Do not Forget Hardware Prefetching When Designing Secure Cache Systems

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

Sumon Nath
21Q050007
sumon@cse.iitb.ac.in

Supervisor:
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2021-2024

Thesis Approval

This thesis entitled

Do not Forget Hardware Prefetching When Designing Secure Cache Systems

by

Sumon Nath
Roll No.: 21Q050007

is approved for the degree of
Master of Science by Research in Computer Science and Engineering

.....	.....	.....
Prof. Biswabandan Panda (Supervisor)	Prof. Mythili Vutukuru (Examiner)	Prof. Manas Thakur (Examiner)

.....	.....
Prof. R. Govindarajan (Examiner)	Prof. Mythili Vutukuru (Chairperson)

Date: July 3, 2024
Place: IIT Bombay

Declaration

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

_____

Sumon Nath
21Q050007

Date: July 3, 2024

Nomenclature

APKI	Accesses Per Kilo Instructions
BTB	Branch Target Buffer
DDR	Double Data Rate
DRAM	Dynamic Random-Access Memory
GHR	Global History Register
GM	GhostMinion
ILP	Instruction Level Parallelism
IP	Instruction Pointer
ISA	Instruction Set Architecture
L1D	Level 1 Data (Cache)
L2C	Level 2 Cache
LLC	Last Level Cache
LQ	Load Queue
LRU	Least Recently Used
MLP	Memory-Level Parallelism
MPKI	Misses Per Kilo Instruction
MSHR	Miss Status Holding Register
MT/s	MegaTransfers per Second
PHT	Pattern History Table
PQ	Prefetch Queue

ROB Reorder Buffer

RSA Rivest–Shamir–Adleman (encryption algorithm)

SDRAM Synchronous Dynamic Random-Access Memory

SGX Software Guard Extensions

STLB Second-Level Translation Lookaside Buffer

SUF Secure Update Filter

TAGE TAgged GEometric history length predictor

TLB Translation Lookaside Buffer

TSB Timely Secure Berti

VA Virtual Address

VM Virtual Memory

X-LQ eXtended Load Queue

Abstract

Speculative execution attacks like Spectre and its variants can cause information leakage through a cache hierarchy. Mitigation techniques have been proposed to ensure cache system does not cause information leakage through speculative side channels. GhostMinion is one of the state-of-the-art mitigation techniques that ensures strictness ordering mitigating speculative attacks through a cache system. Similar to a cache system, hardware prefetchers can also cause speculative information leakage. To mitigate it, GhostMinion advocates on-commit prefetching on top of strictness ordering in the cache system. Our experiments show that the GhostMinion cache system when coupled with hardware prefetching leads to redundant traffic between different levels of cache causing contention leading to performance loss. Next, we observe that prefetching on commit leads to performance loss as it affects the prefetcher timeliness.

We perform a thorough analysis of state-of-the-art prefetching techniques on a secure cache system. Then, we propose two microarchitectural solutions that ensure high performance while designing secure prefetchers. The first solution detects and filters redundant traffic when updating the cache hierarchy non-speculatively. The second solution adjusts the timeliness of the prefetcher to compensate for the delayed triggering of prefetch requests at commit, resulting in a secure yet high-performing prefetcher. Our experiments show that our filter consistently improves the performance of secure cache systems like GhostMinion in the presence of state-of-the-art prefetchers (by 1.9% for single-core and 19.0% for multi-core for the top-performing prefetcher). We see a synergistic behavior of the filter with our proposed secure prefetcher, which leads to a further increase in performance by 6.3% and 23.0%, for single-core and multi-core systems, respectively. Our enhancements are extremely lightweight incurring a storage overhead of 0.59 KB per core.

Contents

Chapter 1

Introduction

Speculative execution attacks, pioneered by Spectre [43] but followed rapidly by other attacks [19, 22, 13, 55, 23] take advantage of the cache state affected by *transient* instructions. Transient instructions are speculative instructions that do not commit. Speculative execution is a fundamental technique used by high-performance processors, and hence, it cannot be disabled in pursuit of security. To mitigate the speculative execution attacks that exploit the cache, various proposals [77, 15, 41, 14, 60, 61, 59, 75, 80] strive to provide security with minimal performance loss. In general, there are two kinds of mitigation techniques proposed in the literature: delay-based and invisible speculation. In delay-based approaches, the *transmission* of secret-dependent values is stalled until it is considered *safe* to proceed. The determination of safety can be complex and requires sophisticated mechanisms to accurately identify when an instruction can be considered safe for execution. In invisible speculation, secret-dependent loads are permitted to execute. However, the effects of these executions are concealed from the cache hierarchy and other microarchitectural structures, instead of postponing the instruction altogether.

Among all the proposals, GhostMinion [15], Speculative Taint Tracking (STT) [80], and Non-speculative Data Access (NDA) [75] are the strictest as they mitigate backward-in-time attacks such as speculative interference attacks [19]. Between STT, NDA, and GhostMinion, Ghostminion is the lightweight and high-performing mitigation technique. GhostMinion is an invisible speculation technique that enforces a strictness ordering that ensures the mitigation of varieties of speculative execution attacks through the cache system: cache hierarchy, miss status holding registers (MSHRs), and hardware prefetchers [14]. GhostMinion uses a small speculative cache (GM) that stores the data corresponding to speculative loads, and when a load commits, the data is communicated to L1D. When the same data is evicted from L1D,

the data is communicated to the L2, and on eviction from L2, the data is communicated to the LLC. On average, GhostMinion incurs a performance loss of around 5% compared to a non-secure cache system.

Data prefetchers are important in improving cache performance by converting cache misses into hits. Recent advances in data prefetchers have pushed the limit of single-thread performance with average performance boosts of 3% to 5% [20, 17, 53, 51]. In the last decade, two ISCA championships on data prefetching [2, 6] have helped in this trend. Unfortunately, hardware prefetchers, which are trained and triggered on speculative loads, can also be used as a source of information leakage even on a secure cache system [14, 15]. A speculative attack using prefetchers works as follows: (i) The attacker primes the cache; The corresponding cache has a prefetcher; (ii) The victim loads secret data similar to the Spectre attack; (iii) The speculative load generated by the victim trains and triggers the hardware prefetcher; (iv) The prefetcher request data as per its address prediction that comes to the cache; (v) Finally, the attacker probes the cache.

GhostMinion makes a case for secure hardware prefetching through *on-commit* prefetching: A secure prefetcher should be trained on commit and prefetching should only happen on commit. This way the prefetcher will not affect the cache and MSHR state speculatively, and transient instructions cannot exploit the prefetcher for information leakage. We show that data prefetching can indeed alleviate the performance loss of a secure cache system. Despite the importance of data prefetching, no detailed study has been carried out about the impact of secure prefetching techniques.

We analyze for the first time the interaction between a wide range of state-of-the-art hardware prefetchers and a high-performing secure cache system. We evaluate IP-stride [16], the well-known prefetcher used in industry, Bingo [17], SPP+PPF [20], IPCP [53] (winner of the 3rd data prefetching championship [6]), and Berti [51], on a secure cache system like GhostMinion. Berti is the state-of-the-art L1D prefetcher (with an accuracy of almost 90%) that orchestrates its requests across the cache hierarchy. We discover that prefetchers interact negatively with secure cache systems like GhostMinion. We find that two main factors prevent them from reaching their optimal performance and propose microarchitectural solutions to overcome them.

Our observations. First, we analyze the performance improvements of the evaluated prefetchers both on a non-secure cache system and a secure cache system like GhostMinion for both SPEC CPU 2017 and GAP workloads (see Section 6.1 for simulation details). We observe that prefetching techniques improve performance both for secure and non-secure

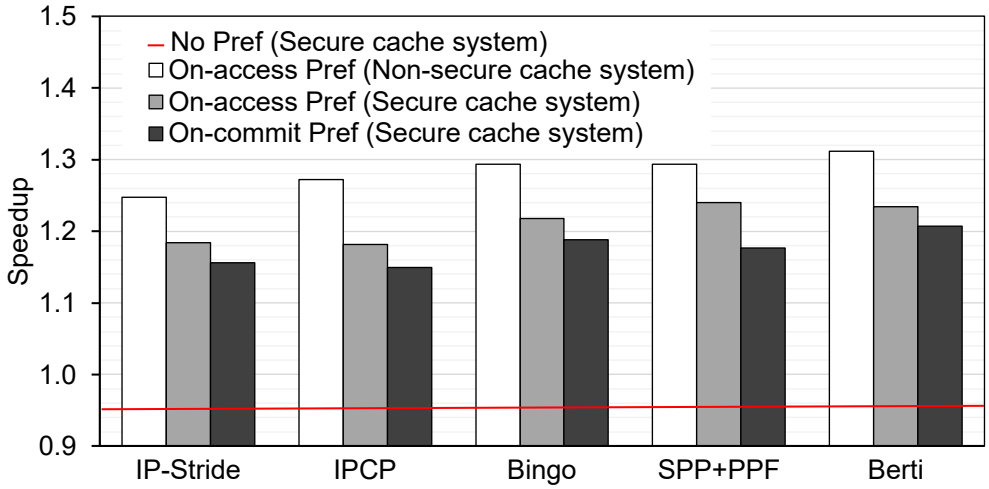


Figure 1.1: Speedup of state-of-the-art prefetchers normalized to a non-secure cache system with no prefetching.

cache systems, but the gap between them is high. The performance gap is because of an increase in memory access latency due to additional memory traffic introduced to update the cache hierarchy with invisible loads. On average, GhostMinion introduces additional traffic of more than 1.5× to L1D when compared to a non-secure cache system with hardware prefetchers (Section 4).

Next, we analyze the impact of implementing a secure prefetcher on a secure cache system, that is, the impact of training and prefetching on-commit, instead of on-access. Figure 1.1 shows the performance improvements obtained by our prefetchers in a secure cache system when they are trained and triggered on-access (second bar) and on-commit (third bar). We observe a consistent performance loss of 3%-4% for all prefetchers with training/prefetching on-commit compared to prefetching on-access. We find that the key factor is timeliness, not the inability to capture the applications' access patterns (Section 4). A major part of the performance loss for on-commit prefetching is due to a new class of late prefetch requests which we coin as "commit-late": misses whose prefetching had not been initiated when the processor requested the data, but that would have been initiated if the prefetch request had been triggered on access. In summary, on average, compared to on-access prefetching on a non-secure cache system, we see a performance loss of around 10% with on-commit prefetching on a secure cache system.

Our contributions. We shed some light on the reasons behind low-performance secure prefetchers and propose a low-cost yet effective solution to recover the performance loss and enable the full potential of secure prefetching, closing the gap to non-secure cache systems.

We bring the following contributions:

- We show that prefetchers lose relative performance on a secure cache system due to (i) the additional memory traffic introduced by the secure cache system and (ii) prefetch timeliness issues (Chapter 4).
- We propose a mechanism to filter out the superfluous non-speculative updates performed in a secure cache system. Our filter is lightweight and incurs a storage overhead of 0.12KB (Chapter 5).
- We propose a mechanism to ensure the timeliness of a prefetcher that is trained and issues prefetch requests at commit, making a case for a secure yet timely and high-performing prefetcher. The end result is the first high-performance secure prefetcher with a storage overhead of 0.47KB (Chapter 5).
- We show that our enhancements on top of the state-of-the-art prefetcher helps in bridging the performance gap between a non-secure cache system and a secure cache system. For SPEC CPU2017 and GAP benchmarks, our enhancements improve performance by 6.3% (our filter contributes to around 30% of the improvement and the rest comes from the better-trained on-commit prefetcher). For a 4-core system, our mechanisms improve performance by 23.0% over on-commit state-of-the art prefetcher in a secure cache system (Chapter 6).

Chapter 2

Background

This chapter discusses all the necessary background information required to understand the rest of the document. We discuss the various performance optimization techniques used to improve processor performance to date, as well as the security vulnerabilities they expose. Modern processors employ many techniques to get high performance from the available silicon. It is not just the increasing clock speed that has led to the high performance, but the ability to pump out multiple instructions in a single clock cycle while maintaining the program's correctness has led to performance improvement by numerous folds. Techniques like out-of-order execution, speculative execution, multi-threading, caching, and hardware prefetching are some key contributors.

The traditional approach of increasing clock speeds has reached its limits. Hence, strategies to enhance the processor's overall throughput are now employed. Out-of-order and speculative execution enable processors to execute instructions concurrently, reducing idle time and enhancing efficiency. Multi-threading allows for the parallel execution of multiple threads, further leveraging the processing power of modern CPUs. Caching, a fundamental aspect of processor design, involves the use of high-speed memory to store frequently accessed data and instructions closer to the execution units. This minimizes the need for repeated access to slower main memory, accelerating processing speed. Hardware prefetching, another optimization technique, involves predicting and preloading data into the cache before it is explicitly requested, reducing latency and improving overall responsiveness.

However, this singular approach of improving performance over the past few decades without critically analyzing their side effects has led to an insecure foundation. Before delving into the security implications, a thorough exploration of these performance-boosting techniques is imperative. By understanding the intricacies of how modern processors achieve

their impressive speeds, we can better appreciate the trade-offs and challenges that arise in security.

2.1 Instruction Level Parallelism

Running instructions one after the other does not fully utilize the available hardware. Pipelining enables instructions to overlap, enabling them to execute[1] simultaneously, improving the throughput of processors. Such parallelism in executing instructions is called instruction-level parallelism (ILP). However, different kinds of hazards restrict the level of parallelism by causing stalls. Various software and hardware techniques are used to minimize the stall caused by such hazards. Hardware techniques include data forwarding and bypassing, out-of-order execution, hardware speculation, memory disambiguation, etc., whereas software ideas include compiler optimizations like loop unrolling, dependence analysis, and scheduling. We narrow our discussion to out-of-order execution and speculative execution, which are relevant to this work.

2.1.1 Out-of-order Execution

Modern processors have multiple execution units. Multiple instructions get pushed into the execution stage and executed in parallel utilizing the execution units, which leads to ILP. However, in an in-order pipeline, instructions dependent on the previous instructions are stalled, which stalls every instruction following it. On the other hand, in an out-of-order pipeline, independent instructions that enter the pipeline in order are executed out-of-order to maximize utilization of all the execution units, increasing ILP. The blocking instructions get bypassed, and the non-blocking instructions execute proactively. Figure 2.1 shows that I4 can start execution as its operands are available, even if I2 is stalled due to a data dependency. I2 waits till its operands are ready. But this does not come for free.

First, it is necessary to respect data dependencies while keeping program order in mind. It should not happen that an older instruction uses the data of a younger instruction. As shown in Figure 2.1, I3 should get the value of R2 from the result of I2 and not I4, even if I4 completes execution first. **Register renaming** solves this by renaming all the architectural registers to temporary registers to preserve data dependencies. I2 is renamed to store its result in T2 instead of R4. I3's operands are renamed from R4 to T2. I4 is renamed to store

[1]Note that this is not the execute stage, rather the execution of instruction as a whole

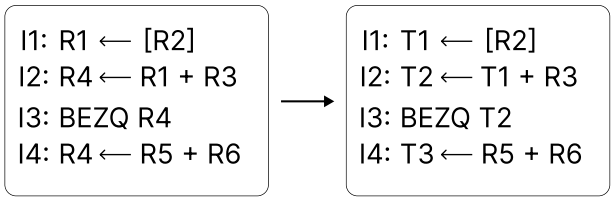


Figure 2.1: Register renaming in action

its results in T3 instead of R4. This preserves the dependency of I3 to I2 for the value of R4.

The second problem is that of **imprecise exception**, where, due to out-of-order completion of instructions, the processor cannot restore its state just before the exception occurs. Such cases also happen in case of a **branch mis-predictions** (discussed in Section 2.1.2). So, there is a need for a check-pointing mechanism to help roll back the effects of instructions following exception-generating instructions. Such a rollback is possible by storing the state of the processor in every instruction. But as it sounds, it will be very inefficient, and hence, there is a need for an efficient approach. This is where the **Reorder buffer**, in short **ROB**, comes into play. The ROB, along with supporting microarchitecture units, stores all instructions in program order and the associated values. On completion of execution, updates are made to the ROB and not copied to the register file immediately. The results are also broadcasted to instructions that are waiting for them.

With the reorder buffer, a new concept is introduced called **instruction commit**, where instructions move their results from temporary registers into the architectural register. That is, an instruction that has completed execution stores its values in the ROB till all the instructions before it commits. Afterward, the results from ROB are copied to the register file. This easily handles the problem of imprecise exceptions. In case of an exception or a branch mis-prediction, the processor flushes the ROB. All instructions before the exception point are committed and would have already reflected changes made to the register file.

Executed instructions at the head of the ROB commits and are removed from the ROB as shown in Figure 2.2. However, instructions at the head of the ROB can halt the following instructions from getting out of the pipeline till the time it commits. Even while the pipeline is stopped due to a long latency instruction, the following instructions can complete their execution and update the ROB entries as required. This allows bursts of completed instructions to commit when they reach the head of the ROB, resulting in improved ILP. We abstract out some of the things related to the ROB. It is not just the ROB that enables

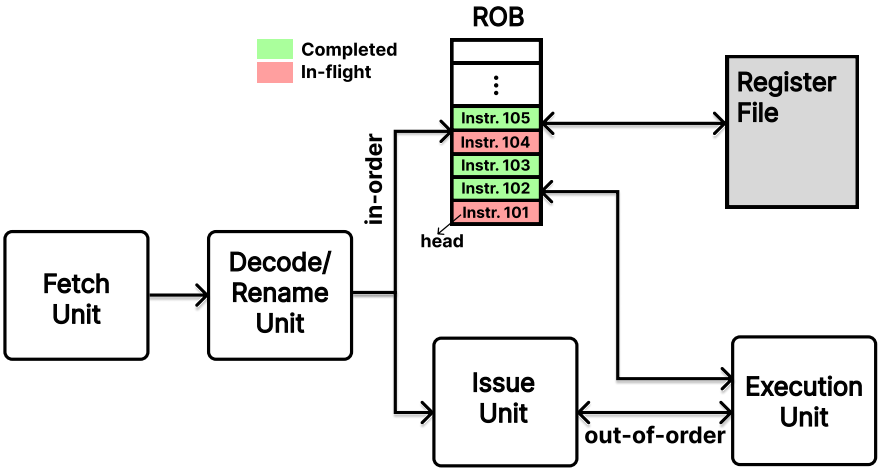


Figure 2.2: Out-of-order processor

out-of-order execution. Along with the ROB, we use structures like Issue Queue, reservation stations, and register rename map tables to enable out-of-order execution. However, it is out of the scope of this thesis.

