UNIVERSITY OF CALIFORNIA
RIVERSIDE

Architectural Support for Securing Systems Against Micro-Architectural Attacks

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Esmaeil Mohammadian Koruyeh

September 2023

Dissertation Committee:

Dr. Nael Abu Ghazaleh, Chairperson
Dr. Rajiv Gupta
Dr. Chengyu Song
Dr. Mohsen Lesani

The Dissertation of Esmaeil Mohammadian Koruyeh is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

My PhD journey was a transformative experience, filled with challenges but also moments of insight and growth. Throughout it all, I was fortunate to never feel alone, surrounded by those who guided and supported me.

First and foremost, I owe a deep debt of gratitude to Professor Nael Abu-Ghazaleh. Your kindness and patience have been my constant throughout this journey. In every challenge, your belief in me and your steady guidance made all the difference. Beyond the academic insights, it was your genuine care and understanding that shaped my experience most profoundly. You have been much more than an advisor to me; you've been a guiding light in both my academic and personal growth. For all of this and more, thank you from the depths of my heart.

I extend my sincere thanks to my committee members, Professors Rajive Gupta, Chengyu Song, and Mohsen Lesani, for their invaluable feedback. Your combined expertise and insights have been instrumental in refining this work to its present form.

To my lab mates: Shafiur Rahman, Fatemah Alharbi, Hoda Naghibi Jouybari, Hodjat Asghari-Esfeden, Sankha Dutta, Ahmed Abdo, Shirin Haji Amin Shirazi, Sakib Md Bin Malek, Jason Zellmer, Abdulrahman Bin Rabiah, and notably, Khaled Khaswaneh - our times together, from coffee breaks to shared successes, were invaluable. Your individual insights and collective support profoundly enriched both my academic work and my personal experiences during our time together.

To my friends, Hadi Zamani, Mohammad Farajollahi, Alireza Abdoli, Sajjad Bahrami, Mehdi Kohansal, Valeh Ebrahimi, and Shahrzad Haji Amin Shirazi - each one of you holds a special place in this journey. Together, you became more than friends; you were the family I found away from home. Whether it was the unwavering support, shared laughter, or quiet moments of understanding, you all made here feel familiar and comforting.

Lastly, Being miles away in a different country has underscored the importance and value of family. To my parents, Mohammad Ali and Maryam; my siblings, Parisa, Ebrahim, and Ali; and my beloved nephew and niece, Soroush and Rosha Heidari – despite the distance, your love, guidance, and unwavering support have always made me feel close to home. Your constant encouragement and the sacrifices you've made have been the bedrock upon which my resilience was built. While I embarked on this academic journey in a foreign land, it was your unwavering faith in me that kept my compass directed towards my goals.

To my family.

ABSTRACT OF THE DISSERTATION

Architectural Support for Securing Systems Against Micro-Architectural Attacks

by

Esmaeil Mohammadian Koruyeh

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2023
Dr. Nael Abu Ghazaleh, Chairperson

Cybersecurity threats continue to grow as the number of attacks on all layers of computing systems by motivated and sophisticated attackers continues to grow over the past several years. The recent Meltdown and Spectre attacks have shown that computer architecture and hardware also offer software-exploitable attack surfaces that can be used to compromise systems. This dissertation investigates the boundary between hardware and software with respect to computer security, exploring attacks that originate in the hardware, and conversely architecture support for securing systems and software.

In this dissertation, we introduce SpectreRSB, a new Spectre attack that we developed targeting the return stack buffer used to optimize the execution of return instructions on modern CPUs. We show that both local attacks (within the same process such as Spectre 1) and attacks on SGX are possible by constructing proof of concept attacks. We also analyze additional types of the attack on the kernel or across address spaces and show that under some practical and widely used conditions they are possible.

Having demonstrated the possibility of Spectre attacks, the dissertation explores general defense approaches to counter this important vulnerability class. The first defense we contribute is SpecCFI, a new CPU design principle that secures modern processors against Spectre attacks with the help of program analysis while retaining the benefits of speculative execution. SpecCFI represents a new approach to securing architecture by using techniques that protect software to enforce secure operation even during speculative execution.

We extended the idea of using program analysis during speculation, to defend against more variants of transient execution attacks. More specifically, we proposed the Speculative Execution Regulation (SER) as a general class of defense. Since speculative execution states are accessible to an attacker, SER seeks to ensure that security invariants are enforced even during speculation.

The third contribution of the dissertation is a general approach to securing processors against transient execution attacks by making speculation leakage free in a principled way, enabling CPUs to retain the performance advantages of speculation while removing the security vulnerabilities it exposes. Our defense, SafeSpec, is a design principle where speculative state is stored in temporary shadow structures, that are not accessible to committed instructions.

The final contribution of my dissertation is the possibility of side-channel attacks on new emerging memories to find potential vulnerabilities. More specifically we showed the possibility of side-channel attacks when Intel Optane persistent memory operates as the main memory in the system and DRAM is considered as the last level cache. The timing difference between accessing the DRAM and Non-Volatile RAM (NVRAM) can create a side channel.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of hardware vulnerabilities has increased significantly in the past few years. With the development of countermeasures against software vulnerabilities, more attention from attackers has been dedicated to finding and exploiting vulnerabilities in hardware. In modern computing systems, hardware is considered the root of trust which provides many security-critical services such as secure boot, secure computation, cryptography acceleration, key management, memory access control, and more. Exploiting hardware vulnerabilities could undermine all the security guarantees offered by the hardware. In addition, hardware designers have proposed and utilized different optimization techniques to improve performance and energy efficiency. There is a tendency, however, to overlook the security of these new optimizations in the design process.

In today's computing devices, caches are well-studied optimizations that aim at hiding memory access latency. However, researchers have shown that computer systems are vulnerable to side-channel attacks which can leak sensitive data like cryptography keys due to timing differences between cache accesses and main memory accesses. More specifically, an adversary can gain information about a system if they can measure leakage in its physical implementation, even if the implementation is otherwise fully secure; such an attack is called a side channel attack. Micro-architectural side channel attacks occur when leakage is measured through process interactions through the shared resources on a processor, including caches and other structures. These attacks are more dangerous on the cloud since tenants share resources.

The recent transient execution attacks such as Spectre and Meltdown, which target speculative execution, a computer architecture optimization technique used for performance, demonstrated how speculative execution can be exploited to enable the disclosure of secret data across both software and hardware isolation boundaries. Specifically, attackers can misguide the processor to speculatively execute a read instruction with an address under their control. Although the speculatively read values are not visible to programs through the architectural state, since the mis-speculation effects are eventually undone, they can be communicated out using a covert channel.

New computing models with combinations of processors augmented by a variety of interconnect and memory architectures lead to new attacks. Accelerators as co-processors in such systems can be either the target of attacks themselves or serve as a vector for launching or amplifying attacks that compromise the main processor or the whole system, bypassing existing mitigations already in place. Having investigated the security of processors, the secure design beyond the CPUs to include cross-processor (CPU-GPU, CPU-FPGA, Multi-

GPU, ...) attack scenarios and consider all types of processors, memory, and interconnect components in heterogeneous computing systems need to get more attention.

This dissertation pursues two directions:

- Exploring vulnerabilities in computer hardware that are exploitable by software.
- Designing secure systems to protect against these vulnerabilities and, where beneficial, leveraging program analysis.

1.1 Contributions of the Dissertation

This dissertation investigates the boundary between hardware and software with respect to computer security, exploring attacks that originate in the hardware, and conversely architecture support for securing systems and software.

1.1.1 Exploring vulnerabilities in computer hardware

SpectreRSB

Spectre attacks rely on the attacker manipulating the branch predictors within the processor. The first exploration and disclosure of the attack identified and attacked two predictors: the direction predictor (Spectre v1) and the branch target predictor (Spectre v2). In this thesis, we identified a new Spectre class that targets the return stack buffer (RSB), a structure within modern processors used to predict the target of return instructions. Specifically, we show how an attacker can manipulate the state of the RSB to cause speculative execution of a payload attack gadget that reads and exposes sensitive information from kernel space or otherwise protected data.

Intel Optane Side-channel attack

The advancement in data-intensive and high-performance computing, e.g. Large scale machine learning and large-scale graph analytics workloads, has increased the demand for more efficient and scalable memory systems besides specialized accelerators. Emerging Non-Volatile Memories (NVMs) are promising candidates for bridging the bandwidth gap between processor and memory. NVMs can be integrated into different levels of memory hierarchy from caches to main memory. Due to promising aspects, emerging NVMs are already being commercialized by industries e.g Intel's 3D Xpoint and Intel's Optane. The unique characteristics of these emerging memories may lead to security and privacy issues that need to be investigated. In this dissertation, we have studied the possibility of side-channel attacks on these new memories to find the potential vulnerabilities. More specifically we showed the possibility of side-channel attacks when Intel Optane persistent memory operates as the main memory in the system and DRAM is considered as the last level cache. The timing difference between accessing the DRAM and Non-Volatile RAM (NVRAM) can create a side channel.

1.1.2 Designing secure systems against Micro-architectural attacks

Over the past few years, many hardware-level security vulnerabilities such as Spectre, Meltdown, and more recently MDS have been identified which exploit speculative execution and out-of-order execution of processors. Processors are vulnerable to these attacks since they do not consider all available information from software at the time of speculation and so they have to guess program execution paths or use stale data from their internal buffers. To mitigate these vulnerabilities, we proposed to apply program analysis (e.g static analysis) techniques at the compiler level. These techniques can provide enough information for the processor to disambiguate speculation or only forward data from its internal buffers to the correct instructions. Each class of aforementioned attacks targets different components of the microarchitecture and has unique properties. To mitigate them, the source of speculation should be identified and the proper program analysis techniques should be applied to disambiguate each speculation source. In this dissertation, we proposed SpecCFI, a lightweight solution to prevent the two most dangerous Spectre variants: Spectre-BTB (v2) and Spectre-RSB (v5). SpecCFI prevents these attacks by using control-flow integrity (CFI) principles to identify when a prediction is likely erroneous and constraints speculation if it is, making it the first software-hardware approach to protecting systems against Spectre attacks. We extended the idea of using program analysis during speculation, to defend against more variants of transient execution attacks like MDS and LVI attacks. More specifically, we proposed the Speculative Execution Regulation (SER) as a general class of defense. Since speculative execution states are accessible to an attacker, SER seeks to ensure that security invariants are enforced even during speculation.

We also introduced a new model (SafeSpec) for supporting speculation in a way that is immune to the side-channel leakage necessary for attacks such as Meltdown and Spectre. In particular, SafeSpec stores side effects of speculation in separate structures while the instructions are speculative. The speculative state is then either committed to the main CPU structures if the branch commits, or squashed if it does not, making all direct side effects of speculative code invisible.

This dissertation is organized as follows:

Chapter 2 Background and Related Work: This chapter starts by providing a brief overview of cache side-channel attacks. We then describe a general overview of transient execution attacks.

Chapter 3 SpectreRSB: This chapter introduces a new variant of Spectre attack which exploits the Return Stack Buffer (RSB) in modern processor's pipeline. We will discuss the principle of the attack and then show that both local attacks (within the same process such as Spectre 1) and attacks on SGX are possible by constructing proof of concept attacks. We also analyze additional types of attacks on the kernel or across address spaces and show that under some practical and widely used conditions they are possible.

Chapter 4 - Speculative Execution Regulation (SER): This chapter deals with the utilization of program analysis against transient attacks. In this chapter, We will describe SpecCFI, which aims to restrict the control flow of the program to a predefined set of targets. We then show the effectiveness of this approach against Spectre attacks with a very low performance overhead.

Chapter 5- SpecASAN*: Protecting Data Speculation using Speculative Address Sanitizer* Expanding upon the concept of Speculative Execution Regulation (SER) that leverages program analysis to defend against transient attacks, this chapter presents SpecASAN. As an embodiment of SER, SpecASAN uses the Data Flow Integrity principle to mitigate a broad spectrum of transient attacks, including Meltdown, MDS, and some variants of the Spectre attacks.

Chapter 6 - SafeSpec: We introduce (SafeSpec), a new model for supporting speculation in a way that is immune to side-channel leakage by storing side effects of speculative instructions in separate structures until they commit. Additionally, we address the possibility of a covert channel from speculative instructions to committed instructions before these instructions are committed. We develop a cycle-accurate model of the modified design of an x86-64 processor and show that the performance impact is negligible.

Chapter 7 - Intel Optane side-channel attacks: We report the results of reverse engineering of Intel Optane memory when it operates in memory mode. We then show several possibilities that can lead to side-channel attacks based on reverse engineering results. In end, we discuss the side-channel attacks that we built against Intel Optane memory.

Chapter 2

Background

Speculative execution has been an important part of processor architecture starting from the 1950s. The IBM Stretch processor implemented a predict not-taken branch predictor to avoid stalling a processor pipeline when a branch is encountered [30]. Computer architecture advanced rapidly starting in the early 1980s leading to rapid increase in the amount of speculation that is exploited with aggressive out-of-order execution. This speculation is supported by sophisticated branch predictor designs [126, 219, 277] that are highly successful in predicting both the branch direction and its target address. In particular, the number of pipeline stages in production CPUs has continued to grow to the point where modern pipelines commonly have between 15 and 25 stages. With out-of-order execution, when a branch instruction stalls (e.g., due to a cache miss on which it depends), instructions that follow the branch are continuously being issued speculatively. Thus, the speculation window where instructions are getting executed speculatively can be large, typically limited by the size of structures such as the reorder buffer, which can hold a few hundred instructions that are being executed within the pipeline.

Speculation is designed to not affect the correctness of a program. Although branch mispredictions occur and speculative instructions can ignore execution faults (e.g., permission error for memory access) these semantics were not considered harmful as mis-speculation will eventually be detected and the erroneously executed instructions will be squashed, leaving no directly visible changes to the program state held in structures such as registers and memory. Micro-architectural structures such as caches and Translation Look-aside Buffers (TLB) are affected by speculative operations, but the contents of such structures typically only affect performance, not the correctness of a program. In fact, prior work has shown that there are beneficial prefetching side-effects to speculatively executed instructions even those that are eventually squashed [179].

This section overviews some background on modern micro-architecture in modern processors: Out of order execution, branch predictor and cache structures.

2.1 Out of Order execution (OoO)

Modern processors implement many features designed to avoid delays in processing due to such things as data dependencies, branch/control flow resolutions, and accessing memory. One such feature is out-of-order (O3) execution, in which instructions are often executed speculatively (i.e., as a prediction/guess) in cycles that would otherwise be a stall in the execution pipeline in hopes of gaining performance by removing the stalls if

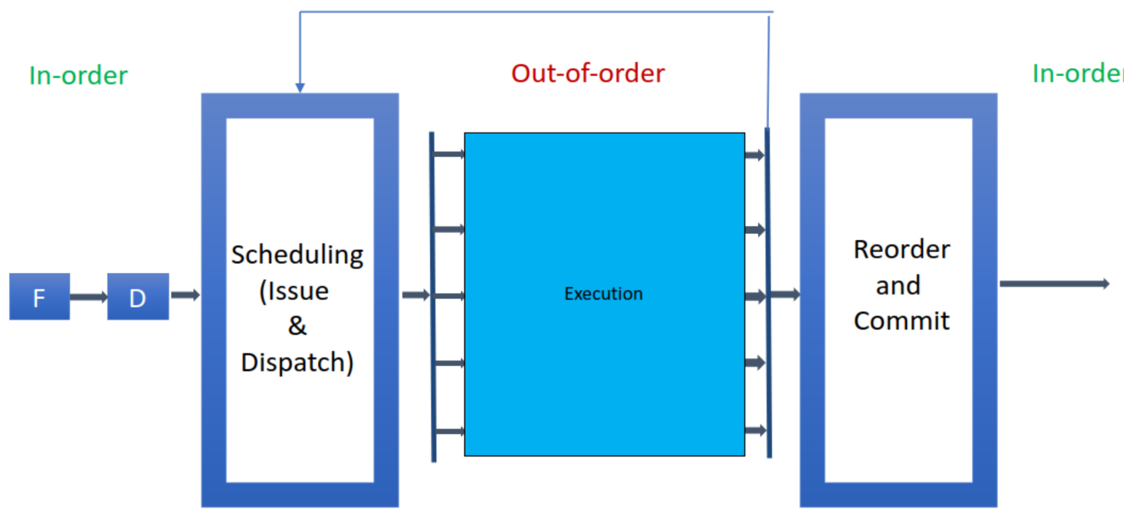


Figure 2.1: Out-Of-Order (O3) Execution

the prediction is correct, and at worst removing the speculatively executed (i.e., transient) instructions if the prediction is incorrect.

This is accomplished by issuing instructions and dispatching them to the proper functional units in program order (i.e., ”in-order), allowing them to execute out of program order (i.e., ”out-of-order”), and then committing them in program order again when they are finished executing.

To accomplish this, a reorder buffer (ROB) is used to keep track of program order and any dependencies, and instructions are only allowed to commit when they are at the top of the ROB.

In addition, with regard to memory instructions such as loading from or storing to memory, buffers such as load queues (LQ) and stores queues (SQ) are often utilized to keep track of in-flight memory instructions, which are often grouped into a load-store queue

(LSQ) to be able to monitor instructions that alias to the same memory location in an attempt to resolve memory dependencies earlier in the pipeline.

