	3.5	0.8	99.6	98.8
	Average	9.0	6.8	83.7	87.8	7.3	5.4	1.5	2.4	99.3	99.5

Table 6.6: Characterization of InvisiSpec's operation under TSO. Sp and Fu stand for *IS-Sp* and *IS-Fu*.

Table 6.6 shows some statistics related to InvisiSpec's operation under TSO. It shows data on 3 SPEC applications, the average of SPEC, 3 PARSEC applications, and the average of PARSEC. Columns 3-8 break down the total number of exposures and validations into exposures, validations that hit in the L1 cache, and validations that miss in the L1 cache. We see that a modest fraction of the transactions are exposures (e.g., 16% in SPEC and 7% in PARSEC for *IS-Fu*). Further, a big fraction are validations that hit in L1 (e.g., 73% in SPEC and 88% in PARSEC for *IS-Fu*). This is why overall validation stall is negligible.

Columns 9-12 show the hit rates in the L1-SB, and in the LLC-SB. The LLC-SB hit rates only apply to validations and exposures. On average, L1-SB hit rates are very low (1.9%

in SPEC and 2.4% in PARSEC for *IS-Fu*), while LLC-SB hit rates are very high (99.8% in SPEC and 99.5% in PARSEC for *IS-Fu*). High LLC-SB hit rates boost performance.

	Application Name	# Squashes Per 1M Instructions		Reason for Squash (%)					
				Branch Misprediction		Consistency Violation		Validation Failure	
		Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu
SPEC	sjeng	73,752	73,996	100.0	100.0	0.0	0.0	0.0	0.0
	libquantum	0	0	100.0	100.0	0.0	0.0	0.0	0.0
	omnetpp	15,890	16,139	100.0	100.0	0.0	0.0	0.0	0.0
	Average	14,816	14,905	97.1	97.4	2.9	2.6	0.0	0.0
PARSEC	bodytrack	1,974	1,735	95.4	94.3	4.6	5.7	0.0	0.0
	fluidanimate	4,961	4,983	99.8	99.9	0.2	0.1	0.0	0.00
	swaptions	4,556	4,724	61.8	62.4	38.2	37.6	0.0	0.0
	Average	4,527	4,524	87.7	87.7	12.3	12.3	0.0	0.0

Table 6.7: Characterization of InvisiSpec's execution squashes under TSO. Sp and Fu stand for *IS-Sp* and *IS-Fu*.

Table 6.7 shows statistics related to the squash events under *IS-Sp* and *IS-Fu* under TSO. It shows data for the same SPEC and PARSEC applications as in Table 6.6. Columns 3-4 show the number of squashes per 1M instructions. On average, PARSEC applications have a lower squash rate than SPEC applications. Columns 5-10 break these events into the reason for the squash: branch misprediction, consistency violation, and validation failure. The large majority are caused by branch mispredictions (e.g., 97% in SPEC and 88% in PARSEC for *IS-Fu*), and only a few by consistency violations (e.g., 2.6% in SPEC and 12% in PARSEC for *IS-Fu*). There are practically no validation failures.

To give an idea of the range of values observed, the table also shows data for a few individual applications. Finally, we collected the same statistics under RC. Under RC, there are practically no validations (i.e., practically all are exposures). Further, there are very few consistency violations, and practically all the squashes are due to branch mispredictions.

6.5.6 Estimating Hardware Overhead

InvisiSpec adds two main per-core structures, namely, the L1-SB (a cache) and the LLC-SB (a CAM). We use CACTI 5 [141] to estimate, at 16nm, their area, access time, dynamic read and write energies, and leakage power. These estimates, shown in Table 6.8, do not include their random logic. Overall, these structures add modest overhead.

Metric	L1-SB	LLC-SB
Area (mm2)	0.0174	0.0176
Access time (ps)	97.1	97.1
Dynamic read energy (pJ)	4.4	4.4
Dynamic write energy (pJ)	4.3	4.3
Leakage power (mW)	0.56	0.61

Table 6.8: Per-core hardware overhead of InvisiSpec.

6.6 CONCLUSION

This chapter presented InvisiSpec, a novel approach to defend against hardware transient execution attacks in multiprocessors by making speculation invisible in the data cache hierarchy. In InvisiSpec, unsafe speculative loads read data into a speculative buffer, without modifying the cache hierarchy. When the loads are safe, they are made visible to the rest of the system through validations or exposures. We proposed an InvisiSpec design to defend against Spectre-like attacks, and one to defend against futuristic attacks where any speculative load may pose a threat. Our evaluation showed that, under TSO, using fences to defend against Spectre attacks slows down execution by 82% relative to a conventional, insecure processor; InvisiSpec reduces the execution slowdown to only 5%. Using fences to defend against futuristic attacks slows down execution by 231%; InvisiSpec reduces the slowdown to only 17%.

CHAPTER 7: CONCLUSION

This thesis studied cache-based side channel attacks. We proposed secure hardware processor designs to effectively and efficiently address the side channel vulnerabilities of modern cache hierarchies. This thesis made the following contributions, which help the community to better understand the threats.

First, we studied the role of cache interference in different cache attacks and proposed effective solutions to mitigate shared cache attacks by limiting malicious interference. The thesis provided an important insight: the key in active cache attacks is creating inclusion victims, since they give an attacker visibility into a victim's private cache. We used this insight to design an effective and practical defense mechanism (SHARP) for inclusive cache hierarchies. In addition, we showed that inclusion victims also exist in non-inclusive cache hierarchies and that the non-inclusive property is insufficient to stave off cache-based side channel attacks. We showed that the directory structure is a unified attack surface for all types of cache hierarchies. Moreover, we proposed the first scalable secure directory design (SecDir) to address the directory side channel vulnerability.

Second, we studied how to effectively defend against transient execution attacks in cache hierarchies. This thesis developed a comprehensive attack model—the Futuristic attack model, which considers attacks beyond Spectre and Meltdown. We then proposed "Invisible Speculation" (InvisiSpec), the first robust hardware defense mechanism against transient execution attacks in multiprocessor cache hierarchies. InvisiSpec is a novel and effective strategy for multiprocessors. It protects against cache-based attacks triggered by any speculative load. The lessons learned from the design have inspired our recent work, Speculative Taint Tracking [93], and will influence more future work on designing comprehensive defense solutions.

Moving forward, we would like to ask whether micro-architecture side channels are a solved problem. The answer is obviously no. We are now entering a new era of computer architecture when security and trustworthiness play an increasingly important role. It is insufficient to merely patch existing systems for known attacks. We need to redesign processor hardware and systems to defeat future attacks. There are a lot of research opportunities and challenges ahead. We conclude by discussing three open challenges.

First, while this thesis did a thorough study on cross-core cache attacks, there exist other attack contexts, which require different security and performance considerations. The majority of side channel vulnerabilities stem from the sharing of system resources. It has been shown that same-core attacks and even same-thread attacks are feasible in Javascript [3, 41] and SGX [142,143] environments. In these attack contexts, the attacker and the victim share more system resources, including the private caches and pipeline structures. For example, prior mitigation techniques which leverage cache partitioning or randomization have to be redesigned to achieve satisfactory security and performance tradeoffs. Overall, how to design holistic defense solutions is still an open research problem.

Second, even though this thesis proposed hardware-only defense mechanisms, it is worthwhile to think about how software and hardware co-designs can help address side channel vulnerabilities. For example, many hardware-only defense mechanisms such as SHARP are ineffective in passive cache attacks, where the attacker exploits the victim's self-conflicts or self-reuses of cache lines. To address this problem, we could redesign the software and hardware interfaces to provide programmers easier ways to control cache interference. As another example, when defending against transient execution attacks, InvisiSpec considers that all speculative loads are vulnerable, and that their side effects need to be hidden. It is promising to leverage software analysis to identify safe loads in advance, and selectively enable InvisiSpec to reduce its performance overhead.

Third, we note the lack of effective and practical methodologies to verify security properties [144] of hardware designs. Currently, we have to manually analyze security properties of hardware defense mechanisms based on our understanding of the existing attacks. Since the attack strategies always evolve, the security goals become moving targets. Instead of using a case-by-case analysis, it is necessary to pursue formal analysis. One possible direction is to use information flow tracking [145, 146] to enforce the non-interference property between different security domains. However, in practice, such a strong security property may only be achievable at the cost of performance and programmability. Another direction is to design effective tools to quantify information leakage if non-interference cannot be attained.

APPENDIX A: CACHE TELEPATHY

This chapter presents another example of cache-based side channel attacks that targets Deep Neural Networks (DNNs). A DNN's architecture (i.e., its hyper-parameters) broadly determines the DNN's accuracy and performance, and is often confidential. We propose Cache Telepathy, an efficient mechanism to help obtain a DNN's architecture using the cache side channel. We show that our attack is effective at helping obtain the architecture by very substantially reducing the search space of target DNN architectures.

For the past several years, Deep Neural Networks (DNNs) have increased in popularity thanks to their ability to attain high accuracy and performance in a multitude of machine learning tasks — e.g., image and speech recognition [147, 148], scene generation [149], and game playing [150]. An emerging framework that provides end-to-end infrastructure for using DNNs is Machine Learning as a Service (MLaaS) [151,152]. In MLaaS, trusted clients submit DNNs or training data to MLaaS service providers (e.g., an Amazon or Google datacenter). Service providers host the DNNs, and allow remote *untrusted* users to submit queries to the DNNs for a fee.

Despite its promise, MLaaS provides new ways to undermine the privacy of the hosted DNNs. An adversary may be able to learn details of the hosted DNNs beyond the official query APIs. For example, an adversary may try to learn the DNN's architecture (i.e., its *hyper-parameters*). These are the parameters that give the network its shape, such as the number and types of layers, the number of neurons per layer, and the connections between layers.

The architecture of a DNN broadly determines the DNN's accuracy and performance. For this reason, obtaining it often has high commercial value. Furthermore, once a DNN's architecture is known, other attacks are possible, such as the model extraction attack [153] (which obtains the weights of the DNN's edges), and the membership inference attack [154, 155] (which determines whether an input was used to train the DNN).

Yet, stealing a DNN's architecture is challenging. DNNs have a multitude of hyper-parameters, which makes brute-force guesswork unfeasible. Moreover, the DNN design space has been growing with time, which is further aggravating the adversary's task.

This chapter demonstrates that despite the large search space, attackers can quickly reduce the search space of DNN architectures in the MLaaS setting using the cache side channel. Our insight is that DNN inference relies heavily on tiled GEMM (Generalized Matrix Multiply), and that DNN architecture parameters determine the number of GEMM calls and the dimensions of the matrices used in the GEMM functions. Such information can be leaked through the cache side channel.

We present an attack that we call *Cache Telepathy*. It is the first cache side channel attack targeting modern DNNs on general-purpose processors (CPUs). The reason for targeting CPUs is that CPUs are widely used for DNN inference in existing MLaaS platforms, such as Facebook's [156] and Amazon's [157].

We demonstrate our attack by implementing it on a state-of-the-art platform. We use Prime+Probe and Flush+Reload to attack the VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective at helping obtain the architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

This chapter makes the following contributions:

- It provides a detailed analysis of the mapping of DNN hyper-parameters to the number of GEMM calls and their arguments.
- It implements the first cache-based side channel attack to extract DNN architectures on general purpose processors.
- It evaluates the attack on VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries.

A.1 BACKGROUND ON DNN AND DNN PRIVACY ATTACKS

A.1.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of Machine Learning (ML) algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation [158]. There are several major types of DNNs in use today, two popular types being fully-connected neural networks (or multi-layer perceptrons) and Convolutional Neural Networks (CNNs).

DNN Architecture The *architecture* of a DNN, also called the *hyper-parameters*, gives the network its shape. DNN hyper-parameters considered in this chapter are:

a) Total number of layers.

b) Layer types, such as fully-connected, convolutional, or pooling layer.

c) Connections between layers, including sequential and non-sequential connections such as shortcuts. Non-sequential connections exist in recent DNNs, such as ResNet [148]. For example, instead of directly using the output from a prior layer as the input to a later layer, a shortcut involves summing up the outputs of two prior layers and using the result as the input for a later layer.

d) Hyper-parameters for each layer. For a fully-connected layer, this is the number of neurons in that layer. For a convolutional layer, this is the number of filters, the filter size, and the stride size.

e) The activation function in each layer, e.g., `relu` and `sigmoid`.