2.1.2 Speculative Execution

The processor may not always anticipate the future instruction stream. This occurs when a conditional branch instruction is encountered in the pipeline. In such cases, processors guess the most probable path the branch might take using a structure called the **branch predictor** and start fetching and executing instructions along the predicted path. These instructions are called **speculative instructions**. When the branch eventually resolves, the processor either accepts the changes made by the speculative instruction stream or rejects and rolls them back.

In the best case, if the predicted path is correct, the results of the speculatively executed instructions are committed, and execution continues as usual. If the prediction is wrong, the processor restores its state to the one just before executing the conditional branch using the help of the in-order commit mechanism discussed in section 2.1.1. This way, in the best-case scenario, the processor can execute instructions ahead of time, resulting in high performance. In the worst case, it flushes out the entire pipeline of wrongly predicted instructions without affecting the correctness of the program. Such wrong path instructions are termed as **transient instructions**.

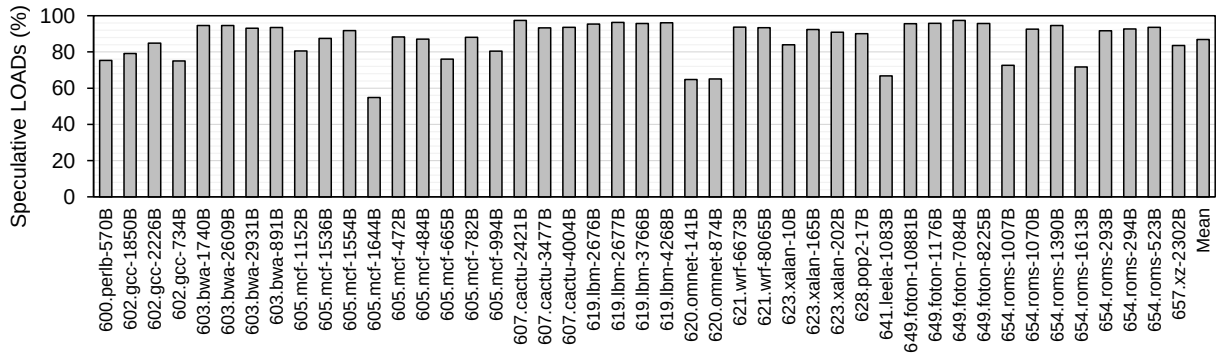


Figure 2.3: Percentage of speculative load instructions in the SPECCPU 2017 benchmarks. We use the hashed-perceptron branch predictor and collect statistics for 200M sim-point instructions after a 50M-instruction warm-up.

Branch predictor: As discussed above in Section 2.1.2, processors speculatively execute instructions rather than waiting for branches to resolve, using a structure called the branch predictor. It uses a structure called the **direction predictor** to predict the direction of a branch instruction. Modern branch predictors are very sophisticated and use the history of previous branch outcomes to learn and predict the outcome of a newly encountered branch. These predictors can predict both indirect and direct branch outcomes. The branch predictor uses another structure called the **branch target buffer (BTB)**, which buffers all the target addresses referred to by previous branch instructions. It uses the BTB to predict target addresses for branches where the target address is unknown until branch resolution. Most branch predictors in modern processors use neural [36, 39, 37] or TAGE-based[64, 63] techniques, yielding high prediction accuracy. Most branch predictors in modern systems have a prediction accuracy of more than 95%.

Interestingly, almost all instructions are executed speculatively, as shown in the [60]. Any instruction that causes an exception can lead to speculative execution. Apart from control instructions, loads/stores cause exceptions, too. A load or store accessing an unmapped memory region leads to a page fault; access to an unauthorized memory region leads to an exception. In addition to this, coherency protocols can cause exceptions in the form of invalidation requests. Apart from these, floating point operations and integer divisions can also cause exceptions. All such instructions are termed as **shadow casting** instructions and cast a **speculative shadow** to the following in-order instructions. All instructions under the speculative shadow are speculative. Figure 2.3 shows that, on average, 88% loads are speculative in the SPEC CPU2017 benchmark suite.

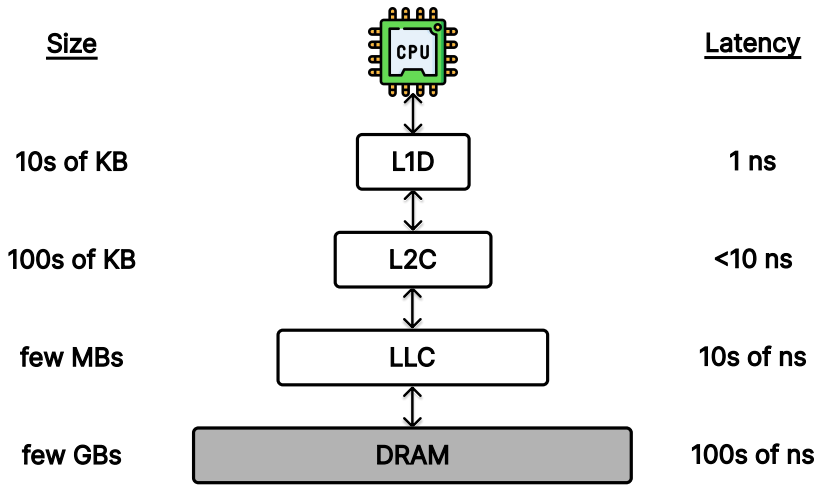


Figure 2.4: Memory hierarchy. As we move higher in the memory hierarchy, size increases; however, speed decreases.

2.2 Memory Hierarchy

Computing in the current age is massively data-driven, which increases the need for fast and large memory units. Most of this data the processor actively uses resides in a DRAM (dynamic random access memory). The ideal scenario for programmers is access to a fast memory unit with unlimited storage. However, DRAM technology has not been able to cope with the rapid increase in processing speed. The latest DRAM chips can provide a bandwidth of around 8800 MT/s (70.4 GB/s) [9, 12]. However, processors can generate memory references with a peak bandwidth of more than 500 GB/s[2]. This massive gap between CPU and memory speeds leads to stalls in the instruction pipeline. This is termed the memory wall problem.

Programs tend to access data (or instruction) that are close together in space or are frequently accessed. For example, an array element is re-referenced multiple times during the execution of a program. This is known as **temporal locality**. **Spatial locality**, on the other hand, means addresses that are closer together spatially are more likely to be used in the near future. These two locality properties have been used extensively to tackle the memory wall problem.

[2]Lastest Intel i7 processor can generate two 64-bit memory references per core; with eight cores operating at 3.4 GHz clock rate, it can generate 54.4 billion data memory references per second along with 27.2 billion instruction references, resulting in a peak bandwidth of 870.4 GB/s.

2.2.1 Cache Hierarchy

Architects proposed to use faster and smaller memory units called caches to store data corresponding to frequently accessed memory addresses that can be accessed faster, matching the speed of CPUs. The cache organizes data into units of blocks. A cache block contains a fixed number of bytes/words. When data is found in the cache, known as a cache hit, it quickly serves it to the processor in a very short span of time; this is called the hit time. However, the faster the cache is, the smaller it is; thus, it can only store a subset of the working set. The working set is the memory used actively by a process in its lifetime. The small cache incurs misses, in which case, the processor probes the DRAM to fetch the block into the cache, incurring a high miss penalty. Architects use the metric called average memory access time to get an overall performance metric of the memory subsystem, which includes the cache hierarchy. The average memory access time is defined as follows:

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

Increasing the cache size decreases the miss rate at the expense of higher hit latency and power consumption. So, the primary dilemma is whether to have larger caches to reduce the miss rate at the expense of higher hit latency or to have a smaller, faster cache that matches the speed of processors. To mitigate this problem, architects add a few levels of cache, which can bridge the gap between the fast and small cache and the large and slow DRAM. The first level cache, or the L1 cache, can match the processor's fast clock cycle, enabling a higher data transfer bandwidth. The second level cache (L2C) can cover most of the misses at the L1 cache with slightly higher latency, thus decreasing the number of accesses going to the DRAM by many folds. Typically, systems use three levels of caches with sizes that increase incrementally, namely L1, L2, and the Last Level Cache (LLC). The cache closest to the processor is typically called the "lowest" cache level, the subsequent levels are "higher" levels, and the one closest to the DRAM is the "highest" cache level. Figure 2.4 shows the typical memory hierarchy employed in a modern system and the typical size-latency of the memory units.

2.2.2 Memory Level Parallelism

A cache miss stalls the processor. As the processor waits for the missing data, it fetches instructions from the instruction cache. A non-blocking cache, sometimes called a lockup-free

cache, permits the data cache to keep delivering cache hits even in the event of a miss. Known as "hit under miss", this optimization actively assists during a miss, lowering the effective miss penalty. Another complex option is for the cache to "miss under miss", reducing the effective miss penalty by overlapping several misses. An out-of-order processor (discussed in section 2.1.1) can issue multiple loads/stores that can be in-flight simultaneously, which may lead to multiple misses. Caches use a special array of registers called the **miss status holding register (MSHR)**, which enables the cache to manage multiple misses concurrently, facilitating memory-level parallelism (MLP). MLP enables the processor to overlap latencies from multiple memory accesses and execute independent instructions, advancing computation within the core.

2.2.3 Data Access Flow

1. The processor indexes the L1 cache in parallel with accessing the Translation look-aside buffer (TLB) to obtain the address translation, which masks the indexing latency if found.

2. When there is a miss in the TLB, the page table walker traverses the page tables to locate the required address translation.

3. A page fault occurs if the page is not found in physical memory, triggering the operating system (OS) trap and retrieve the page from disk to memory and store the translation in the page table and TLB.

4. Upon re-execution of the instruction, if there is a hit in the TLB but a miss in the L1D cache, the processor stores information about the instruction corresponding to the miss in one of the MSHRs (Miss Status Handling Registers) of the L1D cache.

5. Subsequently, if there is a miss in the L2 cache, the processor stores information in one of the MSHRs of the L2 cache. Similarly, in the case of LLC as well.

6. The DRAM controller is requested for the required data. The DRAM controller reads the block from the DRAM and sends it back to the LLC.

7. The cache controller of the LLC receives the block, searches its address in the MSHR, determines the victim block to evict, evicts the block, and fills the incoming block in the LLC.

8. The cache controller writes it back to a higher level cache or memory if the victim block is dirty

9. The cache controller then removes the entry in the MSHR corresponding to the miss and returns the block to the L2 cache.

10. Steps 7-9 are repeated for the L2 cache and then for the L1D cache.

11. Finally, the data requested by the load operation is returned to the processor.

2.2.4 Inclusion Policies

The cache hierarchy can be classified depending on whether a particular cache level contains all, none, or some blocks in its immediate higher-level cache. In an inclusive cache, all blocks are present in its higher-level cache. In an exclusive cache, none of the blocks are present in its higher level. A non-inclusive cache can contain some of the blocks present at the higher level but not necessarily all or none of the blocks. All three inclusion policies have their benefits and problems.

Inclusive Cache Hierarchy. On a cache miss at a particular level, the block is filled into all the higher-level caches as it returns to the lowest cache level. For example, if a request misses all the cache levels, the block is filled into LLC, L2, and L1. In case of eviction at the cache level, the corresponding block is invalidated from all the lower levels. Invalidating blocks in multiple levels of the cache consumes time and power and is one of the major drawbacks of an inclusive cache hierarchy. There is no need for invalidation when evicting a block from the L1 cache. Only a dirty block must be written back to the higher-level cache or memory on eviction. Writeback of clean blocks is unnecessary as the blocks will be in the higher-level cache. Another drawback of the inclusive hierarchy is the amount of data replication across the cache hierarchy. The effective size of the cache hierarchy is the size of the LLC.

Exclusive Cache Hierarchy. Data replication is nullified in an exclusive cache hierarchy. This increases the effective size of the cache hierarchy to the sum of the size of all the caches. On a cache miss, blocks do not fill into the higher levels of cache. For example, on a miss at L1, L2, and LLC, the cache block is bypassed (not filled) from L2, LLC and filled only into L1. Also, in case of a hit in one of the higher-level caches, the block is passed on

to the lower-level cache, and the block is invalidated in the higher-level cache. This avoids data replication. The higher levels of cache acts as a victim cache. On eviction, all blocks are copied to the immediate higher-level cache. Victim caches preserve the evicted blocks of the lower level, reducing conflict misses.

Non-inclusive Cache Hierarchy. A non-inclusive cache is almost similar to an inclusive cache except that there are no invalidations in the lower-level caches on evictions. This simplifies the eviction process and increases the effective size of the cache hierarchy by relaxing the property of inclusion.

2.3 Timing Channel Attacks

Information leakage in computing systems can happen through storage or timing channels. Storage channels try to exploit the functional correctness of software/hardware behavior, i.e., faults that are directly visible to programs through registers or memory addresses. Architects and hardware manufacturers ensure the internal architectural changes are hidden from any external agents, guaranteeing functional correctness and eliminating storage channels. This functional correctness is proven formally in literature [50]. On the other hand, timing channels exploit variability in timing behavior. This variability can depend on a hidden state, which can, in turn, lead to the leakage of sensitive information. Even though hardware designers have successfully hidden the CPU's internal state in terms of functional behavior, the timing behavior is often exposed. Consistency of timing behavior can not be proven formally, making it impossible to guarantee security.

```
1    function exponent(b, e, m):
2        x = 1
3        for i in (|e|-1) to 0:
4            x = x * x        // Square
5            x = x % m        // Reduce
6            if (e[i] = 1):
7                x = x * b    // Multiply
8                x = x % m    // Reduce
9        ret x
```

Listing 2.1: Exponentiation by Square-and-Multiply

In the past decade, researchers proposed attacks that exploit such timing channels [47, 52, 79, 43, 46, 19]. Depending on the threat model, such timing channel exploits are mainly

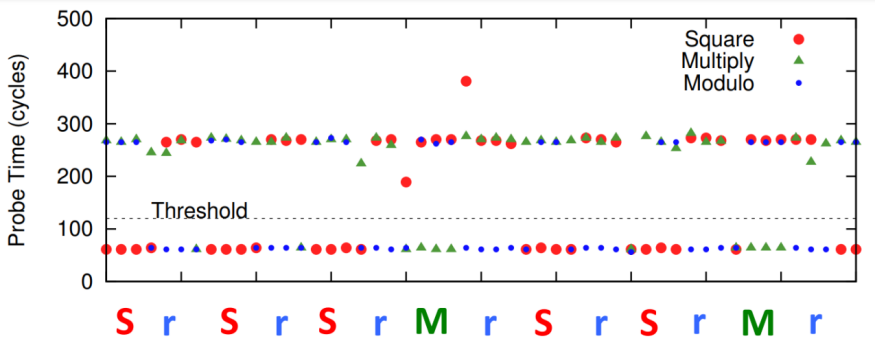


Figure 2.5: Cache access time (probe time) for square, multiply, and modulo functions (adapted from [79])

categorized into covert and side-channel attacks.

A **covert channel** is an intentional communication channel that uses a medium not originally intended for that purpose. For example, the sender can send bit information encoded in the form of memory accesses that lead to controlled eviction of receiver data present in the cache. The receiver can decode the information by timing its memory accesses, wherein a hit or miss in the cache indicates either of the two bits. Although it is essential to mitigate such channels, they do not compromise **confidentiality**, i.e., they do not leak sensitive information from the computer system.

On the other hand, **side channels** compromise the confidentiality of computing systems where an attacker leaks sensitive information from a victim. Similar to covert channels, the transmission medium is an unintended communication channel; however, in a side channel attack, the sender does not intend to share any information.