2.1.1 Microarchitectural vs Architectural Effects

Many of the optimizations involved in out-of-order execution utilize temporary storage devices that aren't directly visible during normal processing in a system. The ROB, LSQ, and SQ/LQ are examples of these, but there are many others, to include caches, line-fill buffers, return stack buffers (RSBs), as well as Branch Target Buffers (BTBs) and Pattern History Tables (PHTs). The transient instructions and their data stores in these temporary storage devices are referred to microarchitectural states as they differ from the traditionally visible architectural states (such as registers and DRAM), which are visible throughout execution.

2.2 Branch prediction

Branch prediction is a critical component of modern processors that support speculative out-of-order execution. When a control flow instruction (branch, call or return) is encountered, the result of the instruction (e.g., whether or not a conditional branch will be taken or what the target value is of an indirect branch or a return) is generally not known at the front end of the pipeline. As a result, to continue to fill the pipeline and utilize the available resources of the processor, branch prediction is used.

Modern processors employ sophisticated predictors (shown in Figure 2.2) which typically consist of three components:

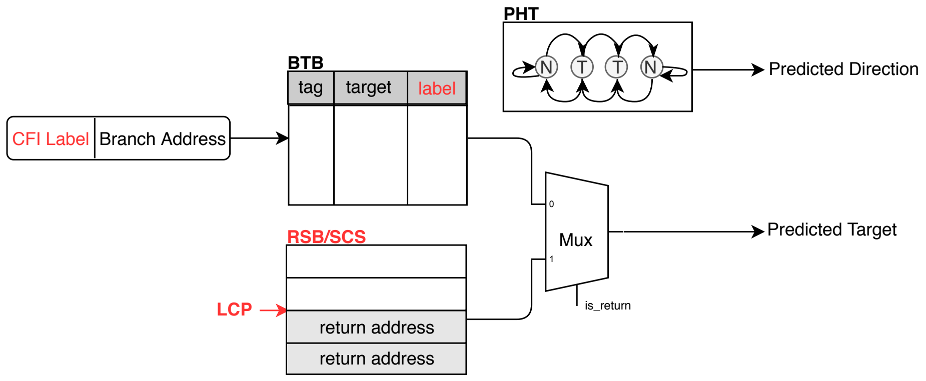


Figure 2.2: Branch Predictor Unit consists of three different predictors: (1) PHT for conditional branch direction; (2) BTB for indirect branch addresses; and (3) RSB for return addresses.

- *Direction predictor:* is responsible for predicting the direction of a conditional branch. Although a number of implementations have been studied, modern predictors typically implement a two-level context sensitive predictor [68]. The first level is a simple predictor that hashes each branch address to a direction predictor (typically a 2-bit saturating counter). This predictor is used either when a branch is not being successfully predicted or when the predictor has not been trained yet. When the predictor is trained, it typically uses a second prediction algorithm, often a variant of a gshare predictor [277], which uses the global history of a branch in addition to its address to hash to a direction predictor as before. The advantage is that the same branch can have different predictions based on the control flow path used to reach it.
- *Target predictor:* is used by indirect jump and indirect call instructions which jump to an address held in a register or a memory location, which is unknown at the front

end of the pipeline. This predictor typically uses the hash of the branch address to index a cache holding the branch targets called the branch target buffer (BTB). BTBs are shared across threads on a virtual core: one value used by a process could be used by another process whose branch has a matching address in the BTB [70].

- *The return address stack:* Since returns are not well predicted using the BTB, and often follow strict call-return semantics, their target is predicted using a return address stack of fixed size. When a call instruction executes, the return address is pushed on this hardware stack; if overflow happens, previous entries are overwritten [148]. When a return is encountered the top of the stack is popped and used as the return target.

2.3 Speculation Attacks

In this section we review two main class of Speculation attacks i.e Spectre and Meltdown attacks. Modern processors use different micro-architectural elements to help with branch prediction. Researcher have recently shown that the speculation behavior of modern processors can be exploited. In general, these attacks exploit four properties:

- **P1**: branch prediction validation happens in deep in the CPU pipeline. As a result, speculative instructions near the branch can access unprivileged memory locations.

- **P2**: speculative instructions leave side-effects in micro-architectural structures such as caches, which can be inferred using well-known timing side channel attacks like Prime+Probe and Flush+Reload [274].

- **P3**: the branch predictor can be mistrained (Spectre 1), or directly polluted (Spectre 2). It is shared across all programs running on the same physical core [70, 95, 145], allowing code running in one privilege domain to manipulate branch prediction in another domain (e.g., kernel, VM, hypervisor, another process, or SGX enclave). SpectreRSB attacks replace this step with speculation control through the RSB.

- **P4**: permission checks are performed deep in the pipeline and execution fault is generated only if the instruction is committed, enabling speculative instructions to access data outside its privilege domain;

```
if (offset < array1_size)
    y = array2[array1[offset] * 64];
```

Figure 2.3: Spectre attack variant 1

2.3.1 Spectre-PHT

Spectre-PHT is presented in Figure 2.3. In this code, a victim process reads values from `array1` using the `offset` provided by the attacker. Then, the resulting value is used to perform an access into `array2`. As we discussed above, accesses into the `array2` can be used by the attacker to deduce the value of the index. The index, in its turn, is controlled by the attacker since attacker controls the `offset`. Therefore, the attacker can use a carefully selected value of `offset` to read arbitrary memory address which then will result in cache access observable by the attacker. However, the `if` statement ensures there are no out of bounds memory accesses allowed. Unfortunately, the attacker can exploit speculative execution and behavior of branch predictor to force the victim process to perform an out of bounds memory access in the following way:

a) The attacker mistrains the branch predictor by executing the code several times with the value of the `offset` such that the `if` statement is true *(branch instruction not-taken)*.

b) Next, to make the speculative window larger, the attacker evicts `array1_size` from the cache, so that the CPU has to load the value from memory. Since the speculation result will not be resolved until this value arrives, forcing it to come from memory expands the size of the speculation window to allow more elaborate speculative gadgets to be executed.

c) Finally, the attacker chooses the malicious `offset` such that it be larger than `array1_size`. The trained branch predictor unit predicts the branch not-taken, so that the CPU executes two memory accesses speculatively and discloses the secret value through the cache side channel.

2.3.2 Spectre-BTB

The Spectre-BTB attack, exploits the Branch Target Buffer (BTB) in processors. The BTB's role in facilitating branch prediction within speculative execution becomes a critical point of vulnerability, as an attacker can manipulate the BTB to influence the control flow of a victim's program. By carefully selecting a vulnerable indirect branch address in the victim's program, the attacker can cause mispredictions of indirect branch targets, which forces the CPU to speculatively execute a maliciously chosen sequence of instructions, effectively steering the victim's control flow down an unintended path. Although this speculative execution is rolled back later, it leaves behind microarchitectural traces, including alterations in cache state, that can be analyzed to obtain sensitive information. Prior work [70] had shown that the branch predictor is shared among processes on the same core. So, one thread can pollute it for another across protection boundaries (including across VMs). Thus, the attacker can poison the branch target predictor for the victim and force it to speculatively execute the gadget which reveals the sensitive data within the victim. This is a dangerous attack because it allows cross process/cross VM Spectre attacks.

2.3.3 Spectre-STL

This variant of the Spectre attack exploits the fact that memory disambiguation predictor may wrongly determine that a load operation doesn't depend on a preceding store operation when they share the same physical address but different virtual addresses. Due to this incorrect prediction, the load operation proceeds before the store operation is complete. Since the load operation is incorrectly scheduled before the store, it ends up reading an outdated value from the cache. An attacker can craft specific code to exploit this behavior and access data they shouldn't be able to see.

2.4 Meltdown, LVI and MDS attacks

The closely related Meltdown attack relies on the fact that a permission check for memory access during normal out-of-order execution of an instruction can happen late in the instruction execution due to pipelining and instruction reordering (P4) allowing the CPU to load the privileged data until the permission is later checked. Unlike Spectre variants, Meltdown does not rely on using misspeculation. Since an exception eventually will be raised, this attack requires the ability to tolerate and recover from the raised exception.

Researchers have demonstrated that several variants of Meltdown-type attacks and Micro-architectural Data Sampling (MDS) attacks exhibit shared characteristics. These common traits enable attackers to extract sensitive data from specific internal micro-architectural elements, notably the L1 Data Cache (L1D), Line Fill Buffer, Load Port, Store Buffer, and Register Files. In their comprehensive study, Gruss et al. [83] categorize these attacks based on their distinctive micro-architectural behaviors.

- *Deferred Permission Check* Some Meltdown-type attacks exploit the fact that permission checks occur late in the pipeline, allowing the attacker to gain access to higher-privilege data. The original Meltdown attack (Meltdown-US) enables an attacker with user privilege to access kernel data due to improper checking of the US (User Accessible bit).

- *Incorrect Use of Intermediate Values* In another type of Meltdown attack, attackers aim to misuse intermediate values. For example, in the Foreshadow attack [240], if the present bit is not set and the processor raises an exception, the processor will still copy the physical address(intermediate value) into the load buffer entry. By exploiting this, the attacker will be able to index into the L1 Dcache and other internal buffers, potentially leaking sensitive data.

- **Use After Free** More recent variations of Meltdown-type attacks seem to leverage the `use-after-free` type vulnerability. This flaw leads to the use of outdated values, as observed in instances such as ZombieLoad [211], RIDL [244], and Fallout [178].

We explain the meltdown and MDS general attack flow based on 2.4. When a load micro-operation is placed into the re-order buffer, an entry in the load-buffer (or more broadly, a memory-order buffer for a load memory operation) is allocated (❶) to guarantee the proper sequencing of memory load visibility. When the load micro-op is scheduled for the load-data execution unit, it accesses its corresponding entry in the load buffer during step ❶. At this stage, the load buffer entry still holds stale data: a stale register number, outdated virtual address information, and a lingering physical address. During Step ❷, the load-buffer entry is refreshed with the register number and the virtual address details from the load micro-op, such as the virtual page number and offset.

in Step ❸, the system employs the virtual address details to conduct a search in the store buffer, line-fill buffer, and L1 data cache. Data from the entry with the highest priority is returned—prioritizing matching store buffer entries over the line-fill buffer and L1 data cache entries. Concurrently, a TLB lookup is carried out to obtain the physical address corresponding to the virtual address. If the TLB lacks the needed entry, a microcode assist initiates a page-table walk.

During Step ❹, the system examines the page-table details. In the context of the original Meltdown attack, the `present bit` is set, while the `user-accessible` bit is not set. As a result, the processor triggers a fault but concurrently continues to refresh the Physi-

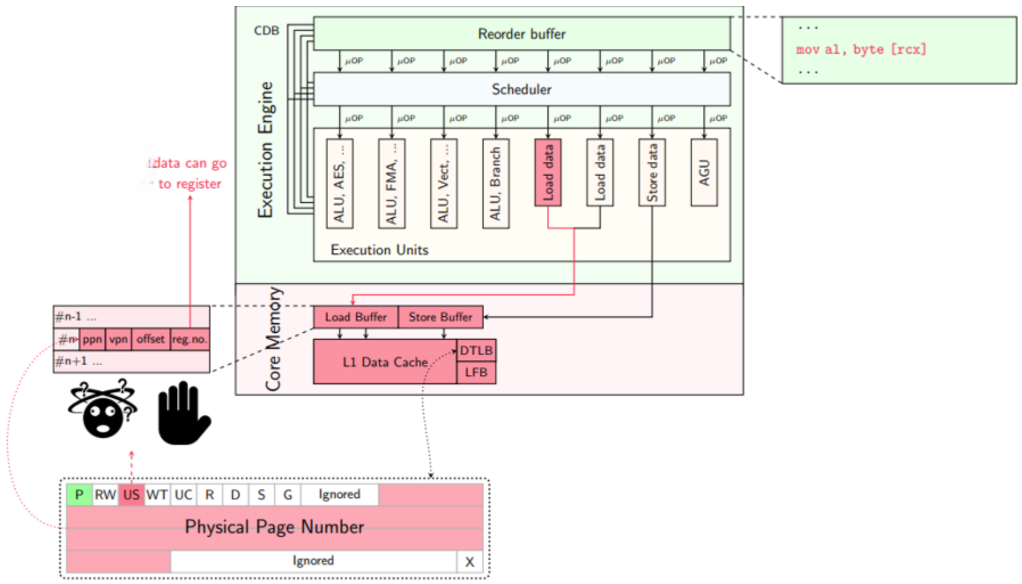


Figure 2.4: MeltDown and MDS attack [83]

cal Page Number (PPN) in the load buffer because updating the physical page number is a typical benign memory access scenario, and secondly, this update to the load buffer is deemed harmless since the outcome will not be used architecturally if the load is aborted. Currently, the data obtained in step ❸ are prepared to be routed to the register. If the physical address corresponds to the data fetched, for instance, from the L1 data cache in case of the original Meltdown attack, the data are forwarded to the register. As a result, an attacker will be able to leak these data through a side channel attack.

The same attack procedure is observed across different variants of Meltdown, including Foreshadow [240] and NULL-LVI [33]. While there are subtle variations, the core mechanism behind most Meltdown-type attacks remains fundamentally consistent. For instance in Foreshadow attack [240], the first 3 steps of the attack will remain the same, however, in step ❹, in the page table details, `present bit` is no set, which means the rest of the page table entry information is not valid. This triggers the processor to raise a fault. Yet, similar to the original Meltdown scenario, the physical address is still transferred to the load buffer, and eventually data from L1 Data cache will be forwarded to register.

Most MDS-type attacks follow a similar pattern, albeit with a distinction in the origin of the data leak. Rather than sourcing data from the L1 Data cache, these attacks leak from other micro-architectural buffers, notably the LFB or the store buffer. More specifically, the first 3 steps remain the same; however, in Step ❸, the virtual address information is used to search the store buffer, line-fill buffer, and L1 data cache. The L1 data cache lookup encounters a conflict and does not succeed. This causes an abort in Step and the load operation is reissued, which transforms the current load into a *zombie load* [211]. Data with the highest priority is then returned – that means data from matching store buffer entries are retrieved before data from matching line-fill buffer and L1 data cache entries. In the subsequent step, the outdated physical page number is utilized (indicative of a ”use after free” scenario) to verify the physical address tag of the data obtained in the earlier step. If there’s a match between the physical address and the tag, the data is then routed to the intended register, making it accessible for subsequent side-channel attack [83].

Chapter 3

Spectre Returns! Speculation Attacks using the Return Stack Buffer

3.1 Introduction

Speculative execution is a microarchitectural technique used pervasively to improve the performance of all modern CPUs. Recently, it has been shown that speculatively executed instructions can leave measurable side-effects in the processor caches and other shared structures even when the speculated instructions do not commit and their direct effects are not visible. Moreover, since these instructions are speculative, normal permission checks do not take effect until the instruction is committed. The recent Spectre attack [95,145,168] has shown that this behavior can be exploited to expose information that is otherwise inaccessible. In the two variants of Spectre attacks, attackers either mistrain the branch predictor unit or directly pollute it to force the speculative execution of code that can enable exposure of the full memory of other processes and hypervisor.

Chen et al. demonstrated that known Spectre variants are able to expose information from SGX enclaves [46]. New variants of Spectre that utilize other triggers for speculative execution have been introduced including speculative store bypass [114].

In this work, we introduce a new attack vector Spectre like attacks that are not prevented by deployed defenses. Specifically, the attacks exploit the Return Stack Buffer (RSB) to cause speculative execution of the payload gadget that reads and exposes sensitive information. The RSB is a processor structure used to predict return address by pushing the return address from a call instruction on an internal hardware stack (typically of size 16 entries). When the return is encountered, the processor uses the top of the RSB to predict the return address to support speculation with very high accuracy.

We show that the RSB can be easily manipulated by user code: a `call` instruction, causes a value to be pushed to the RSB, but the stack can subsequently be manipulated by the user so that the return address no longer matches the RSB. We describe the behavior of RSB in more details in Section 3.2.

In Section 3.3, we show an RSB based attack that accomplishes the equivalent of Spectre variant 1 through manipulation of the RSB instead of mistraining the branch predictor; we use this scenario to explain the principles of the attack, even though it may not be practical. The RSB is shared among hardware threads that execute on the same virtual processor enabling inter-process (or even inter-vm) pollution of the RSB. Thus, in

Section 3.4, we develop an attack that targets a different thread or process on the same machine. In Section 3.5, we present a third type of SpectreRSB: an attack against an SGX compartment where a malicious OS pollutes the RSB to cause a misspeculation that exposes data outside an SGX compartment. This attack bypasses all software and microcode patches on our SGX machine. Section 3.6 overviews another potential attack targeting an unmatched return in the kernel code; we present this attack for completeness because it relies on a number of ingredients that are difficult to find in practice.

We show how these attacks interact with deployed defenses concluding that several practical deployments are vulnerable to SpectreRSB. Thus, we believe that SpectreRSB is as a dangerous speculation attack, that in some instances is not mitigated by the primary defenses against Spectre. It extends our understanding of the threat surface of speculation attacks, allowing future defenses to more effectively mitigate their risks. We discuss the implications of the attack in Section 6.7.

Disclosure: We reported these attacks to the security team at Intel. Although we did not demonstrate attacks on AMD and ARM processors, they also use RSBs to predict return addresses. Therefore, we also reported our results to AMD and ARM.