DNN Weights The computation in each DNN layer involves many multiply-accumulate operations (MACCs) on input neurons. The DNN *weights*, also called *parameters*, specify operands to these multiply-accumulate operations. In a fully-connected layer, each edge out of a neuron is a MACC with a weight; in a convolutional layer, each filter is a multi-dimensional array of weights, which is used as a sliding window that computes dot products over input neurons.

DNN Usage DNNs usage has two distinct phases: training and inference. In training, the DNN designer starts with a network architecture and a training set of labeled inputs, and tries to find the DNN weights to minimize mis-prediction error. Training is generally performed offline on GPUs and takes a relatively long time to finish, typically hours or days [156, 159]. In inference, the trained model is deployed and used to make real-time predictions on new inputs. For good responsiveness, inference is generally performed on CPUs [156,157].

A.1.2 Prior Privacy Attacks Need the DNN Architecture

To gain insight into the importance of DNN architectures, we discuss prior DNN privacy attacks [153–155,160]. There are three types of such attacks, each with a different goal. All of them require knowing the victim's DNN architecture. In the following, we refer to the victim's network as the oracle network, its architecture as the oracle DNN architecture, and its training data set as the oracle training data set.

In the *model extraction attack* [153], the attacker tries to obtain a network that is close enough to the oracle network. It assumes that the attacker knows the oracle DNN architecture at the start, and tries to estimate the weights of the oracle network. The attacker creates a synthetic data set, requests the classification results from the oracle network, and uses such results to train a network that uses the oracle architecture.

The *membership inference attack* [154, 155] aims to infer the composition of the oracle training data set, which is expressed as the probability of whether a data sample exists in the training set or not. This attack also requires knowledge of the oracle DNN architecture. Attackers create multiple synthetic data sets and train multiple networks that use the oracle architecture. Then, they run the inference algorithm on these networks with some inputs in their training sets and some not in their training sets. They then compare the results to find the patterns in the output of the data in the training sets. The pattern information is used to infer the composition of the oracle training set. Specifically, given a data sample, they run the inference algorithm of the oracle network, obtain the output and check whether the output matches the pattern obtained before. The more the output matches the pattern, the more likely the data sample exists in the oracle training set.

The *hyper-parameter stealing attack* [160] steals the loss function and regularization term used in ML algorithms, including DNN training and inference. This attack also relies on knowing the oracle DNN architecture. During the attack, attackers leverage the model extraction attack to learn the DNN's weights. They then find the loss function that minimizes the training misprediction error.

A.2 THREAT MODEL

This chapter develops a cache-timing attack that quickly reduces the search space of DNN architectures. The attack relies on the following standard assumptions.

Black-box Access. We follow a black-box threat model in an MLaaS setting similar to [153]. In a black-box attack, the DNN model is only accessible to attackers via an official query interface. Attackers do not have prior knowledge about the target DNN, including its hyper-parameters, weights and training data.

Co-location. We assume that the attacker process can use techniques from prior work [16, 39, 161–165] to co-locate onto the same processor chip as the victim process running DNN inference. This is feasible, as current MLaaS jobs are deployed on shared clouds. Note that recent MLaaS, such as Amazon SageMaker [166] and Google ML Engine [167] allow users to upload their own code for training and inference, instead of using pre-defined APIs. In this case, attackers can disguise themselves as an MLaaS process and the cloud scheduler will have difficulty in separating attacker processes from victim processes.

Code Analysis. We also assume that the attacker can analyze the ML framework code and linear algebra libraries used by the victim. These are realistic assumptions. First, open-source ML frameworks are widely used for efficient development of ML applications. The frameworks supported by Google, Amazon and other companies, including Tensorflow [168], Caffe [169], and MXNet [170] are all public. Our analysis is applicable to almost all of these frameworks. Second, the frameworks' backends are all supported by high-performance and popular linear algebra libraries, such as OpenBLAS [171], Eigen [172] and MKL [173]. OpenBLAS and Eigen are open sourced, and MKL can be reverse engineered, as we show in Section A.6.

A.3 ATTACK OVERVIEW

The goal of Cache Telepathy is to substantially reduce the search space of target DNN architectures. In this section, we first discuss how our attack can assist other DNN privacy attacks, and then give an overview of the Cache Telepathy attack procedure.

Cache Telepathy's Role in Existing DNN Attacks In settings where DNN architectures are not known, our attack can serve as an essential initial step for many existing DNN privacy attacks, including model extraction attacks [153] and membership inference attacks [154].

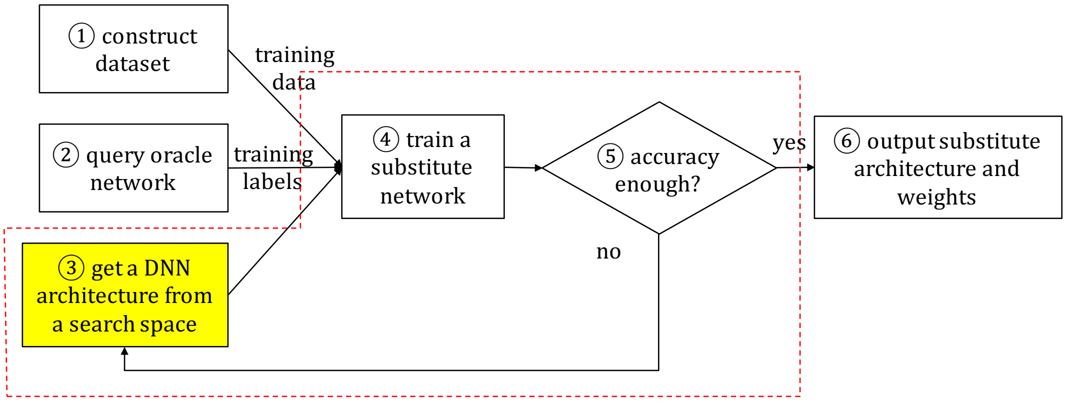

① construct dataset
training data
② query oracle network
training labels
③ get a DNN architecture from a search space
④ train a substitute network
⑤ accuracy enough?
yes
no
⑥ output substitute architecture and weights

Figure A.1: Cache Telepathy assists model extraction attacks.

Figure A.1 demonstrates how Cache Telepathy makes the model extraction attack feasible. The final goal of the model extraction attack is to obtain a network that is close enough to the oracle network (Section A.1.2). The attack uses the following steps. First, the attacker generates a synthetic training data set (①). This step can be achieved using a random feature vector method [153] or more sophisticated techniques, such as hill-climbing [154]. Next, the attacker queries the oracle network via inference APIs provided by MLaaS providers to get labels or confidence values (②). The synthetic data set and corresponding query results will be used as training data and labels later. In the case that the oracle architecture is not known, the attacker needs to choose a DNN architecture from a search space (③) and then train a network with the chosen architecture (④). Steps ③-④ repeat until a network is found with sufficient prediction accuracy (⑤).

This attack process is extremely compute intensive, since it involves many iterations of step ④. Considering the depth and complexity of state-of-the-art DNNs, training and validating each network can take hours to days. Moreover, without any information about the architecture, the search space of possible architectures is often intractable, and thus, the model extraction attack is infeasible. However, Cache Telepathy can reduce the architecture search space (③) to a tractable size and make the attack feasible in settings where DNN architectures are unknown.

Membership inference attacks suffer from a more serious problem if the DNN architecture is not known. Recall that the attack aims to figure out the composition of the oracle training data set (Section A.1.2). If there are many different candidate architectures, the attacker needs to consider the results generated by all the candidate architectures and statistically summarize inconsistent results from those architectures. A large search space of candidate architectures, not only significantly increases the computation requirements, but also potentially hurts attack accuracy. Consider a candidate architecture which is very different from the oracle architecture. It is likely to contribute incorrect results, and in turn, decrease the attack accuracy. However, Cache Telepathy can reduce the search space to a reasonable size. Moreover, the candidate architectures in the reduced search space have the same or very similar hyper-parameters as the oracle network. Therefore, they perform very similarly to the oracle network on various data sets. Hence, our attack also plays an important role in membership inference attacks.

Overall Cache Telepathy Attack Procedure Our attack is based on two observations. First, DNN inference relies heavily on GEMM (Generalized Matrix Multiply). We conduct a detailed analysis of how GEMM is used in ML frameworks, and figure out the mapping between DNN hyper-parameters and matrix parameters (Section A.4). Second, high-performance GEMM algorithms are vulnerable to cache-based side channel attacks, as they are all tuned for the cache hierarchy through matrix blocking (i.e., tiling). When the block size is public (or can be easily deduced), the attacker can use the cache side channel to count blocks and learn the matrix sizes.

The Cache Telepathy attack procedure includes a cache attack and post processing steps. First, it uses a cache attack to monitor matrix multiplications and obtain matrix parameters (Sections A.5 and A.6). Then, the DNN architecture is reverse-engineered based on the mapping between DNN hyper-parameters and matrix parameters (Section A.4). Finally, Cache Telepathy prunes the possible values of the remaining undiscovered hyper-parameters and generates a pruned search space for the target DNN architecture (Section A.8.3). We consider the attack to be successful if we can generate a reasonable number of candidate architectures whose hyper-parameters are the same or very similar to the oracle network.

A.4 MAPPING DNNS TO MATRIX PARAMETERS

DNN hyper-parameters, listed in Section A.1.1, can be mapped to GEMM execution. We first discuss how the layer type and configurations within each layer map to matrix parameters, assuming that all layers are sequentially connected (Section A.4.1 and A.4.2). We then generalize the mapping by showing how the connections between layers map to GEMM execution (Section A.4.3). Finally, we discuss what information is required to extract the activation functions of Section A.1.1 (Section A.4.4).

A.4.1 Analysis of DNN Layers

There are two types of neural network layers whose computation can be mapped to matrix multiplications, namely fully-connected and convolutional layers.

Fully-connected Layer

In a fully-connected layer, each neuron computes a weighted sum of values from all the neurons in the previous layer, followed by a non-linear transformation. The ith layer computes $out_i = f_i(in_i \otimes \theta_i)$ where in_i is the input vector, θ_i is the weight matrix, $\otimes$ denotes a matrix-vector operation, f is an element-wise non-linear function such as tanh or sigmoid, and out_i is the resulting output vector.

The feed-forward computation of a fully-connected DNN can be performed over a batch of a few inputs at a time (B). These multiple input vectors are stacked into an input matrix In_i. A matrix multiplication between the input matrix and the weight matrix (θ_i) produces an output matrix, which is a stack of output vectors. We represent the computation as $O_i = f_i(In_i \cdot \theta_i)$ where In_i is a matrix with as many rows as B and as many columns as N_i (the number of neurons in the layer i); O_i is a matrix with as many rows as B and as many columns as N_{i+1} (the number of neurons in the layer $i+1$); and θ_i is a matrix with N_i rows and N_{i+1} columns. Table A.1 shows the number of rows and columns of all the matrices.

Matrix	**n_row**	**n_col**
Input: In_i	B	N_i
Weight: θ_i	N_i	N_{i+1}
Output: O_i	B	N_{i+1}

Table A.1: Matrix sizes in a fully-connected layer.

Convolutional Layer

In a convolutional layer, a neuron is connected to only a spatial region of neurons in the previous layer. Consider the upper row of Figure A.2, which shows the computation in the ith layer. The layer generates an output out_i (right part of the upper row) by performing convolution operations on an input in_i (center of the upper row) with multiple filters (left part of the upper row). The input volume in_i is of size $W_i \times H_i \times D_i$, where the depth ($D_i$) also refers to the number of channels of the input. Each filter is of size $R_i \times R_i \times D_i$.

To see how a convolution operation is performed, the figure highlights the process of generating one output neuron in out_i. The neuron is a result of a convolution operation – an elementwise dot product of the filter shaded in dots and the subvolume in in_i shaded in dashes. Both the subvolume and the filter have dimensions $R_i \times R_i \times D_i$. Applying one filter on the entire input volume (in_i) generates one channel of the output (out_i). Thus, the number of filters in layer i (D_{i+1}) is the number of channels (depth) in the output volume.

The lower row of Figure A.2 shows a common implementation that transforms the multiple convolution operations in a layer into a single matrix multiply. First, as shown in arrow ①, each filter is stretched out into a row to form a matrix F_i'. The number of rows in F_i' is the number of filters in the layer.

Second, as shown in arrow ②, each subvolume in the input volume is stretched out into a column. The number of elements in the column is $D_i \times R_i^2$. For an input volume with dimensions $W_i \times H_i \times D_i$, there are $(W_i - R_i + P_i)(H_i - R_i + P_i)$ such columns in total, where P_i is the amount of zero padding. We call this transformed input matrix in_i'. Then, the convolution becomes a matrix multiply: $out_i' = F_i' \cdot in_i'$ (③).