To understand how a side-channel attack works, let us take a simple example based on the Flush+Reload attack [79], discussed in detail in Section 2.3.1. The victim and attacker share the same crypto library (say, using mmap). Listing 2.1 shows the GnuPG [1] implementation of RSA [57]. e is the secret key the attacker wants to leak. GnuPG uses the square and multiply technique to encrypt the data using the cipher e. Depending on the cipher bits, it does a square-reduce operation if the bit is clear. Otherwise, it does a sequence of square-reduce-multiply-reduce for a set bit. One can easily deduce the secret key by observing the access pattern of code lines, i.e., the sequence of instructions fetched and executed. Like the covert channel, the attacker flushes the code lines of the shared library, waits for the victim to perform the encryption, reloads the code lines, and times its access. Figure 2.5 shows the probe times of the attacker. The code lines having probe time less than the threshold

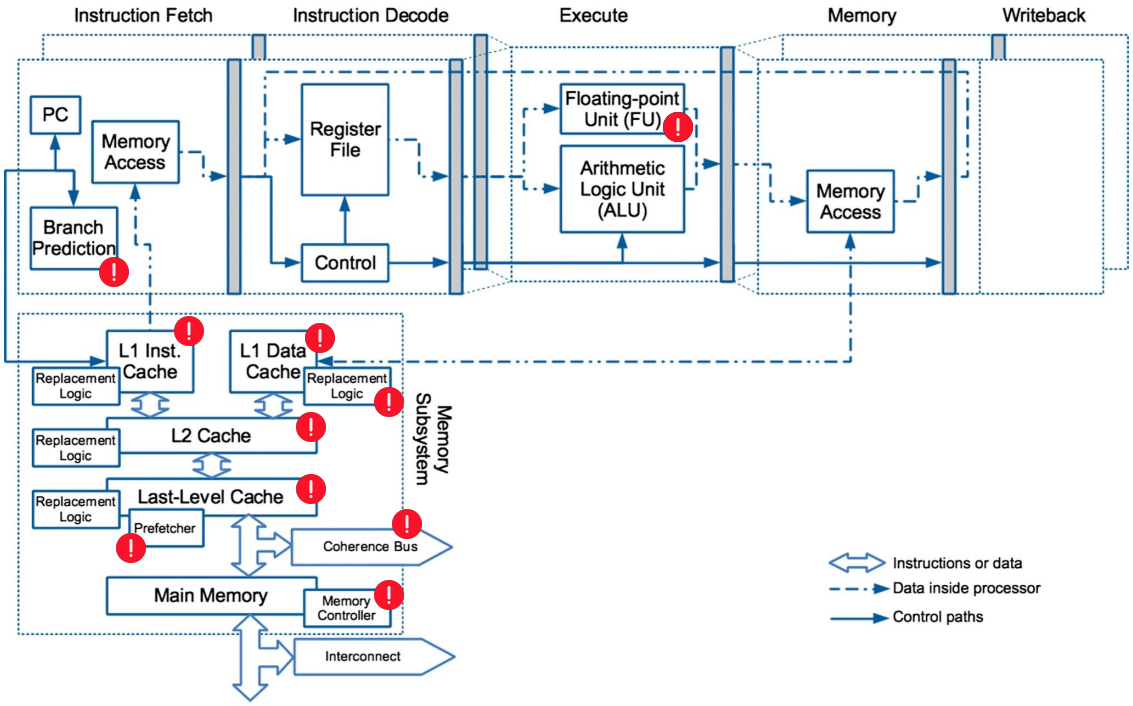


Figure 2.6: Sources of timing channels in a processor (adapted from [72])

indicate access by the victim. The attacker can recover the sequence of operations using the probe times, hence leaking the secret key. For this thesis, we focus on side channels where the victim is unaware of any confidentiality breach. Next, we look into different sources in microarchitecture that can lead to a timing channel, as shown in Figure 2.6.

- **Instruction with Different Execution Timing.** Programs use a lot of different kinds of instructions having different execution times. The type of instructions executed is often secret-dependent, which changes the program execution time on different runs, making it vulnerable to secret leaks.

- **Contention for Functional Units.** Attackers can create contention at shared functional units, which changes the timing of the victim using the shared functional unit, leading to a timing channel.

- **Stateful Functional Units.** Many functional units in the processor store some history for making predictions to boost performance. Such functional units are often shared and can be used to leak the history stored in them by observing their predictions.

- **Memory Hierarchy.** Timing differences of accesses to the memory hierarchy are prevalent and are the most common source of timing channels. Various parts of the memory hierarchy can lead to information leakage. They are as follows:
  - ✦ Instruction as well as data cache.
  - ✦ Cache replacement policies
  - ✦ Hardware prefetchers
  - ✦ Translation look-aside buffer
  - ✦ Cache coherence directory
  - ✦ Read, write queues and other intermediate buffers
  - ✦ Memory controller and interconnect

Among these, the data cache hierarchy is the most commonly exploited microarchitecture structure as a timing channel to leak sensitive information. Consequently, researchers have proposed numerous attacks that leverage caches to establish a timing channel.

2.3.1 Cache Timing Attacks

Cache timing attacks have attracted much attention in the microarchitecture community as it is difficult to write software with constant time behavior. In addition, timing attacks based on caches can achieve a high bandwidth of more than 1 Mbps. Interest in cache attacks prevails to date due to speculative execution attacks, which is the main focus of this thesis, which we discuss in Section 2.4. This Section discusses some of the traditional side and covert channel attacks based on caches.

Flush + Reload [79]. It is the first cache-based timing attack that provides high resolution, bandwidth, and low noise. This attack requires memory deduplication, allowing programs to share identical read-only pages. Firstly, the attacker **flushes** the cache line containing shared data/code line. Then, it allows the victim to access critical data. Depending on the secret data, the flushed line is either accessed or not accessed by the victim—the attacker then **reloads** the data/code line and times its access. A lower access time indicates that the victim indeed accessed the line. As the initially flushed line access depends on the victim's secret information, the attacker can easily leak the secret by timing its reloads. This results in a high resolution and low noise cache timing channel. But the catch here is

that the sharing of address space as a communication channel is necessary for the attack to be successful. Another requirement is the availability of flush instruction by the underlying instruction set architecture (ISA). These requirements are not met in most practical systems, reducing the scope of this attack.

Evict + Time [52]. This attack relaxes the use of the flush instruction, expanding the range of systems it can affect. Attackers use huge pages to easily determine which addresses are mapped to specific cache sets. The attacker fills the cache set where the shared addresses are mapped to with random data, evicting any shared memory blocks. The victim accesses/does not access the shared address depending on the secret data. The attacker then reloads the shared address, where a fast access reveals the victim accessed the data, and a slow access shows the victim did not access the data. Similar to flush + reload, it requires memory deduplication. In addition, such attacks cannot deal with **LLC slicing** [27], where the address mapping to slices is selected using some undocumented function. It also needs the cache to be inclusive, as it does not use flush instruction, and the data can be present in an upper-level cache.

Prime + Probe [47]. This is an agile attack that negates the requirement of shared memory between the victim and the attacker, increasing the scope of the attack to cloud platforms and across VMs. The attacker first identifies the target set the victim utilizes to perform critical operations. This step, called the eviction set creation, is the most important and complex. Once the eviction set is created, it **primes** (or fills) the eviction set. Depending on the secret data, the victim accesses or does not access a cache line mapped to the eviction set. The attacker then **probes** (or re-accesses) the eviction set. If any of the lines are evicted previously, then the access will be slower, deducing that the victim accessed some line that was brought in the eviction set, hence leaking the secret. The eviction set creation and the probe step are prone to noise compared to the simpler flush+reload and evict+time attack. Prime + probe attack can even bypass LLC slicing to identify cache sets used by the victim to perform critical operations.

2.4 Speculative Execution Attacks

Speculative execution attacks can cause critical data leakage across many security boundaries. Starting from the Spectre [43] and Meltdown [46] attacks discovered in 2018, a wave

of speculative execution attacks came encompassing various threat models. As discussed in Sections 2.1, 2.2, modern processors use numerous performance-boosting techniques, and for the past few decades, performance has been the sole focus of the microarchitecture community. Many such performance-boosting methods are based on aggressive speculation, leading to speculative execution. With all such techniques in place, it is well-proven that programs behave correctly at the ISA level. However, speculative execution modifies the underlying microarchitecture, and data leaks through micro-architectural states. Speculative execution attacks use this loophole. Secret data is accessed by the victim during the transient execution phase and encoded into a covert channel. Then, the attacker extracts the secret data using the covert channel. Although they use a covert channel to decode the secret, speculative execution attacks are inherently different from traditional side channel or covert channel attacks. In conventional attacks, the data is accessible by the victim, whereas, in speculative execution attacks, the data accessed by the victim during speculative execution is inaccessible, which makes such attacks more dangerous. In Sections 2.4.1, 2.5.1, we discuss how such an attack is possible with relevant examples. As described in the paper [76], we can divide speculative execution attacks into three phases:

Setup phase. In this phase, the attacker modifies some microarchitectural state, which forces the target code (also termed the disclosure gadget) to execute in the wrong path (transient execution). For example, in Spectre v1, the attacker executes the victim code multiple times, mistraining the branch prediction unit such that it makes a wrong prediction, leading to transient execution.

Transient Execution Phase. The disclosure gadget executes in this phase. The disclosure gadget is the target code that accesses the secret data and transmits it to a covert channel. The disclosure gadget can be both a piece of the victim and an attacker code. Even if the architectural effects by the transient execution of the target code are rollbacked by the processor eventually on misprediction detection, the microarchitectural state changes persist. Such information can be decoded by the attacker later using one of the traditional covert channels discussed in Section 2.3.1.

Decoding Phase. In this phase, the attacker retrieves the secret data using a traditional covert channel.

The scope of such attacks is immense, ranging from (i) one user program leaking data of another program, (ii) a user-level program dumping the entire kernel memory, (iii) attacks across VMs co-located on the same machine, (iv) attackers residing in the VM attacking the hypervisor and many more.

2.4.1 Spectre attack

The attacker searches for victim code or the operating system's shared libraries for instruction sequences that can violate memory isolation boundaries between the victim and attacker's address space. The attacker tricks the CPU into executing the instruction sequence transiently. Upon transient execution of such instruction sequence, secret data is transmitted to the victim's memory via a covert channel transmitter. As discussed in Section 2.4, Spectre follows the three-step process to complete an attack successfully. Many variants of the Spectre attack use various disclosure gadgets and covert channels. Next, we will discuss the first variant, which uses instruction sequences with conditional branches as a disclosure gadget and Flush + Reload or Prime + Probe as a covert channel, depending on the threat model.

Spectre variant I

```
1    if(x < array1_size)
2        y = array2[array1[x] * 4096];
```

Listing 2.2: Spectre variant I

Listing 2.2 shows the victim code (disclosure gadget) the attacker uses. This can be part of a system call or shared library. Here, x is attacker-controlled, i.e., the piece of code can be called by the attacker with different values of x. The *array1* is of size *array1_size* and *array2* can be shared with the victim or not, depending on the threat model. The threat model can be such that (i) the attacker and the victim use a shared library by memory deduplication, with *array2* being a part of it, or (ii) there is no use for shared memory. A bounds check is performed first on x, preventing any out-of-bounds access of *array1*. The motive of the attacker is to access an out-of-bounds value of array1 (array1[x]), which resolves to a secret byte in the victim's memory k. It is assumed that *array1_size* and *array2* are not in the cache, but k is. The attack assumes this cache configuration is for demonstration purposes, which attackers can easily achieve in real systems. Below are the three phases of the attack:

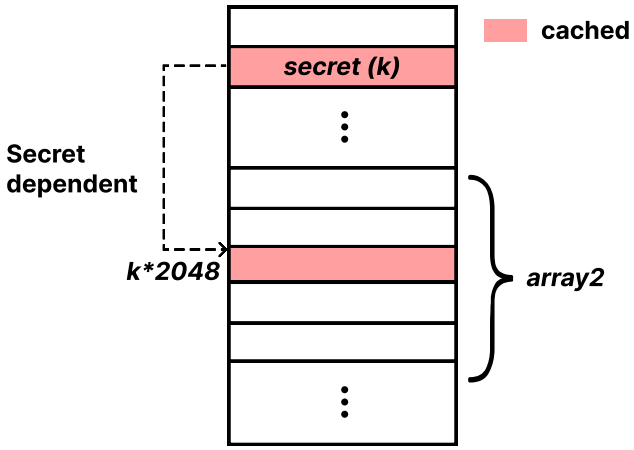


Figure 2.7: Secret dependent access of shared data structure leading to leakage of secret in Spectre variant 1

Setup phase: The attacker invokes the disclosure gadget (we are using the terms target/victim code and disclosure gadget interchangeably) with inbound values of **x**, executing line no. 1 and hence (mis-)training the branch predictor to predict TRUE on future invocations. In addition, the attacker also prepares the covert channel, which will be used to transmit the secret. Depending on the threat model, the attacker (i) either flushes (from flush + reload attack) all the cache lines comprising *array2* in case a shared memory is used, (ii) or finds the eviction set, which will map *array2*'s access from the victim's address space to the attacker's address space. Then, the attacker primes (from Prime + Probe attack) the eviction set.

Transient execution phase: The attacker invokes the disclosure gadget with the malicious value of x. The branch instruction is started, but the branch does not resolve immediately as *array1_size* is uncached. The mistrained branch predictor wrongly predicts that the branch will be taken, starting speculative execution of following instructions. The secret k residing in the address *base (array1) + x* is retrieved quickly due to a cache hit. Load at *array2[k * 2048]* is started, which is dependent on the secret k, and in the meantime branch resolves while the load is still in-flight. On branch resolution, the processor realizes the misprediction and reverts the processor to a state just before the branch is executed. However, the access to *array2* using the secret k populated the cache and persisted even after the rollback of the pipeline. Here, the cache acts as the covert channel.

Decoding phase: The attacker finally retrieves the secret key from the covert channel. Depending on the threat model, the attacker, in case of (i) shared memory, reloads (Flush+Reload) each element of *array2* at a distance of 4096 (*array2[x' * 4096]*) and times each of the accesses. The secret key is decoded from the location where a cache hit is experienced. (ii) With no shared memory; the attacker probes (Prime + Probe) each element of the eviction set created in the setup phase. Unlike Flush + Reload, the location experiencing a cache miss indicates the secret key.

The Spectre attack can achieve a high bandwidth of around 10 KB/s with an error rate lower than 0.01%. Although this is a proof of concept attack, it is tested on multiple processors ranging from x86-based Intel and AMD to ARM-based Qualcomm's Snapdragon and Samsung's Exynos processors and is able to compromise security successfully. However, a practical rendition of such an attack is hard and involves a lot of challenges. To list a few, (i) it is hard to search for a disclosure gadget that accesses some sensitive information, (ii) the covert channel itself can be noisy, (iii) tricking the victim into executing speculatively and accessing the secret can be tricky, because of noise affecting the cache state. Nevertheless, it is just a matter of time before some black hat hacker finds all the right ingredients, cooks up a speculative execution attack, and starts leaking your passwords.

2.5 Backwards-in-Time Attacks

Attacks like Speculative Interference [19] and SpectreRewind [34] change the timing behavior of committed instructions. Younger transient instructions can change the timing of bound-to-retire[3] older instructions. This can change the relative order of bound-to-retire memory operations, change the cache state (for example, replacement bits), and create a covert channel.

2.5.1 Speculative Interference Attack

Transient load accesses the secret and then forwards the secret to following transient instructions, which contend for various microarchitecture resources depending on the secret. Such a sequence of instructions is termed an interference gadget. The contention created by the transient instruction changes the timing behavior of older bound-to-retire instructions, hence

[3]Correctly predicted speculative instructions that will eventually commit or non-speculative instructions

encoding the secret. The change in timing behavior of bound-to-retire memory operations ultimately changes the cache state, creating a covert channel.

```
z = ...
A = f(z)     // takes F cycles
y = load(A) // interference target
B = g(z)     // takes G > F cycles
v = load(B)
if (i < N) {                          // mispredicted branch
    secret = load(&TargetArray[i])  // secret access
    x = load(&S[secret * 64])       // Interference Gadget
    f'(x)
}
```

Listing 2.3: Speculative interference attack by contention at non-pipelined execution unit

A speculative interference attack involves a three-step process. First, an interference gadget competes for resources conditionally, based on the secret. This resource contention induces timing variations in the behavior of an interference target, typically a non-speculative instruction, leading to changes in the cache state. Finally, the attacker exploits these cache state changes to infer the secret.

The variants of speculative interference attacks, shown in Listing 2.3, focus on resource contention at a non-pipelined execution unit. In this scenario, two functions, f and f', utilize the same non-pipelined execution unit, resulting in resource contention. The cache state of $S[0]$ is assumed to be uncached, while $S[1]$ is cached. Consequently, the transmitter load (interference gadget) shows secret-dependent behavior.

If the secret is 0, the transmitter load quickly returns, allowing f' to execute in the non-pipelined execution unit. This stalls the execution of f, delaying the load of A. At the same time, the non-speculative load of B completes before A, resulting in the order of non-speculative loads as $B \rightarrow A$. On the other hand, if the secret is 1, the transmitter waits for the load to return, avoiding immediate contention for the execution unit. This enables f to execute, completing the load of A first, followed by B, resulting in the order of non-speculative loads as $A \rightarrow B$.

This reordering of loads translates into persistent cache state changes, primarily through cache replacement state. The attacker decodes the ordering of non-speculative loads from the cache replacement state, utilizing methods like RELOAD+REFRESH [21]. As a result, the secret is leaked.

Chapter 3

Related works

3.1 Mitigating speculative execution attacks

This section explores the different techniques proposed to defend against speculative execution attacks. The community has suggested both software and hardware mitigations to tackle this issue. One challenge with these attacks is that they exploit micro-architectural vulnerabilities that are difficult to detect or mitigate through software. Software solutions, such as KAISER [35], Retpoline [73], and Memory fences [28], aim to prevent secret leakage by isolating addresses or limiting speculative execution in critical code areas. However, implementing these solutions requires rewriting software or the operating system code and is often incompatible with existing code. Moreover, software-based approaches are tailored to specific attack scenarios and may not provide comprehensive protection against new threats. Additionally, they can lead to significant performance slowdowns, sometimes as high as 50% [54].

Given these limitations, there is growing interest in hardware-based mitigations, which offer more effective protection against speculative execution attacks while minimizing performance impact. Hence, we focus on hardware mitigations. Speculative attacks consist of three steps, as outlined in Section 2.4. Shown in Figure 3.1, the transient execution step can be broken down into two parts: first is the **ACCESS** instruction, which accesses the secret value, and second is the **TRANSMIT** instruction, which transmits the secret value into the covert channel. The decoding phase is also called the **INFERENCE** phase, where the attacker decodes the transmitted secret from the covert channel.