3.1.1 Defenses against Meltdown/Spectre

After the disclosure of Spectre and Meltdown in January, 2018 [143,157], a number of defenses were suggested.

Intel proposed defenses: Intel released a whitepaper [107] suggesting three types of defenses.

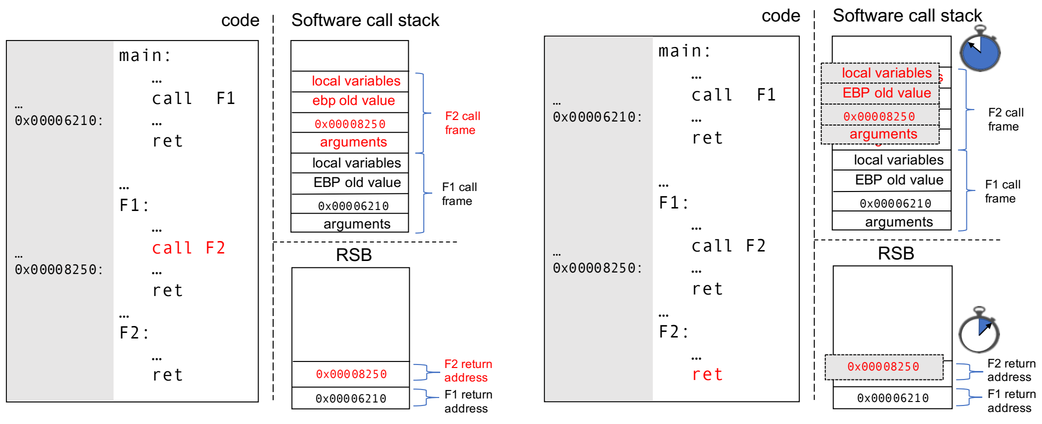
- To mitigate Spectre V1 attack, Intel recommends inserting a `LFENCE` instruction after the branch as a barrier to stop speculative execution. This defense mechanism has now been adopted by compilers such as GCC [162] and MSVC [176]. However, this does not prevent attacks where the attacker controls the program and does not use `LFENCE` instructions.
- To mitigate Spectre V2 attack, Intel introduced three new processor interfaces through microcode updates [115]:
  - Indirect Branch Restricted Speculation (**IBRS**) prevents software running in higher privileged mode from using prediction results from software running in lower privileged mode.
  - Single Thread Indirect Branch Predictors (**STIBP**) prevents code executing on one logical processor from impacting the indirect branch prediction of code executing on another logical processor.
  - Indirect Branch Predictor Barrier (**IBPB**) stops software running before the barrier from affecting the indirect branch prediction of software running after the barrier.
- To mitigate Meltdown, Intel recommends unmapping more privileged domain (kernel space) during the execution of less privileged software, which has been adopted by all popular operating systems, including Windows, Linux, and macOS. This is the KPTI defense described below.

Kernel Page-Table Isolation (KPTI): Gruss et. al [84] introduced a protection technique called KAISER to protect against side channel attacks bypassing kernel level address space randomization (KASLR) [85]. The protection is based on unmapping kernel pages while in user mode, and remapping them on a mode switch to the kernel. As a result, misspeculation from user code is not able to access kernel memory, preventing Meltdown. It has been reported that KPTI can introduce substantial performance overhead [81]. KPTI cannot prevent attacks within the same privilege mode (e.g., to access memory outside a sandbox) [42, 215].

Return Trampoline (retpoline): retpoline [239] is a software-based mitigation technique against indirect branch target injection attack (i.e., Spectre V2). It "exploits" two properties of the branch target prediction engine: (1) when executing a `ret` instruction, the predictor will utilize the return stack buffer (RSB) instead of the BTB; and (2) RSB cannot be polluted by attackers. The retpoline technique essentially swaps indirect branches for returns and deliberately pollutes the RSB with a useless gadget to control speculative execution. Retpoline protection requires access to source code and recompilation.

RSB refilling (also known as RSB stuffing) [108]: on Intel's Core i7 processors starting from Skylake (which are called Skylake+), an underfill condition in the RSB where a return occurs when the RSB is empty causes the processor to speculate the return address through the branch predictor. Thus, defenses deployed to protect indirect branches against Spectre variant 2 fail in this situations since return instructions can cause a misspeculation through the branch predictor. To counter this situation, Skylake+ processors also implement RSB refilling (a software patch): every time there is a switch into the kernel, the RSB is intentionally filled with the address of a benign delay gadget (similar to Retpoline) to avoid the possibility of misspeculation. RSB refilling interferes with SpectreRSB, although it was designed for a completely different purpose. However, we note that all Core i7 processors prior to Skylake are not patched with RSB refilling and that different processor lines, importantly including the Intel Xeon which are the primary platform used on Intel-based cloud computing systems and servers, are also unpatched, leaving them vulnerable to SpectreRSB.

3.2 Attack Principles: Reverse Engineering the Return Stack Buffer



(a) Calling a function (F2)

(b) Return from a function (F2)

Figure 3.1: Example of function call and return effect on software call stack and RSB

In this section, we explain the operation of the Return Stack Buffer (RSB), which is the microprocessor structure our attacks exploit to implement speculation attacks that bypass all existing defenses. On modern processors, sophisticated branch predictors are used to predict the direction and target of conditional and indirect branches and calls. Return instructions challenge such predictors because the return address depends on the call location from which a function invoked, which for many functions that are called from different locations of a program can lead to poor branch predictor performance. For example, consider a function such as `printf()` which may be called from many different locations of a program. Relying on the previous history of where it returned to can lead to very low prediction performance through the branch predictor. We verify each of these mechanisms on two Intel processors (a Haswell and a Skylake).

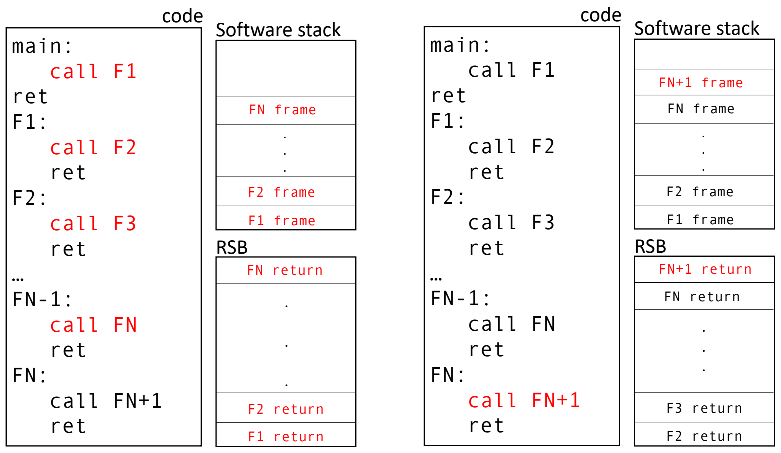
3.2.1 RSB Overview

To overcome this problem, the return address is predicted using the RSB as follows. The RSB is a hardware stack buffer where the processor pushes the return addresses every time a call instruction is executed and uses that as a return target prediction when the matching return is encountered. Figure 3.1a shows an example of the state of the RSB after two function calls (F1 and F2) have been executed. The figure also shows the state of the software stack for the program where the stack frame information and the return address of the function are stored. Figure 3.1b shows how the values on these stacks are used when the return instruction from function F2 is executed. At this point, the return address from the fast shadow stack is used to speculate about the return address location quickly. The instructions executed at this point are considered speculative. Meanwhile, the return address is fetched from the software stack as part of the teardown of the function frame. The return address is potentially in main memory (not cached) and is received several hundred cycles later. Once the return address from the software stack is resolved, the result of the speculation is determined: if it matches the value from the RSB, the speculated instructions can be committed. If it does not, then a misspeculation has occurred and the speculatively executed instructions must be squashed. This behavior is similar to speculation through the branch predictor, except it is triggered by return instructions. Note that the misspeculation window could be substantially larger since the return could be issued out of order, and other dependencies have to be resolved before it is committed.

3.2.2 RSB sources of misspeculation

The RSB misspeculates when the return address value in the RSB does not match the return address value in the software stack, leading the program to misspeculate to the address in the RSB. If this misspeculation can be triggered intentionally by an attacker, spectre like attacks become possible through the RSB. Thus, in this subsection, we explain the sources of misspeculation through the RSB, and discuss whether they provide a vector for attackers to trigger speculation attacks. We label these sources as S1 to S4 to be able to refer to them in the attack descriptions.

S1: Overfill or Underfill of the RSB due to limited structure size: The RSB structure is typically sized to match common nesting depths of call stacks in programs. On low-end machines, the RSB can be as shallow as 4 entries in size. More typically, on desktops, it is in the range of 16 entries, and for server class processors, it can be larger (e.g., 24 entries on the AMD Ryzen [72]). As illustrated in Figure 3.2, when the RSB overfills, it typically overwrites the older entries in the stack. Eventually, when the stack is unrolled as



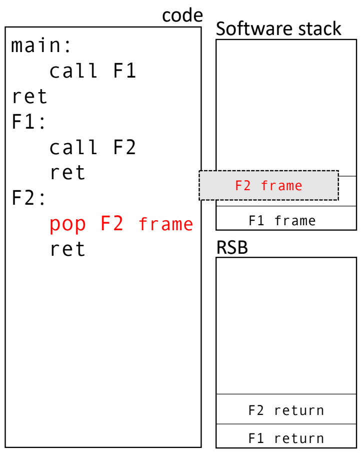
(a) Executing N nested function (b) N+1 nested function call

Figure 3.2: Example of overfill of RSB

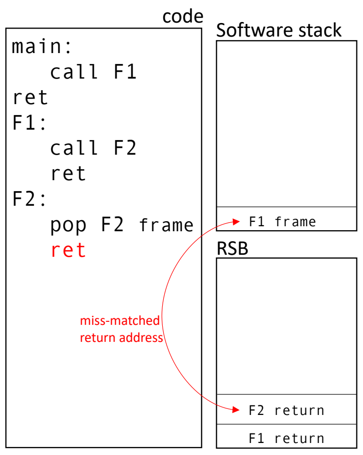
the nested calls return, we reach the function whose value has been overwritten causing an underfill of the stack (in Figure 3.2, the entry for F1 got overwritten).

In an underfill, there is no value available on the RSB to guide speculation. Different CPUs handle this situation differently. For example, the Intel CPUs that we checked switch over to the branch predictor if the RSB is empty, which can be used to trigger attacks through the branch predictor [154]. However, AMD appears not to follow this strategy.

S2: Direct pollution of the RSB: This is the primary vector that we use in our proof of concept attacks. Call instructions implicitly push a return address to the RSB and the software stack. However, an attacker can then replace the address on the software stack (by writing directly to that location), or just remove it altogether (as shown in Figure 3.3a). In this case, the value in the RSB remains and does not match a value on the software stack



(a) Removing the current return



(b) Mismatch between RSB, Stack

Figure 3.3: Example of direct pollution of the RSB

causing misspeculation when a return is executed (as shown in Figure 3.3b). By controlling the call address, the attacker can control the misspeculation address.

It is also possible to convert a call instruction into a push and jmp, in which case a return value exists on the software stack that is not matched by a value in the RSB. A return could also be replaced by a pop and a jump, causing a value to remain in the RSB that has been removed from the software stack.

S3: Speculative pollution of the RSB: speculatively executed calls push a value on the stack, although the details are specific to the architecture. Once misspeculation is discovered the call is squashed but the speculatively pushed return address remains on the RSB. This provides the opportunity for a malicious attacker to push a return address that is outside the address space accessible by the program (e.g., a kernel address) without raising an exception or having to handle the side effects of a call. [1]

[1]We did not use this vector, but its conceivable to use it to bypass Supervisor Mode Execution Prevention (SMEP) [76] to jump to a kernel gadget. For example, the user may attempt to jump to user code in PhysMap

S4: RSB use across execution contexts: on a context switch the RSB values left over from an executing thread are reused by the next thread. Once we switch to a new thread, if the thread executes a return, then it will misspeculate to an address provided by the original thread. The same is true with a switch over to the Operating System (provided RSB refilling is not implemented), or to an SGX context.

3.3 SpectreRSB: Basic attack example

In this section, we illustrate the attack principles by showing a basic speculation attack launched from a process to part of its address space that it cannot directly access (similar to Spectre variant 1 [95]). This attack represents the simplest instance of SpectreRSB and therefore we use it to explain the attack in detail. It is unlikely to be practical: it is difficult to implement the gadget to manipulate the stack using high level sandboxing primitives to allow the attack to break sandbox boundaries. On an unpatched machine, this attack enables the attacker to read kernel memory via the Meltdown bug. However, KPTI prevents using it to allow user code to read kernel data. We note that this attack does not rely on any speculation through branches or the branch predictor. For this reason, the attack bypasses defenses that focus only on securing speculation through the branch predictor. Most of our experiments were conducted on the machines shown in Table 3.1; the i7-6700 machine is a Core i7 Skylake with SGX2.

implementing a ret2dir [132] rather than a ret2usr [131] attack; the difficulty is that the user cannot pollute the RSB with kernel addresses using S2 without raising an exception, but may be able to do that using S3 (we note that PhysMap is marked as non-executable in most recent Linux distributions.)

Table 3.1: Experiment Environment

	CPU Model	Kernel version	kernel patch	Intel patch
Machine1	Intel Xeon(R)-E51620	4.15.0-22-generic	Retpoline, Kpti	✓
Machine2	Intel Core(TM)-i7-6700	4.4.117	Retpoline, Kpti, RSB refilling	✓

Figure 3.4 presents an overview of a basic SpectreRSB attack. The attack starts at line 22 with the call to `speculative`, with an argument which is the memory address of the sensitive data to be read. `speculative` calls `gadget`, which serves two purposes: (1) the return address is pushed to the RSB (the return address is to line 17 where we have the payload gadget to be executed speculatively); and (2) we jump to the (inline assembly) function `gadget` which will manipulate the software stack to create the mismatch between the RSB and the software stack. In this case, gadget cleans up the effects of the function call to itself, popping off the frame including the return address.

At this point, before the return, the stack state is consistent with a return from speculative back to main. However, the RSB holds a return value from `gadget` to `speculative`. Thus, in line 12 when the return executes, the CPU speculatively executes at line 17. The flush of the top of the stack (line 10) ensures that the true value of the return address will be fetched from memory rather than from the caches creating a large speculation window. Note that the speculation window based on the return. Speculative execution at line 17 reads the secret which can be any mapped address even if inaccessible to the user process during normal execution and then communicates it out through the flush reload cache side channel by accessing a data dependent index in the Array (line 18). Finally, the real return value is obtained, and the misspeculation is squashed, returning us to line 23, where we probe the cache to identify which data dependent cache set was accessed to expose the value of the secret.

```
1.  Function gadget()
2.  {
3.      push %rbp
4.      mov %rsp, %rbp
5.      pop %rdi     //remove frame/return address
6.      pop %rdi     //from stack stopping at
7.      pop %rdi     //next return address
8.      nop
9.      pop %rbp
10.     clflush (%rsp)  //flush the return address
12.     retq     //triggers speculative return to 17
13. }            //committed return goes to 23
14. Function speculative(char *secret_ptr)
16.     gadget();       //modify the Software stack
17.     secret = *secret_ptr; //Speculative return here
18.     temp &= Array[secret * 256]; //Access Array
20. Function main()
22.      speculative(secret_address);
23.      for (i = 1 to 256)   //Actual return to here
24.      {
25.          t1 = rdtscp();
26.          junk = Array[i * 256]; //check cache hit
27.          t2 = rdtscp();
28.      }
```

Figure 3.4: SpectreRSB basic attack example

3.4 Attacks across different threads/processes

In this section, we investigate different vectors of SpectreRSB which exploit S4 (RSB use across execution context) to pollute the RSB. These attacks potentially allow an attacker to attack another process (Similar to Spectre V2), perhaps even across VMs, making the attack dangerous on the cloud. In general, these attacks require a machine not implementing RSB refilling (pre-Skylake, or Xeon, for example), to make sure that a context switch does not overwrite the polluted addresses from the RSB (Figure 3.5).

The attacker establishes co-location with the victim on the same core similar to Spectre 2. The attack pattern proceeds as follows. (1) after a context switch to the attacker, s/he flushes shared address entries (for flush reload). The attacker also pollutes the RSB with the target address of a payload gadget in the victim's address space; (2) the attacker yields the CPU to the victim; (3) The victim eventually executes a return, causing speculative execution at the address on the RSB that was injected by the attacker. Steps 4 and 5 switch back to the attacker to measure the leakage.