Finally, the out_i' matrix is reshaped back to its proper dimensions of the out_i volume (arrow ④). Each row of the resulting out_i' matrix corresponds to one channel in the out_i volume. The number of columns of the out_i' matrix is $(W_i - R_i + P_i)(H_i - R_i + P_i)$, which is

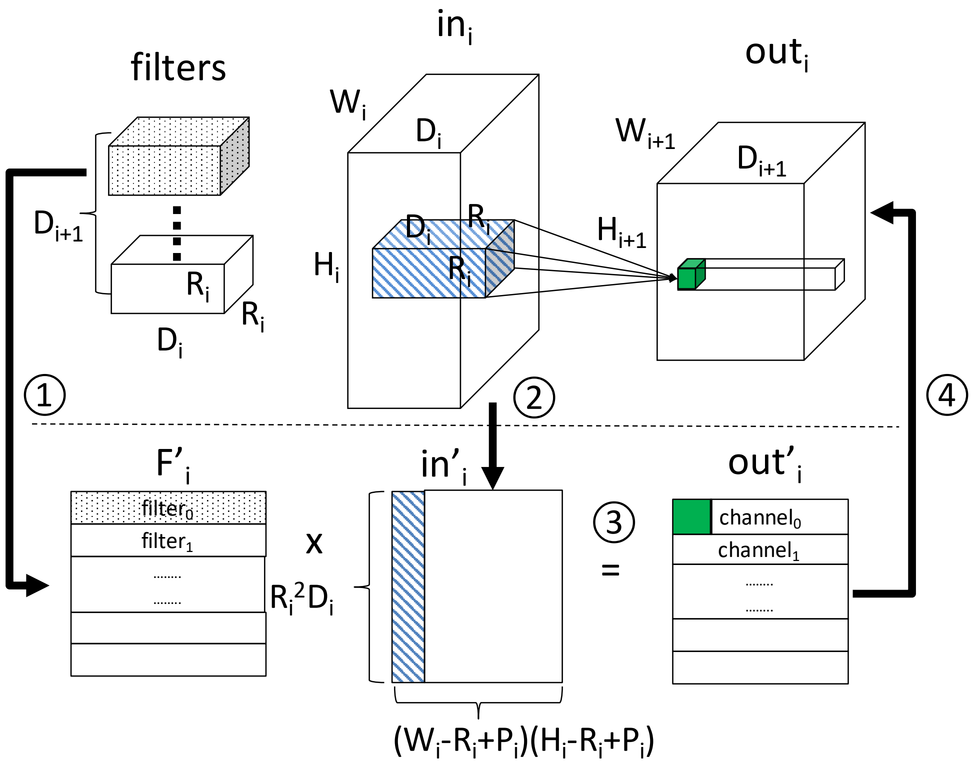


Figure A.2: Mapping a convolutional layer (upper part of the figure) to a matrix multiplication (lower part).

the size of one output channel, namely, $W_{i+1} \times H_{i+1}$. Table A.2 shows the number of rows and columns of the matrices involved.

Matrix	**n_row**	**n_col**
in'_i	$D_i \times R_i^2$	$(W_i - R_i + P_i)(H_i - R_i + P_i)$
F'_i	D_{i+1}	$D_i \times R_i^2$
out'_i	D_{i+1}	$(W_i - R_i + P_i)(H_i - R_i + P_i) = W_{i+1} \times H_{i+1}$

Table A.2: Matrix sizes in a convolutional layer.

The matrix multiplication described above processes a single input. As with fully-connected DNNs, CNN inference can consume a batch of B inputs in a single forward pass. In this case, a convolutional layer performs B matrix multiplications per pass. This is different from fully-connected layers, where the entire batch is computed using only one matrix multiplication.

A.4.2 Resolving DNN Hyper-parameters

Based on the previous analysis, we can now map DNN hyper-parameters to matrix operation parameters assuming all layers are sequentially connected.

Fully-connected Networks

Consider a fully-connected network. Its hyper-parameters are the number of layers, the number of neurons in each layer (N_i) and the activation function per layer. As discussed in Section A.4.1, the feed-forward computation performs one matrix multiplication per layer. Hence, we extract the number of layers by counting the number of matrix multiplications performed. Moreover, according to Table A.1, the number of neurons in layer i (N_i) is the number of rows of the layer's weight matrix (θ_i). The first two rows of Table A.3 summarize this information.

Structure	**Hyper-Parameter**	**Value**
FC network	# of layers	# of matrix muls
FC layer$_i$	N_i: # of neurons	$n\_row(\theta_i)$
Conv network	# of Conv layers	# of matrix muls / B
Conv layer$_i$	D_{i+1}: # of filters	$n\_row(F'_i)$
	R_i: filter width and height[1]	$\sqrt{\frac{n\_row(in'_i)}{n\_row(out'_{i-1})}}$
	P_i: padding	difference between: $n\_col(out'_{i-1}), n\_col(in'_i)$
Pool$_i$ or Stride$_{i+1}$	pool or stride width and height	$\approx \sqrt{\frac{n\_col(out'_i)}{n\_col(in'_{i+1})}}$

Table A.3: Mapping between DNN hyper-parameters and matrix parameters. FC stands for fully connected.

Convolutional Networks

A convolutional network generally consists of four types of layers: convolutional, Relu, pooling, and fully connected. Recall that each convolutional layer involves a batch B of matrix multiplications. Moreover, the B matrix multiplications that correspond to the same layer, always have the same dimension sizes and are executed consecutively. Therefore, we

[1] Specifically, we learn the filter spatial dimensions. If the filter is not square, the search space grows depending on factor combinations (e.g., 2 by 4 looks the same as 1 by 8). We note that filters in modern DNNs are nearly always square.

can count the number of consecutive matrix multiplications which have the same computation pattern to determine B.

In a convolutional layer i, the hyper-parameters include the number of filters (D_{i+1}), the filter width and height (R_i), and the padding (P_i). We assume that the filter width and height are the same, which is the common case. Note that for layer i, we consider that the depth of the input volume (D_i) is known, as it can be obtained from the previous layer.

We now show how these parameters for a convolutional layer can be reverse engineered. From Table A.2, we see that the number of filters (D_{i+1}) is the number of rows of the filter matrix F'_i. To attain the filter width (R_i), we note that the number of rows of the in'_i matrix is $D_i \times R_i^2$, where D_i is the number of output channels in the previous layer and is equal to the number of rows of the out'_{i-1} matrix. Therefore, as summarized in Table A.3, the filter width is attained by dividing the number of rows of in'_i by the number of rows of out'_{i-1} and performing the square root. In the case that layer i is the first one, directly connected to the input, the denominator (out'_0) of this fraction is the number of channels of the input of the network, which is public information.

Padding results in a larger input matrix (in'_i). After resolving the filter width (R_i), the value of padding can be deduced by determining the difference between the number of columns of the output matrix of layer $i-1$ (out'_{i-1}), which is $W_i \times H_i$, and the number of columns of the in'_i matrix, which is $(W_i - R_i + P)(H_i - R_i + P)$.

A pooling layer can be located in-between two convolutional layers. It down-samples every channel of the input along width and height, resulting in a small channel size. The hyper-parameter in this layer is the pool width and height (assumed to be the same value), which can be inferred as follows. Consider the channel size of the output of layer i (number of columns in out'_i) and the channel size of the input volume in layer $i+1$ (approximately equals to the number of columns in in'_{i+1}). If the two are the same, there is no pooling layer; otherwise, we expect to see the channel size reduced by the square of the pool width. In the latter case, the exact pool dimension can be found using a similar procedure used to determine R_i. Note that a non-unit stride operation results in the same dimension reduction as a pooling layer. Thus, we cannot distinguish between non-unit striding and pooling. Table A.3 summarizes the mappings.

A.4.3 Connections Between Layers

We now examine how to map inter-layer connections to GEMM execution. We consider two types of inter-layer connections, i.e., sequential connections and non-sequential connections.

Mapping Sequential Connections

A sequential connection is one that connects two consecutive layers, e.g., layer i and layer $i + 1$. The output of layer i is used as the input of its next layer $i + 1$. According to the mapping relationships in Table A.3, a DNN places several constraints on GEMM parameters for sequentially-connected convolutional layers.

First, since the filter width and height must be integer values, there is a constraint on the number of rows of the input and output matrices in consecutive layers. Considering the formula used to derive the filter width and height in Table A.3, if layer i and layer $i + 1$ are connected, the number of rows in the input matrix of layer $i + 1$ ($n\_row(in'_{i+1})$) must be the product of the number of rows in the output matrix of layer i ($n\_row(out'_i)$) and the square of an integer number.

Second, since the pool size and stride size are integer values, there is another constraint on the number of columns of the input and output matrix sizes between consecutive layers. According to the formula used to derive pool and stride size, if layer i and layer $i + 1$ are connected, the number of columns in the output matrix of layer i ($n\_col(out'_i)$) must be very close to the product of the number of columns in the input matrix of layer $i+1$ ($n\_col(in'_{i+1})$) and the square of an integer number.

The two constraints above help us to distinguish non-sequential connections from sequential ones. Specifically, if one of these constraints is not satisfied, we are sure that the two layers are not sequentially connected.

Mapping Non-sequential Connections

In this chapter, we consider that a non-sequential connection is one where, given two consecutive layers i and $i + 1$, there is a third layer j, whose output is merged with the output of layer i and the merged result is used as the input to layer $i + 1$. We call the extra connection from layer j to layer $i + 1$ a shortcut, where layer j is the source layer and layer $i + 1$ is the sink layer. Shortcut connections can be mapped to GEMM execution.

First, there exists a certain latency between consecutive GEMMs, which we call inter-GEMM latency. The inter-GEMM latency before the sink layer in a non-sequential connection is longer than the latency in a sequential connection. To see why, consider the operations that are performed between two consecutive GEMMs: post-processing of the prior GEMM's output (e.g., batch normalization) and pre-processing of the next GEMM's input (e.g., padding and striding). When there is no shortcut, the inter-GEMM latency is linearly related to the sum of the prior layer's output size and the next layer's input size.

However, a shortcut requires an extra merge operation that incurs extra latency between GEMM calls.

Second, the source layer of a shortcut connection must have the same output dimensions as the other source layer of the non-sequential connection. For example, when a shortcut connects layer j and layer $i+1$, the output matrices of layer j and layer i must have the same number of rows and columns. This is because one can only merge two outputs whose dimension sizes match.

These two characteristics help us identify the existence of a shortcut, its source layer, and its sink layer.

A.4.4 Activation Functions

So far, this section discussed how DNN parameters map to GEMM calls. Convolutional and fully-connected layers are post-processed by elementwise non-linear functions, such as `relu`, `sigmoid` and `tanh`, which do not appear in GEMM parameters. We can distinguish `relu` activations from `sigmoid` and `tanh` by monitoring whether the non-linear functions access the standard mathematical library `libm`. `relu` is a simple activation which does not need support from libm, while the other functions are computationally intensive and generally leverage libm to achieve high performance. We remark that nearly all convolutional layers use `relu` or a close variant [148, 174–177].

A.5 ATTACKING MATRIX MULTIPLICATION

We now design a side channel attack to learn matrix multiplication parameters. Given the mapping from the previous section, this attack will allow us to reconstruct the DNN architecture.

We analyze state-of-the-art BLAS libraries, which have extensively optimized blocked matrix multiply. Examples of such libraries are OpenBLAS [171], BLIS [178], Intel MKL [173] and AMD ACML [179]. We show in detail how to extract the desired information from the GEMM implementation in OpenBLAS. In Section A.6, we generalize our attack to other BLAS libraries, using Intel MKL as an example.

A.5.1 Analyzing GEMM from OpenBLAS

Function `gemm_nn` from the OpenBLAS library performs blocked matrix-matrix multiplication. It computes $C = \alpha A \cdot B + \beta C$ where α and β are scalars, A is an $m \times k$ matrix, B is a $k \times n$ matrix, and C is an $m \times n$ matrix. Our goal is to extract m, n and k.

Like most modern BLAS libraries, OpenBLAS implements Goto's algorithm [180]. The algorithm has been optimized for modern multi-level cache hierarchies. Figure A.3 depicts the way Goto's algorithm structures blocked matrix multiplication for a three-level cache. The *macro-kernel* at the bottom performs the basic operation, multiplying a $P \times Q$ block from matrix A with a $Q \times R$ block from matrix B. This kernel is generally written in assembly code, and manually optimized by taking the CPU pipeline structure and register availability into consideration. The block sizes are picked so that the $P \times Q$ block of A fits in the L2 cache, and the $Q \times R$ block of B fits in the L3 cache.