Two primary strategies are employed to interrupt the information leakage: halting transient execution or eliminating covert channels transmitting the secret. Delay-based ap-

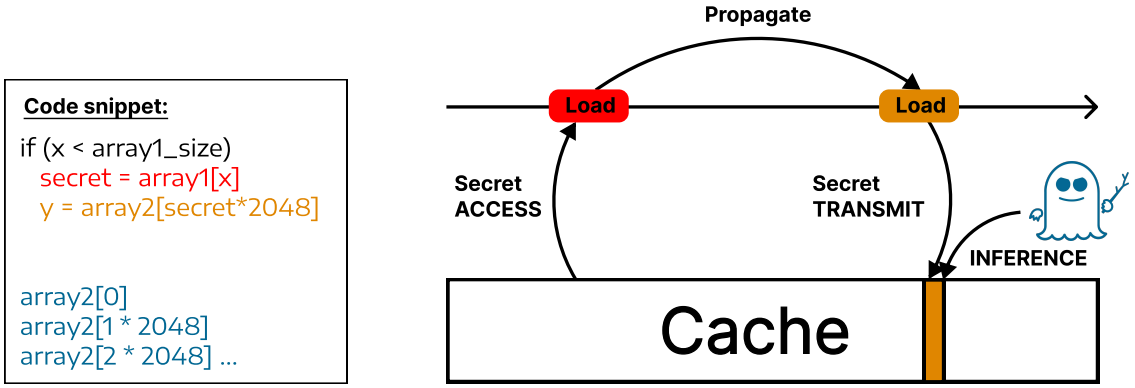


Figure 3.1: Timeline of a speculative execution attack

proaches halt transient execution, while others focus on invisible speculation to remove covert channels.

3.1.1 Delay-based Mitigations

The first approach involves delay-based techniques, wherein the transmission of secret-dependent values is stalled until it is considered safe to proceed [80, 75]. Safety determination can be complex and requires sophisticated mechanisms to accurately identify when an instruction can be considered safe for execution.

Speculative Taint Tracking [80]. A conservative approach to stop secret transmission in the speculative execution window is by delaying the instruction following secret access. Speculative Taint Tracking (STT) considers it safe to forward the secret to dependent speculative instructions unless those instructions form a covert channel. STT safeguards this protection with minor frontend and backend processor design modifications, eliminating any covert channel creation.

It *taints* the output registers of an ACCESS instruction, and every register dependent on such a tainted register is marked as tainted during the program execution. Any instruction with tainted input registers is delayed for execution, while untainted ones are allowed to execute. The tainted instructions are executed when its registers become untainted, i.e., when the ACCESS instruction becomes non-speculative. The ACCESS instruction broadcasts all dependent instructions to untaint its registers once the ACCESS instruction becomes safe to execute.

One of the major challenges identified by the author was to identify and block all kinds of covert channels. The authors categorized covert channels into explicit and implicit covert channels. Explicit covert channels are created when the secret is directly passed on to a transmitting instruction. These channels are blocked by blocking instructions with tainted input registers till they become untainted. On the other hand, implicit channels do not require direct transmission of secret, rather instruction execute depending on the secret creating a secret dependent resource contention. Implicit covert channels come in two forms: explicit branch and implicit branch. In explicit branch cases, instructions execute based on a secret-dependent branch predicate. In the case of an implicit branch, there is an implicit secret-dependent branch that dictates the sequence of events, forming the covert channel. To block implicit channels, side effects of misprediction detection are stalled. For explicit branches, squashes and branch predictor updates are delayed until the predicates are untainted. Similarly, for implicit branches, implicit branch resolution, such as squashes, is postponed, effectively blocking prediction and resolution-based channels in both cases.

Non Speculative Data Access [75]. Data can be leaked through a covert channel when a chain of speculative instructions accesses and transmits the secret. To break the chain, Non-Speculative Data Access (NDA) tries to block secret propagation from an unsafe instruction. It restricts data propagation by preventing tag broadcast of destination registers of unsafe instructions to source registers of all its dependent instructions, delaying the wakeup of their dependents until source instruction becomes safe. NDA suggests different policies for deciding which instructions are unsafe, allowing dependent instructions to proceed. These policies offer varying levels of security and come with different performance impacts, and are as follows:

- **Strict data propagation** marks all instructions following an unresolved branch unsafe. The tag bits of destination registers of unsafe instructions are not broadcasted to dependent instructions, breaking the data propagation chain. On branch resolution, the unsafe instruction is declared safe and broadcasts tags to dependent instructions.
- **Permissive data propagation** considers only LOAD instructions candidates for an ACCESS instruction. Thus, it marks only LOAD instructions following an unresolved branch unsafe.
- **Load Restriction** marks all LOADs as unsafe till it reaches the head of ROB.

3.1.2 Invisible Speculation

Invisible speculation restricts covert channels by blocking the INFERENCE step. Previously proposed invisible speculation schemes [77, 14, 15] focus on speculative attacks using *cache-based* covert channels. Speculative LOADs are allowed to ACCESS secret and TRANSMIT them, making such a scheme efficient in case of correct speculation. However, it restricts speculative instructions to modify the state of the cache system[1], restricting the creation of a covert channel. The cache state is updated once the LOADs are deemed safe.

InvisiSpec [77] uses a speculative buffer (SB) to store all speculative data. When a LOAD request occurs, the speculative buffer is checked first. If the requested data is not found, it is invisibly retrieved from the cache hierarchy. This process ensures that the cache state remains unchanged; caches are not filled in case of misses, and replacement states are left untouched. The speculative buffer is cleared in case of a misprediction to prevent attackers from inferring information during the INFERENCE step. However, if the prediction is correct, the speculative LOADs are reissued to update the cache states.

To maintain memory consistency, InvisiSpec employs either validation or exposure methods. Validation involves checking for any violations during the speculative window. After reissuing LOADs, the system validates whether the data in the speculative buffer matches the reissued LOAD. If a mismatch is detected, the specific instruction and all subsequent instructions are squashed to preserve memory consistency. In the Exposure method, no such check is conducted after the completion of LOAD reissue and is utilized when no memory consistency violation is anticipated.

GhostMinion [15] has been proposed to mitigate Spectre attacks with minimal performance impact. It introduces a small cache called the GhostMinion (GM) cache, which stores speculative data and is flushed in a single cycle upon a domain boundary switch or branch misprediction. The rest of the cache hierarchy remains unaffected by speculative data requests, and only committed instructions write data back to the cache hierarchy.

Backward in-time channel attacks discussed in Section 2.5 exploit the timing effects of speculative instructions on logically earlier instructions. To mitigate these attacks, GhostMinion proposes a more generalized solution. Like MuonTrap [14], it uses a small cache (GM cache) to store speculative requests. However, GhostMinion employs multiple restrictions

[1]This includes the state of the cache hierarchy, replacement policy state, and coherence state modifications.

on the GM cache to defend against newer attack variants. GhostMinion's distinguishing factors include parallel access to the L1 cache and temporal ordering of instructions using techniques like TimeGuarding and LeapFrogging to prevent backward in-time attacks. In Section 3.1.4, we discuss GhostMinion in more detail.

3.1.3 Other Techniques

Techniques like Delay-on-Miss [62], CleanupSpec [59] and DoppelGanger [44] do not fall directly into the category of delayed-based or invisible speculation techniques. In this Section we discuss these three techniques.

Delay-on-Miss. Inspired by both delay-based and invisible speculation methods, Delay-on-Miss (DOM) identifies extra costs associated with reissuing LOADs in the invisible speculation-based approach and with delaying all instructions after a branch instruction in the delay-based approach. It introduces a hybrid approach that operates invisibly upon an L1D hit without changing cache states and only delays execution upon an L1D miss. DOM allows speculative data accesses that encounter a hit in L1 to continue execution without updating its replacement or coherence states, while in case of a miss, delay the LOAD until it becomes non-speculative. Additionally, the authors enhance this idea by introducing a value predictor that predicts values on an L1D miss, eliminating the delay in case of a correct prediction.

CleanupSpec. It allows speculative changes to the cache hierarchy and adopts an undo-based approach in case of misspeculation. Cleanupspec handles the uncommon case of misspeculation ($\approx$5%) and rollbacks cache state changes made during speculation. It uses partitioned L1D along with a random replacement policy and randomized L2 and LLC to mitigate cross-core or multi-thread covert channels.

DoppelGanger. It enhances delay-based mitigation techniques and improves performance by reducing delay. It generates a doppelganger for any LOAD it can, employing an address predictor. This doppelganger is then preloaded into the LOAD's destination register. If the prediction is accurate, the preloaded value is propagated to the LOAD. However, if the prediction fails, the LOAD is replayed. In cases where it cannot produce a duplicate (unable to make a prediction), it resorts to either the NDA or STT mechanism to safeguard the

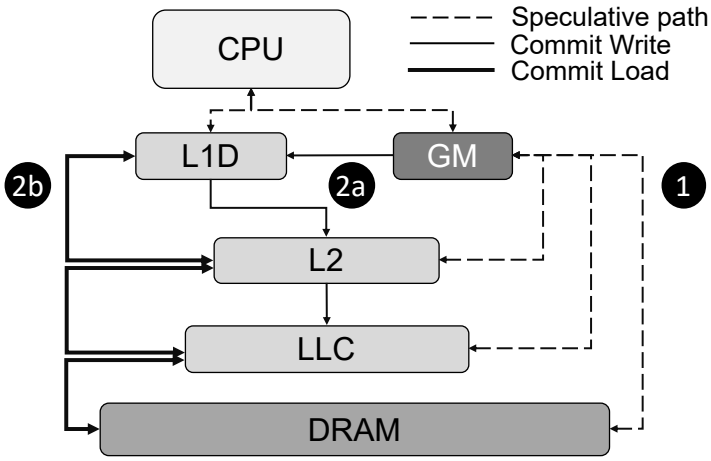


Figure 3.2: GhostMinion secure cache system

load. In case of correct prediction, DoppelGanger eliminates the delay incurred in case of delay-based approaches improving performance.

3.1.4 GhostMinion

GhostMinion uses a small 2KB cache called the GM, accessed concurrently along with the L1D, that stores the data of speculative instructions till they commit (or retire). On a demand miss generated by a speculative instruction at GM, it searches for the data at L1D, L2, and LLC, similar to a conventional cache hierarchy. However, on a hit at L1D, L2, or LLC, the cache state (replacement policy priority bits) is not updated. On a miss at the L1D, L2, and LLC, the response is directly filled into GM, bypassing L1D, L2, and LLC (Figure 3.2, 1). On a commit, the rest of the cache hierarchy (L1D, L2, and LLC) gets the data of committed instructions from GM if it is a GM hit, through *on-commit* writes (Figure 3.2, 2a). In case of a GM miss, *re-fetching* of data is done into the non-speculative cache hierarchy (L1 to LLC) (Figure 3.2, 2b). GM is neither inclusive nor exclusive to the rest of the cache hierarchy.

Within GM, instructions are restricted to see the eviction or insertion of others depending on their temporal order. The temporal order is maintained based on the timestamp. Time-Guarding used by GhostMinion ensures the insertions and the evictions are invisible under multiple speculations. To hide contention at MSHRs, the timestamp metadata propagates into the MSHRs at each cache level, allowing younger loads to be canceled and replaced by the older loads (leapfrogging). Also, in GhostMinion, a block can only be in a shared or

Table 3.1: Summary of mitigation techniques.

Mitigation techniques	**Classification**	**Secure?**	**Storage overhead**	**Perf. slowdown**
CleanupSpec [59]	Undo-based	No	<1KB	Medium
NDA [75]	Delay-based	Yes	≈ 150 Bytes	High
STT [80]	Delay-based	Yes	≈ 1.4 KB	Medium
NDA + Doppelganger [44]	Delay-based	Yes	≈ 13.5 KB	Medium
DoM [61]	Delay + invisible speculation-based	No	≈ 0.4 KB	High
DoM + Doppelganger [44]	Delay + invisible speculation-based	No	≈ 13.9 KB	High
STT + Doppelganger [44]	**Delay-based**	**Yes**	**≈ 14.9 KB**	**Low**
InvisiSpec [77]	Invisible speculation	No	≈ 9.5 KB	High
MuonTrap [14]	Invisible speculation	No	2 KB	Low
GhostMinion* [15]	**Invisible speculation**	**Yes**	**2 KB**	**Low**

invalid state and the coherence states of GM and non-speculative caches are not altered until an instruction is committed.

Table 3.1 summarizes recent mitigation techniques, keeping security, performance, storage, and implementation complexity in mind. We use the secure implementation of GhostMinion as suggested in Pensieve [78]. DoM, MuonTrap, Invisispec, and Cleanupspec are not secure as these techniques do not mitigate the speculative interference [19] attacks. Cleanupspec does not fit into delay or invisible speculation-based mitigation techniques. The degree of implementation complexity indicates the extent of hardware changes required within the processor core and memory hierarchy. High implementation complexity indicates changes are required in both the core and memory hierarchy. We categorize performance slowdown into three bins: low (<5%), medium (5% to 10%), and high (>10%).

As mentioned in Section 3.1, mitigation techniques fall into one of the two broad approaches: delay-based [80, 75, 61] and invisible speculation [77, 15, 14, 60]. For performance evaluation, we use SPEC CPU2017 [70] and GAP [10] benchmarks. Invisispec [77] uses a speculative buffer similar to GM, but it does not provide strictness ordering. It incurs higher performance overhead than GhostMinion [15]. Among all the delay-based approaches, STT provides security guarantees with minimum performance overhead. Doppelganger improves the performance of delay-based approaches, which includes STT. Compared to delay-based

techniques that incur high storage overhead, invisible speculation techniques like GhostMinion entail lower performance penalties with minimum storage overhead. Therefore, in this paper, we select GhostMinion as our secure cache system.

3.2 Hardware Prefetching

Cache prefetching is a technique to mitigate expensive off-chip memory latency by predicting upcoming memory accesses and retrieving the data that is not currently in the cache before it's requested by the processor. In particular, we focus on hardware prefetchers, which observe load/store access patterns and prefetches data based on past access behavior. Let us consider a practical scenario illustrated in Figure 3.3. ❶ denotes the cache access pattern by the CPU. The prefetcher learns this pattern and predicts future cache accesses. ❷ indicates the cache access predicted by the prefetcher, which is issued to memory ahead of time, indicated by ❸. The data is prefetched into the cache ❹. When the CPU actually issues the request ❺, it gets a cache hit instead of a cache miss. This improves performance significantly. Some of the commonly used metrics to evaluate a prefetcher are:

$$Coverage = Misses\ eliminated\ by\ prefetching/Total\ misses \tag{3.1}$$

$$Accuracy = Useful\ prefetches/Total\ prefetches \tag{3.2}$$

$$Lateness = Late\ prefetches/Total\ prefetches \tag{3.3}$$

Here misses eliminated is the difference between the number of cache misses without a prefetcher and the number of cache misses with a prefetcher. A coverage of one indicates the ideal case where all misses are covered by the prefetcher, which is impractical. Hence, a higher coverage is desirable. However, a prefetcher with high coverage and low accuracy can degrade performance by generating unnecessary memory traffic. Another important metric is the prefetcher lateness which is the percent of late prefetches. Here, late prefetches are accurately predicted prefetches that were issued late and are present in the MSHR of the corresponding cache level.

State-of-the-art hardware prefetchers have significantly enhanced single-thread performance, achieving average performance boosts ranging from 3% to 5% [20, 17, 53, 51]. Most proposed storage-efficient prefetchers are designed for the L2 cache [17, 20]. Exceptions include multi-lookahead offset prefetching (MLOP) [65], instruction pointer classifier-based

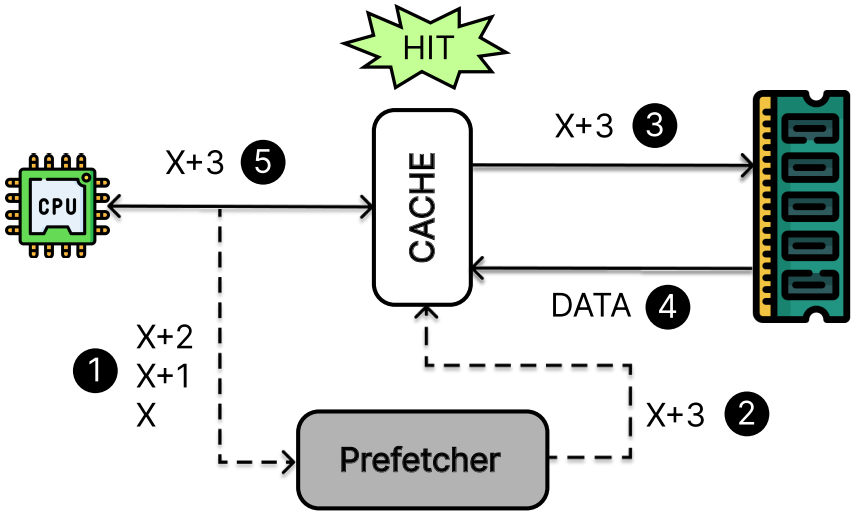


Figure 3.3: Hardware prefetching

prefetching (IPCP) [53], and Berti [51], which are L1D prefetchers. It is well-established that an L1D prefetcher offers superior performance compared to an L2 prefetcher because prefetched lines are brought closer to the core, as demonstrated by IPCP [53]. Additionally, an L1D prefetcher observes unfiltered memory access patterns, enabling more accurate predictions of future accesses than an L2 or LLC prefetcher. Berti stands out as the state-of-the-art L1D prefetcher, boasting a high accuracy of almost 90%.

Prefetchers lead to covert channels, as shown in [25, 29], which can, in turn, lead to information leakage through transient execution. As discussed in Chapter 1, having a secure cache system is not enough to mitigate speculative execution attacks as speculatively triggered prefetches can modify the cache state, creating a covert channel. Next, we discuss five state-of-the-art hardware prefetchers, which we later compare with our proposed technique for evaluation.

IP-stride. Stride-based prefetching is a simple prefetching technique where the prefetcher tries to learn constant strides across memory accesses. IP-stride prefetcher is a stride prefetcher that associates a stride with a particular Instruction Pointer (IP). This prefetcher tries to learn the stride by subtracting the current address from a previous address associated with a particular IP. As a particular stride occurs for the same IP, the confidence is incremented. When confidence reaches a threshold, prefetches are issued for the particular IP with the learned stride. It can learn multiple strides across multiple IPs with varying stride patterns.