3.4.1 Attack 2a: Attack across two colluding threads

In this attack, the attacker and the victim are two colluding threads following the steps in Figure 3.5. In the first attack, we let the two threads synchronize using `futex` operations to control their interleaving. The RSB pollution happens in the first thread which also flushes the top of the stack of the second thread, while the return happens in the second. The attack succeeded, proving that SpectreRSB works from one thread to another. However, since the return is in user mode, we cannot read kernel data. For the attack to

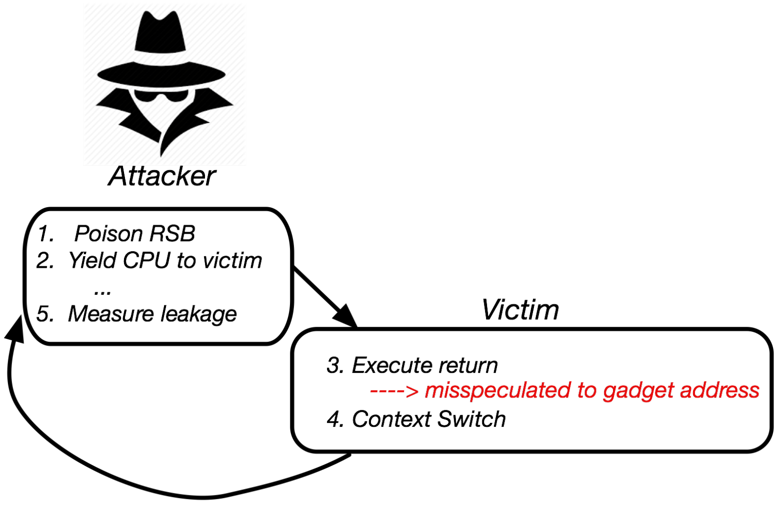


Figure 3.5: Attack 2: Basic Attack Flow

be useful, we should either launch an attack such that the victim colluding thread returns while in the kernel (enabling us to read kernel data while its memory is mapped), or work across process boundaries such that the victim thread is a different process and we leak its sensitive data (attack 2c).

3.4.2 Attack 2b: Attack with two colluding threads with return from inside kernel

Next, we wanted to see if we can use this attack to cause a return while the victim thread is in the kernel mode in step 3. To ensure this, we have the colluding victim execute a blocking system call, which typically has them deep inside a call stack in the kernel before blocking. The attacker after polluting the RSB, waits for the victim to unblock, perhaps even triggering the event that unblocks it. At this point the victim continues execution inside the kernel, and recurses back out of its call stack, with one or more returns, triggering the vulnerability. This attack requires a machine without SMEP enabled. We demonstrated the attack with SMEP disabled.

3.4.3 Attack 2c: Attacks across Process boundary

The attacks above assume two colluding threads. In principle, it can be generalized across different address spaces, but this requires overcoming some challenges. First, the attacker has to be able to identify gadgets existing in the victim binary instead of being able to use their own. This may also require them to recover the ASLR offset of the victim, but there are a number of existing attacks that make that possible. However, once these gadgets are found, the same attack pattern can follow by first polluting the RSB, then using eviction to remove the top of the stack containing the return address from the cache to extend the speculation window. Synchronization is difficult, but can be simplified if the attacker is able to trigger operations in the victim (e.g., if the victim is a server accepting connections). We did not create a PoC of this attack.

3.5 SpectreRSB Atack 3: Attacks on SGX

Having established attack 1 of the SpectreRSB where the attacker pollutes the RSB for its own process to cause misspeculation, we next investigate whether SpectreRSB attacks work on SGX compartments (similar to SGXSpectre [46]).

In this attack we consider, a malicious untrusted user code manipulates the RSB to try to cause misspeculation inside the enclave. In this attack, we pollute the RSB with the target address of a payload gadget from untrusted user code (this can equally be done by a malicious OS). Note that the gadget could be in the untrusted user code since user code and SGX enclaves share the same address space. The next step is to do an enclave call to switch it the trusted execution mode. The enclave call has to have an unmatched return to cause speculation execution at the address that was injected from the untrusted code. Finally, the untrusted code after returning from the enclave call can check the cache to record the leakage.

Triggering an unmatched return: the RSB assumes that strictly paired call-return behavior. In attacks that cross execution boundaries, the attacker pollutes the RSB, but would then like to trigger a return in the victim process code (or OS/SGX code) to which they have no access. However, if the attacker manages to catch the victim inside of a function call, then when the victim executes again, it will encounter an unmatched return. This could rely on timing or a blocking call inside of a function that will cause the scheduler to unschedule the victim. In this Proof of Concept attack, we placed an unmatched return directly in the enclave, but we expect to be able to do that using the strategies above for other enclaves.

3.6 Potential Attack 4: From user to Kernel

In this section, we briefly discuss the possibility of another attack where user code pollutes the RSB and then triggers an unmatched return in the kernel (we call this attack 4). This attack is likely to be difficult, if not impossible, so we describe it only for completeness. The main insight is that a return from the kernel to a polluted address in the RSB will cause speculation while in kernel mode. This means that the kernel address space is still mapped, allowing us to read from kernel. This attack assumes the following ingredients: (1) that the RSB is shared between the user and the kernel: we find that this is the case on two Intel processors; (2) We need to be able to trigger an unmatched return in the kernel. Although some programming constructs such as tail recursion, continuations, setjmp/longjmp and others can break call-return semantics, we have not attempted to find such unmatched returns in the kernel; and (3) We need to figure out the stack address of the kernel, and evict it from the cache. This last step is necessary to make sure that the speculation window is sufficiently large to execute a useful gadget speculatively (without this, we can only execute a gadget a few instructions long speculatively). Luckily, the mapping between the stack kernel address and the physical address is deterministic in Linux on x86-64 (it uses the Physmap address directly instead of double mapping it). This makes deriving the conflict set straightforward once we identify the kernel stack address.

We explore a proof of concept attack with an unmatched return in a kernel module that we build. Later, we discuss concrete possibilities for how to make this happen with multiple threads. The attack is shown in Figure 3.6, and works only on an unpatched machine or a machine not implementing RSB refilling. After polluting the RSB in steps 1-3, and flushing the top of the kernel stack in step 4, before issuing a system call to our kernel module with the unmatched return . The mismatched return triggers a misspeculation to step 7 to execute in supervisor mode. This attack does not work on patched Skylake+ processors due to RSB refilling but works on the Xeon machine. We also discover that Supervisor Mode Execution Prevention (SMEP) checks are not speculatively bypassed since the speculative program counter is known at the time of speculation. Thus, the attack as

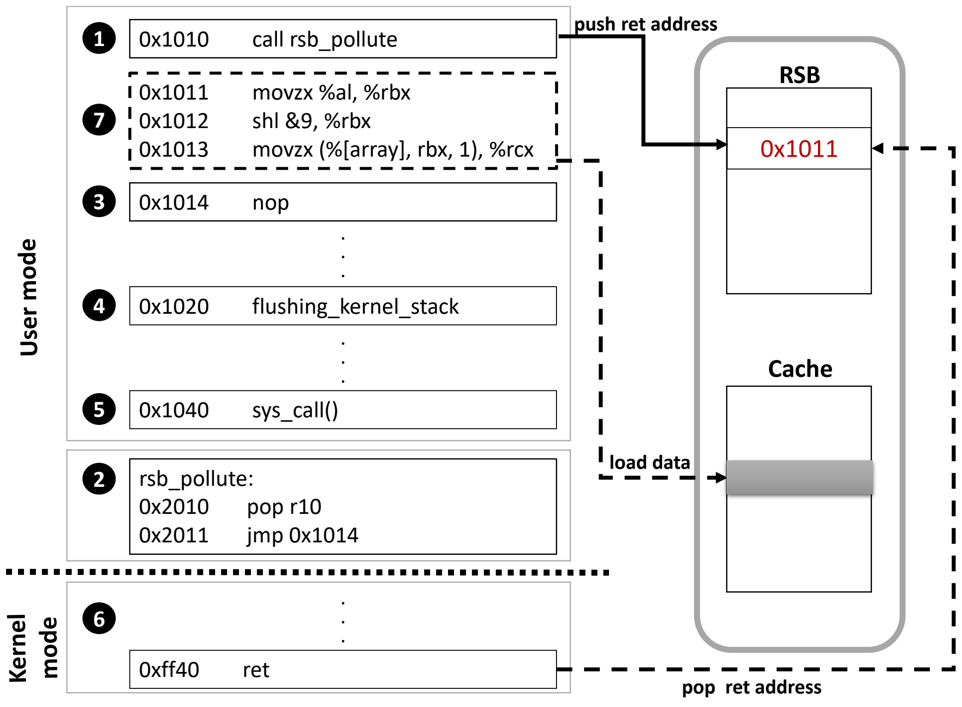


Figure 3.6: Attack 4: Basic Attack Flow

shown requires SMEP to be disabled to enable the kernel to return to user code. An alternative strategy to bypass this limitation is to try to use the return address of the gadget in PhysMap (as discussed under S3 in Section 3), but most Linux distributions disable execution of PhysMap addresses. We only demonstrated the attack with SMEP disabled. We also assumed that we know the address of the kernel stack pointer to flush it in step 4.

Table 3.2: Attack senarios vs. defense mechanisms

Attack no	Attack Name	lfence	IBRS	STUBP	IBPB	retpoline	RSB refilling	SMEP/SMAP
Attack 1	Same-process	✓	✓	✓	✓	✓	✓	✓
Attack 2a	Colluding threads (user)	✓	✓	✓	✓	✓	✗	✓
Attack 2b	Colluding threads (kernel)	✓	✓	✓	✓	✓	✗	✗
Attack 2c	Cross-process	✓	✓	✓	✓	✓	✗	✓
Attack 3	Return in SGX	✓	✓	✓	✓	✓	✓	✓
Attack 4	Kernel from user	✓	✓	✓	✓	✓	✗	✗

3.7 Discussion and Mitigations

In the first attack, a process launches an attack within the same address space either on the kernel or on data outside of its software containment. An attack from user mode to the kernel is possible in the original spectre: the BTB is poisoned, and then an indirect jump in the kernel space triggers the misspeculation to the payload. This attack is prevented by the Retpoline defense which only covers BTB poisoning, and assumes that the user code is compiled to use Retpoline. Importantly, none of the Intel microcode patches seem to restrict speculation through the RSB.

In attack 2 of SpectreRSB, we attempted to carry out the attack across execution threads. Attack 2a demonstrates a practical attack across two colluding threads in the same process. Attack 2b, shows how two colluding threads can cooperate to make an attack like attack 4 more predictably execute a return in kernel mode. Finally, Attack 2c in principle can carry out a SpectreRSB attack across different processes, bypassing all known defenses. Although we did not demonstrate this attack completely yet, we believe that none of the defenses stop it and it should be considered a dangerous threat vector on the cloud.

The attack on SGX bypasses all known defenses, including RSB refilling, and should be considered a dangerous open vulnerability on SGX systems. Changing the microcode patches to protect speculation through the RSB, or implementing RSB refilling upon entrance to SGX enclaves can potentially mitigate this vulnerability.

In Attack 4, our intuition was that a SpectreRSB attack that causes a return in the kernel to misspeculate would defeat both KPTI (in kernel mode, the kernel pages are available) and Retpoline (which only protects indirect jumps and calls, but not returns). While this is generally true, we discovered a number of complications that can be overcome under some conditions. RSB refilling, which is implemented on Intel Skylake+ processors, stops the attack. On other processors, SMEP prevents a return to a gadget in user space, however, a return to a gadget in kernel space if one can be identified is possible. Finally, we need to be able to determine the address of the top of the kernel stack in order to be able to evict, to increase the speculation window.

To mitigate SpectreRSB, we suggest that all processors, not just Skylake+ immediately support the RSB refilling patch which should interfere with all attacks that require a context switch to the kernel (attacks 3 and 4). Adding RSB refilling on an SGX enclave entrance should also be considered to stop attack 2. We also suggest that Intel microcode patches consider extending protection to the RSB, and not just the branch predictor. We summarize the proposed SpectreRSB attacks as well as their ability to bypass defenses in Table 3.2.

Attack results: This attack successfully works on fully patched machines. The attack bypasses all software and microcode patches:it bypasses Retpoline since no indirect jumps are used. It bypasses the microcode patches since they do not appear to limit speculation through the RSB. It bypasses RSB refilling (which is only implemented on Skylake+, but not on the Xeon processors) since no mode switches to the kernel are triggered during the attack. Thus, SGX is vulnerable to SpectreRSB even on fully patched machines.

3.8 Concluding Remarks

In this chapter, we introduced a new type of speculation attacks (SpectreRSB) that is triggered by the Return Stack Buffer (RSB), rather than the branch predictor unit. The RSB is used to predict the address of return instructions. We demonstrated a number of vectors that allow an attacker to cause RSB misspeculation. Using these techniques, we construct a number of attack vectors including attacks within the same process, attacks on SGX enclaves, attacks on the kernel, and attacks across different threads and processes. SpectreRSB bypasses all published defenses against Spectre, making it a highly dangerous vulnerability.

Interestingly, there is a patch that was proposed to protect against the behavior of Intel Core i7 Skylake generation and newer processors called RSB refilling. RSB refilling interferes with SpectreRSB attacks that experience at least one mode switch from user to kernel. We recommend that this patch should be deployed immediately across all processor generations (and not just Skylake+). In the long run, we believe that these patches are ad hoc and that new attack vectors will continue to emerge. Current systems are fundamentally insecure unless speculation is disabled. However, we believe that it is possible to design future generations of CPUs that retain speculation but also close speculative leakage channels, for example by keeping speculative data in separate CPU structures than committed data.

Chapter 4

Speculative Execution Regulation (SER): Leveraging Program Analysis to Secure Speculative Execution

Having demonstrated the possibility of Spectre attacks on modern processors, we will explore the defense mechanisms to counter Spectre attacks and other classes of transient attacks in the upcoming chapters.

The disclosure of Meltdown [157] and Spectre [143] attacks in 2018, and the many variants that have been discovered since then, have significantly undercut confidence in the security of modern architectures. Specifically, they demonstrated dangerous vulnerabilities in speculative execution which is at the heart of speculation and therefore at the core of architectural optimization such as super-scalar and out-of-order execution. Common wisdom was that speculative state is isolated: that is, it is safe to execute any instruction speculatively since any side effects are eliminated as part of recovery from misspeculation. If this isolation holds, then there is no motivation to ensure correctness or security during speculative execution since any effect is temporary and will be invisible to the software once speculation is resolved. Only correctly executed instructions would commit, at which point their effects become visible.

Transient execution attacks showed us that this isolation assumption is untrue: while direct (or architectural) effects of misspeculation are undone, indirect (or microarchitectural) state changes remain. Thus, speculative state can be *transmitted* to committed state through *transmitter* instructions that leave a data dependent microarchitectural signature. When security invariants (or even correctness) are not enforced during speculation, these attacks can speculatively access secret values unavailable on the committed path (e.g., accessing a kernel memory location from user space); then transmit it out using a transmitter instruction by modifying microarchitecture states (e.g., an access of a specific set in the cache). Although the execution is eventually squashed, the secrets are leaked, encoded through the microarchitectural changes.

A number of mitigations against transient execution attacks have been proposed. They can be broadly categorized into two groups. (1) Hide the microarchitectural effects of squashed instructions. Solutions such as InvisiSpec [269], SafeSpec [135], CleanupSpec [206], and Ghost Loads [207] follow this general principle of making sure that speculative transmitter instructions do not leave microarchitectural signatures either by delaying updates to these structures until the instruction commits or by undoing the effects if an instruction is squashed. These approaches must protect each microarchitectural structure separately, and have been shown recently to be vulnerable to speculative interference attacks [127]. (2) Limit potentially dangerous speculation. An extreme version of this approach would prevent all speculative execution, greatly harming performance. Thus, practical defenses focus on identifying situations where speculation could be exploited and delay such speculation until it becomes apparent that the instructions will successfully commit. A subset of these solutions focus on transmitter instructions. With few exceptions, these techniques look for architectural triggers/properties to guarantee safe execution, requiring substantial investments to track these properties. For example, Speculative Taint Tracking (STT) [280], delays transmitter instructions until all their operands are guaranteed to commit requiring support for taint tracking to be able to distinguish speculative from committed values. These techniques are necessarily conservative because they lack semantic information about the program and the data and often require significant overhead making them impractical for real world implementation purposes.

In next two chapters, we explore a new class of defenses against speculative execution attacks which we call *Speculative Execution Regulation (SER)*. Since speculative execution state is accessible to an attacker, SER seeks to ensure that security invariants are enforced even during speculation. While security can be ensured using techniques such as NDA [260] and STT [280] based on coarse-grained separation between speculative state and committed state, we show that this approach is too conservative, thus preventing speculation in many cases where it is safe. Instead, SER bases these decisions on the semantics of the operations being executed speculatively, letting instructions execute earlier if it determines them to be safe (including many that would be deemed unsafe from an architectural perspective) but stopping them if they are unsafe.

SER leverages program analysis techniques that have been developed to protect programs from vulnerabilities (on the committed path) by enforcing security invariants. These include Control Flow Integrity (CFI) [10], Data Flow Integrity (DFI) [41], memory safety checkings [181,216], and Dynamic Information Flow tracking [230]. These techniques have proven effective, leading CPU manufacturers to start implementing support for them in hardware, such as Intel CET [103], Intel MPX [115], and ARM MTE [82]. SER brings these protections to bear on regulating speculative execution, not only committed instructions. This requires us to enforce these defensive invariants earlier in the pipeline prior to instruction issue. Since hardware support for these analyses is increasingly present in commercial CPUs, the cost of supporting these ideas is largely already paid–SER simply re-architects them to provide additional security during speculation.

In this chapter 4 we introduce SpecCFI [150] a recent defense example of SER. It applies control flow integrity in the decode stage to prevent SpectreBTB (v2) and SpectreRSB (v5) [35, 36, 149]. It modifies existing CFI support such as Intel's CET [103, 222] to prevent these vulnerabilities at a substantially lower overhead than other defenses.