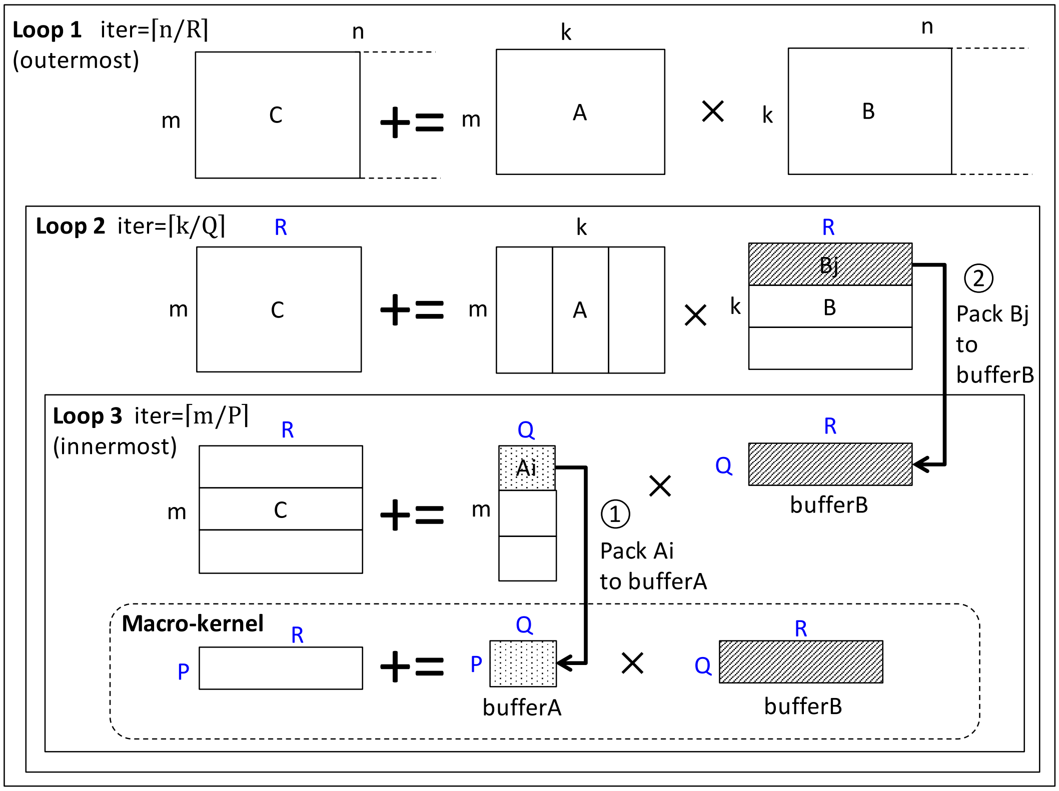


Figure A.3: Blocked GEMM with matrices in column major.

As shown in Figure A.3, there is a three-level loop nest around the macro-kernel. The innermost one is Loop 3, the intermediate one is Loop 2, and the outermost one is Loop 1. We call the iteration counts in these loops $iter_3$, $iter_2$, and $iter_1$, respectively, and are given by:

$$\begin{aligned} iter_3 &= \lceil m/P \rceil \\ iter_2 &= \lceil k/Q \rceil \\ iter_1 &= \lceil n/R \rceil \end{aligned} \tag{A.1}$$

Algorithm A.1 shows the corresponding pseudo-code with the three nested loops. Note that Loop 3 is further split into two parts, to obtain better cache locality. The first part performs only the first iteration, and the second part performs the rest.

Algorithm A.1: `gemm_nn` in OpenBLAS.

```
   Input  : Matrix $A$, $B$, $C$; Scalar $\alpha$, $\beta$; Block size $P, Q, R$; $UNROLL$
   Output: $C := \alpha A \cdot B + \beta C$
 1 for $j = 0, n, R$ do // Loop 1
 2     for $l = 0, k, Q$ do // Loop 2
           // Loop 3, 1st iteration
 3         itcopy($A[0, l]$, $buf\_A$, $P$, $Q$)
 4         for $jj = j, j + R, 3UNROLL$ do // Loop 4
 5             oncopy($B[l, jj]$, $buf\_B + (jj - j) \times Q$, $Q$, $3UNROLL$)
 6             kernel($buf\_A$, $buf\_B + (jj - j) \times Q$, $C[l, j]$, $P$, $Q$, $3UNROLL$)
 7         end
           // Loop 3, rest iterations
 8         for $i = P, m, P$ do
 9             itcopy($A[i, l]$, $buf\_A$, $P$, $Q$)
10             kernel($buf\_A$, $buf\_B$, $C[l, j]$, $P$, $Q$, $R$)
11         end
12     end
13 end
```

The first iteration of Loop 3 (Lines 3-7) performs three steps as follows. First, the data in the $P \times Q$ block from matrix A is packed into a buffer (bufferA) using function `itcopy`. This is shown in Figure A.3 as arrow ① and corresponds to line 3 in Algorithm A.1. Second, the data in the $Q \times R$ block from matrix B is also packed into a buffer (bufferB) using function `oncopy`. This is shown in Figure A.3 as arrow ② and corresponds to line 5 in Algorithm A.1. The $Q \times R$ block from matrix B is copied in units of $Q \times 3UNROLL$ sub-blocks. This breaks down the first iteration of Loop 3 into a loop, which is labeled as Loop 4. The iteration count in Loop 4, $iter_4$, is given by:

$$\begin{aligned} & iter_4 = \lceil R/3UNROLL \rceil \\ or \quad & iter_4 = \lceil (n \bmod R)/3UNROLL \rceil \end{aligned} \tag{A.2}$$

where the second expression corresponds to the last iteration of Loop 1. Note that bufferB, which is filled by the first iteration of Loop 3, is also shared by the rest of iterations. Third, the macro-kernel (function `kernel`) is executed on the two buffers. This corresponds to line 6 in Algorithm A.1.

The rest iterations (line 8-11) skip the second step above. These iterations only pack a block from matrix A to fill bufferA and execute the macro-kernel.

The BLAS libraries use different P, Q, and R for different cache sizes to achieve best performance. For example, when compiling OpenBLAS on our experimental machine (Section A.7), the GEMM function for double data type uses $P = \mathit{512}$; $Q = \mathit{256}$, $R = \mathit{16384}$, and $\mathit{3UNROLL} = \mathit{24}$.

A.5.2 Locating Probing Addresses

Our goal is to find the size of the matrices of Figure A.3, namely, m, k, and n. To do so, we need to first obtain the number of iterations of the 4 loops in Algorithm A.1, and then use Formulas A.1 and A.2. Note that we know the values of the block sizes P, Q, and R (as well as *3UNROLL*) — these are constants available in the open-source code of OpenBLAS.

In this chapter, we propose to use, as probing addresses, addresses in the `itcopy`, `oncopy` and `kernel` functions of Algorithm A.1. To understand why, consider the dynamic invocations to these functions. Figure A.4 shows the Dynamic Call Graph (DCG) of `gemm_nn` in Algorithm A.1.

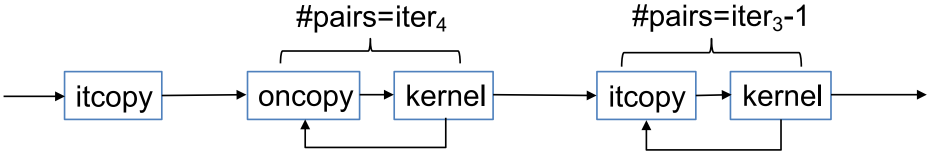


Figure A.4: DCG of `gemm_nn`, with the number of invocations per iteration of Loop 2.

Each iteration of Loop 2 contains one invocation of function `itcopy`, followed by iter_4 invocations of the pair `oncopy` and `kernel`, and then $(\mathit{iter}_3 - \mathit{1})$ invocations of the pair `itcopy` and `kernel`. The whole sequence in Figure A.4 is executed $\mathit{iter}_1 \times \mathit{iter}_2$ times in one invocation of `gemm_nn`. We will see in Section A.5.3 that these invocation counts are enough to allow us to find the size of the matrices.

We now discuss how to select probing addresses inside the three functions—`itcopy`, `oncopy` and `kernel`—to improve attack accuracy. The main bodies of the three functions are loops. To distinguish these loops from the GEMM loops, we refer to them in this chapter as *in-function* loops. We select addresses that are located inside the in-function loops as probing addresses. This strategy helps improve attack accuracy, because such addresses are accessed multiple times per function invocation and their access patterns can be easily distinguished from noise (Section A.8.1).

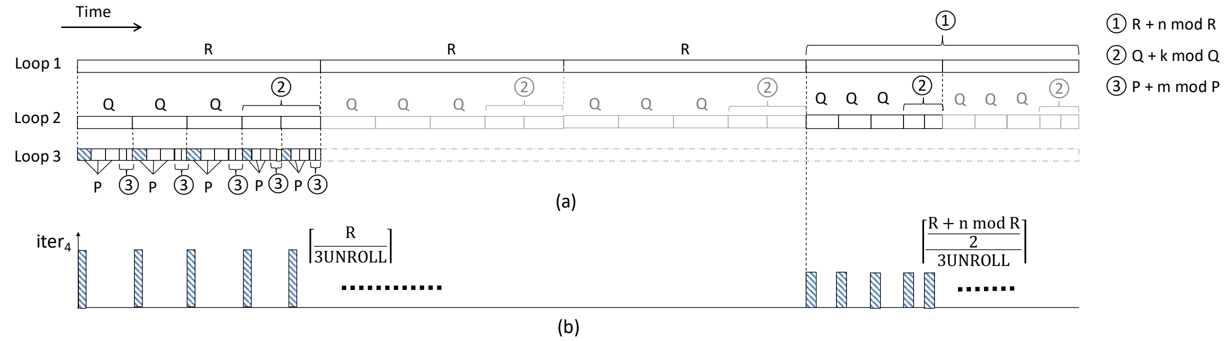


Figure A.5: Visualization of execution time of a `gemm_nn` where Loop 1, Loop 2, and Loop3 have 5 iterations each (a), and value of $iter_4$ for each first iteration of Loop 3 (b).

A.5.3 Procedure to Extract Matrix Dimensions

To understand the procedure we use to extract matrix dimensions, we show an example in Figure A.5(a), which visualizes the execution time of a `gemm_nn` where Loop 1, Loop 2 and Loop 3 have 5 iterations each. The figure also shows the size of the block that each iteration operates on. Note that the OpenBLAS library handles the last two iterations of each loop in a special manner. When the last iteration does not have a full block to compute, rather than assigning a small block to the last iteration, it assigns two equal-sized small blocks to the last two iterations. In Figure A.5(a), in Loop 1, the first three iterations use R-sized blocks, and each of the last two use a block of size $(R + n \bmod R)/2$. In Loop 2, the corresponding block sizes are Q and $(Q + k \bmod Q)/2$. In Loop 3, they are P and $(P + m \bmod P)/2$.

Figure A.5(b) shows additional information for each of the first iterations of Loop 3. Recall that the first iteration of Loop 3 is special, as it involves an extra packing operation that is performed by Loop 4 in Algorithm A.1. Figure A.5(b) shows the number of iterations of Loop 4 in each invocation ($iter_4$). During the execution of the first three iterations of Loop 1, $iter_4$ is $\lceil R/3UNROLL \rceil$. In the last two iterations of Loop 1, $iter_4$ is $\lceil ((R + n \bmod R)/2)/3UNROLL \rceil$, as can be deduced from Equation A.2 after applying OpenBLAS' special handling of the last two iterations.

Based on these insights, our procedure to extract m, k, and n has four steps.

Step 1: Identify the DCG of a Loop 2 iteration and extract $iter_1 \times iter_2$**.** By probing one instruction in each of `itcopy`, `oncopy`, and `kernel`, we repeatedly obtain the DCG pattern of a Loop 2 iteration (Figure A.4). By counting the number of such patterns, we obtain $iter_1 \times iter_2$.

Step 2: Extract $iter_3$ **and determine the value of** m**.** In the DCG pattern of a Loop 2 iteration, we count the number of invocations of the `itcopy-kernel` pair (Figure A.4).