Bingo. Spatial data prefetching exploits the recurrence of access patterns over memory regions to prefetch future memory references. Most existing prefetchers associate observed access patterns to either short events with a high probability of recurrence or long events with a low probability of recurrence, which leads to low accuracy or very little prediction opportunity, respectively. Bingo [17] solves this problem by considering both short and long events to achieve high accuracy with minimum loss of prediction opportunities. Bingo is inspired by the TAGE predictor based on the idea of storing multiple cascaded history tables where each entry is associated with different events. Bingo stores the page access footprint, associating them to long and short events. It uses the longest event while prediction, increasing accuracy without losing on prefetch opportunities by relying on shorter events.

Signature Path Prefetcher. An effective prefetching algorithm must accommodate a broad spectrum of memory access patterns. While simple stride prefetching techniques can detect sequences of addresses with constant differences, they fall short of capturing diverse delta patterns. Offset-based prefetchers like the Best Offset prefetcher [49] assess multiple offsets at runtime to issue prefetches, maximizing the likelihood of use. However, they overlook temporal ordering between delta patterns and struggle with accuracy on complex address patterns. Look-ahead prefetchers like Signature Path Prefetcher (SPP) [42] encode access relationships to make future predictions. By recursively referring to a pattern table, they generate timely prefetches for complex access patterns.

Instruction Pointer Classifier-based Spatial Prefetching. Instruction pointers show different access patterns. Capitalizing on this fact, Instruction Pointer Classifier-based Spatial Prefetching (IPCP) [42] classifies IPs having different access patterns into three classes: constant stride, complex stride, and global stream. IPCP classifies accesses based on IP showing one of the three stride patterns at the L1D cache. It integrates the three types of prefetching concepts mentioned earlier, selecting the most suitable one depending on the situation to enhance performance.

Berti. Offset-based prefetchers operates by adding offsets to the current access and prefetching the resultant address. It selects an offset based on the maximum likelihood of its use in the future. Berti [51] is an offset-based prefetcher that proposes to use local (per IP) deltas instead of having a single global offset. Delta is the difference between two cache

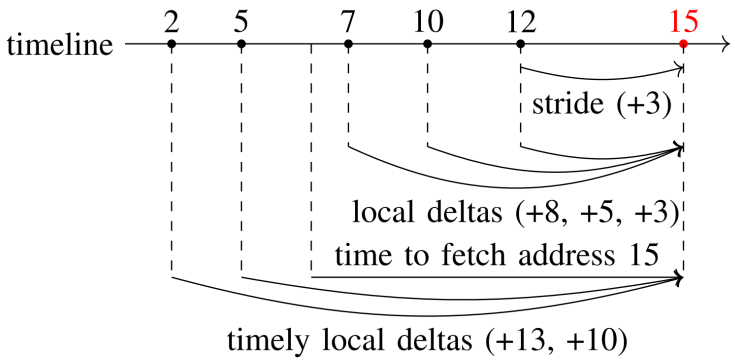


Figure 3.4: Strides, local deltas & timely local deltas. The values on the timeline (2, 5, 7...) represent the addresses referenced by the same instruction (adapted from [51])

line addresses. It not only selects deltas having the maximum probability of usage but also factors in if the deltas will lead to a timely prefetch. Berti stands out as the most efficient and lightweight among all the state-of-the-art prefetchers. Therefore, we opt for it as the preferred prefetcher. Section 3.2.1 describes the Berti prefetcher in detail.

3.2.1 Berti prefetcher: 10K feet view

Training the prefetcher. Berti trains for deltas for a given IP, which is known as the local deltas. For an IP, local deltas are defined as the difference between the cache line addresses of two demand accesses. The goal of the training mechanism is to estimate the coverage of each seen delta per IP, considering only those deltas that would result in a timely prefetch as shown in Figure 3.4. The training consists of the following actions: measuring fetch latency, learning timely deltas, and computing the coverage of the deltas.

1. Measuring fetch latency. To learn the deltas that are *timely*, it is necessary to measure the time required to fetch data to the L1D. This measurement is performed for any cache line in L1D, both for demand misses and prefetch requests. Fetch latency can be measured by keeping a timestamp for any L1D miss inserted into the MSHR and any prefetch request inserted into the PQ. On an L1D fill, the latency is simply computed by subtracting the stored timestamp from the current cycle.

2. Learning timely and accurate deltas. Once the fetch latency is obtained for each L1D fill, Berti looks up for past accesses (from the same IP) that could have triggered a timely prefetch for the current request, given the history of accesses and their recorded timestamps. (Access time of timely + fill latency of current request $<$ access time of current request)

Timely deltas are then computed by subtracting the address of each timely access in the history from the current address.

3. Computing the coverage of deltas. On every search in history, Berti obtains a set of timely deltas. Deltas frequently appearing in the search would cover a significant fraction of misses, while deltas that rarely appear would result in low coverage. It is important to note that high local (per IP) coverage translates into high global accuracy.

Issuing prefetch requests. For a given IP, deltas with the highest coverage are selected and added to the current load address to form the prefetch requests. Berti orchestrates the prefetch requests across the cache hierarchy depending on the coverage of each delta and the L1D MSHR occupancy. If the coverage of a delta is above a *high-coverage* watermark and the L1D MSHR occupancy is below the *occupancy* watermark, then prefetch requests using that delta gets filled at all the cache levels till L1D. Otherwise, if the coverage is above a *medium-coverage* watermark, irrespective of the L1D MSHR occupancy, prefetch requests get filled till L2. Finally, if the coverage is above a *low-coverage* watermark, requests get filled only in the LLC.

In the subsequent chapter, we delve into the challenges posed by the negative interactions between a secure cache system and a prefetcher. Additionally, we address the inherent issues associated with designing a secure prefetcher.

Chapter 4

Motivation

4.1 Threat model

We assume the following capabilities in our *transient execution* attacker:

(i) (S)he is capable of mounting attacks like Spectre and speculative interference [19] through the cache system.

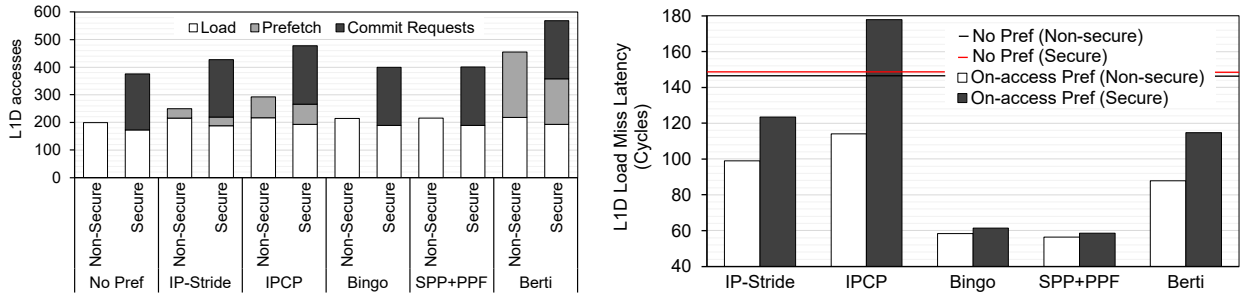
(ii) (S)he can exploit a hardware prefetcher by speculative training and prefetching that can change the cache state, as mentioned in Muontrap [14].

(iii) The attacker and the victim can be part of the same process or two different processes. The attacker can run arbitrary code but cannot access secret data directly, i.e., the attacker is running within a sandbox either at the user or at the kernel level.

(iv) (S)he is also capable of mounting attacks like Augury [74] and Gofetch [24] using data memory-dependent prefetchers. There are timing-based side and covert channels involving hardware data prefetchers and caches [67, 30, 26] that can be mitigated by existing spatial isolation techniques [31, 58].

4.2 Secure Prefetching

The focus of this work is to build a secure classical prefetcher complementing a secure cache system. Specpref [68] and Ghost Loads [60] design a secure prefetcher for a specific cache hierarchy like MuonTrap [14] and Ghost Loads [60]. Specpref prefetches speculatively but conservatively to the small speculative cache of Muontrap (similar to the GM) and prefetches on commit to the rest of the cache hierarchy. Overall, Specpref is a throttling technique that



(a) L1D traffic

(b) L1D Load miss latency

Figure 4.1: Average L1D accesses per kilo instructions and average L1D load miss latency with on-access prefetching

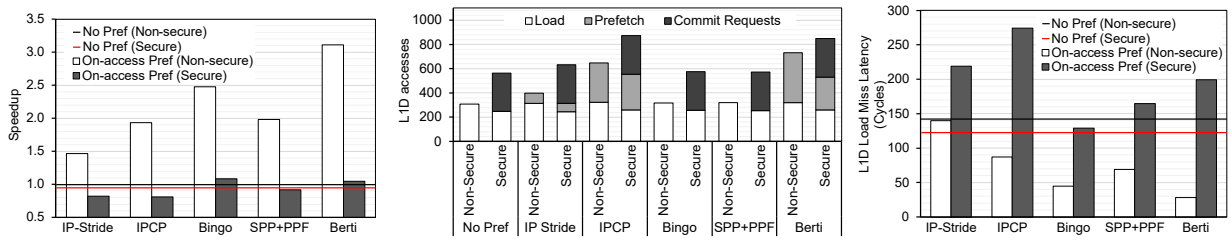
is applied on top of existing prefetchers, which ignores some of the fundamental observations that we present.

4.3 Impact of secure cache system on prefetching

This section analyzes what prevents the prefetcher effectiveness on secure cache systems like GhostMinion and observes that the main cause is the traffic generated on restoring the cache state on commit.

Figure 4.1 shows the increase in L1D accesses for the prefetchers evaluated in this work with a GhostMinion secure cache system and for on-access prefetching. On a non-secure system with no prefetching, the average L1D accesses per kilo instructions (APKI) is 199, which goes up to 375 in GhostMinion because of commit requests that update the cache state. With Berti on GhostMinion, the APKI goes up to 570. The trend persists for all the prefetchers. For L2 prefetchers like Bingo and SPP+PPF there is no access from the prefetcher to L1D as the prefetch requests are generated from L2.

The increase in APKI results in additional traffic causing an increase in L1D miss latency as shown in Figure 4.1. One of the primary contributors to this additional miss latency is the following interesting trend that makes the latency worse especially in the presence of hardware prefetching. On average, for the Berti prefetcher, with a secure cache system, there is a 10.4% increase in L1D MSHR occupancy and the L1D MSHR becomes full for an additional 8.7% of the time. Furthermore, without prefetching, the L1D MSHR occupancy decreases by 15.9% when we move from a non-secure to a secure cache system because demand misses are first served by the GM.



(a) Speedup (b) L1D traffic (c) L1D Load miss latency

Figure 4.2: Speedup normalized to a non-secure cache system with no prefetching, traffic (in terms of APKI), and miss latency with on-access prefetching for 605.mcf_s-1554B trace

To dig deep into the interesting interactions, we choose `605.mcf_s-1554B` and perform a detailed analysis. Figure 4.2(a) shows the normalized performance with respect to a non-secure baseline without prefetching. A significant reduction in performance is observed when the prefetchers are applied to a secure cache system (for Berti, it is more than 300%). Figure 4.2(b) shows the increase in traffic at L1D contributed by load, prefetch, and commit requests from GhostMinion. Figure 4.2(c) shows a significant increase in L1D miss latency. When we analyze the MSHR occupancy numbers, without prefetching, L1D MSHR occupancy decreases by 16.2% when we move from a non-secure to a secure cache system because the demand requests are first served by GM. However, with prefetching, there is an increase in L1D MSHR occupancy of 10.1% when we move from a non-secure to a secure cache system. This happens because, with a non-secure cache system, the L1D MSHR only has to deal with demand and prefetch requests, while with a secure cache system, it also has to handle prefetch requests on top of GhostMinion requests, which increases the pressure on the MSHR. Without prefetching, L1D MSHR is almost never full. However, with prefetching, there is an increase in the percentage of time L1D MSHR is full (from 6.3% to 20%). In Section 5.1, we propose a mechanism that resolves the additional traffic-induced performance loss when hardware prefetching is enabled.

4.4 Impact of secure hardware prefetching

As described in GhostMinion [15], on-commit prefetching results in no information leakage due to speculative execution. However, as shown in Figure 1.1, simply moving state-of-the-art prefetchers to the commit stage results in 3%-4% performance loss compared to on-access prefetching. This section analyses the reason.

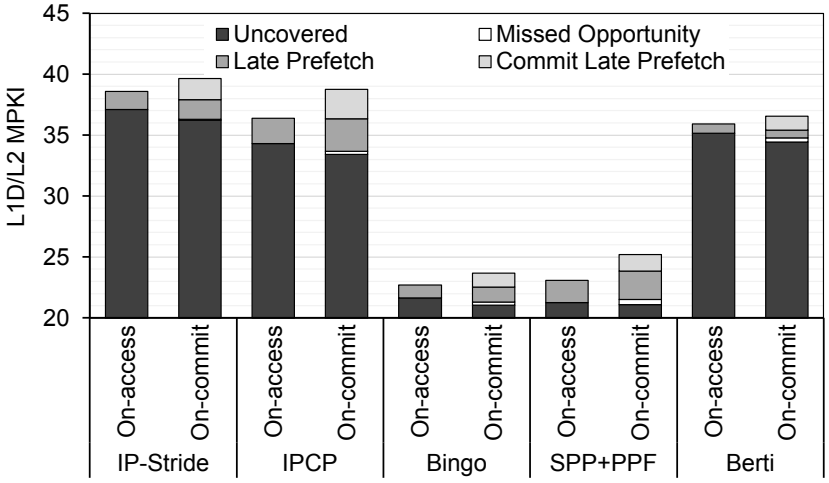


Figure 4.3: Average L1D/L2 demand MPKI in terms of coverage and lateness.

Figure 4.3 shows the average demand misses per kilo instructions (MPKI) across the workloads analyzed in this work. The MPKI is shown for the cache level where the prefetcher works, namely, L1D for IP-stride, IPCP, and Berti, and L2C for Bingo and SPP+PPF. In addition, each prefetcher is evaluated both with on-access and on-commit prefetching. The MPKI has been divided into the following four categories:

- *Commit late prefetch*: This is a new kind of late prefetch request that we introduce in this work and it only appears when the prefetcher is placed at the commit stage. We define it as follows: at the time of a demand cache miss, a prefetch request for the target cache line has not been triggered yet by the on-commit prefetcher, but it would have been triggered by an on-access prefetcher. Importantly, this type of prefetch does not fall in the traditional late prefetch category since in fact the prefetcher request has not been triggered yet when the access takes place.

- *Late prefetch*: This is the typical late prefetch, where a demand miss finds in the MSHR a prefetch request for the target cache line, and merges both requests.

- *Missed opportunity*: The demand miss is for a cache line that would have been predicted correctly by an on-access prefetch but it was missed by on-commit prefetch as it is trained in a different order. This kind of prefetch is also only present for on-commit prefetching and gives information about the negative impact of training at commit.

- *Uncovered*: Demand misses that did not fall in any of the previous categories.

We observe a common trend for all evaluated prefetchers: the uncovered demand misses are reduced when moving the prefetcher to on-commit. Even if we add the missing opportunity bar to the uncovered one, in general, the resulting MPKI (excluding the MPKI coming from the commit late prefetch requests) is lower for on-commit prefetchers. The Uncovered misses of L1D/L2-MPKI are reduced with on-commit as some uncovered ones are now classified as on-commit late.

Despite this trend, performance is worse when compared to on-access prefetching. The reason is timeliness. Although traditional late prefetch requests practically do not increase when moving from on-access to on-commit, our new defined class of *commit late* is the culprit of the increase in *overall* MPKI for on-commit prefetching. That is, prefetch requests must be triggered earlier to compensate for the delays entailed by on-commit prefetching. Fortunately, as we show in this work, it is possible to compensate for this lack of timeliness. Section 5.2 proposes a mechanism that mitigates the lack of timeliness.

In chapter 5, we address the challenges associated with the impact of secure cache systems on prefetching and the issues inherent in secure hardware prefetching. To mitigate the negative interactions between secure cache systems and prefetchers, we introduce the Secure Update Filter (SUF). Additionally, we propose Timely Secure Berti (TSB), a timely and secure adaptation of the state-of-the-art hardware prefetcher Berti, which addresses the issues related to secure hardware prefetching.

Chapter 5

Secure Prefetching for Secure Cache Systems

5.1 Prefetch-friendly secure cache system

Secure cache systems, based on invisible speculation, update the cache hierarchy when memory instructions are not speculative, e.g., when committing. In the case of GhostMinion, this entails re-fetching the cache line on a miss in the GM or sending on-commit write requests (for clean cache lines) to the rest of the cache hierarchy on a hit in the GM. The goal of this extra data movement is to populate the cache hierarchy, which was left intact when the data was speculatively requested by the core, to minimize cache misses in the subsequent accesses.

Both re-fetching and write propagation have an important impact on memory hierarchy traffic. The extra traffic, however, does not come with a noticeable performance degradation in a memory system that is not heavily contended, as one with prefetching mechanisms. However, prefetching mechanisms stress the cache hierarchy queues and MSHRs, preventing the prefetcher from improving performance as shown in Section 4.3. We observe that many requests aimed at restoring the cache hierarchy are indeed not necessary and cause severe contention. For example, triggering a re-fetch for data that was provided by the L1D would consume L1D ports to just update the LRU replacement policy. In the same context, the on-commit write requests propagate up in the memory hierarchy until the data is already found in a cache level. The access to the cache level already containing the cache line could be therefore avoided.