In chapter 5, we seek to establish the generality of this approach by applying SER to Memory Safety, a defense which we call SpecASan to prevent more classes of transient execution attack like MDS.

4.1 SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation

4.1.1 Introduction

The recent Spectre [143] attacks have demonstrated how speculative execution can be exploited to enable disclosure of secret data across isolation boundaries. Specifically, attackers can misguide the processor to speculatively execute a read instruction with an address under their control. Although the speculatively read values are not visible to programs through the architectural state, since the misspeculation effects are eventually undone, they can be communicated out using a *covert channel*. Since their introduction, a large number of attacks following the same pattern (temporary read of sensitive data through speculation, followed by disclosure of this data through a covert channel (e.g., [101])) have been discovered which enable bypassing different permissions using a number of different speculation triggers [21, 28, 68, 94, 141, 148, 167, 210, 229, 240, 259]; it is clear that this is a general class of vulnerability that requires deep rethinking of processor architecture.

Since speculation is essential for the performance of modern processors, to mitigate this threat without severely restricting speculation, some solutions such as InvisiSpec [269] and SafeSpec [136] propose separating speculative data from committed data. Such an approach, rather than attempting to limit speculation, would isolate possible leakage. However, the principle has to be applied to every micro-architectural structure (e.g., cache, TLB, DRAM row buffer), and it is unclear if this approach could prevent leakage through contention, for example, using the functional unit port side-channel [28].

Another direction to mitigate this threat is to restrict speculation if a potentially dangerous gadget can be executed speculatively. For example, Intel and AMD suggest inserting serialization instructions like `lfence` to prevent loading potentially secret data [12, 109]. Because blindly inserting serialization instructions will have the same effect as disabling speculation, thus severely reducing performance [107], a better solution is to conditionally insert barriers. The MSVC C compiler [176], oo7 [248], and Respectre [102] use static analysis to identify dangerous gadgets and only insert `lfence` before the identified gadgets. Context-Sensitive Fencing [233] dynamically inserts serialization instructions when a load instruction operates on untrusted data (address), but only for Spectre-PHT.

Our observation is that Spectre-like attacks rely on manipulating the processors' prediction structures (see Section 2.2 for details) to coerce speculation to an attacker-chosen code gadget. Therefore, these attacks can potentially be defeated more efficiently by identifying and preventing erroneous speculation when the prediction structures produce a wrong prediction. As a first step towards this direction, we propose SpecCFI, a lightweight solution to prevent the two most dangerous Spectre variants: Spectre-BTB (v2) and Spectre-RSB (v5). SpecCFI prevents these attacks by using control-flow integrity (CFI) principles to identify when a prediction is likely erroneous and constrains speculation if it is.

In contrast to traditional CFI, even hardware supported proposals, whose purpose is to prevent illegal control flow within the primary architecturally visible control flow of a program, SpecCFI pushes CFI to the speculation level, where it can be used to determine whether a speculative execution path should be allowed or limited. Compared to existing solutions against Spectre-BTB and Spectre-RSB, such as the recent microcode update from Intel [109] and retpoline [239], SpecCFI introduces less performance degradation as it still allows correct speculation to proceed, while these existing solutions blindly "disable" all indirect branch prediction.

We also like to argue that defenses against Spectre-BTB and Spectre-RSB serve as the foundation for defense against Spectre-PHT (v1) attacks. The reason is that serialization instructions can be viewed as a special type of inline reference monitor and, therefore, it is crucial to make sure that these inserted barriers are never bypassed. However, without protections against Spectre-BTB (forward indirect branches) and Spectre-RSB (returns), attackers can easily bypass the barriers to carry out the attacks [28]. Furthermore, as demonstrated in return-oriented programming [220], by jumping to the middle of an x86 instruction, attackers can use unintended gadgets, in our case speculatively, to launch attacks. For this reason, we envision SpecCFI being combined with existing solutions against v1 attacks [40, 188, 233] to provide comprehensive protection against Spectre attacks.

The SpecCFI principle can leverage any CFI implementation (e.g., coarse-grained such as Intel's CET [116], or fine-grained such as HAFIX [54]), with small differences in implementation and leading to the enforcement of the respective version of CFI. We present our baseline design for forward edge protection in Section 4.1.7 and backward edge protection in Section 4.1.9. We investigate two versions of SpecCFI: SpecCFI-base that implements CFI only for speculation, and SpecCFI-full that also supports CFI for the committed control flow (i.e. conventional goal of CFI). Section 4.1.12 evaluates performance and complexity of the design. We show that SpecCFI-base eliminates dangerous misspeculation (where the predicted target label does not match the destination), without impacting performance.

SpecCFI-full incurs an additional small overhead, on par with other hardware CFI implementations [50, 54, 55]. We also analyze the implementation complexity and find that the overhead is modest.

Although some software and hardware solutions have started to appear to defend against this class of attacks, we believe that our solution is elegant along with a number of interesting properties. We believe that it also combines well with other proposed defenses, such as SafeSpec [136] and InvisSpec [269] which limit the speculative side effects once misspeculation occurs, by limiting the opportunities for harmful speculation. Section **??** compares SpecCFI to these and other works.

In summary, the contributions of the chapter include:

- We present a new defense against Spectre variants that rely on polluting the BTB and RSB, by embedding CFI principles into the branch prediction decisions.
- We analyze the security of the proposed designs showing that it protects against all variants of Spectre-BTB (v2) and Spectre-RSB (v5) attacks. Combined with solutions such as context-sensitive fencing, we believe that we can completely secure the system against Spectre attacks.
- We analyze the performance and complexity of SpecCFI, showing that it leads to little overhead. Compared to a defense that prevents speculation around indirect jumps,

Table 4.1: Spectre attack variants and their targeted branch prediction components

Spectre	**Element exploited**
Spectre-PHT (v1) [143]	Pattern History Table (PHT)
Spectre-PHT (v1.1) [141]	Pattern History Table (PHT)
Spectre-BTB (v2) [143]	Branch Target Buffer (BTB)
Spectre-RSB (v5) [148, 167]	Return Stack Buffer (RSB)

indirect calls and returns, SpecCFI provides equivalent security yet still avoids the large performance overheads. The hardware complexity is also negligible.

4.1.2 Background

Spectre Attacks Spectre attacks exploit the branch and aliasing predictors to fool them to access unauthorized data speculatively [28,35,94,141,143,148,167]. The main properties that the attack exploits in speculative execution are: (1) lazy permission checks on speculation: while instructions are being executed speculatively, the processor will not check the permissions until the commit stage; (2) Speculative instructions have unintended side-effects on micro-architectural states even if they do not get committed; and (3) attackers can deliberately mislead execution into attacker-intended gadgets by mistraining branch predictors, and use the previous property to leak sensitive information. Specifically, an attacker selected gadget is executed speculatively to perform unauthorized access and leak the value through a side-channel [21, 107, 143]. Based on the prediction structure being attacked, variants of the Spectre attacks that are addressed in this work are shown in Table 4.1. Mitigations for other variants of the Spectre attacks as well as variants of the Meltdown attack have been discussed in detail by Canella et al. [35].

Control-flow Integrity

Control-flow integrity (CFI) [10, 195] is a state-of-the-art solution to mitigate control-flow hijacking attacks. In such attacks, attackers corrupt/overwrite control data (i.e., data that controls indirect control transfer, function pointers and return addresses for instance) to divert the victim program's execution to carry out attacker-chosen logic, for example, to enable malware or open a backdoor. CFI prevents such attacks by enforcing a basic safety property: *software execution must follow only legal paths within a control-flow graph (CFG) determined ahead of time* [10]. Hence, a CFI mechanism always consists of two components: one that computes the CFG of the program and one that regulates the control transfer while it is executing.

Constructing CFG. The security guarantee of a CFI mechanism directly depends on the accuracy of the CFG, which can be constructed through static or dynamic analysis. Coarse-grain CFI mechanisms [283,284] generate the CFG using static analysis: any address-taken function can be a legitimate target for any indirect call; any address taken basic block can be a legit target for any indirect jump; and the address of the next instruction after any call can be a legit target for a return. Although coarse-grained CFI can eliminate most illegal control transfer targets, follow-up research has shown that the CFG used is too permissive/inaccurate that it still allows attacks [38, 77]. Fine-grained CFI solutions improve the accuracy of the CFG by incorporating type information [185,196,235,243,249]. Unfortunately, the CFG may still allow illegal control transfers [37, 67]. More recently, researchers have proposed utilizing run-time information to further improve the precision of the CFG [60, 187, 241], which can even achieve perfect accuracy [99] (i.e., one possible target per indirect control transfer site).

Regulating control-flow. Once the CFG is calculated, legitimate control transfers can be grouped into equivalence sets. Within the same set, control-flow can be transferred from any source location (e.g., a call site or return site) to any target location (e.g., target function or call site). By assigning each equivalence set a unique ID/label, run-time control-flow can be regulated with a simple check—source label must match destination label. Such checks can be implemented using either software or hardware. Some hardware extensions only support a single label [23, 116, 130] thus can only enforce coarse-grained CFI. Others support multiple labels [50, 55] and fine-grained CFI. Some hardware extensions also include a shadow stack to enforce unique return target [50, 54, 116, 130].

Adoption. Because of its effectiveness against control-flow hijacking attacks, CFI has been adopted by both commodity software and hardware. Tice et al. [235] introduced forward-edge CFI to LLVM and GCC in 2014. Android adopted this implementation in 8.1 to protect its media stack and extended the protection in Android 9 to more components and the OS kernel. Microsoft introduced its own CFI implementation, control-flow guard in Visual Studio 2015 and has been utilizing it to protect important OS components, including the web browser. In Windows 10 (V1730), Microsoft extended the protection to the OS kernel and hypervisor (Hyper-V). On the hardware side, Intel introduced Control-flow Enforcement Technology (CET) [116] and ARM introduced a similar mechanism, Branch Target Indicators (BTI), in ARMv8.5-A [23].

4.1.3 SpecCFI System Model

This section first overviews the threat model we assume in this work. It also describes the extensions to the Instruction Set Architecture (ISA) to support SpecCFI and the compiler modifications to use them.

4.1.4 Threat Model

The main goal of SpecCFI is to prevent attackers from launching branch target injection attacks (i.e., Spectre-BTB and Spectre-RSB). We assume a strong local adversary model with a shared BTB across different hardware threads (i.e., hyperthread) and protection domains (address space, privilege level, and SGX enclaves). We assume the RSB is not shared between hardware threads, consistent with existing CPU designs, but it is shared between different protection domains. Specifically, we assume adversaries can inject arbitrary branch targets into BTB in an attempt to control the predicted branch target in the victim protection domain.

Meltdown style attacks [157, 178, 210, 211, 240, 244] are outside the threat model since they occur due to speculation on the value to be used within the execution of the same instruction; privileged kernel memory [157], L1 cache contents [240], fill buffer [211], in-flight data in modern CPUs (for example: Re-Order Buffer and Line Fill Buffers) [244], and store buffer [178, 210]. Moreover, misspeculation through the direction predictor (which leads to Spectre-PHT) does not result in a control flow violation, since both conditional branch directions are legal control flow paths. Luckily, existing works have already developed protections against Spectre-PHT, primarily by limiting speculation around conditional branches that can lead to dangerous misspeculation [40,102,176,233,248]. Similarly, Spectre-STL is out-of-scope but can be mitigated by disabling speculative store bypass [11,20,109]. To the best of our knowledge, SpecCFI is the first hardware design that targets the more dangerous Spectre-BTB and Spectre-RSB attacks even when they use different side-channels (e.g., contention-based side-channel in SMT processors [28]).

We further assume that target software is protected with hardware-enforced CFI, which marks valid indirect control transfer targets (e.g., `ENDBRANCH` in CET). Although the target software may contain memory vulnerabilities (e.g., buffer overflows) that could be exploited to achieve arbitrary read and write (i.e., the traditional threat model for CFI), such attacks are out-of-scope of this work.

4.1.5 Instruction Set Architecture (ISA) Extension

Most hardware CFI extensions [23, 50, 54, 55, 116] use target labeling to enforce forward-edge CFI, and a shadow stack to enforce backward-edge CFI. Without the loss of generality, we assume two modifications to the ISA to inform the hardware of the labels from the CFG analysis:

- Extending the indirect `jmp` and `call` instructions to include CFI labels. For coarse-grained CFI enforcement (e.g., Intel CET [116] and ARM BTI [23]), the label at jump and call sites can be omitted.
- Adding a new instruction to mark legitimate indirect branch targets with corresponding labels. For coarse-grained CFI enforcement, the label can be omitted (e.g., the

case of Intel CET) or collapsed to two labels: one for jump targets and the other for call targets (e.g., the case of ARM BTI).

The shadow stack is generally transparent to the program and will not be directly manipulated. However, certain language features such as exception handling, `setjmp/longjmp`, require manipulation of the shadow stack. To support these features, additional instructions are needed, but since they do not interact with SpecCFI, we omit their details. The Intel CET specifications [116] provide an example of such instructions. Table 5.1 summarizes required ISA changes.

Table 4.2: ISA Extensions to support CFI.

Instruction	**Description**
`call [dest],label`	Target class-aware call
`jmp [dest],label`	Target class-aware jump
`cfi_lbl`	Verify CFI integrity

4.1.6 Compiler Modification

SpecCFI relies on the compiler to mark valid indirect control transfer targets with labels. Fortunately, because these required modifications are the same as CFI, they are already available as part of commodity compilers. For example, both LLVM and GCC include support for (1) software-enforced fine-grained forward-edge CFI [235], (2) Intel CET, and (3) ARM BTI. Therefore, SpecCFI requires little or no modifications to the compilers. SpecCFI is compatible with any label based CFI implementation.

4.1.7 Forward-Edge Defense

In this section, we describe the component of SpecCFI responsible for preventing both misspeculation as well as control-flow that breaks CFI on the forward-edge (i.e., on indirect calls and indirect jumps). This defense is responsible for preventing Spectre-BTB (v2) both within the same address space and across different address spaces. It is also responsible for maintaining CFI integrity on committed instructions (the traditional use of CFI).

Preventing Spectre-BTB (within the same address space)

In this attack, the attacker pollutes the target BTB entry by repeatedly executing an indirect branch in its own address space that hashes into the same entry. The attacker can use script engines like the JavaScript engine in browsers and the BPF JIT engine in the kernel. When the victim branch is executed speculatively, the polluted entry will direct the victim to a malicious gadget. Our goal is to prevent the victim from jumping speculatively to the malicious gadget.

Our first design considers augmenting the BTB to hold a CFI label for the target. This design extends indirect `call/jmp` instruction execution to update the BTB to add the CFI label of the branch. Later in the speculation path, all indirect calls and jumps are indexed to the BTB to predict their target as before, but with an additional check against the inserted CFI label. This defense prevents attacker-controlled misspeculation since the label of the attacker's instruction does not match the true target. For benign programs, such misspeculation is likely to occur only when the BTB is cold (has not been initialized yet), or when branch aliasing causes collisions in the BTB structure. While these cases should be rare, in both cases the value in the BTB is not the correct target. Limiting such erroneous speculation might result in performance improvement since we do not waste time on fetching instructions from what is likely to be the wrong path.

Since only committed indirect branches update the BTB, possible targets that may be used by attackers are limited to gadgets starting with a `cfi_lbl` instruction with an identical label to that of the `call/jmp` instruction's label. Note that a label may be shared by multiple locations in the code in CFI, and misspeculation among these locations is still possible (i.e., control flow bending [38]); as known from CFI solutions, this set is much smaller than the potential targets set without CFI.

Preventing Spectre-BTB (cross-address-spaces)

```
0x09: load rax,
    0x25
0x10: call *rax,
    L1
...
0x25: cfi_lbl L1
0x26: add rbx,1

        (Attacker)
```

```
0x09: load rax,
    0x50
0x10: call *rax,L1
...
0x25: load rbx,[
    secret]
0x50: cfi_lbl L1

        (Victim)
```

Figure 4.1: Example attack across address spaces

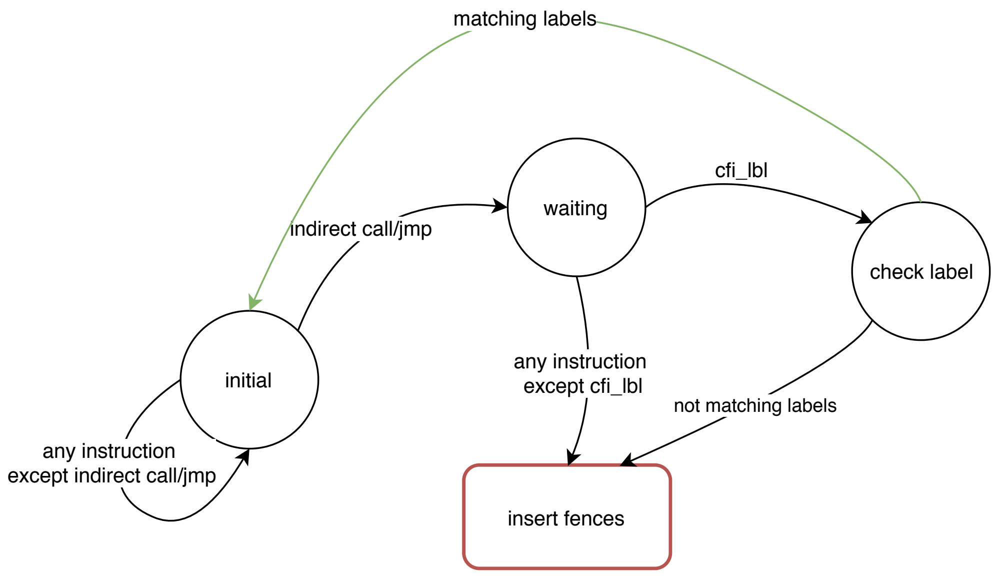


Figure 4.2: State machine for forward edge protection

Storing CFI labels in BTB entries mitigates attacks within the same address space, but not those across address spaces, when attackers pollute the globally shared BTB from another program. In this case, if attackers know the label used by the victim program (e.g., through offline analysis), they can craft an entry in the BTB with the same label and bypass the protection. Consider the example in Figure 4.1. The attacker inserts L1 and 0x25 in the 0x10 index of BTB, by selecting the label and location of a branch. When the CPU context switches to the victim space, the victim call at location 0x10 is indexed to BTB and uses the BTB entry, inserted by the attacker to predict its target. Since the label matches, the CPU continues speculative execution of the malicious gadget from 0x25, and the attacker successfully redirects the control flow and executes the malicious gadget to reveal the secret.