This count plus 1 gives $iter_3$. Of all of these $iter_3$ iterations, all but the last two execute a block of size P; the last two execute a block of size $(P + m \bmod P)/2$ each (Figure A.5(a)). To estimate the size of this smaller block, we assume that the execution time of an iteration is proportional to the block size it processes — except for the first iteration which, as we indicated, is different. Hence, we time the execution of a "normal" iteration of Loop 3 and the execution of the last iteration of Loop 3. Let's call the times t_{normal} and t_{small}. The value of m is computed by adding P for each of the ($iter_3$ - 2) iterations and adding the estimated number for each of the last two iterations:

$$m = (iter_3 - 2) \times P + 2 \times \frac{t_{small}}{t_{normal}} \times P \tag{A.3}$$

Step 3: Extract $iter_4$ and $iter_2$, and determine the value of k. In the DCG pattern of a Loop 2 iteration (Figure A.4), we count the number of `oncopy-kernel` pairs, and obtain $iter_4$. As shown in Figure A.5(b), the value of $iter_4$ is $\lceil R/3UNROLL \rceil$ in all iterations of Loop 2 except those that are part of the last two iterations of Loop 1. For the latter, $iter_4$ is $\lceil ((R + n \bmod R)/2)/3UNROLL \rceil$, which is a lower value. Consequently, by counting the number of DCG patterns that have a low value of $iter_4$, and dividing it by 2, we attain $iter_2$. We then follow the procedure of Step 2 to calculate k. Specifically, all Loop 2 iterations but the last two execute a block of size Q; the last two execute a block of size $(Q + k \bmod Q)/2$ each (Figure A.5(a)). Hence, we time the execution of two iterations of Loop 2 in the first Loop 1 iteration: a "normal" one (t'_{normal}) and the last one (t'_{small}). We then compute k like in Step 2:

$$k = (iter_2 - 2) \times Q + 2 \times \frac{t'_{small}}{t'_{normal}} \times Q \tag{A.4}$$

Step 4: Extract $iter_1$ and determine the value of n. If we take the total number of DCG patterns in the execution from Step 1 and divide that by $iter_2$, we obtain $iter_1$. We know that all Loop 1 iterations but the last two execute a block of size R; the last two execute a block of size $(R + n \bmod R)/2$ each. To compute the size of the latter block, we note that, in the last two iterations of Loop 1, $iter_4$ is $\lceil ((R + n \bmod R)/2)/3UNROLL \rceil$. Since both $iter_4$ and $3UNROLL$ are known, we can estimate $(R + n \bmod R)/2$. We neglect the effect of the ceiling operator because $3UNROLL$ is a very small number. Hence, we compute n as:

$$n = (iter_1 - 2) \times R + 2 \times iter_4 \times 3UNROLL \tag{A.5}$$

Our attack cannot handle the cases when m or k are less than or equal to twice their corresponding block sizes. For example, when m is less than or equal to $2 \times P$, there is no iteration of Loop 3 that operates on a smaller block size. Our procedure cannot compute the exact value of m, and can only say that $m \leq 2P$.

A.6 GENERALIZATION OF THE ATTACK ON GEMM

Our attack can be generalized to other BLAS libraries, since all of them use blocked matrix-multiplication, and most of them implement Goto's algorithm [180]. We show that our attack is still effective, using the Intel MKL library as an example. MKL is a widely used library but is closed source. We reverse engineer the scheduling of the three-level nested loop in MKL and its block sizes. The information is enough for us to apply the same attack procedure in Section A.5.3 to obtain matrix dimensions.

Constructing the DCG We apply binary analysis [39, 181] techniques to construct the DCG of the GEMM function in MKL, shown in Figure A.6. The pattern is the same as the DCG of OpenBLAS in Figure A.4. Thus, the attack strategy in Section A.5 also works towards MKL.

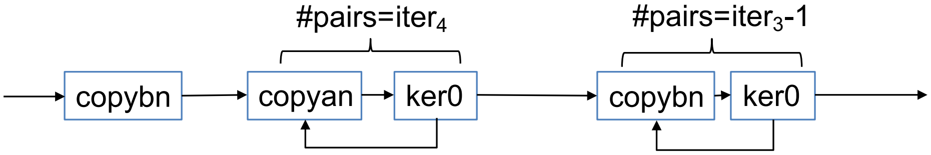


Figure A.6: DCG of blocked GEMM in Intel MKL, with the number of invocations per iteration of Loop 2.

Extracting Block Sizes Similar to OpenBLAS, in MKL, there exists a linear relationship between matrix dimensions and iteration count for each of the loops, as shown in Formulas A.1 and A.2. When the matrix size increases by a block size, the corresponding iteration count increases by 1. We leverage this relationship to reverse engineer the block sizes for MKL. Specifically, we gradually increase the input dimension size until the number of iterations increments. The stride on the input dimension that triggers the change of iteration count is the block size.

Special Cases According to our analysis, MKL follows a different DCG when dealing with small matrices. First, instead of executing three-level nested loops, it uses a single-level loop, tiling on the dimension that has the largest value among m, n, k. Second, the kernel computation is performed directly on the input matrices, without packing and buffering operations.

For these special cases, we slightly adjust the attack strategy in Figure A.5. We use side channels to monitor the number of iterations on that single-level loop and the time spent for each iteration. We then use the number of iterations to deduce the size of the largest dimension. Finally, we use the timing information for each iteration to deduce the product of the other two dimensions.

A.7 EXPERIMENTAL SETUP

Attack Platform We evaluate our attacks on a Dell workstation Precision T1700, which has a 4-core Intel Xeon E3 processor and an 8GB DDR3-1600 memory. The processor has two levels of private caches and a shared last level cache. The first level caches are a 32KB instruction cache and a 32KB data cache. The second level cache is 256KB. The shared last level cache is 8MB. We test our attacks on a same-OS scenario using Ubuntu 4.2.0-27, where the attacker and the victim are different processes within the same bare-metal server. Our attacks should be applicable to other platforms, as the effectiveness of Flush+Reload and Prime+Probe has been proved in multiple hardware platforms [6, 48].

Victim DNNs We use a VGG [176] instance and a ResNet [148] instance as victim DNNs. VGG is representative of early DNNs (e.g., AlexNet [175] and LeNet [182]). ResNet is representative of state-of-the-art DNNs. Both are standard and widely-used CNNs with a large number of layers and hyper-parameters. ResNet additionally features shortcut connections.

There are several versions of VGG, with 11 to 19 layers. All VGGs have 5 types of layers, which are replicated a different number of times. We show our results on VGG-16.

There are several versions of ResNet, with 18 to 152 layers. All of them consist of the same 4 types of modules, which are replicated a different number of times. Each module contains 3 or 4 layers, which are all different. We show our results on ResNet-50.

The victim programs are implemented using the Keras [183] framework, with Theano [184] as the backend. We execute each DNN instance with a single thread.

Attack Implementation We use Flush+Reload and Prime+Probe attacks. In both attacks, the attacker and the victim are different processes and are pinned to different cores, only sharing the last level cache.

In Flush+Reload, the attacker and the victim share the BLAS library via page deduplication. The attacker probes one address in `itcopy` and one in `oncopy` every 2,000 cycles. There is no need to probe any address in `kernel`, as the access pattern is clear enough. Our Prime+Probe attack targets the last level cache. We construct two sets of conflict addresses for the two probing addresses using the algorithm proposed by Liu et al. [6]. The Prime+Probe uses the same monitoring interval length of 2,000 cycles.

A.8 EVALUATION

We first evaluate our attacks on the GEMM function. We then show the effectiveness of our attack on neural network inference, followed by an analysis of the search space of DNN architectures.

A.8.1 Attacking GEMM

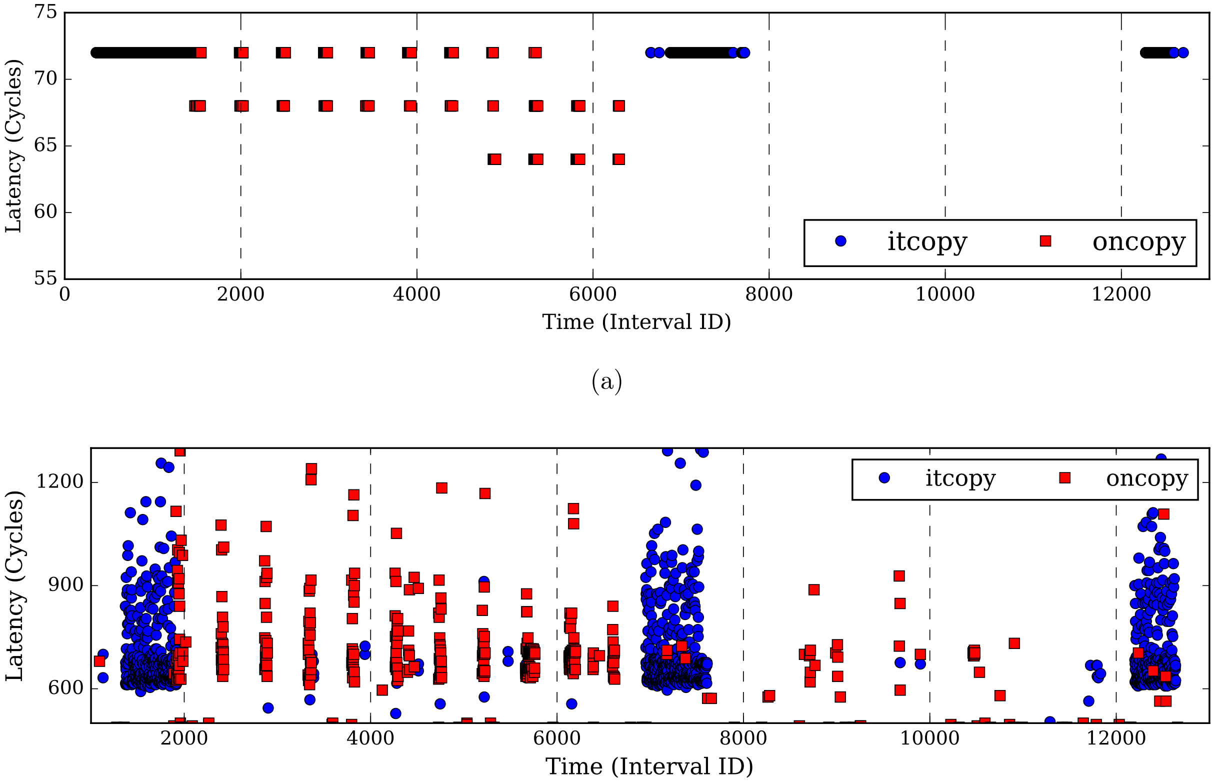
Attack Examples Figure A.7 shows raw traces generated by the two side channel attacks, i.e. Flush+Reload and Prime+Probe, when monitoring the execution of the GEMM function in OpenBLAS. Due to space limitations, we only show the traces for one iteration of Loop 2 (Algorithm A.1).

Figure A.7(a) is generated under Flush+Reload. It shows the latency of the attacker's reload accesses to the probing addresses in the `itcopy` and `oncopy` functions for each monitoring interval. In the figure, we only show the instances where the access took less than 75 cycles. These instances correspond to cache hits and, therefore, cases when the victim executed the corresponding function. Figure A.7(b) is generated under Prime+Probe. It shows the latency of the attacker's probe accesses to the conflict addresses. We only show the instances where the accesses took more than 500 cycles. These instances correspond to cache misses of at least one conflict address. They are the cases when the victim executed the corresponding function.

Since we select the probing addresses to be within in-function loops (Section A.5.2), a cluster of hits in the Flush+Reload trace (or misses in the Prime+Probe trace) indicates the time period when the victim is executing the probed function.

In both traces, the victim calls `itcopy` before interval 2,000, then calls `oncopy` 11 times between intervals 2,000 and 7,000. It then calls `itcopy` another two times in intervals 7,000 and 13,000. The trace matches the DCG shown in Figure A.4. We can easily derive that $iter_4 = 11$ and $iter_3 = 3$.

Handling Noise Comparing the two traces in Figure A.7, we can observe that Prime+Probe suffers much more noise than Flush+Reload. The noise in Flush+Reload is generally sparsely and randomly distributed, and thus can be easily filtered out. However, Prime+Probe has



(a)

(b)

Figure A.7: Flush+Reload (a) and Prime+Probe (b) traces of the GEMM execution. The monitoring interval is 2,000 cycles.