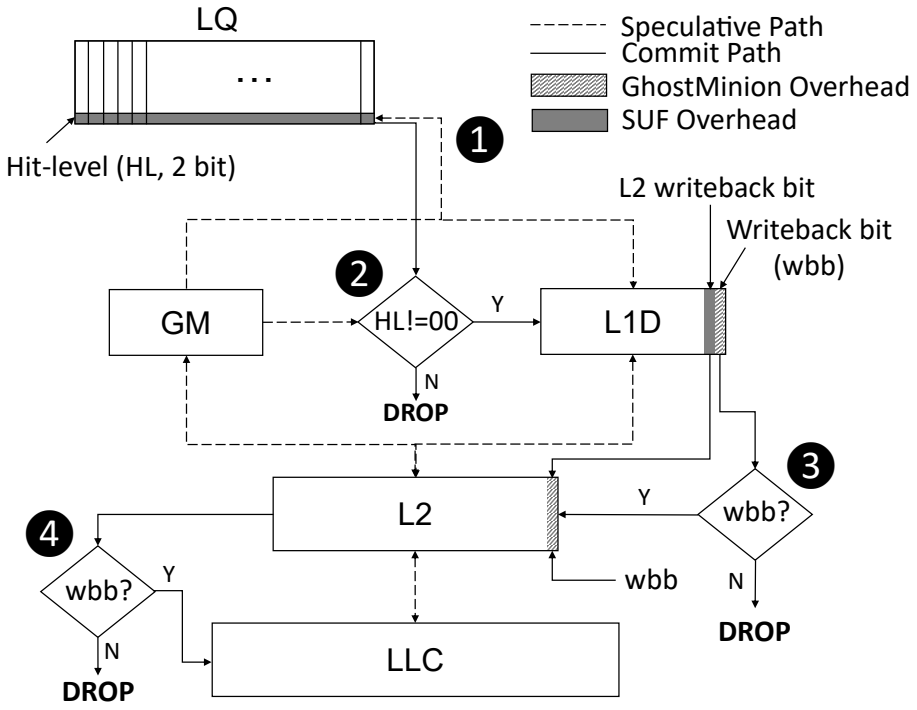


Figure 5.1: Overview of the secure update filter (SUF)

Driven by this observation, we propose the secure update filter (SUF). SUF records the cache level that provided the data when requested. Then, at commit time, either filters the re-fetching when the data was provided by the L1D or stops the on-commit write propagation at the level previous to the one that provided the cache line. In case the SUF mispredicts, because the fetched cache line may have been evicted in the interim, a subsequent fetch request would incur extra latency since it will be served from a higher level. Thanks to SUF's high accuracy, the number of cache accesses is reduced, and consequently, the amount of traffic generated when restoring the cache state. SUF works independently of the underlying prefetching technique, in a transparent way.

Identifying the cache level holding a cache line. SUF uses the lower level (L1D is the lowest level and LLC is the highest level of the cache) holding a cache line to decide if filtering should be employed. The cache level can be learned when the processor requests the data, by propagating down the hierarchy the cache level that served the cache line. That information is encoded using 2 bits indicating if the data comes from L1D (or GM, which is accessed in parallel), L2C, LLC, or DRAM. The 2-bit hit-level information is stored along with the requested data in the memory operation entry at the load queue (LQ) (Figure 5.1, step ❶).

Filtering updates. Once a speculative load is committed, it checks the GM in order to decide if re-fetching the cache line down the hierarchy (GM miss) or propagating the cache line up in the hierarchy (GM hit) is required. SUF checks the hit-level field, and proceeds as follows. In the case of the data being provided by the L1D (value 00), SUF drops the update (both for re-fetching and on-commit propagation). Otherwise, the re-fetch or propagation is done as usual that causes the cache line to move from GM to L1D (step ❷). Upon eviction of the cache line from either L1D or L2, the decision to propagate the writeback block is determined by the GhostMinion *writeback bit*[1]. The value of the GhostMinion writeback bit at L1D and L2 are evaluated at commit time using the hit-level and propagated along with the writeback block. Each cache line at L1D stores the L2 writeback bit as well, so that it gets propagated to L2 upon writeback (step ❸). Finally, during the eviction from L2, the GhostMinion writeback bit is once again employed to determine whether to propagate or not (step ❹).

Security guarantees. SUF selectively filters out commit requests. These commit requests do not expose any speculative information, as GhostMinion's Timeguarding mechanism ensures that the timing of instructions bound to commit remains unaffected by transient instructions. The primary consequence is that the cache hierarchy will not be updated as intended, potentially resulting in a subsequent miss for the same address within the cache hierarchy, leading to performance degradation. However, due to the high accuracy of SUF, this performance loss is minimal.

Another potential issue is that the replacement policy state of the cache will not be updated due to filtered requests, which could also result in performance loss. On average, this impact is negligible. Notable exceptions include 605.mcf_s-782B, 628.pop2_s-17B, and 654.roms_s-523B, where there is potential for performance improvements of 1.23%. Nonetheless, the overall performance impact due to the positive replacement state update is outweighed by the substantial reduction in traffic.

Applicability. SUF is applicable to any secure cache system based on invisible speculation that updates the cache hierarchy on commit.

Storage overhead. SUF is implemented with only 0.12 KB of additional storage: 0.03 KB at the LQ and 0.09 KB at the L1D. Each of the 128 entries in the LQ is extended with a two-bit hit-level field and each of the 768 entries of the L1D is extended with a single L2 writeback bit.

[1]The GhostMinion paper does not mention this bit. However, we assume it is implicit.

5.2 Timely secure prefetcher

As discussed in Section 4.4, transitioning prefetchers from on-access to on-commit leads to an average performance loss of 3% to 4%. This performance degradation stems from the new timeliness constraints, which introduces *commit-late* prefetch requests and missed prefetching opportunities. These factors contribute to lower prefetch accuracy and coverage. We observe that this behavior can be fixed by adjusting prefetch timeliness for prefetchers like IP-stride and IPCP. One of the ways to improve prefetch timeliness is to increase the prefetch distance to cover more distant memory requests, which can lead to fewer commit-late prefetch requests. However, we find that even with adaptive approaches, all the prefetchers fail to beat the Berti prefetcher.

Another option for addressing the timeliness issue is to modify the learning process of the hardware prefetcher. We pursue this approach with Berti. We focus on Berti because (i) it shows the best performance improvements over its peers and (ii) its learning is unaffected by the order of memory access streams (on-access vs on-commit) as it operates on timely deltas, which is not the case with other prefetchers.

5.2.1 Issues with the secure prefetcher

As explained in the previous subsection, Berti relies on the fetch latency to guide its learning mechanism. For secure prefetching, we identify two main issues: (i) the latency seen by the prefetcher is a misleading on-commit latency (instead of the actual on-access latency) and (ii) the deltas selected are timely at commit but not at access when data are actually needed.

To understand these problems better we start by describing the on-commit version of Berti implemented on GhostMinion. Berti is located at the L1D cache and utilizes post-commit L1D accesses and fills for training. Since GhostMinion speculatively fills the GM and moves that cache line to L1D on commit, Berti observes the on-commit write latency from the GM to L1D, instead of the fetch latency from a higher level of the memory hierarchy to GM. This alteration in fetch latency disrupts the learning process, leading to inaccurate delta learning and prefetch requests, which in turn translates into performance degradation. Figure 5.2 (in red) provides a visual example of the training and prefetch request issuance process for on-commit Berti in GhostMinion. The timeline shows a +1 delta for a given load. All L1D accesses are misses that take three cycles to fill GM and one cycle to perform an on-commit write from GM to L1D. We focus on the process of how on-commit Berti issues

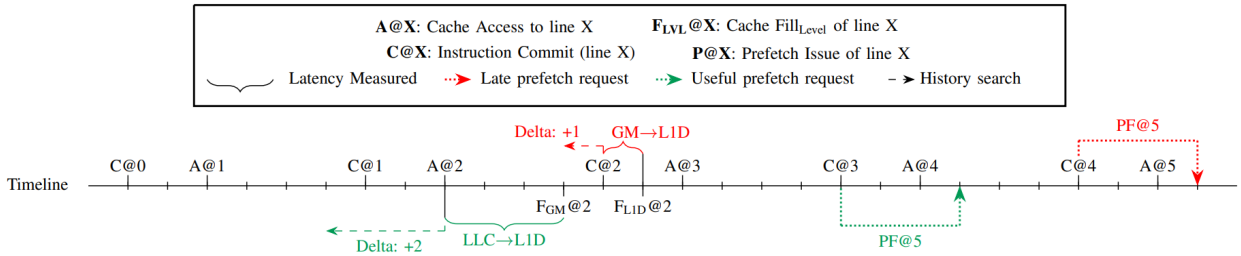


Figure 5.2: Timeline representing working of on-commit Berti (red) and TSB (green). The timeline represents the access/commit of consecutive executions of the same load instruction, with each step representing one cycle.

a prefetch request for cache line @5.

Upon the commit of the load for @2 (C@2), the learning process initiates. When cache line @2 fills the L1D cache from the GM (via on-commit write), the latency of one cycle is recorded. Consequently, Berti searches for the nearest commit instruction capable of triggering a timely prefetch request. In this example, this instruction is C@1. Berti then learns the appropriate delta, for the writeback latency, which in this case is +1. From this point forward, Berti triggers a prefetch request with delta +1 upon each instruction commit. When the load for @4 is committed (C@4), Berti issues a prefetch request for line@5 (PF@5). However, this prefetch request takes three cycles to fill the cache, resulting in a late prefetch request since the access to cache line @5 is performed two cycles later. This occurs because the learning process is performed (i) with the writeback latency, not with the fetch latency, and (ii) the deltas selected are timely at commit (C@2) but not at access (A@2). Note that this late prefetch will occur even if Berti searches for deltas based on the cache access time (A@2) rather than the commit time (C@2). Hence, both problems should be addressed in order to achieve timely secure prefetching.

5.2.2 Timely training of the secure prefetcher

Driven by the previous two observations, we propose Timely Secure Berti (TSB), a new timely training mechanism for Berti that utilizes the fetch latency to GM and computes the right delta using the access times. This way the training mechanism emulates the latency that future demand accesses will face.

TSB works as follows: first, when the demand load miss happens, it speculatively saves the necessary information for training Berti correctly, including the access time and the fetch latency to GM. At the time of commit, the commit time for each demand miss is saved

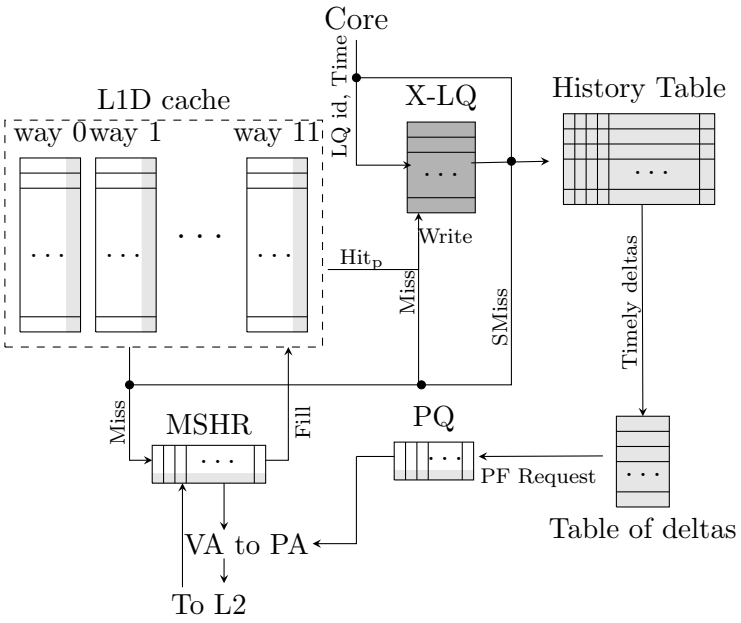


Figure 5.3: TSB design overview. Berti original hardware extensions are shown in light gray. TSB hardware extensions are shown in gray.

in history table. While training at commit, TSB searches committed instructions, which could have triggered a timely prefetch for the current load. Timely deltas are calculated by subtracting the address of each timely commit from the current address. As we show in Figure 5.2(in green), when cache line @2 fills the GM, the fill latency is calculated as the difference between the fill timestamp and the cache access timestamp and saved until commit. When instruction @2 commits (C@2) and Berti is ready to be trained: the fill latency is used to search the commit that can trigger a timely prefetch request that will hit in the access of cache line @2; once the commit is known (in our example, C@0), the delta is calculated (+2). Unlike the default on-commit version, TSB will trigger prefetch requests with delta +2, which means that C@3 triggers the prefetch request for cache line @5, resulting in a timely prefetch request.

Storage overhead. TSB uses X-LQ, an extension of the LQ needed to propagate the actual fetch latency. The *X-LQ* is dual-ported and it contains as many entries as the LQ (128 entries in our modeled system) and it is indexed with the LQ entry id (one-to-one mapping). Each entry contains a valid bit, a bit indicating that the access was a hit on a prefetched cache line (Hit_p), a 16-bit access timestamp, and a 12-bit fetch latency. TSB incurs a storage overhead of only 0.47 KB over Berti (3.01 KB over no-prefetch). Figure 5.3 shows an overview of TSB, where the light gray represents the Berti hardware and the dark gray represents the additional TSB hardware.

On an L1D miss, the valid bit is set, and the access timestamp is filled with information from the clock of the local processor (the last 16 bits of the current cycle). When the cache fill in the GM is done, the fetch latency is also recorded. On a hit to a prefetched cache line, both the valid bit and the Hit_p bit are set. In this case, the access timestamp is also recorded, but the fetch latency corresponds to the latency of the prefetched line (which has been previously computed and stored along with the L1D cache [51]). On a regular hit, the valid bit is not set, since no action will be taken at commit. When a load commits, the history table is filled for both misses and prefetch hits using commit timestamp, and the timely deltas are searched by using the fetch latency and access timestamp in X-LQ.

TSB security guarantees. TSB is trained and triggered on commit, which ensures that it is not trained on any transient instructions, and hence no prefetch requests can be generated based on transient information. In the speculative phase, the access time and fill latency of a particular request are stored in the X-LQ. The only vulnerable information is fill latency, which may be altered by a transient instruction. Any aberration to the fill latency through resource contention by a transient instruction could lead to prefetcher-based side channels. If the fill latency of a bound-to-commit instruction is altered by a transient instruction (e.g. backward-in-time attack), then the security is compromised. However, this is not possible with GhostMinion as a transient instruction can not affect the timing of a bound-to-commit instruction thanks to strictness ordering ensured by time-guarding. Also, as the X-LQ is flushed on a domain switch, the transient information stored in the X-LQ cannot be exploited by a malicious process. Moreover, the information stored by a particular load instruction in the X-LQ is accessible only by that instruction and only at commit time. No other instruction can access the information corresponding to any other instructions. This makes it secure as there is no possibility of data leakage.

TSB applicability. TSB applies to all secure cache systems (both invisible speculation and delay-based). Note that on-access prefetching is not secure with delay-based techniques like STT. STT defines "ACCESS" instructions as instructions deemed capable of accessing a secret. In STT, a load that only reads a (potential) secret but does not transmit one (i.e., "ACCESS" instruction), executes without delay. Now, if we train the prefetcher on-access, the data corresponding to the executed "ACCESS" instructions get filled into the cache hierarchy, training the prefetcher with speculative information; modifying the prefetcher state, making it insecure. As the "ACCESS" instructions are executed speculatively, the utility of our proposal is to train prefetchers on commit and cover misses that would happen at access.

5.2.3 Timely training of non-self-timing prefetchers

Berti is a self-timing prefetcher. However, prefetchers like IP-stride, IPCP, Bingo, and SPP+PPF are not. To make these prefetchers secure, they should be trained and triggered on commit. The prefetch tables should not be updated on speculative requests. To compensate for the performance loss as shown in Figure 1.1, all these prefetchers should be adapted to mitigate on-commit lateness, making them timely.

IP-stride and IPCP. For prefetchers like IP-stride and IPCP, the prefetch distance should be tuned based on the lateness. We use a mechanism that increases the distance with an increase in prefetch lateness. We calculate prefetch lateness as the ratio of late prefetch requests to useful prefetch requests. We monitor prefetch lateness every 512 misses (size of the L1 in terms of the number of cache lines) and if the prefetch lateness increases for two consecutive intervals, then we increment the prefetch distance by one. Updating distance based on the lateness of only the previous interval leads to noisy decision-making.

SPP+PPF. For SPP+PPF, we use the same mechanism of prefetch lateness-driven adaptive distance selection as done with IP-stride and IPCP. However, SPP is a different prefetcher that uses the predicted delta in the signature used for finding out the next delta, recursively. To make it adaptive, we continue the learning of SPP with on-commit requests. However, we skip the next k deltas before we start prefetching, where k is driven by the prefetch lateness. Based on empirical analysis, we find that for timely prefetching, the value of k is between two to five. As SPP is an L2 prefetcher, the monitoring interval used is 4096 misses (one-half of the size of the L2).

Bingo. Bingo is a region-based prefetcher, similar to SMS [69] where introducing timeliness is a non-trivial task. We extend Bingo with temporal information as suggested in Tempo [71] using a local tempo buffer and global tempo buffer. We then change its distance dynamically based on the prefetch lateness.

For all the prefetchers, we use the lateness threshold of 0.14, which is just less than the average lateness while we perform on-commit prefetching. However, with Bingo, the prefetch lateness threshold that we use is 0.05. In general, with Bingo, the number of late prefetch requests is lower than IP-stride, IPCP, and SPP+PPF, as shown in Figure 4.3. We also use a phase change detector as used in prior works [40] and on an application phase change, we reset the prefetch distance to the base distance used.

5.2.4 Secure data memory-dependent prefetchers

So far we have discussed how to design secure and high-performing classical prefetchers. In recent years, there are attacks like Augury [74] and Gofetch [24] that use a data memory-dependent prefetcher (DMP). DMP is a specialized prefetcher for prefetching an array of pointers and captures access patterns like A[B[i]] that classical prefetchers do not cover. The Augury attack attempts to leak the secret, B[i]. The DMP generates a prefetch request to address A[B[i]]. This is similar to the Spectre attack with the difference being that a DMP prefetcher is activated to prefetch the secret dependent load address.