To prevent cross-address-space attacks, one possibility is to randomize the mapping of addresses to the BTB (e.g., similar to the CASESAR solution for caches [202]) to make it difficult for attackers to guess the label or the location associated with the target branch. However, as this approach only provides probabilistic guarantees against attacks, we decided to use an alternative implementation that avoids using labels in the BTB. Specifically, our implementation enforce the CFI check by ensuring that the first speculatively executed instruction after an indirect branch is a legal `cfi_lbl` instruction with a matching label, guaranteeing that the speculation target is a legal target in the program's Control Flow Graph. We note that this is the standard implementation of hardware acceleration of CFI. However, since we are using CFI to constrain speculation (not just the committed instructions), this approach requires pushing the check earlier in the pipeline to the decode stage of the first instruction on the speculative path. However, as our experimental analysis shows, this change results in negligible impact on performance legal speculation is not delayed.

With respect to performance, the two implementations operate differently, but are likely to perform similarly. The first implementation requires modifications to the critical BTB structure and can potentially slow down the execution pipeline, favoring the target label-checking implementation. A small disadvantage of the second implementation is that the target instructions have to be speculatively fetched (if not cached) to be able to check the label, which could be avoided if the label-mismatch is detected by the BTB in the first implementation.

The state machine implementing the check in the decode stage of the pipeline is shown in Figure 4.2. Starting at the initial state, any indirect `call/jmp` instruction in the decode stage sets the `CFI_REG` register with its own CFI label and causes the CPU to wait for a `cfi_lbl` instruction. The decode stage makes sure that the next instruction is a `cfi_lbl` instruction. This restricts potential gadgets to those starting with a `cfi_lbl` instruction. Moreover, the CPU will confirm that the `CFI_REG` value and the label of the `cfi_lbl` instruction are equal. In this way, potential gadgets are further restricted to those with a matching label. When the instruction following the `call/jmp` is not a `cfi_lbl` instruction or when the label of the `cfi_lbl` instruction does not match the label of the `call/jmp`, an `lfence` micro-op is inserted into the pipeline to guarantee prevent execution from the wrong speculative path.

4.1.8 Enforcing CFI for Committed Instructions

SpecCFI is essentially hardware-supported CFI, but with CFI enforcement during speculation. Thus, given the similarity in the hardware support to traditional CFI, we also extend the design to support standard CFI to enforce the CFI rules on committed instructions and defend against control flow hijacking attacks. This support is achieved by enforcing the CFI check during the commit stage of the pipeline: if an indirect `call/jmp` instruction is not followed by a `cfi_lbl` instruction with a matching label, the CPU raises a CFI violation exception.

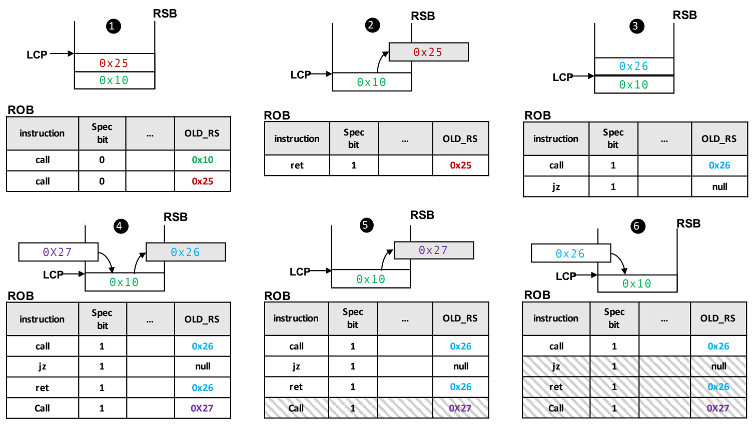
4.1.9 Backward-Edge Defense

The backward-edge defense component of SpecCFI protects misspeculation on return instructions. Return instructions typically obtain their predicted addresses from a hardware stack called the Return Stack Buffer (RSB). The RSB has been shown to be vulnerable to a range of Spectre attacks [148,167]. To provide protection for the backward-edge, hardware CFI proposals use a Shadow Call Stack (SCS), which is protected from normal memory reads and writes, and can only be manipulated through special instructions [116]. Similar to RSB, the SCS is used to retain the return addresses of previously executed calls. The differences are: (1) SCS is in memory, so it is saved and restored across context-switch; while RSB is a special cache in the CPU and its content is shared across different context. (2) SCS is only used for CFI enforcement and its size is configurable; while RSB is only used for speculation, and since misspeculation was thought to be only a performance problem, RSB is a best effort structure that is not maintained precisely and has a limited size.

Combined Speculation-consistent RSB/SCS: Overview

To provide defenses against Spectre-RSB attacks, we combine the traditional RSB and SCS into a unified structure RSB/SCS acting as both RSB and call stack. Conceptually, RSB in our design can be viewed as the in-processor cache for the in-memory SCS. We note that this is different from other SCS implementations that retain the RSB separately. By getting speculation targets from the precisely maintained SCS, consistent with the philosophy of SpecCFI, we move the CFI guarantees to the speculation stage, closing the Spectre-RSB vulnerability.

The overall design of RSB/SCS has additional requirements from the design of conventional SCS. Specifically, since we have to be able to use it to obtain speculation targets, it must track additional speculative state without affecting the committed state of the SCS. We describe the overall design in the remainder of this section.



ROB

instruction	Spec bit	...	OLD_RS
call	0		0x10
call	0		0x25

ROB

instruction	Spec bit	...	OLD_RS
ret	1		0x25

ROB

instruction	Spec bit	...	OLD_RS
call	1		0x26
jz	1		null

ROB

instruction	Spec bit	...	OLD_RS
call	1		0x26
jz	1		null
ret	1		0x26
Call	1		0X27

ROB

instruction	Spec bit	...	OLD_RS
call	1		0x26
jz	1		null
ret	1		0x26
Call	1		0X27

ROB

instruction	Spec bit	...	OLD_RS
call	1		0x26
jz	1		null
ret	1		0x26
Call	1		0X27

Figure 4.3: Example of the operation of the combined RSB/SCS

When a context switch occurs, the committed RSB/SCS entries must be saved such that they can be restored when the program runs again. To be able to keep the state of this structure consistent, we extend the reorder buffer (ROB, which is the structure in the CPU used to track speculative instructions and their register values before they commit) to track this state. Specifically, we add a logical register `OLD_RS` which (is subject to renaming and) holds the return address that is pushed to the RSB/SCS by a call instruction, or popped by a return instruction from the RSB/SCS. In addition, we keep track of a pointer to the last committed entry (LCP) of the RSB/SCS so as to save and restore the state of committed entries in this structure in the case of context switch or a spill overflow to memory. At the decode stage, If the instruction is a call, the next address is “speculatively” pushed to the RSB/SCS structure. When this instruction commits, the LCP is updated to point to the last committed entry. If the instruction is decoded as a return it “speculatively” pops a return value from the RSB/SCS structure into `OLD_RS` (without changing LCP) and sets the program counter to this address. To support conventional CFI, when the return instruction reaches the commit stage, the value of the `OLD_RS` register is compared with the top of the traditional software stack. If these two values do not match, a CFI violation exception is raised.

We considered the need to provision the stack with additional ports since it is used not only to serve committed instructions, but also to handle speculative calls and returns. However, we found that additional ports do not result in performance benefits because the speculative SCS state is held primarily in the port-rich reorder buffer. When the in-processor cache (RSB) overflows or the current thread is about to be swapped out, we spill it over to the hardware-protected in-memory SCS. When the RSB underflows or a new thread is swapped in, we load entries from the SCS. We did not explore optimization to prefetch values from the SCS when RSB is close to empty, or to push some values proactively to memory when RSB gets close to full.

Misprediction Recovery

Every `ret` instruction utilizes the RSB/SCS to predict its jump target. Since the state of RSB/SCS is modified by speculative call and ret instructions, in case of misspeculation, the CPU has to recover the correct state of the structure.

When misspeculation is detected, we need to flush all the speculated instructions from the pipeline. As a part of this process, we have to annul all the corresponding entries from the ROB. During annulment, for every call or return instruction, we not only remove the ROB entry but also update the RSB/SCS to preserve the consistent state of the structure. If the instruction is a call, the top of the RSB/SCS is be popped. In the case of a `ret` instruction, the value of `OLD_RS` will be pushed back to the RSB/SCS.

RSB/SCS Work Flow

To clarify how this structure works, we step through the example code sample presented in Figure Figure 4.4.

```
0x09:       call Function1;

0x10:

0x24:       Function1:

                   call Function2;

0x25:              call Function3;

0x26:              call Function4;

0x27:

0x36:       Function2:

                   ret;

0x74:       Function3:

                   jz 0x86;

0x86:       ret;
```

Figure 4.4: Code sample to illustrate the operation of RSB/SCS

Assume both calls to *function1* and *function2* have pushed their return values to the RSB/SCS. By committing these instructions at ❶, the LCP is updated to point to the last committed value and then the corresponding entries are evicted from ROB. In the second step ❷, the return instruction from the first call is being executed speculatively, saving the return address in the ROB, and eventually getting committed. The following speculative call to *function3* at ❸, will push its return address to RSB/SCS. At step ❹, the execution of the return instruction and the following call to *function4*, change the RSB/SCS state. Assume that a misspeculation on the `jz` instruction has been detected at ❺ and every instruction executed after the branch has to be flushed. Therefore, the recovery process starts annulling instructions from the last entry in ROB until the misspeculated instruction has been reached. Annulling the last call in the ROB at ❺, the value at the top of RSB/SCS is popped and at ❻, annulling the return, the OLD_RES value of the instruction saved in ROB is pushed back to the RSB/SCS to reset the state to the previous state before the misspeculation.

Preventing RSB Poisoning

Since the RSB/SCS is not shared between different threads and preserved across context switches, the attacker is not able to poison this structure. Although we allow special instructions to manipulate the SCS to take care of cases such as `setjmp/longjmp`, we assume these instructions are only available to code within the trusted computing base to prevent them from being abused to arbitrarily manipulate the RSB/SCS (which is not a Spectre vulnerability).

4.1.10 Comparison to Intel CET

A few days before the submission of this work, Intel published a new specification of its CET [116] extensions. The new specification includes a paragraph (section 3.8) indicating their plans to include a check that an indirect branch executed speculatively targets a legal `Branch_end` target. Intel suggested this solution, which is essentially the configuration of SpecCFI using CET as the CFI implementation, concurrently with our work.

We believe that Intel's interest in this solution validates it practicality as a defense against transient speculation attacks. While the updated CET specifications document describes only the general idea, our work contributes a reference implementation and assessment of both the performance and security of the solution. In addition, SpecCFI provides substantial security advantages over the new CET, including:

- Backward edge protection using the speculation aware shadow stack. While Intel CET uses a shadow stack to protect the backward edge for committed instructions, the specifications describe no plans to use it for limiting speculation. It is not trivial to extend the shadow stack to track the speculative state, as we describe in Section 4.1.9.

- Generalized CFI protection and limiting control flow bending. CET only enforces that control flow (whether committed or, in the new specifications, speculative) happens to the start of a legal basic block. As a result, it allows arbitrary control flow bending [37], which does not meaningfully restrict the attack opportunities. In contrast, SpecCFI admits any CFI implementation, which can substantially shrink the control bending attack possibilities. Specifically, from a given indirect control flow instruction, only the gadgets with matching CFI label are reachable. State-of-the-art CFI systems such as PathArmor/Context Sensitive CFI can be supported [241] substantially limiting the control flow opportunities. In particular, we intend to explore supporting uCFI [99] in our future work, leaving no control flow bending opportunities available.

4.1.11 Security Analysis

In this section, we analyze whether SpecCFI can achieve its primary security goal: preventing attackers from exploiting branch target injection to ultimately launch Spectre attacks.

Guarantees against Branch Target Injection

Branch target injection attacks target two prediction components: the branch target buffer (BTB) and the return stack buffer (RSB). Similar to CFI, SpecCFI does not prevent such injections: we assume attackers can still insert arbitrary targets into the BTB, for example by executing branches inside their own protection domain [70]. What SpecCFI guarantees is that if the injected target is not a valid indirect control transfer target in the victim protection domain, then the injected prediction target will not be executed speculatively, i.e., they cannot speculatively execute arbitrary code gadgets. For RSB, SpecCFI essentially converts it into a precise shadow call stack (SCS) and maintains it across context switches, such that both in-address-space injection and cross-address-space injection are no longer possible.

Impact of Imprecise CFG: One weakness of static CFG construction is imprecision, leading to having multiple possible targets with the same label. This ambiguity may still allow attackers to launch attacks using permitted function-level gadgets [34, 37, 67, 209]. Since

SpecCFI also relies on the CFI analysis to provide valid targets for forward-edge indirect control transfer, it also inherits the same limitation: *mis-prediction is still possible to any of the targets sharing a valid label.* Since SpecCFI is compatible with any label based CFI, it can benefit from improvements in CFI systems that are increasing the precision in tracking the legal control flow.

Incorporating Defense against Spectre-PHT

SpecCFI on its own can only mitigate Spectre-BTB and Spectre-RSB attacks. In this subsection, we discuss how SpecCFI can be (and *should* be) combined with Spectre-PHT defenses to complete the defense against known Spectre variants. In particular, to defend against Spectre-PHT attacks, researchers have proposed code analysis techniques [102, 176, 248] to (1) identify dangerous code gadgets that can be used to leak information and (2) conditionally insert serialization instructions (e.g., `lfence`) to prevent these dangerous code gadgets from being executed speculatively. One tricky part of such analysis is that, although on the committed path, direct control transfer is always correct; during speculation, even direct control transfer can be wrong. As a simple example, consider a direct call behind a conditional branch: if the prediction on the conditional branch is wrong, then the following direct call is also wrong. For this reason, when analyzing the code to identify potential dangerous gadgets for Spectre-like attacks, one must perform inter-procedural analysis (for both direct and indirect calls) to account for gadgets that may span across function calls. The unique opportunity here is that, if the static analysis to identify and eliminate Spectre gadgets uses the same CFG for CFI enforcement, then malicious gadgets at the beginning of function should already be eliminated. As a result, when combined with such defenses, even if SpecCFI allows misspeculation due to imprecise CFG, the wrong target cannot be used to launch attacks, because the gadgets have already been eliminated.

At the same time, defenses against Spectre-PHT attacks have to use SpecCFI-like techniques to be sound. The reason is the same reason inline reference monitors like Software Fault Isolation [174,276] have to enforce some control-flow regulation—if attackers can hijack the control-flow to arbitrary locations, then they can easily bypass the inserted checks and bypass the protection. This is especially dangerous to variable length ISA like x86 where attackers can jump to the middle of an instruction to find unintended instructions forming exploitable gadgets. Similarly, SpecCFI provides the same runtime guarantee to Spectre-PHT defenses: by enforcing that even speculative control-flow cannot deviate from the CFG used in static analysis, *the code being analyzed and instrumented will be the same as that executed.*

4.1.12 Performance and Complexity Evaluation

In this section, we evaluate SpecCFI in terms of performance and hardware complexity. All performance experiments were conducted using the MARSSx86 (Micro Architectural and System Simulator for x86) [193], a widely used cycle accurate simulator. MARSSx86 is built using PTLsim [279] and does a full system simulation (including the OS) on top of the QEMU [27] emulator. First, we configured MARSSx86 to simulate an Intel Skylake processor; configurations are shown in Table 6.1. We then integrated SpecCFI

Table 4.3: Configuration of the simulated CPU

Parameter	**Configuration**
CPU	SkyLake
Issue	6-way issue
IQ	96-entry Issue Queue
Commit	Up to 6 Micro-Ops/cycle
ROB	224-entry Reorder Buffer
iTLB	64-entry instructions Translation Lookaside Buffer
dTLB	64-entry data Translation Lookaside Buffer
LDQ	72-entry Load Queue
STQ	56-entry Store Queue
RSB	16-entry Return Stack Buffer
I-Cache	32 KB, 8-way, 64B line, 4 cycle hit
D-Cache	32 KB, 8-way, 64B line, 4 cycle hit

into the simulator to model all new operations realistically and in full details, in order to retain hardware faithful cycle accurate modeling of the extended processor pipeline.