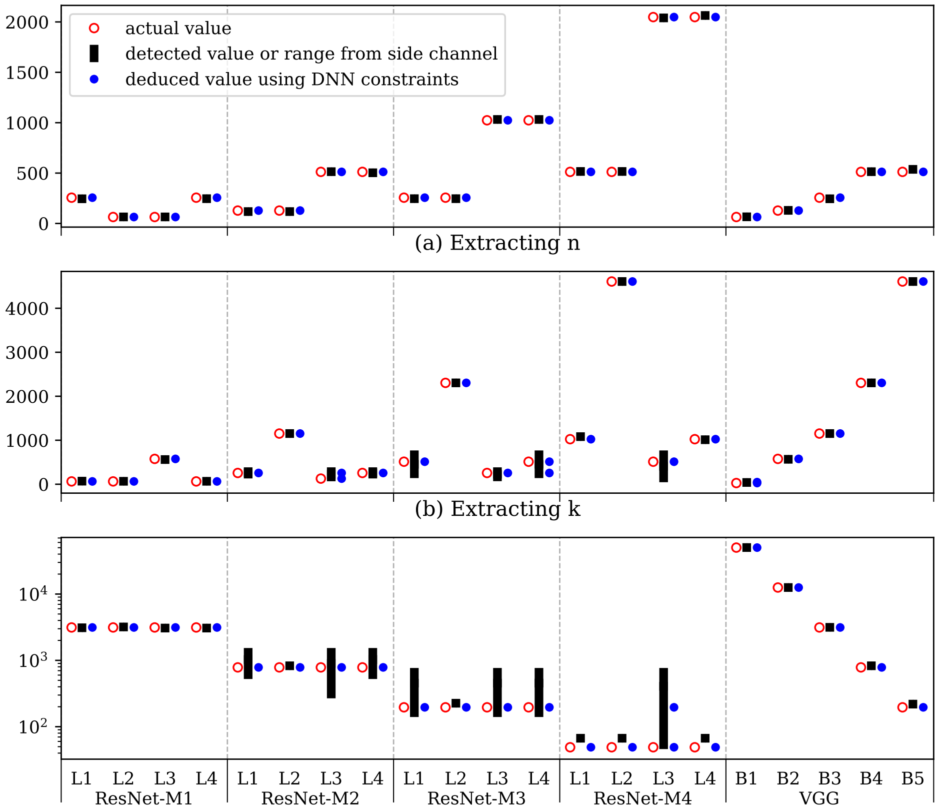
noise in consecutive monitoring intervals, as shown in Figure A.7(b). It happens mainly due to the non-determinism of the cache replacement policy [185]. When one of the cache ways is used by the victim's line, it takes multiple "prime" operations to guarantee that the victim's line is selected to be evicted. It is more difficult to distinguish the victim's accesses from such noise.

We leverage our knowledge of the execution patterns in GEMM to handle the noise in Prime+Probe. First, recall that we pick the probing addresses within tight loops inside each of the probing functions (Section A.5.2). Therefore, for each invocation of the functions, the corresponding probing address is accessed multiple times, which is observed as a cluster of cache misses in Prime+Probe. We count the number of consecutive cache misses in each cluster to obtain its size. The size of a cluster of cache misses that are due to noise is smaller than size of a cluster of misses that are caused by the victim's accesses. Thus, we discard the clusters with small sizes. Second, due to the three-level loop structure, each probing function, such as `oncopy`, is called repetitively with consistent interval lengths between each invocation (Figure A.4). Thus, we compute the distances between neighboring clusters and discard the clusters with abnormal distances to their neighbors.

These two steps are effective enough to handle the noise in Prime+Probe. However, when tracing MKL's special cases that use a single-level loop (Section A.6), we find that using Prime+Probe is ineffective to obtain useful information. Such environment affects the accuracy of the Cache Telepathy attack, as we will see in Section A.8.3.

A.8.2 Extracting Hyper-parameters of DNNs

We show the effectiveness of our attack by extracting the hyper-parameters of VGG-16 [176] and ResNet-50 [148]. Figures A.8(a), A.8(b), and A.8(c) show the extracted values of the n, k, and m matrix parameters, respectively, using Flush+Reload. In each figure, we show the values for each of the layers (*L1*, *L2*, *L3*, and *L4*) in the 4 distinct modules in ResNet-50 (*M1*, *M2*, *M3*, and *M4*), and for the 5 distinct layers in VGG-16 (*B1*, *B2*, *B3*, *B4*, and *B5*). We do not show the other layers because they are duplicates of the layers shown.



(a) Extracting n

(b) Extracting k

(c) Extracting m

Figure A.8: Extracted values of the n, k, and m matrix parameters for VGG-16 and ResNet-50 using Flush+Reload on OpenBLAS.

The figures show three data points for each parameter (e.g., m) and each layer (e.g., L1 in ResNet-M2): a hollowed circle, a solid square or rectangle, and a solid circle. The hollowed circle indicates the *actual* value of the parameter. The solid square or rectangle indicates the value of the parameter *detected* with our side channel attack. When the side channel attack can only narrow down the possible value to a range, the figure shows a rectangle. Finally, the solid circle indicates the value of the parameter that we *deduce*, based on the detected value and some DNN constraints. For example, for parameter m in layer L1 of ResNet-M2, the actual value is 784, the detected value range is $[524, 1536]$, and the deduced value is 784.

We will discuss how we obtain the solid circles later. Here, we compare the actual and the detected values (hollowed circles and solid squares/rectangles). Figure A.8(a) shows that our attack is always able to determine the n value with negligible error. The reason is that, to compute n, we need to estimate $iter_1$ and $iter_4$ (Section A.5.3), and it can be shown that most of the noise comes from estimating $iter_4$. However, since $iter_4$ is multiplied by the small *3UNROLL* parameter in the equation for n, the impact of such noise is small.

Figures A.8(b) and (c) show that the attack is able to accurately determine the m and k values for all the layers in ResNet-M1 and VGG, and for most layers in ResNet-M4. However, it can only derive ranges of values for most of the ResNet-M2 and ResNet-M3 layers. This is because the m and k values in these layers are often smaller than twice of the corresponding block sizes (Section A.5.3).

In Figure A.9, we show the same set of results by analyzing the traces generated using Prime+Probe. Compared to the results from Flush+Reload, there are some differences of detected values or ranges, especially in ResNet-M3 and ResNet-M4.

In summary, our side channel attacks, using Flush+Reload or Prime+Probe, can either detect the matrix parameters with negligible error, or can provide a range where the actual value falls in. We will next show that, in many cases, the imprecision from the negligible error and the ranges can be eliminated after applying DNN constraints (Section A.8.3).

A.8.3 Size of Architecture Search Space

In this section, we compare the number of architectures in the search space without Cache Telepathy (which we call *original* space), and with Cache Telepathy (which we call *reduced* space). In both cases, we only consider reasonable hyper-parameters for the layers as follows. For fully-connected layers, the number of neurons can be 2^i, where $8 \leq i \leq 13$. For convolutional layers, the number of filters can be a multiple of 64 ($64 \times i$, where $1 \leq i \leq 32$), and the filter size can be an integer value between 1 and 11.

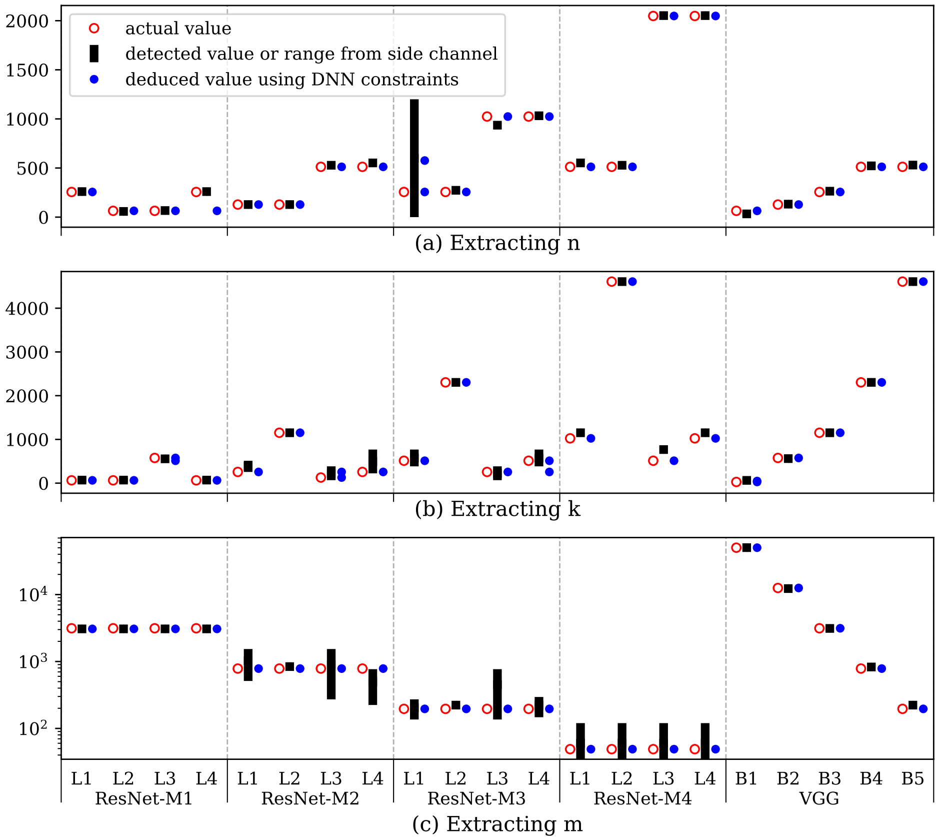


Figure A.9: Extracted values of the n, k, and m matrix parameters for VGG-16 and ResNet-50 using Prime+Probe on OpenBLAS.

Size of the Original Search Space

To be conservative, when computing the size of the original search space, we assume that the attacker knows the number of layers and type of each layer in the oracle DNN. There exist 352 different configurations for each convolutional layer without considering pooling or striding, and 6 configurations for each fully-connected layer. Moreover, considering the existence of non-sequential connections, given L layers, there are $L \times 2^{L-1}$ possible ways to connect them.

A network like VGG-16 has five *different* layers (*B1*, *B2*, *B3*, *B4*, and *B5*), and no shortcuts. If we consider only these five different layers, the size of the search space is about 5.4×10^{12} candidate architectures. A network like ResNet-50 has 4 *different* modules (*M1*, *M2*, *M3*, and *M4*) and some shortcuts inside these modules. If we consider only these four different modules, the size of the search space is about 6×10^{46} candidate architectures. Overall, the original search space is intractable.

Determining the Reduced Search Space

Using the detected values of the matrix parameters in Section A.8.2, we first determine the possible connections between layers by locating shortcuts. Next, for each possible connection configuration, we calculate the possible hyper-parameters for each layer. The final search space is computed as

$$search\ space = \sum_{i=1}^{C} (\prod_{j=1}^{L} x_j) \tag{A.6}$$

where C is the total number of possible connection configurations, L is the total number of layers, and x_j is the number of possible combinations of hyper-parameters for layer j.

Determining Connections Between Layers We show how to reverse engineer the connections between layers using ResNet-M1 as an example.

First, we leverage inter-GEMM latency to determine the existence of shortcuts and their sinks using the method discussed in Section A.4.3. Figure A.10 shows the extracted matrix dimensions and the inter-GEMM latency for the 4 layers in ResNet-M1. The inter-GEMM latency after M1-L4 is significantly longer than expected, given its output matrix size and the input matrix size of the next layer. Thus, the layer after M1-L4 is a sink layer.

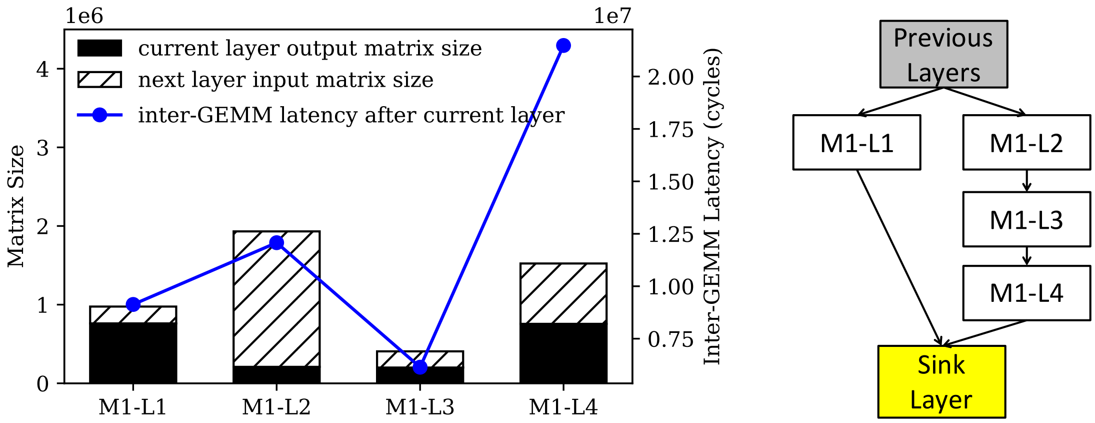


Figure A.10: Extracting connections in ResNet-M1.