Secure cache systems like GhostMinion can mitigate these attacks with minor changes, which are as follows. To design a secure interaction of the DMP prefetcher with the GhostMinion, we advocate that the prefetch addresses generated by the DMP prefetcher should be filled into the GM cache only. As there is no instruction associated with a prefetch request generated by the DMP prefetcher, the prefetched address will never get written back to the non-speculative cache hierarchy, on commit. On a domain switch, the GM cache is flushed out as usual, leaving no footprint in the cache hierarchy. Also, on a domain switch, the DMP prefetcher is flushed. To make DMP secure and high-performing, the prefetch distance of the DMP prefetcher has to be adjusted based on lateness. Note that prefetching into the small GM cache is possible with the DMP prefetcher because it is not activated on every load request and rather only on indirect accesses. We use the indirect memory prefetcher (IMP) [81] as the DMP prefetcher. We use IMP with a degree two and adaptive distance (d) to prefetch A[B[i+d]]. The adaptive distance takes care of timeliness issues. The baseline IMP has a fixed distance of 16.

In chapter 6, we provide a detailed analysis of our techniques namely secure update filter (SUF) and timely secure berti (TSB) in comparison to on-commit prefetcers. In addition, we conduct sensitivity studies to demonstrate the benefits of TSB with delay based techniques like STT. We also highlight the benefits of timely-secure versions of non-self-timing prefetchers over their on-commit counterparts.

Chapter 6

Evaluation

6.1 Methodology

We use ChampSim [8], a trace-driven simulator used for the 2nd and 3rd Data Prefetching Championships (DPC-2 [2] and DPC-3 [6]). Recent prefetching proposals are also coded and evaluated on ChampSim [20, 17, 53, 65, 51]. The version employed in DPC-3 has been extended with a decoupled front-end [56], a detailed memory hierarchy support for address translation, and a faithful DRAM model that accounts for the variable access time due to bank conflicts, close/open page, page hit/miss, etc. We calculate the dynamic energy consumption of the memory hierarchy (caches and DRAM) with CACTI-P [45] and the Micron DRAM [3] power calculator on 7 nm process technology. Table 6.1 details our baseline system configuration, similar to an Intel Sunny Cove microarchitecture [33, 4, 5].

We employ publicly available traces [7, 10] from the SPEC CPU2017 [70] and single-threaded GAP [18] benchmark suites. We limit our study to the 65 memory-intensive traces (45 from SPEC CPU2017 and all from GAP) that exhibit at least one miss per kilo-instruction (MPKI) at the LLC in our baseline system. We run both single- and multi-core simulations. We collect statistics for 200M sim-point instructions after a 50M-instruction warm-up [66]. For multi-core experiments, we simulate 150 randomly generated heterogeneous mixes of SPEC CPU2017 and GAP traces and report weighted speedup.

We evaluate the effectiveness of SUF and TSB on a GhostMinion [15] secure cache system with a 2KB GM with 1 cycle latency for different data prefetchers: IP-Stride [32] (the Intel and AMD L1D prefetcher), IPCP [53] (the winner of the DPC-3 competition), Bingo [17], SPP+PPF [20], and Berti [51]. We use the tuned implementations of each prefetcher using the parameters listed in Table 6.2.

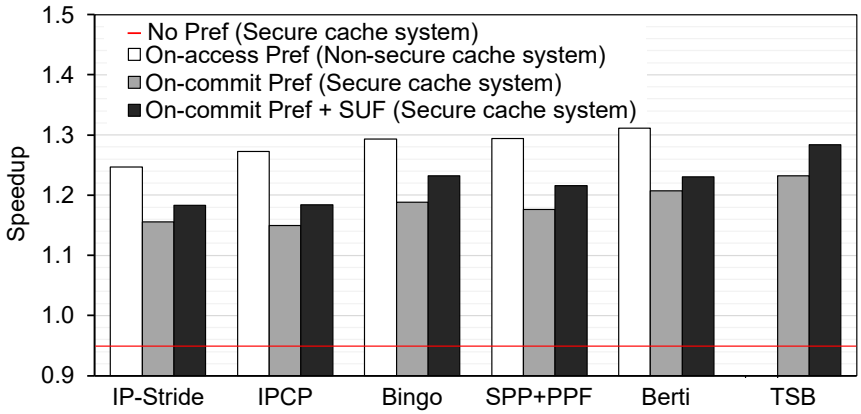


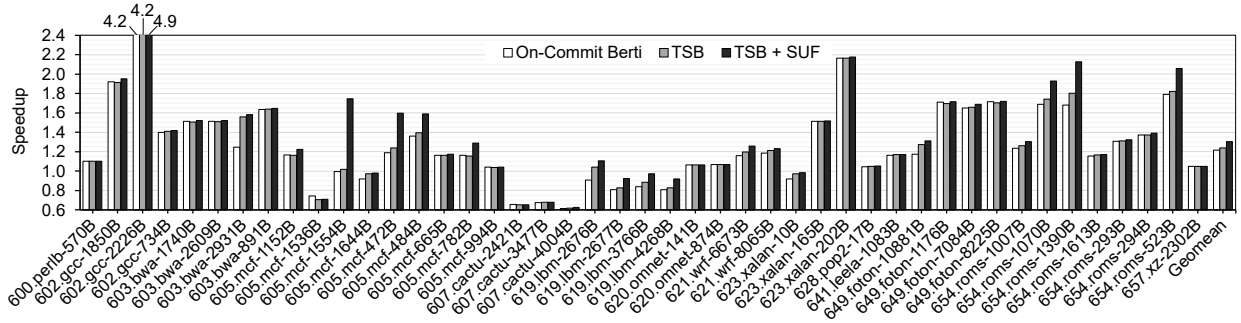
Figure 6.1: Speedup normalized to non-secure cache system with no prefetching.

Core	Out-of-order, hashed perceptron branch predictor [38], 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle STLB: 1536 entries, 12-way, 8 cycles
L1I	32 KB, 8-way, 4 cycles, 8 MSHRs, LRU
L1D	48 KB, 12-way, 5 cycles, 16 MSHRs, LRU
L2	512 KB, 8-way, 15 cycles, 32 MSHRs, LRU, non-inclusive
LLC	1 bank per core, each bank: 2 MB, 16-way, 35 cycles, 64 MSHRs, LRU, non-inclusive
DRAM	Controller: One channel/4-cores, 6400 MTPS [11], FR-FCFS, reads prioritized over writes, write watermark: 7/8th Chip: 4 KB row-buffer per bank, open page, burst length 16, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

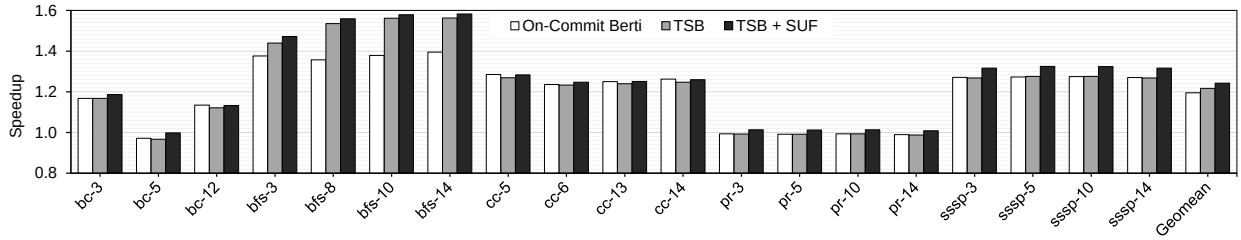
Table 6.1: Simulation parameters of the baseline system.

6.2 Results

This section shows the benefits of our two main contributions: the secure update filter (SUF) and the timely secure Berti (TSB) prefetcher. All normalized graphs are relative to a non-secure system without prefetching. If a red line is present, it represents a GhostMinion secure cache system without prefetching. When averaging results, we use the geometric mean when normalizing values and the arithmetic mean otherwise. In graphs showing average numbers, each bar represents a prefetch configuration: on-access prefetch in a non-secure cache system (white bar), on-commit prefetch in a GhostMinion cache system (gray bar), and on-commit prefetch in a GhostMinion system with the SUF mechanism (black bar). The last prefetcher is the timely secure Berti (TSB).



(a) SPEC CPU2017



(b) GAP

Figure 6.2: Speedup of Berti, TSB, and TSB+SUF normalized to a non-secure system without prefetcher. The higher the better.

6.2.1 Performance

Overall speedup. The average single-thread speedup of the prefetchers evaluated for both non-secure and secure systems is shown in Figure 6.1. The first (white) bar shows the speedup of the non-secure version of the prefetchers. The second (gray) bar shows the speedup obtained in a GhostMinion secure cache system by the same prefetchers, now being secure. All prefetchers exhibit a performance loss (between 7.3% and 9.6%) when transitioning from non-secure to secure, in part due to the ≈5% performance degradation of GhostMinion (red line). The third (black) bar shows the speedup achieved by SUF, which improves the performance of all secure prefetchers with the highest improvement of 3.7% for Bingo and the lowest of 1.9% for Berti.

The last set of bars illustrates the improvements of our TSB proposal, only for secure scenarios. TSB without SUF achieves the same speedup as secure Berti + SUF (23.0%). When SUF is added to TSB, the speedup increases by 4.2%, reaching a total of 28.4% (over baseline). Among the on-access prefetchers in a non-secure system, only Berti offers a clear performance advantage over our secure TSB+SUF; SPP+PPF, Bingo, and IPCP offer similar performance (less than a 0.8% performance difference) and IP-Stride has lower performance

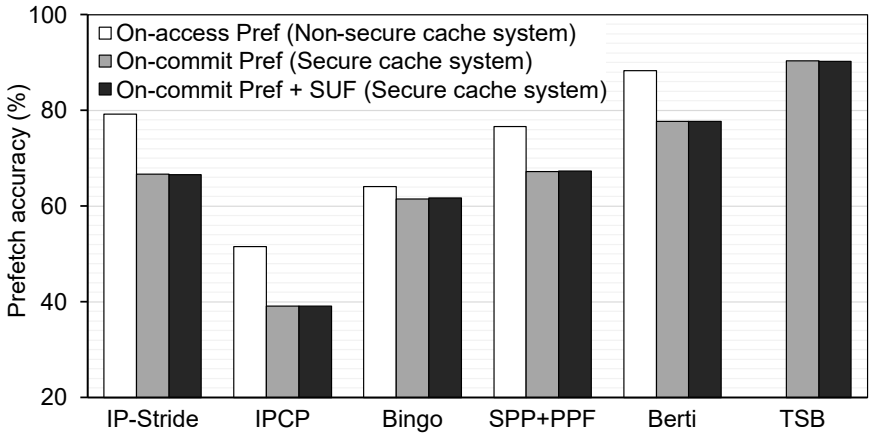


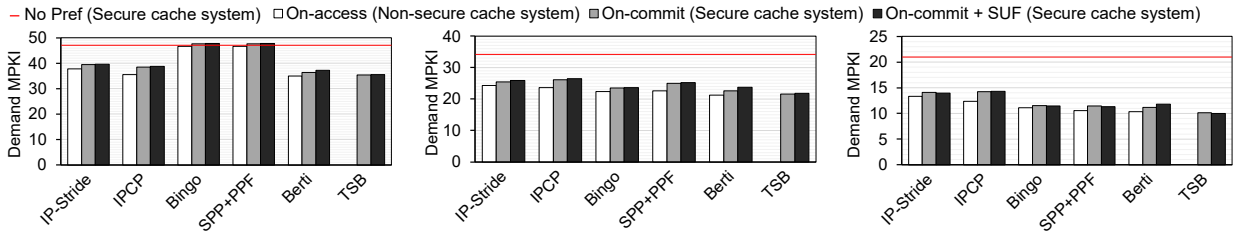
Figure 6.3: Average prefetch accuracy. The higher the better.

Prefetcher	**Configuration**	**Size**
IP-Stride	1024 entries	8KB
IPCP [53]	128-entry IP table, 8-entry RST table, and 128-entry CSPT table	0.87KB
SPP+PPF [20]	256-entry ST, 512-entry 4-way PT, 8-entry GHR, Perceptron weights: 4096×4, 2048×2, 1024×2, and 128×1 entries, 1024-entry prefetch and reject tables	39.2 KB
Berti [51]	128-entry History Table, 16-entry Delta table with 16 deltas	2.55 KB
Bingo [17]	2 KB region, 64/128/16K-entry FT/AT/PHT	124 KB

Table 6.2: Configurations of evaluated prefetchers

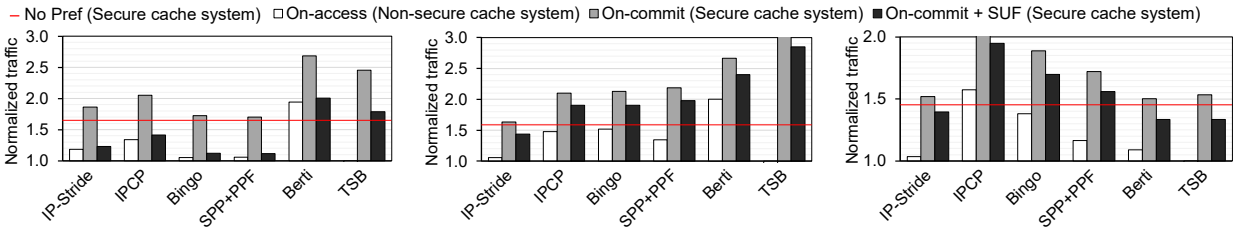
(2.9%). Importantly, TSB+SUF mitigates the performance degradation of using a secure system from 5.1% (in a system without prefetching) to 2.1% (in a system with prefetching). Finally, TSB can also be applied to non-secure cache systems, thus removing any speculative side-channel attack induced by the prefetcher. In that case TSB performs on par with respect to on-access Berti (speedup for TSB 1.310 (not shown in Figure 6.1) vs. speedup for Berti 1.311).

Individual speedup. Figure 6.2 shows the individual speedup for on-commit Berti, TSB, and TSB+SUF. For SPEC traces, TSB improves performance by more than 5% in 7 out of 45 traces (15.6% of all traces) when compared with the on-commit Berti. For *603.bwaves_s-2931B*, performance improves by 24.9% over Berti, because it has a large fetch latency which is learned correctly in TSB. Our new learning system allows TSB to learn better and more accurate deltas, which provides better accuracy and coverage. TSB+SUF



(a) L1D (b) L2 (c) LLC

Figure 6.4: Average L1D, L2, and LLC demand MPKIs. The lower the better.

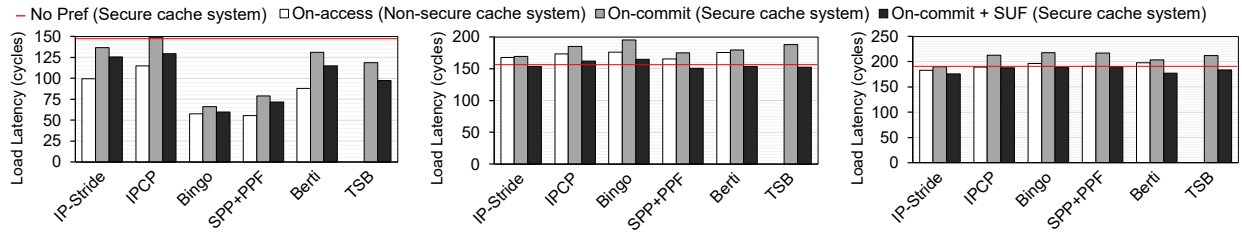


(a) L1D (b) L2 (c) LLC

Figure 6.5: Normalized L1D, L2 and LLC traffic. The lower the better.

achieves more than 5% performance improvement in 18 out of 45 traces (40% of all traces), with a maximum improvement of 75.8% in *605.mcf_s-1554B*. After analyzing its behavior, we detected that this improvement comes from the reduction in the number of cycles that the L2 MSHR is found full, which drops by 42.2%. TSB and TSB+SUF only sees a performance drop of more than 1% in one application, *605.mcf_s-1536B*. As for GAP, TSB achieves better performance in all *bfs* traces with an average improvement of 10.8%, also because of their large fetch latency. TSB+SUF achieves slightly better performance in all benchmarks, with more than 3.8% average performance improvement in *sssp* traces due to the inclusion of the SUF filter. Interestingly, TSB and TSB+SUF do not degrade performance in any trace. This is because, on average, SUF filters accurately for 99.3% of the time, with the maximum accuracy of 99.9% for `654.roms_s-1613B` and a minimum of 87.26% for `605.mcf_s-1554B`, improving the effectiveness of GhostMinion with prefetching.

Prefetch accuracy. Figure 6.3 shows the accuracy of the different prefetchers. Compared to the on-access prefetcher, on-commit prefetchers experienced a decrease in accuracy in all prefetchers, with a maximum of 24.0% in IPCP and a minimum of 4.1% in Bingo. Because SUF does not affect the timeliness of the prefetcher, it does not modify the accuracy of any prefetcher. The improvements in the learning system for TSB and TSB+SUF are



(a) L1D (b) L2 (c) LLC

Figure 6.6: Demand load latency in cycles at L1D, L2 and LLC for all prefetchers. The lower the better.

reflected in their accuracy, which is 2.4% better than on-access Berti in non-secure systems, achieving an accuracy of 90.3%.