Performance Evaluation

We use the SPEC2017 benchmarks [4] for evaluation, which is a standard benchmark suite used to evaluate the impact of processor modification on a range of representative applications that exhibit a range of different behaviors. All benchmarks were compiled using an LLVM compiler that is modified to mark valid indirect control transfer targets with labels. Unfortunately, since there is no official LLVM front-end for FORTRAN [5], we were not able to compile 8 out of the 23 SPEC2017 benchmarks as they contain FORTRAN code.

One option to prevent Spectre attacks is to insert fences to stop speculation around indirect control flow instructions. In order to evaluate SpecCFI performance, we compare it against the following design points:

- Baseline: this is the case of an unmodified unprotected machine. Specifically, we compile and run the SPEC2017 benchmarks using unmodified version of LLVM compiler and MARSSx86 simulator. In all of our experiments, we use the Instructions committed Per Cycle (IPC), a common metric for evaluating the performance of processors, to report performance. The IPC values of the defenses are normalized to this baseline implementation without defenses; thus, a higher normalized value than 1 indicates better than baseline performance.
- *Retpoline-style software fencing*: we implement a system adding fences to indirect branches using software. The compiler is modified to substitute all the indirect branches and return instructions with a sequence of instructions which ensure that the target of the branches are resolved before any following instruction that might touch the cache (i.e, load) are issued. For protecting the forward edges (i.e. indirect call and jumps) This is done by converting each indirect call to the three following instructions: ❶ a `load` preparing the value of the target register/memory, ❷ an `lfence` making sure that no future load is issued before the branch is resolved and ❸ the actual call to the address specified in the target register. Taking the same approach for securing backward edges (i.e. returns) we substitute any `ret` instruction with a sequence of ❶ a `pop` from top of the software stack to the target register, ❷ an `lfence` making sure to stop the speculation before the actual target of `ret` resolved and ❸ a `jmp` to

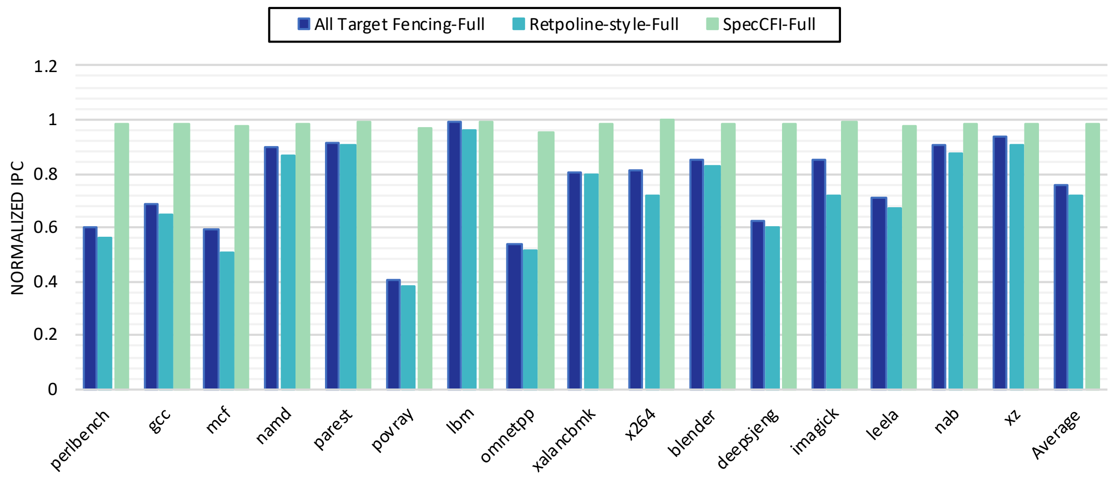


Figure 4.5: Performance Impact

transfer the control to the target. Conceptually, this solution is similar to the Retpoline defense [239] which essentially replaces speculation on indirect branches with an empty stall gadget. Different from Retpoline, we also insert the fences for returns (Retpoline does not protect returns, and leaves the code vulnerable to Spectre-RSB attacks).

This software approach has the advantage of not modifying the underlying hardware but imposes a noticeable overhead in the number of instructions and code size.

- *All Target Fencing*: In this approach, we show one implementation with an `lfence`, inserted in hardware, at target of each indirect branch and return (the all target fencing) since such a defense is possible without CFI. This is done by detecting every indirect call, jump, or return in the decode stage of the pipeline and inserting an `lfence` at target of them to make sure that the branch is resolved before issuing further instructions.

The implementations discussed above prevent speculation by inserting `lfence` into the pipeline. SpecCFI offers a more intelligent and targeted way of using fences for securing forward edges (as discussed in Section 4.1.7), as well as a new method for making backward edges secure (as explained in Section 4.1.9). To study the effect of different serializing instruction we use two different types of `lfence` instructions in our experiments:

- *Strict* `lfences`, are highly restrictive and prevent any instruction to pass through them until the fence retires [233]. This type of fences impose high overhead to the system. All the x86 serialization instructions including the `lfence` we use in our experiment, categorize as strict fences.
- *Relaxed* `lfences`, only stop certain types of instructions until the fence gets retired [233], while letting the others through. For example, LSQ-LFENCE [233], prevents any subsequent load instruction from being issued speculatively out of the load-/store queue but allows any other instruction to pass it. LSQ-LFENCEs are secure against Spectre because they prevent the speculative loads, and have the advantage of letting speculation on other types of instructions proceed, substantially reducing the performance impact.

Figure 4.5 shows the performance overhead of SpecCFI-full (securing both forward and backward edges) in comparison to the *All Target Fencing* and *Retpoline-style software fencing* approaches. We note that in general, inserting serializing instructions (e.g, `lfence`) in the target of every indirect branch is expensive, imposing performance overhead of 39% and 48% on average for *All Target Fencing* and *Retpoline style* respectively. Using SpecCFI, by inserting `lfence` only when the CFI check fails, the number of inserted

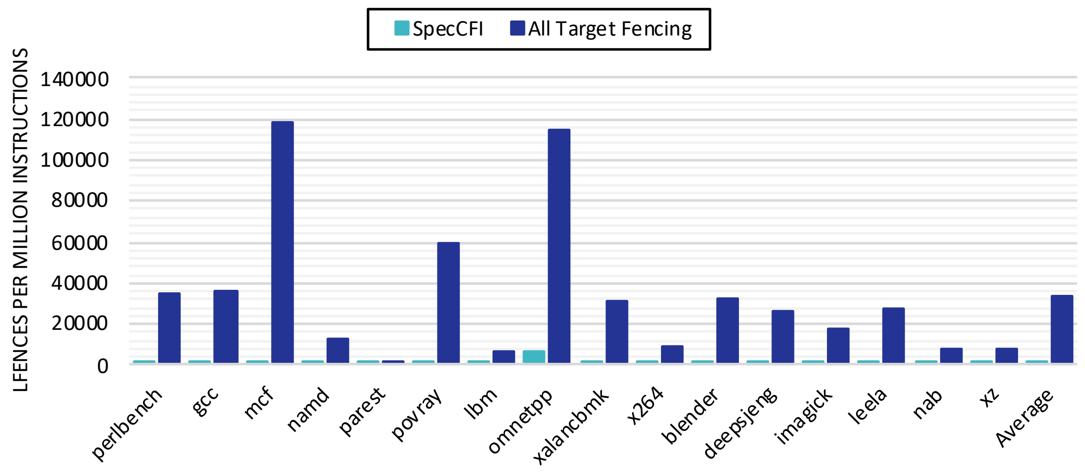


Figure 4.6: Number of lfences inserted by different defenses

`lfence` drops significantly thus reducing the performance overhead to less than *1.9%* on average.

To illustrate the reason behind the performance reduction in the different approaches, we study the number of `lfence` instructions inserted in each approach in Figure 4.6. Note that benchmarks such as `mcf` and `omnet`, are C++ benchmarks which use a large number of indirect branches due to the common use of virtual function calls and function pointers. As a result, this leads to a large number of `lfence` being inserted into the pipeline, and to a substantial performance impact compared to the baseline implementation. The only exception to this trend is `Provay` which suffers the highest overhead for all the defenses but does not have huge number of `lfence` compared to the other benchmarks. Looking more closely at this benchmark, we found out that it is a memory intensive benchmark with the highest number of load and store micro-ops among all the benchmarks. Intel manuals [115] indicate that an `lfence` is committed only when there is no preceding

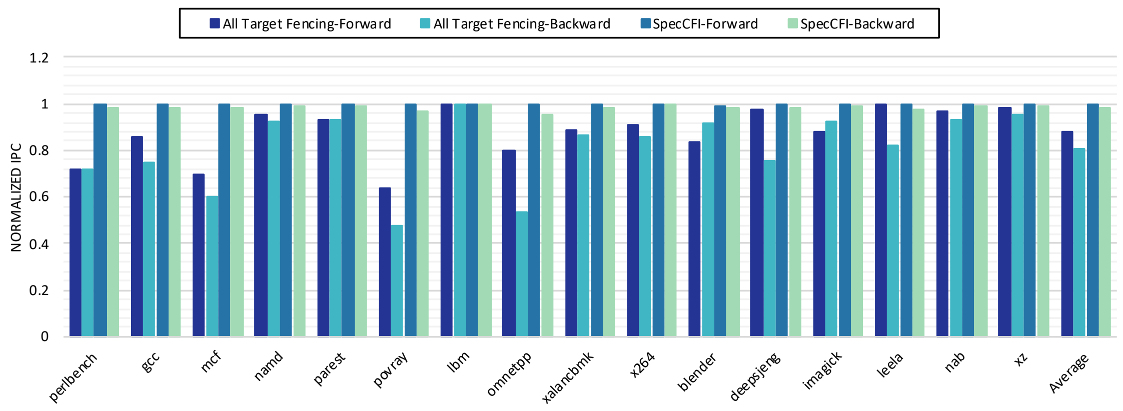


Figure 4.7: Overhead breakdown for forward and backward edge

outstanding store. Thus, for this benchmark, each `lfence` instruction remains active for a longer period of time until it gets committed which explains the high performance impact. It is also worth mentioning that unlike the *All target fencing* and *Retpoline-style* which insert `lfence` for each indirect branch, the `lfence` instructions for SpecCFI occur due to mis-prediction detected as a label mismatch causing the insertion of the `lfence`. This means that the higher the rate of mis-prediction, the more `lfence` instructions are inserted.

In Figure 4.7, we study the effect of securing the forward and backward edges separately since they use separate mechanisms for protection. Note that in *Retpoline-style*, all return instructions are converted to a sequence of instructions terminating with a `jmp`, meaning that there is no remaining `ret` instruction (i.e. backward-edge) in the code compiled in this setting. Therefore, the overhead measured as the overhead of *Retpoline-style*-full is equivalent to only *Retpoline-style*-forward overhead and the overhead on the backward-edge is zero. The results from the breakdown show that as expected, the overhead in general increases with the number of indirect branches in *All Target Fencing*. As for

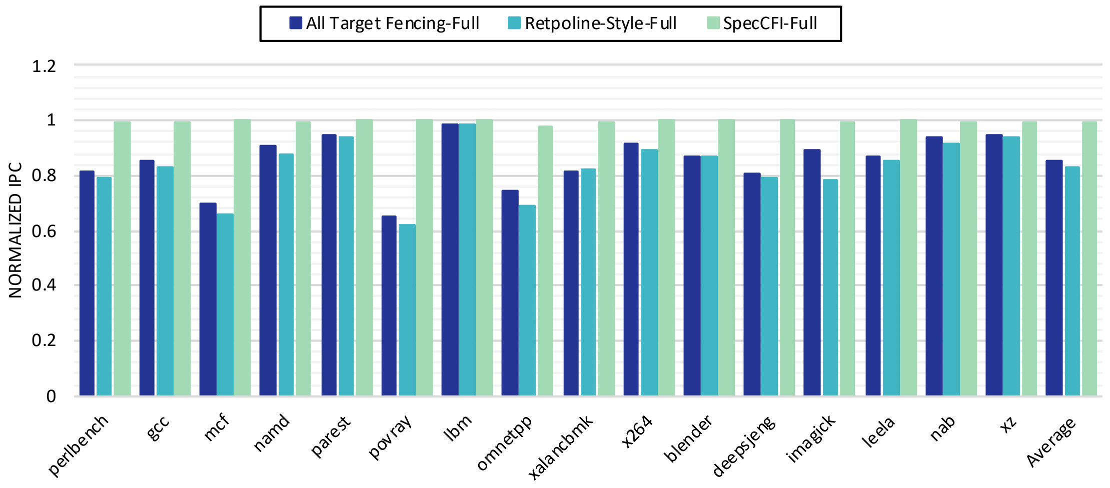


Figure 4.8: Performance using relaxed fences

SpecCFI, the overhead caused from forward edge defense is typically low: the overhead is incurred only on CFI mismatches which indicate misprediction of the branches. Therefore, the major part of the SpecCFI overhead is the overhead of SpecCFI-full on the backward-edge which is associated with maintaining the RSB/SCS hardware structure. It is important to consider that this maintenance effort also includes procedures to make sure the committed path is secure and therefore only a portion of this overhead is associated with defense against Spectre attacks.

Since strict `lfence` imposes a higher overhead on the system and relaxed `lfence` provides the same security guarantee with lower overhead, we implemented all discussed defenses with relaxed `lfence` as well to study the differences in overhead. Figure 4.8 examines the effect of relaxed `lfence`. The results show that the overhead caused by strict `lfence` is much higher than that of relaxed `lfence`. Also as expected, using strict instead of relaxed causes far more performance degradation when the benchmark is memory intensive (i.e., has a lot of stores in this case). Our results show that just by changing the type of the `lfence` from strict to relaxed, the average overhead drops down from 48.9% to 22.6% for *Retpoline-style* and from 39.9% to 18.82% for *All Target Fencing*. However, these overheads are still substantially higher than those of SpecCFI.

Hardware Implementation Overhead

To estimate the hardware overheads of SpecCFI, we implemented the primary hardware structures and integrated them within an open core to estimate the area and timing overhead. Specifically, the implementation consists of adding two `CFI_REG` registers in two locations of the pipeline: (1) decode stage, to support detecting CFI violations for speculative instructions and (2) commit stage, to support detecting CFI violations for committed instructions. Since `CFI_REG` is used to store the CFI labels its size should be the same as the maximum CFI label size (32-bits for our design). Furthermore, we need to add two comparators; one in decode and one in commit stage of the pipeline. These comparators will be used by `cfi_lbl` instruction to compare its label to the `CFI_REG` (todetect violations).

Additionally, SpecCFI needs a `LCP` register to point to the last entry of the RS-B/SCS from a committed call, used to distinguish between entries from speculative and committed instructions. Since RSB/SCS has 16 in-processor cache entries, the `LCP` size is 4-bit. Moreover, at two stages of the pipeline, new entries can be added to the RSB/SCS: (1) while executing call instruction and (2) load the preserved RSB/SCS entries from memory in case of underflow. Therefore, we had to update the number of write ports from 1 to 2. The same thing applies to the number of read ports, as we may use RSB/SCS to fetch next instruction while spilling over to memory in case of RSB/SCS overflow. In addition, to

Table 4.4: SpecCFI hardware implementation overhead

	Static power	**Dynamic power**	**Area**	**Cycle time**
SpecCFI	0.4%	0.4%	0.1%	0.0%

preserve the correct behaviour of RSB/SCS, we provided two `LCP` update mechanisms: (1) -/+1: for regular push/pop operations and (2) -/+4: for handling overflow and underflow of the structure. The cost of the RSB/SCS itself did not lead to a noticeable increase in complexity or area.

To measure the impact of SpecCFI implementation on power, area, and cycle time, we modified the open source processor (AO486) [17] to include SpecCFI design using Verilog. To synthesize the implementation of integrating SpecCFI to the processor on a DE2-115 FPGA board [2] we used Quartus 2 17.1 software. The results shown in Table 4.4 prove that SpecCFI indeed has low implementation complexity. In terms of power, there is a 0.4% increase in core dynamic and static power. Although it is difficult to measure power accurately, we applied the power analysis tool provided by Quartus to measure power after synthesis to get more accurate results. In terms of area, there is a 0.1% increase in total logic elements. Moreover, since SpecCFI design is simple, it fits within the optimized frequency of the core. Thus, it has no effect on cycle time. The AO486 processor is an implementation of the 80486 ISA using a 32-bit in-order pipeline. Thus, these results are relative to the small pipelined core; the overheads will be much smaller if compared to a modern out-of-order superscalar core.

4.1.13 Empirical Security Evaluation

Against real exploits

To verify our analysis, we evaluated the effectiveness of SPECCFI against real-world exploits. We ran previously disclosed Spectre-BTB [143], Spectre-RSB [148], and SMoTHerSpecter [28] PoC inside the emulator. Table 4.5 summarizes the results, using the same classification scheme proposed in [35]. The experiment results show that SPECCFI was able to prevent all information leaks.