Next, we check the output matrix dimensions of previous layers to locate the source of the shortcut. Note that a shortcut only connects layers with the same output matrix dimensions. Based on the extracted dimension information (values of n and m) in Figure A.8, we determine that M1-L1 is the source. In addition, we know that M1-L1 and M1-L2 are not sequentially connected by comparing the output matrix of M1-L1 and the input matrix of M1-L2 (Section A.4.3).

Figure A.10 summarizes the reverse engineered connections among the 4 layers, which match the actual connections in ResNet-M1. We can use the same method to derive the possible connection configurations for the other modules. Note that this approach does not work for ResNet-M3 and ResNet-M4. In these layers, the input and output matrices are small and operations between consecutive layers take a short time. As a result, the inter-GEMM latency is not effective in identifying shortcuts.

Determining Hyper-parameters for Each Layer We plug the detected matrix parameters into the formulas in Table A.3 to deduce the hyper-parameters for each layer. For the matrix dimensions that cannot be extracted precisely, we leverage DNN constraints to prune the search space.

As an example, consider reverse engineering the hyper-parameters for Layer 3 in ResNet-M2. First, we extract the number of filters. We round the extracted $n_{M2\text{-}L3}$ from Figure A.8(a) (the number of rows in F') to the nearest multiple of 64. This is because, as discussed at the beginning of Section A.8.3, we assume that the number of filters is a multiple of 64. We get that the number of filters is 512. Second, we use the formula in Table A.3 to determine the filter width and height. We consider the case where L2 is sequentially connected to L3. The extracted range of $k_{M2\text{-}L3}$ from Figure A.8(b) (the number of rows in in' of current layer) is $[68, 384]$, and the value for $n_{M2\text{-}L2}$ from Figure A.8(a) (the number of rows in out' of the previous layer) is 118. We need to make sure that the square root of $k_{M2\text{-}L3}/n_{M2\text{-}L2}$ is an integer, which leads to the conclusion that the only possible value for $k_{M2\text{-}L3}$ is 118 (one of the solid circles for $k_{M2\text{-}L3}$), and the filter width and height is 1. The same value is deduced if we consider, instead, that L1 is connected to L3. The other solid circle for $k_{M2\text{-}L3}$ is derived similarly if we consider that the last layer in M1 is connected to layer 3 in M2.

We apply the same methodology for the other layers. With this method, we obtain the solid circles in Figures A.8 and A.9.

Determining Pooling and Striding We use the difference in the m dimension (i.e., the channel size of the output) between consecutive layers to determine the pool or stride size. For example, in Figure A.8(c) and A.9(c), the m dimensions of the last layer in ResNet-M1 and the first layer in ResNet-M2 are different. This difference indicates the existence of a pool layer or a stride operation. In Figure A.8(c), the extracted value of $m_{M1\text{-}L4}$ (the number of columns in out' for the current layer) is 3072, and the extracted range of $m_{M2\text{-}L1}$ (the number of columns in in' for the next layer) is $[524, 1536]$. We use the formula in Table A.3 to determine the pool or stride width and height. To make the square root of $m_{M1\text{-}L4}/m_{M2\text{-}L1}$

an integer, $m_{M2\text{-}L1}$ has to be 768, and the pool or stride width and height have to be 2.

Size of the Reduced Search Space

Using Equation A.6, we compute the number of architectures in the search space without Cache Telepathy and with Cache Telepathy. Table A.4 shows the resulting values. Note that we only consider the possible configurations of the *different* layers in VGG-16 (*B1*, *B2*, *B3*, *B4*, and *B5*) and of the *different* modules in ResNet-50 (*M1*, *M2*, *M3*, and *M4*).

DNN		ResNet-50	VGG-16
Original: No Cache Telepathy		$> 6 \times 10^{46}$	$> 5.4 \times 10^{12}$
Flush+Reload	OpenBLAS	512	16
	MKL	6144	64
Prime+Probe	OpenBLAS	512	16
	MKL	5.7×10^{15}	1936

Table A.4: Comparing the original search space (without Cache Telepathy) and the reduced search space (with Cache Telepathy).

Using Cache Telepathy to attack OpenBLAS, we are able to significantly reduce the search space from an intractable size to a reasonable size. Both Flush+Reload and Prime+Probe obtain a very small search space. Specifically, for VGG-16, Cache Telepathy reduces the search space from more than 5.4×10^{12} architectures to just 16; for ResNet-50, Cache Telepathy reduces the search space from more than 6×10^{46} to 512.

Cache Telepathy is less effective on MKL. For VGG-16, Cache Telepathy reduces the search space from more than 5.4×10^{12} to 64 (with Flush+Reload) or 1936 (with Prime+Probe). For ResNet-50, Cache Telepathy reduces the search space from more than 6×10^{46} to 6144 (with Flush+Reload) or 5.7×10^{15} (with Prime+Probe). The last number is large because the matrix dimensions in Module M1 and Module 4 of ResNet-50 are small, and MKL handles these matrices with the special method described in Section A.6. Such method is not easily attackable by Prime+Probe. However, if we only count the number of possible configurations in Modules M1, M2, and M3, the search space is 41472.

Implications of Large Search Spaces A large search space means that the attacker needs to train many networks. Training DNNs is easy to parallelize, and attackers can request many GPUs to train in parallel. However, it comes with a high cost. For example, assume that training one network takes 2 GPU days. On Amazon EC2, the current price for a single-node GPU instance is ∼\$3/hour. Without Cache Telepathy, since the search space is so huge, the cost is unbearable. Using Cache Telepathy with Flush+Reload, the reduced search space for the different layers in VGG-16 and for the different modules in ResNet-50 running OpenBLAS means that the training takes 32 and 1024 GPU days, respectively. The resulting cost is only ∼\$2K and ∼\$74K. When attacking ResNet-50 running MKL, the attacker needs to train 6144 architectures, requiring over \$884K.

A.9 COUNTERMEASURES

We overview possible countermeasures against our attack, and discuss their effectiveness and performance implications.

We first investigate whether it is possible to stop the attack by modifying the BLAS libraries. All BLAS libraries use extensively optimized blocked matrix multiplication for performance. One approach is to disable the optimization or use less aggressive optimization. However, it is unreasonable to disable blocked matrix multiplication, as the result would be very poor cache performance. Using a less aggressive blocking strategy, such as removing the optimization for the first iteration of Loop 3 (lines 4-7 in Algorithm A.1), only slightly increases the difficulty for attackers to recover some matrix dimensions. It cannot effectively eliminate the vulnerability.

Another approach is to reduce the dimensions of the matrices. Recall that in both OpenBLAS and MKL, we are unable to precisely deduce the matrix dimensions if they are smaller than or equal to the block size. Existing techniques, such as quantization, can help reduce the matrix size to some degree. This mitigation is typically effective for the last few layers in a convolutional network, which generally use small filter sizes. However, it cannot protect layers with large matrices, such as those using a large number of filters and input activations.

Alternatively, one can use existing cache-based side channel defense solutions. One approach is to use cache partitioning, such as Intel CAT (Cache Allocation Technology) [186]. CAT assigns different ways of the last level cache to different applications, which blocks cache interference between attackers and victims [64]. Further, there are proposals for security-oriented cache mechanisms such as PLCache [69], SHARP [18] and CEASER [68]. If these mechanisms are adopted in production hardware, they can mitigate our attack with moderate performance degradation.

A.10 CONCLUSION

In this chapter, we proposed Cache Telepathy, an efficient mechanism to help obtain a DNN's architecture using the cache side channel. We identified that DNN inference relies heavily on blocked GEMM, and provided a detailed security analysis of this operation. We then designed an attack to extract the matrix parameters of GEMM calls, and scaled this attack to complete DNNs. We used Prime+Probe and Flush+Reload to attack VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective at helping obtain the architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

REFERENCES

[1] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, 1973.

[2] C. Percival, "Cache Missing for Fun and Profit," http://www.daemonology.net/papers/htt.pdf, 2005.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[5] M. Neve and J.-P. Seifert, "Advances on Access-driven Cache Attacks on AES," in *Selected Areas in Cryptography*, 2006.

[6] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[7] J. Szefer, "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses," *Journal of Hardware and Systems Security*, 2016.

[8] Z. He and R. B. Lee, "How Secure is Your Cache Against Side-channel Attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2017.

[9] F. Yao, M. Doroslovack, and G. Venkataramani, "Are Coherence Protocol States Vulnerable to Information Leakage?" in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[10] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 2018.

[11] Wikipedia, "Amazon Elastic Compute Cloud," https://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud, 2018.

[12] Wikipedia, "Google Cloud Platform," https://en.wikipedia.org/wiki/Google_Cloud_Platform, 2018.

[13] Wikipedia, "Microsoft Azure," https://en.wikipedia.org/wiki/Microsoft_Azure, 2018.

[14] Wikipedia, "VMware," https://en.wikipedia.org/wiki/VMware, 2018.

[15] Wikipedia, "Docker (software)," https://en.wikipedia.org/wiki/Docker_(software), 2018.

[16] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *Proceedings of the 16th ACM conference on Computer and communications security.* ACM, 2009.

[17] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical Mitigations for Timing-based Side-channel Attacks on Modern X86 Processors," in *30th IEEE Symposium on Security and Privacy.* IEEE, 2009.

[18] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA).* ACM, 2017.

[19] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[20] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "SecDir: A Secure Directory to Defeat Directory Side-channel Attacks," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[21] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 2018.

[22] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures," in *29th USENIX Security Symposium*, 2020.

[23] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *IEEE Symposium on Security and Privacy (SP).* IEEE, 2013.

[24] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High Performing Cache Hierarchies for Server Workloads: Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[25] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "NCID: A Non-inclusive Cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies," in *Proceedings of the 7th ACM International Conference on Computing Frontiers.* ACM, 2010.

[26] Intel, "Intel Xeon Processor Scalable Family Technical Overview," https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, 2017.

[27] Intel, “6th Gen Intel Core X-Series Processor Family Datasheet - 7800X, 7820X, 7900X,” https://www.intel.com/content/www/us/en/products/processors/core/6th-gen-x-series-datasheet-vol-1.html, 2017.

[28] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cross Processor Cache Attacks,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016.

[29] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of Res. and Dev.*, 1967.

[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[31] M. Johnson, *Superscalar Microprocessor Design*. Prentice Hall Englewood Cliffs, 1991.

[32] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[33] K. Gharachorloo, A. Gupta, and J. Hennessy, “Two Techniques to Enhance the Performance of Memory Consistency Models,” in *International Conference on Parallel Processing (ICPP)*, 1991.

[34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “x86-TSO: a Rigorous and Usable Programmer’s Model for x86 Multiprocessors,” *Communications of the ACM*, 2010.

[35] I. SPARC International, *The SPARC Architecture Manual Version 8*. Prentice Hall, 1992.

[36] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors,” in *Proceedings of the 17th annual international symposium on Computer Architecture (ISCA)*, 1990.

[37] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “ARMageddon: Cache Attacks on Mobile Devices,” in *25th USENIX Security Symposium*, 2016.

[38] D. A. Osvik, A. Shamir, and E. Tromer, “Cache Attacks and Countermeasures: The Case of AES,” in *Cryptographers’ Track at the RSA conference*. Springer, 2006.

[39] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant Side-channel Attacks in PaaS Clouds,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[40] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and Its Application to AES,” in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015.

[41] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM, 2015.

[42] D. M. Gordon, "A Survey of Fast Exponentiation Methods," *Journal of Algorithms*, 1998.

[43] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games—Bringing Access-based Cache Attacks on AES to Practice," in *IEEE Symposium on Security and Privacy (SP).* IEEE, 2011.

[44] B. Gülmezoğlu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A Faster and More Realistic Flush+Reload Attack on AES," in *International Workshop on Constructive Side-Channel Analysis and Secure Design.* Springer, 2015.

[45] J. Bonneau and I. Mironov, "Cache-collision Timing Attacks against AES," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2006.

[46] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches," in *24th USENIX Security Symposium*, 2015.