Prefetch coverage. Figure 6.4 shows the demand misses per kilo instructions (MPKI) at the L1D, L2, and LLC (Y axis) with prefetchers (X-axis). All prefetchers exhibit an average MPKI increase of 4.3%, 7.4%, and 8.3% at the L1D, L2, and LLC caches, respectively, when they move from on-access in a non-secure system to on-commit in a secure system. As with accuracy, since SUF does not change the way that prefetchers work, its coverage remains the same for all of the prefetchers. Our secure prefetcher, TSB, achieves an MPKI reduction of 2.4%, 4.9%, and 8.8% at the L1D, L2, and LLC caches, respectively, compared to on-commit prefetcher (Berti). TSB and TSB+SUF have the same coverage as on-access Berti in a non-secure cache system. The superior coverage of TSB can be attributed to the correct latency seen by it, which provides the least late and incorrect prefetch requests.

6.2.2 Memory hierarchy traffic, latency and energy with SUF

Memory hierarchy traffic. GhostMinion adds a significant amount of traffic due to the writeback and re-fetch requests. Figure 6.5 shows the traffic between the different levels of cache (Y-axis) under the different hardware prefetchers (X-axis). All prefetchers increase traffic compared to its on-access version by an average of 54.7%, 46.6%, and 40.4% in L1D, L2, and LLC, respectively. SUF mitigates the increase in traffic generated by GhostMinion writeback and re-fetch actions with all the prefetchers (an average reduction in L1D traffic by 30.1%).

Latency. Figure 6.6 shows the number of cycles that load misses take to fill the L1D, L2, and LLC caches (Y axis). The utilization of a secure system cache increases the latency of all prefetchers at all levels, with a significant increase at the L1D cache. On average,

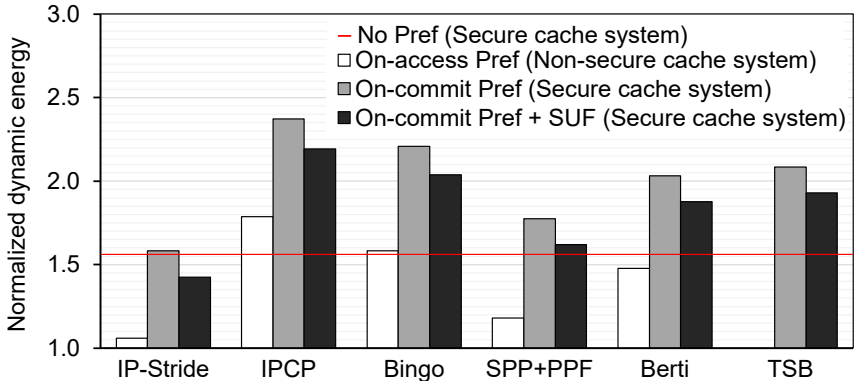


Figure 6.7: Normalized dynamic energy consumption. The lower the better.

the on-commit prefetcher sees an increase in latency of 34.8% in the L1D cache, 5.2% in the L2 cache, and 8.8% in the LLC cache with respect to on-access prefetchers. Of all the prefetchers, Berti is the one whose latency is most affected, with a maximum increase of 49.4% in the L1D cache (from 87.8 cycles to 131.2 cycles). This is because Berti is the most aggressive of all the prefetchers, triggering more prefetch requests and increasing memory traffic. Thanks to the reduction in traffic between cache levels, SUF is able to reduce the latency penalty introduced by GhostMinion by more than 12% at all cache levels, which translates into higher performance.

Energy. Figure 6.7 shows the normalized dynamic energy consumption in the memory hierarchy (L1D, L2C, LLC) (Y-axis) for the different prefetcher configurations (X-axis). There is a direct correlation between traffic and dynamic energy consumption overhead in the memory hierarchy. The secure system has extra traffic generated by GM, which increases the base energy consumption for all prefetchers. The on-commit version of the prefetcher increases energy consumption by an average of 41.8%, compared to the on-access version. SUF is able to reduce this increase in energy from 41.8% to 30.0%. On-commit IP-Stride+SUF is able to consume less than the system without prefetching, thanks to SUF reducing all the redundant traffic from GM. TSB and TSB+SUF show higher dynamic energy consumption than prefetchers like IP-Stride or Berti because they trigger a greater number of prefetch requests, but they also achieve better performance.

6.2.3 Multi-core performance

Figure 6.8 shows the performance of SUF and TSB on a 4-core simulated system. The mixes have been sorted in increasing order of speedup. Compared to the non-secure baseline, GhostMinion incurs an average performance overhead of 16.8%. Only 14 mixes show a

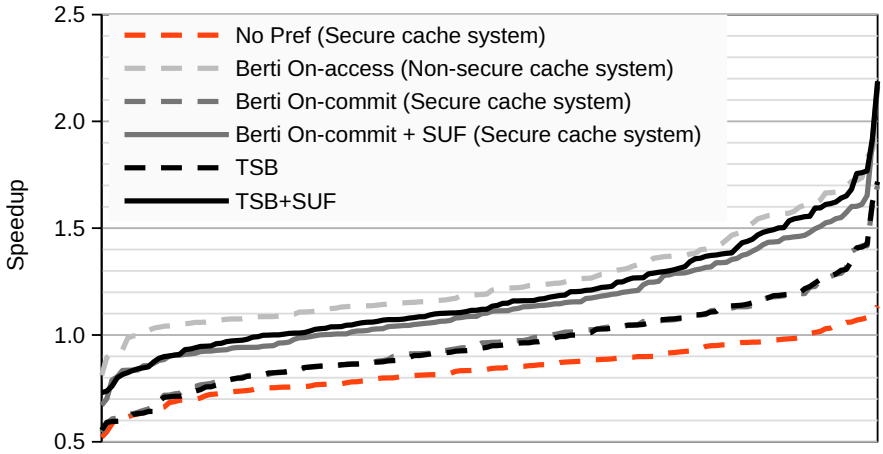


Figure 6.8: Speedup for 4-core mixes normalized to the non-secure cache system with no prefetching.

performance improvement.

As multi-core execution increases the traffic in the higher levels of the cache, it increases the latency of memory requests. Hence, the benefits of SUF reducing the traffic are more acute than in the single-core execution. The SUF filter improves performance over a secure cache system in all mixes. TSB+SUF improves the performance over the non-secure baseline by 16.1%, followed by on-commit Berti+SUF (12.4%). When compared with secure on-commit Berti, TSB+SUF improves performance by 23%. Note that SUF's average accuracy drops marginally from 99.95% in single-core to 99.25% in a multi-core system. This marginal accuracy drop is because of the cross-core evictions at the shared LLC, and this drop does not affect overall performance.

SMT-based multi-core systems. The effectiveness of SUF is driven by its accuracy and in an SMT core, one thread can evict cache lines of other threads from both L1D and L2. When we apply SUF and TSB on a 2-way SMT-based multi-core processor, we find that the average accuracy is still over 99%. The reason is that, on average, it takes 200 cycles from the time a speculative load request is generated till it gets committed. This latency is as low as 46.93 cycles for `603.bwaves-2931B`. So, the probability of an eviction at L1/L2/LLC that can lead to a mis-prediction by SUF is extremely low, which is also evident from high accuracy. There are mixes with multiple copies of `605.mcf-1554B`, `cc-14B`, `bc-0B`, and `bc-5B` where the accuracy dropped to 91.74%. In summary, the effectiveness of SUF and SUF+TSB remains similar in multicore systems as shown in Figure 6.8 and also in SMT-based multicore systems.

6.2.4 Sensitivity studies

So far, we have shown the effectiveness of TSB with GhostMinion. Now, we show the effectiveness of TSB with STT [80]. We also show the effect of timely and secure prefetching with non-self-timing prefetchers.

STT+TSB. On single-core, on average, STT with TSB outperforms STT+on-safe prefetching by 2.3% and as high as 7.1% for `gcc`, with an overall average speedup of 21% compared to non-secure baseline with no prefetching. There is no need for SUF with STT as STT is a delay-based technique.

Timely secure prefetching with non-self-timing prefetchers. Based on the enhancements suggested in Section 5.2.3, we now discuss their effects on naive on-commit IP-stride, IPCP, Bingo, SPP+PPF, and IMP, and name it TS-stride, TS-IPCP, TS-Bingo, TS-SPP+PPF, and TS-IMP, respectively. In single core system, with GhostMinion, TS-stride, TS-IPCP, TS-Bingo, and TS-SPP+PPF outperform on-commit versions of IP-stride, IPCP, Bingo, and SPP+PPF by 3.1%, 2.84%, 1.92%, and 2.64%, respectively. In the case of multi-core systems, the performance improvements go up significantly with an average of more than 11.36% with secure and timely versions of the prefetchers compared to on-commit prefetchers. Compared to on-commit IMP, which prefetches into the cache hierarchy, TS-IMP, which prefetches into the GM cache shows an average performance improvement of 2.73%. We use the gem5 [48] simulator to simulate IMP with GhostMinion. The current version of ChampSim cannot simulate IMP as ChampSim does not simulate data associated with addresses. Note that, overall, TSB is the high-performing and secure prefetcher outperforming the timely secure version of all prefetchers by 4.1%.

Chapter 7

Conclusion and Future Work

Secure cache systems mitigate speculative execution attacks like Spectre by providing strictness ordering. We showed that hardware prefetching is fundamental for hiding the performance overhead of a secure cache system. We performed, for the first time a comprehensive evaluation of the interaction between state-of-the-art prefetch mechanisms on a secure cache system. Our analysis shows that (i) state-of-the-art secure memory hierarchies prevent prefetchers from achieving their true potential and even, in some cases, turning significant speedups into no performance at all, and (ii) prefetching techniques perform sub-optimally when moving to commit due to loss of timeliness. We addressed these two problems and improved the effectiveness of hardware prefetchers. Our proposal improves the single-thread performance by 6.3% and the multi-core by 23.0% (over the top-performing Berti prefetcher) with 0.59 KB of storage overhead per core.

Designing a secure L2 prefetcher presents significant challenges due to the unavailability of information at the L2 cache during the instruction commit phase. This limitation can lead to various issues in prefetching efficiency and accuracy. Additionally, the performance impact of securing other microarchitectural structures remains unexplored and requires further analysis.

Chapter 8

Acknowledgements

Firstly, I express my gratitude to my advisor, **Prof. Biswabandan Panda**, for his support and invaluable guidance throughout my journey in this institution. His expertise, patience, and encouragement have been instrumental in shaping this research project and my academic growth.

I extend my appreciation to **Prof. Alberto Ros** and **Agustin Navarro-Torres** for their collaboration on this work. Their contributions, expertise, and insightful suggestions have immensely enriched the research and added depth to our findings.

I would also like to express my gratitude to my fellow **CASPER** friends, especially **Shubham Roy** and **Nishkarsh Gautam**. Our discussions, brainstorming sessions, and valuable insights have played a crucial role in shaping the direction of this research.

Sumon Nath
IIT Bombay

Bibliography

[1] "Gnu privacy guard," http://www.gnupg.org, 2013.

[2] "The 2nd data prefetching championship (dpc-2)," https://comparch-conf.gatech.edu/dpc2/, Jun. 2015.

[3] "Micron DRAM power calculator," https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf, Dec. 2015.

[4] "SunnyCove microarhcitecture," https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove, May 2018.

[5] "SunnyCove microarhcitecture latency," https://www.7-cpu.com/cpu/Ice_Lake.html, May 2018.

[6] "The 3rd data prefetching championship (dpc-3)," https://dpc3.compas.cs.stonybrook.edu/, Jun. 2019.

[7] "SPEC CPU2017 traces for champsim," https://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/, Feb. 2019.

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

[9] "DDR5 SDRAM," https://en.wikipedia.org/wiki/DDR5_SDRAM, 2020.

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

[11] "DDR standards," https://en.wikipedia.org/wiki/Double_data_rate, Apr. 2024.

[12] "DDR5 memory standard: An introduction to the next generation of dram module technology," https://www.kingston.com/en/blog/pc-performance/ddr5-overview, Jan. 2024.

[13] P. Aimoniotis, C. Sakalis, M. Själander, and S. Kaxiras, "Reorder buffer contention: A forward speculative interference attack for speculation invariant instructions," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 162–165, 2021.

[14] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *47th Int'l Symp. on Computer Architecture (ISCA)*, 2020, pp. 132–144.

[15] S. Ainsworth, "Ghostminion: A strictness-ordered cache system for spectre mitigation," in *54th Int'l Symp. on Microarchitecture (MICRO)*, 2021, pp. 592–606.

[16] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, 2009.

[17] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[18] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, Aug. 2015.

[19] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *26th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2021, pp. 1046–1060.

[20] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13.

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

[22] J. V. 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 *27th USENIX Security Symposium (USENIX Security 18)*, Aug. 2018, pp. 991–1008.

[23] C. Canella, J. V. 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 *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019, pp. 249–266.

[24] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security*, 2024.

[25] Y. Chen, A. Hajiabadi, L. Pei, and T. E. Carlson, "New cross-core cache-agnostic and prefetcher-based side-channels and covert-channels," *arXiv preprint arXiv:2306.11195*, 2023.

[26] Y. Chen, L. Pei, and T. E. Carlson, "Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *28th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2023, pp. 16–32.

[27] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2012.

[28] I. Corporation, "Intel® 64 and ia-32 architectures software developer's manual," https://software.intel.com/sites/default/files/managed/39/c5/325462-sdmvol-1-2abcd-3abcd.pdf, December 2018.

[29] P. Cronin and C. Yang, "A fetching tale: Covert communication with the hardware prefetcher," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 101–110.

[30] ——, "A fetching tale: Covert communication with the hardware prefetcher," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 101–110.

[31] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, 2012.

[32] J. Doweck, "Inside intel core microarchitecture and smart memory access," in *Intel whitepaper*, 2006, pp. 1–13.

[33] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," https://www.agner.org/optimize/microarchitecture.pdf, 2020.

[34] J. Fustos, M. Bechtel, and H. Yun, "Spectrerewind: Leaking secrets to past instructions," in *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, 2020, pp. 117–126.

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

[36] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.

[37] D. A. Jiménez, "Fast path-based neural branch prediction," in *36th Int'l Symp. on Microarchitecture (MICRO)*, 2003, pp. 243–252.

[38] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 197–206.

[39] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Trans. Comput. Syst.*, pp. 369–397, nov 2002.

[40] N. S. Kalani and B. Panda, "Instruction criticality based energy-efficient hardware data prefetching," *IEEE Comput. Archit. Lett.*, vol. 20, no. 2, pp. 146–149, 2021. [Online]. Available: https://doi.org/10.1109/LCA.2021.3117005

[41] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *56th Design Automation Conference (DAC)*, 2019, pp. 1–6.

[42] J. Kim, P. V. Gratz, and A. L. N. Reddy, "Lookahead prefetching with signature path," in *2nd Data Prefetching Championship*, Jun. 2015.

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

[44] A. B. Kvalsvik, P. Aimoniotis, S. Kaxiras, and M. Själander, "Doppelganger loads: A safe, complexity-effective optimization for secure speculation schemes," in *50thInt'l Symp. on Computer Architecture (ISCA)*, 2023, pp. 1–13.

[45] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

[46] 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 *27th USENIX Security Symposium, (USENIX Security 18)*, 2018, pp. 973–990.

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

[48] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[49] P. Michaud, "Best-offset hardware prefetching," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 469–480.

[50] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: From general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 415–429.

[51] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibánez, V. Viñals-Yúfera, and A. Ros, "Berti: an Accurate Local-Delta Data Prefetcher," in *55th Int'l Symp. on Microarchitecture (MICRO)*, 2022, pp. 975–991.

[52] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed., 2006, pp. 1–20.

[53] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 118–131.

[54] Phoronix, "Bisected: The unfortunate reason linux 4.20 is running slower," https://www.phoronix.com/scan.php?page=article&item=linux-420-bisect&num=1, 2018.

[55] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: Attacking arm pointer authentication with speculative execution," in *49th Int'l Symp. on Computer Architecture (ISCA)*, 2022, pp. 685–698.

[56] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[57] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, pp. 120–126, feb 1978.

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

[59] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *52th Int'l Symp. on Microarchitecture (MICRO)*, 2019, pp. 73–86.

[60] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander, "Ghost loads: What is the cost of invisible speculation?" in *16th Int'l Conf. on Computing Frontiers (CF)*, 2019, pp. 153–163.

[61] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 723–735.

[62] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Understanding selective delay as a method for efficient secure speculative execution," *ACM Trans. on Computers*, 2020.

[63] A. Seznec, "A new case for the tage branch predictor," in *44th Int'l Symp. on Microarchitecture (MICRO)*, 2011, pp. 117–127.

[64] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, p. 23, 2006.

[65] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *The 3rd Data Prefetching Championship*, Jun. 2019.

[66] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

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

[68] T. Solanki and B. Panda, "Specpref: High performing speculative attacks resilient hardware prefetchers," in *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2022, pp. 57–60.

[69] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 252–263.

[70] Standard Performance Evaluation Corporation, "SPEC CPU2017," http://www.spec.org/cpu2017, 2017.

[71] M. Sutherl, A. Kannan, and N. E. Jerger, "Not quite my tempo: Matching prefetches to memory access times," in *2nd Data Prefetching Championship*, Jun. 2015.

[72] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, pp. 219–234, 2019.

[73] P. Turner, "Retpoline: a software construct for preventing branch-target injection," https://support.google.com/faqs/answer/7625886, December 2018.

[74] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *2022 IEEE Symposium on Security and Privacy (S&P)*, 2022.

[75] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *52th Int'l Symp. on Microarchitecture (MICRO)*, 2019, pp. 572–586.

[76] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Comput. Surv.*, pp. 1–36, 2021.

[77] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *51th Int'l Symp. on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[78] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, "Pensieve: Microarchitectural modeling for security evaluation," in *50th Int'l Symp. on Computer Architecture (ISCA)*, 2023, pp. 1–15.

[79] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[80] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data," in *52ndInt'l Symp. on Microarchitecture (MICRO)*, 2019, pp. 954–968.

[81] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *48th Int'l Symp. on Microarchitecture (MICRO)*, 2015, pp. 178–190.