[47] T. Hornby, "Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD," in *BackHat*, 2016.

[48] Y. Yarom and K. Falkner, "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *23rd USENIX Security Symposium*, 2014.

[49] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache Storage Channels: Alias-driven Attacks and Verified Countermeasures," in *IEEE Symposium on Security and Privacy (SP).* IEEE, 2016.

[50] 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 (CCS).* ACM, 2016.

[51] D. Gruss, C. Maurice, and K. Wagner, "Flush+Flush: A Stealthier Last-Level Cache Attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[52] Intel, *The Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel, 2016, vol. 2A: Instruction Set Reference A-Z.

[53] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[54] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read Arbitrary Memory Over Network," *arXiv preprint arXiv:1807.10535*, 2018.

[55] Intel, "Q2 2018 Speculative Execution Side Channel Update," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html, 2018.

[56] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *Proceedings of the 12th USENIX Conference on Offensive Technologies (WOOT)*, 2018.

[57] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests," in *IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO)*, 2018.

[58] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," *arXiv preprint arXiv:1903.01843*, 2019.

[59] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[60] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, 2015.

[61] 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.* ACM, 2009.

[62] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection," in *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, 2016.

[63] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[64] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[65] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud," in *USENIX Security Symposium*, 2012.

[66] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks," *IEEE Micro*, 2016.

[67] Z. Wang and R. B. Lee, "A Novel Cache Architecture with Enhanced Performance and Security," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

[68] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[69] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of the 34th annual international symposium on Computer Architecture (ISCA)*. ACM, 2007.

[70] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014.

[71] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks," *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[72] W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *Journal of Computer Security*, 1992.

[73] M. Chiappetta, E. Savas, and C. Yilmaz, "Real Time Detection of Cache-based Side-channel Attacks Using Hardware Performance Counters," *Applied Soft Computing*, 2015.

[74] M. Payer, "HexPADS: A Platform to Detect "Stealth" Attacks," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016.

[75] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014.

[76] M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[77] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 2006.

[78] Intel, "Speculative Execution Side Channel Mitigations," https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[79] Arm, "Cache Speculation Side-channels," https://developer.arm.com/support/arm-security-updates/speculativeprocessor-vulnerability/download-the-whitepaper, 2018.

[80] P. Turner, "Retpoline: a Software Construct for Preventing Branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[81] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-Overhead Defense Against Spectre Attacks via Binary Analysis," *arXiv preprint arXiv:1807.05843*, 2018.

[82] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," *Engineering Secure Software and Systems*, 2017.

[83] Y. Cheng, Z. Chen, Y. Zhang, Y. Ding, and T. Wei, "Oh No! KPTI Defeated Unauthorized Data Leakage is Still Possible," in *BlackHat Asia*, 2019.

[84] 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," *arXiv preprint arXiv:1902.05178*, 2019.

[85] 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," *arXiv preprint arXiv:1806.05179*, 2018.

[86] 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 (MICRO)*, 2019.

[87] 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019.

[88] 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 (ISCA)*, 2019.

[89] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019.

[90] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "ConTExT: Leakage-Free Transient Execution," *arXiv preprint arXiv:1905.09100*, 2019.

[91] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[92] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, “SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[93] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative Taint Tracking: A Comprehensive Protection for Speculatively Accessed Data,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[94] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, “MI6: Secure Enclaves in a Speculative Out-of-Order Processor,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[95] K. E. Fletcher, W. E. Speight, and J. K. Bennett, “Techniques for Reducing the Impact of Inclusion in Shared Network Cache Multiprocessors,” Rice University, Tech. Rep., 1995.

[96] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr, and J. Emer, “Achieving Non-inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[97] Y. Xie and G. H. Loh, “PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[98] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, “Adaptive Insertion Policies for Managing Shared Caches,” in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008.

[99] M. K. Qureshi and Y. N. Patt, “Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[100] W. Liu and D. Yeung, “Using Aggressor Thread Information to Improve Shared Cache Management for CMPs,” in *The 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2009.

[101] M. M. Godfrey, “On the Prevention of Cache-based Side Channel Attacks in a Cloud Environment,” M.S. thesis, Queen’s University, 2013.

[102] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, “The Stanford DASH Multiprocessor,” *Computer*, 1992.

[103] R. Singhal, “Inside Intel Next Generation Nehalem Microarchitecture,” in *Hot Chips*, 2008.

[104] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou, "Hardware-software Integrated Approaches to Defend against Software Cache-based Side Channel Attacks," in *15th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2009.

[105] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *25th USENIX Security Symposium*, 2016.

[106] D. Kumar, C. Yashavant, B. Panda, and V. Gupta, "How Sharp is SHARP?" in *The USENIX Workshop on Offensive Technologies (WOOT)*, 2019.

[107] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A Full System Simulator for Multicore X86 CPUs," in *Proceedings of the 48th Design Automation Conference (DAC)*. ACM, 2011.

[108] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," in *Soviet Physics Doklady*, 1966.

[109] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[110] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[111] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-driven Simulation – A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites," http://www.jaleels.org/ajaleel/workload, 2010.

[112] VMware, "Transparent Page Sharing: New Default Setting," http://blogs.vmware.com/security/2014/10, 2014.

[113] 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.

[114] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-level Cache Complex Addressing Using Performance Counters," in *Research in Attacks, Intrusions, and Defenses*. Springer, 2015.

[115] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors," in *Proceedings of the 2015 Euromicro Conference on Digital System Design (DSD)*, 2015.

[116] W. V. Quine, "The Problem of Simplifying Truth Functions," *The American Mathematical Monthly*, 1952.

[117] D. Molka, D. Hackenberg, and R. Schöne, "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer," in *Proceedings of the Workshop on Memory Systems Performance and Correctness.* ACM, 2014.

[118] M. Clark, "A New X86 Core Architecture for the Next Generation of Computing," in *Hot Chips Symposium.* IEEE, 2016.

[119] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer et al., "Zen: A Next-generation High-performance X86 Core," in *IEEE International Solid-State Circuits Conference (ISSCC).* IEEE, 2017.

[120] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE transactions on computers*, 1978.

[121] A. Gupta and W.-D. Weber, "Cache Invalidation Patterns in Shared-memory Multiprocessors," *IEEE Transactions on Computers*, 1992.

[122] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[123] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *International Conference on Parallel Processing*, 1990.

[124] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[125] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo Directory: A Scalable Directory for Many-core Systems," in *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2011.

[126] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997.

[127] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE micro*, 2016.

[128] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies.* ACM, 2014.

[129] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[130] A. Seznec and F. Bodin, "Skewed-Associative Caches," in *International Conference on Parallel Architectures and Languages Europe (PARLE).* Springer, 1993.

[131] Wikipedia, "Advanced Encryption Standard," https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, 2019.

[132] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.

[133] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP).* IEEE, 2013.

[134] Intel, "Q3 2018 Speculative Execution Side Channel Update," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018.

[135] Intel, "Lazy FP State Restore," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html, 2018.

[136] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[137] G. Bell and M. Lipasti, "Deconstructing Commit," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[138] H. W. Cain and M. H. Lipasti, "Memory Ordering: A Value-based Approach," in *Proceedings of the 31st annual international symposium on Computer architecture (ISCA)*, 2004.

[139] Intel, "Intel Analysis of Speculative Execution Side Channels White Paper," https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[140] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-Speculative Load-Load Reordering in TSO," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[141] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "CACTI 5.1," Advanced Architecture Laboratory HP Laboratories, Tech. Rep., 2008.

[142] Intel, "Intel Software Guard Extensions Programming Reference," https://software.intel.com/en-us/sgx/sdk, 2013.

[143] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling Microarchitectural Replay Attacks," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA).* ACM, 2019.

[144] R. B. Lee, *Security Basics for Computer Architects.* Morgan & Claypool Publishers, 2013.

[145] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, 2009.

[146] J. Yu, L. Hsiung, M. El'Hajj, and C. W. Fletcher, "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing," *The Network and Distributed System Security Symposium (NDSS)*, 2019.

[147] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey et al., "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv preprint arXiv:1609.08144*, 2016.

[148] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

[149] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv preprint arXiv:1511.06434*, 2015.

[150] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, 2016.

[151] Amazon, "Machine Learning on AWS Putting Machine Learning in the Hands of Every Developer," https://aws.amazon.com/machine-learning/, 2019.

[152] Google, "AI and Machine Learning Products," https://cloud.google.com/products/ai/, 2019.

[153] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing Machine Learning Models via Prediction APIs," in *USENIX Security Symposium*, 2016.

[154] R. Shokri, M. Stronati, and V. Shmatikov, "Membership Inference Attacks Against Machine Learning Models," in *IEEE Symposium on Security and Privacy (SP).* IEEE, 2017.

[155] Y. Long, V. Bindschaedler, L. Wang, D. Bu, X. Wang, H. Tang, C. A. Gunter, and K. Chen, "Understanding Membership Inferences on Well-Generalized Learning Models," *arXiv preprint arXiv:1802.04889*, 2018.

[156] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro et al., "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[157] Amazon, "Amazon SageMaker ML Instance Types," https://aws.amazon.com/sagemaker/pricing/instance-types/, 2018.

[158] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, 2015.

[159] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[160] B. Wang and N. Z. Gong, "Stealing Hyperparameters in Machine Learning," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[161] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A Placement Vulnerability Study in Multi-tenant Public Clouds," in *24th USENIX Security Symposium*, 2015.

[162] Z. Xu, H. Wang, and Z. Wu, "A Measurement Study on Co-residence Threat Inside the Cloud," in *24th USENIX Security Symposium*, 2015.

[163] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel, "Malicious Co-residency on the Cloud: Attacks and Defense," in *IEEE Conference on Computer Communications*. IEEE, 2017.

[164] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. M. Marvel, "Catch Me if You Can: A Closer Look at Malicious Co-Residency on the Cloud," *IEEE/ACM Transactions on Networking*, 2019.

[165] P. D. Ezhilchelvan and I. Mitrani, "Evaluating the Probability of Malicious Co-residency in Public Clouds," *IEEE Transactions on Cloud Computing*, 2017.

[166] Amazon, "Amazon SageMaker: Machine Learning for Every Developer and Data Scientist." https://aws.amazon.com/sagemaker/, 2018.

[167] Google, "Cloud ML Engine Overview," https://cloud.google.com/ml-engine/docs/technical-overview, 2018.

[168] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "TensorFlow: A System for Large-scale Machine Learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[169] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional Architecture for Fast Feature Embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia.* ACM, 2014.

[170] Apache, “Apache MXNet,” https://mxnet.apache.org/, 2018.

[171] Z. Xianyi, W. Qian, and Z. Chothia, “OpenBLAS: An Optimized BLAS Library,” http://www.openblas.net/, 2013.

[172] Eigen, “Eigen is a C++ Template Library for Linear Algebra,” http://eigen.tuxfamily.org/, 2018.

[173] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, “Intel Math Kernel Library,” in *High-Performance Computing on the Intel Xeon Phi.* Springer, 2014.

[174] B. Xu, N. Wang, T. Chen, and M. Li, “Empirical Evaluation of Rectified Activations in Convolutional Network,” *arXiv preprint abs/1505.00853*, 2015.

[175] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems*, 2012.

[176] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *International Conference on Learning Representations (ICLR)*, 2015.

[177] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

[178] F. G. Van Zee and R. A. Van De Geijn, “BLIS: A Framework for Rapidly Instantiating BLAS Functionality,” *ACM Transactions on Mathematical Software (TOMS)*, 2015.

[179] AMD, “Core Math Library (ACML),” https://developer.amd.com/amd-aocl/amd-math-library-libm/, 2012.

[180] K. Goto and R. A. v. d. Geijn, “Anatomy of High-performance Matrix Multiplication,” *ACM Trans. Math. Softw.*, 2008.

[181] R. O’Neill, *Learning Linux Binary Analysis.* Packt Publishing Ltd., 2016.

[182] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based Learning Applied to Document Recognition,” *Proceedings of the IEEE*, 1998.

[183] F. Chollet, “Keras,” https://github.com/fchollet/keras, 2015.

[184] Theano Development Team, “Theano: A Python Framework for Fast Computation of Mathematical Expressions,” *arXiv preprint abs/1605.02688*, 2016.

[185] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX,” in *26th USENIX Security Symposium*, 2017.

[186] Intel, “Improving Real-Time Performance by Utilizing Cache Allocation Technology,” https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, 2015.