Table 4.5: Empirical security evaluation of SPECCFI.

		in-place	out-of-place
Spectre-BTB	Cross-address-space	✓	✓
	Same-address-space	✓	✓
Spectre-RSB	Cross-address-space	✓	✓
	Same-address-space	✓	✓
SmotherSpecter	Cross-address-space	✓	✓
	Same-address-space	✓	✓

Impact of CFG precision

To study the difference between coarse-grained CFI (e.g., Inte CET [116]) and fine-grained CFI (e.g., SPECCFI) against BTB injection attacks, we used the SMoTher-Spectre [28] for a demonstration. In this scenario, the attacker has to find a BTI gadget in the victim process which loads a secret in a register and terminates by an indirect branch

Table 4.6: Available SMother Gadgets in Standard Libraries

Standard Libraries	CFI Implementation	
	Coarse-grained	*Fine-grained*
glibc-2.29	314	1
libssl-1.1	21	1
libcrypto-1.1	98	4
ld-2.29	64	0
libstdc++	47	0

to be able to perform BTB injection. By poisoning the BTB, the attacker transfers control to a SMoTHer Gadget to leak the secret. The SMoTHer Gadget starts with a comparison based on the target register followed by a conditional jump which enables SMoTherSpectre to leak the secret through a port contention side-channel. Figure 4.9 compares the required SMoTHer gadgets and feasibility of the attack under coarse-grained and fine-grained CFI.

Table 4.6 shows the number of available SMoTher Gadgets from several standard libraries. Using the constraints for the SMother Gadget identified by Bhattacharyya et al. [28], we scanned for valid SMoTHer gadgets in the first 70 instructions after label instructions (`endbr64` and `cfi_lbl`). For SpecCFI, we used a function signature based approach for generating labels [185, 187]. As we can see, although fine-grained CFI still permits some gadgets, the number is much smaller than that available under coarse-grained CFI.

It is worth mentioning that we only use SMoTHer gadget constraints as an example of practical gadgets. There are no clear systematic approaches to locate generic Spectre gadgets that are exploitable in practice, further analysis is required in order to find more specific constraints. We hope to pursue this question in our future work.

```
Train_BTB:
 0x1:mov rax, 0x20
 0x2:call *rax
foo:
  0x10: endbr64
  0x11: nop
```

Attacker

```
main: //BTI gadget
  0x0:mov rdx,[secret]
  0x1:mov rax,0x10
  0x2:call *rax //baz
        ()
baz: //Smother free
  0x10: endbr64
          ...
  0x14: nop
bar://Smother Gadget
  0x20:endbr64
  0x24:cmp $0, rdx
  0x25:je <>
```

Victim

(a) Coarse-grained enforcement of CFI (e.g. CET)

```
Train_BTB:
 0x1:mov rax, 0x20
 0x2:call *rax, L1
foo:
  0x20: cfi_lbl, L1
  0x21: nop
```

Attacker

```
main: //BTI gadget
  0x0:mov rdx,[secret]
  0x1:mov rax,0x10
  0x2:call *rax, L1//
        baz()
baz: //Smother free
  0x10: cfi_lbl, L1
          ...
  0x14: nop
bar://Smother Gadget
  0x20:cfi_lbl, L2
  0x21:cmp $0, rdx
  0x22:je <>
```

Victim

(b) Fine-grained enforcement of CFI (e.g, SpecCFI)

Figure 4.9: Speculative control-flow bending attack example.

4.2 Concluding Remarks

In this chapter, we presented a new defense that protects speculative processors against misspeculation targeting the branch target buffer (BTB) and the return stack buffer (RSB). These attacks are arguably the most dangerous speculation attacks because they can bypass compiler inserted fences. Prior defenses either excluded these attacks from their threat model, or implemented aggressive limits to speculation that dramatically degraded performance. In contrast, SpecCFI provides complete protection against these dangerous attacks, with little impact on performance, and with minimal hardware complexity.

SpecCFI introduces the idea of using CFI, explored previously as a protection against control-flow hijacking attacks for committed instructions (i.e., even on non-speculative processors), as a defense against speculation attacks. In particular, SpecCFI verifies the forward-edge of CFI on the instructions in the speculative path and only allows speculation if CFI labels match protecting against Spectre-BTB attacks. It also verifies the backward-edge using a unified shadow call stack, protecting against Spectre-RSB attacks. Essentially, SpecCFI moves the CFI check to the decode stage of the pipeline, preventing speculative execution of instructions unless they conform to the CFI annotations. For normal programs, this results in negligible performance degradation since it only prevents speculation with mismatching CFI labels, which will most likely result in misspeculation. By stopping misspeculation, we benefit from avoiding cache pollution and other resource waste during misspeculation.

Combined with recent proposals to mitigate Spectre-PHT, we believe SpecCFI mitigates the threat from known speculation attacks. Moreover, it does so without sacrificing performance due to speculative execution and with minimal modifications to the processor pipeline.

Chapter 5

SpecAsan: Protecting Data Speculation using Speculative Address Sanitizer

5.1 Introduction

In this chapter, we seek to establish the generality of this approach by applying SER to Memory Safety, a defense which we call SpecASAN, after the Address Sanitizer implementation of memory safety. Address Sanitizer (ASan) is an attack prevention technique used to ensure memory safety (reads and writes of data) only to/from instructions that are allowed to do so. We use an ASan implementation that leverages ARM's Memory Tagging Extension (MTE) [82,217,218] and modify it to enforce the ASan check in the early stage of the pipeline where it is applicable (Section 5.3.2). Our security analysis shows that this check mitigates attacks such as Speculative Load Bypass [35, 114] and potential Memory Data Sampling attacks [33, 178, 211, 244] by preventing illegal accesses to unauthorized data (Section 4.1.11). Our preliminary performance evaluation (Section 4.1.12) shows that, similar to SpecCFI, the cost of this defense is low, since MTE is already present in ARM. Moreover, the overhead is small, since speculation is only prevented when the security invariant does not hold, which often indicates misspeculation (or an attack).

5.2 Background

Memory safety bugs have been a persistent challenge in the field of software development and security for more than three decades. A test-driven development process, complemented by dynamic testing tools such as AddressSanitizer and Valgrind, forms a strong first line of defense against memory bugs. Beyond this, techniques such as fuzzing, ideally within a fuzz-driven development framework, and static analysis provide additional layers of scrutiny to uncover more elusive errors. Once a piece of software reaches production, existing memory bugs present potential vulnerabilities. Over time, a range of code-hardening techniques have been developed to make the exploitation of these bugs more challenging. These include stack cookies, non-executable memory, ASLR, and various forms of control flow integrity, such as LLVM CFI, Microsoft CFG, and Shadow Call Stack. More recently, the landscape of memory safety has expanded to include hardware-based solutions. The implementation of ARM Pointer Authentication in Apple's latest hardware provides cryptographic authentication of return addresses, impeding the use of return-oriented programming (ROP) attacks. Additionally, Intel's Control-flow Enforce-

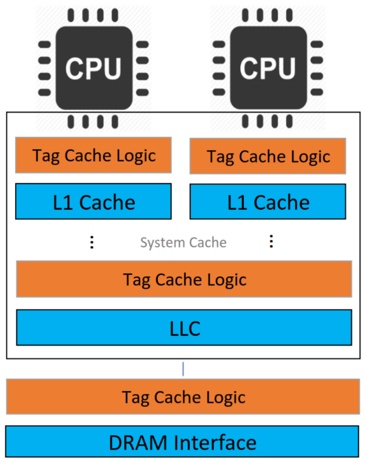


Figure 5.1: ARM Memory Tagging Extension (MTE)

ment Technology aims to combat ROP. Among hardware-based strategies, two notable implementations have emerged: SPARC ADI and ARM MTE, both rooted in the concept of memory tagging or memory coloring [218]. In the remainder of this chapter, we will narrow our focus to the ARM Memory Tagging Extension (MTE), as our proposed solution, SpecASan, is constructed upon this feature. By leveraging and extending the capabilities of ARM MTE, SpecASan aims to prevent a wide class of transient attacks. We will provide a detailed exploration of both the underlying principles of ARM MTE and the innovative ways in which SpecASan builds upon these to offer a solution to defend against this class of attacks.

ARM Memory Tagging Extension (MTE)

ARM's Memory Tagging Extension (MTE) splits memory into 16-byte tag granules and assigns a 4-bit tag to each 16-byte tag granule, referred to as the lock.

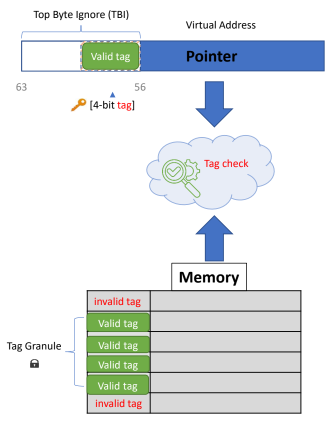


Figure 5.2: MTE: Memory Tagging Extension

Every pointer attempting to access memory is assigned a 4-bit tag, denoted as the key in Figure 5.2. In a 64-bit architecture, this tag is stored in the top byte of the pointer. This is permissible because the top byte of the virtual address is typically ignored during the address-translation process, leaving this space available for other uses such as storing the tag.

With each memory access, ARM's Memory Tagging Extension (MTE) requires that the 4-bit tag (referred to as the 'key'), hidden in the top byte of the pointer, must match the corresponding 4-bit tag in the memory (known as the 'lock') for the memory access to be permitted. This matching process serves as a security measure, ensuring that only authorized accesses occur.

Several instructions have been added to the ARM architecture to support Memory Tagging Extension (MTE). We will briefly explain the most important ones, highlighting their roles and significance in enhancing the functionality of MTE. additional instructions.

Table 5.1: ISA Extensions to support MTE.

Instruction	**Description**
`IRG <Xd>, <Xn>, <Xm>`	Insert random tag
`STG <Xt>, [<Xn|SP>,#0]`	store allocation tag

The *IRG* instruction takes a pointer from *Xn*, inserts a random tag into it, and writes the result to *Xd*. The *Xm* register can be used to specify a mask that controls which bits of the original value are preserved.

The *STG* instruction stores a tagged 64-bit value from the source register *Xt* to the address contained in the base register *Xn* or *SP*. The tag stored in memory is the same as the tag in the top byte of *Xt*. In addition to the ISA modification, an updated version of the AMBA 5 Coherent Hub Interface (CHI) specification is being developed to specifically support the transport and coherency requirements of ARM's Memory Tagging Extension (MTE) as shown in Figure 5.1.

Normal *Load* and *Store* instructions: When executing a load instruction, the processor initiates a read request for specific data located in memory. This request is then directed towards the appropriate level of cache or main memory, which acts as the responder. upon this request, the requested data along with the associated tag will be returned to the processor. In this scenario, the processor, acting as the requester, verifies the tag embedded in the load instruction (or pointer) against the tag of the returned memory location. If there is a mismatch between the tags, the system response is determined by the particular implementation. The processor might trigger a fault and terminate the program, or it might record the fault and continue the execution Figure 5.1.

For the *store* instruction, the processor takes on the role of the requester and initiates a write request to modify a particular data entry in memory. As responder, the corresponding memory unit is tasked with comparing the tag associated with the pointer to that of the target memory location. If there is a match, the memory unit writes the new data to the specified location Figure 5.1.

5.3 Speculative Execution Regulation (SER)

The program analysis techniques utilized by ASan/HWASan, in conjunction with ARM's Memory Tagging Extension (MTE), serve to ensure the correctness of a program during the commit stage of the pipeline. However, these techniques are inadequate in enforcing correctness for instructions executed speculatively, due to the nature of speculative execution. To address this shortfall, we introduce Speculative Execution Regulation (SER). SER aims to adapt traditional program analysis methodologies, extending enforcement into the speculative path of the pipeline, thereby enhancing the integrity of the speculative execution process. As an implementation of SER, we extend the ARM MTE architecture to support MTE's tags for internal micro-architectural buffers, such as the Line Fill Buffer, Load/Store Queue, load buffer, and caches. We refer to this approach as SpecASAN, which helps with the prevention of attacks such as MDS, Spectre v1, and several others.

5.3.1 Threat Model

We consider an adversary who wants to abuse speculative execution vulnerabilities to disclose some confidential information, such as private keys, passwords. We also assume the victim system is protected against other classes of (e.g., software) vulnerabilities. Finally, we assume the attacker can only run unprivileged code on the victim system (e.g., JavaScript sandbox, user process, VM, or SGX enclave), but seeks to leak information across arbitrary privilege levels and address spaces. In our analysis, we adhere to the threat models assumed by the original attacks in question.

The primary objective of SpecASan is to inhibit the forwarding of data from caches and internal micro-architectural buffers to the destination instruction (often a faulty load) during speculation, especially when such data should not be forwarded according to the program's data flow. This goal is accomplished by comparing the tag stored in the instruction (such as a load) with the tag stored in each internal buffer when a value needs to be forwarded. By ensuring a match between these tags, SpecASan reinforces control over the speculative path and prevents unauthorized data transmission. As we discussed in **??**, meltdown-type attacks including Micro-architectural sampling attack (MDS) try to force the processor to forward data from internal buffers by issuing a faulty load. So by applying the SpecASAN principles we are able to prevent most variants of Meltdown/MDS attacks.

Moreover, although SpecASAN cannot prevent misspeculation through the direction predictor (a scenario leading to Spectre-PHT), it can effectively stop the leakage of out-of-bounds data from the cache. Furthermore, SpecASAN is capable of stopping Spectre V4 (Speculative Store Bypass).

For the Spectre-BTB and Spectre-RSB attacks, which manipulate the control flow of the victim's program, SpecASAN cannot halt the pollution of the branch predictor. However, it can minimize the number of leakage gadgets that these attacks might exploit to leak data. Fortunately, other implementations of SER, SpecCFI, can provide preventive measures against these attacks. In conclusion, the combination of SpecCFI and SpecASAN, as dual implementations of SER, can act as robust defenses against a broad class of transient execution attacks.

In Section 5.4, we will explain in detail how SpecASAN can be utilized to prevent each class of attacks, providing an in-depth security analysis.

5.3.2 SpecASAN System Model, Architectural Modifications

In this section, we describe the modifications made at the micro-architecture level to enable SpecASAN. Specifically, these changes involve propagating and storing the tag at the required micro-architectural levels, and performing the tag check whenever necessary. In the existing architecture of the ARM MTE, tags are stored only in memory together with the data, and tag check logic will be handled on the processor or memory controller, following the requester, responder model explained in the background Section 5.2.
As illustrated in Figure 5.3, SpecASAN augments this architecture by not only storing tags in memory but also propagating them alongside the corresponding data within micro-architectural structures associated with *load* and *store* instructions. Whenever these entities are accessed during speculative execution, a tag check is performed. This ensures that data is not forwarded from these internal micro-architectural structures to a malicious load in

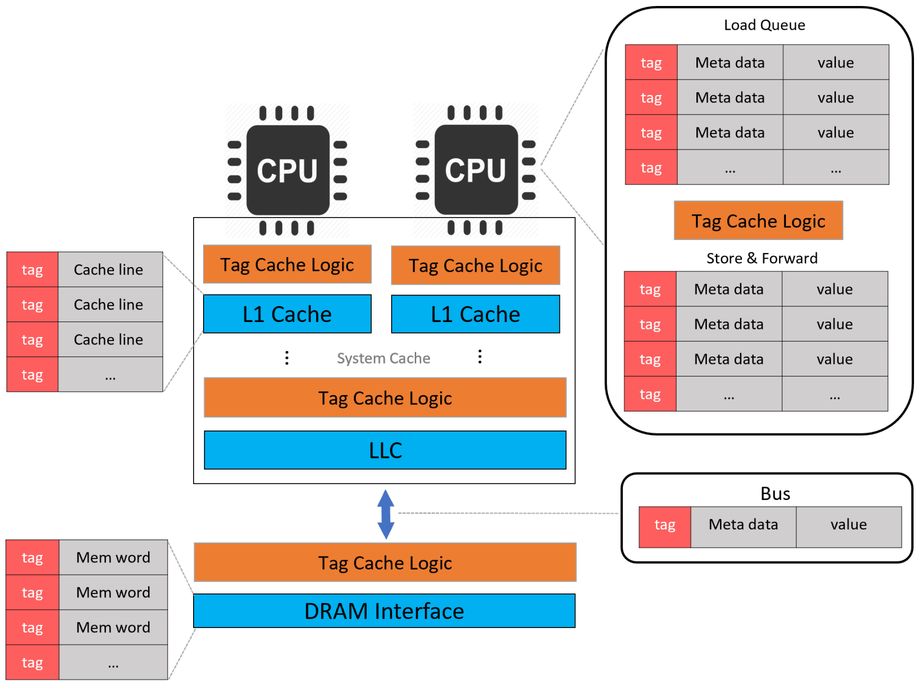


Figure 5.3: SpecASAN Architecture Overview

the event of a tag mismatch, thereby stopping a wide range of transient attacks. Subsequent sections discuss the specific modifications required for each structure in detail.

Data Cache(Dcache)

SpecASAN augments the D-cache to accommodate tags alongside data. Given that each 4-bit tag corresponds to a 16-byte data granule, a 64-byte cache block/line will contain four MTE tags, as shown in Figure 5.4.

In the following sections, we explain how SpecASAN will handle memory access request with respect to cache properties and policies.

Read Request

The processor sends a read request to the cache for a specific memory address and one of

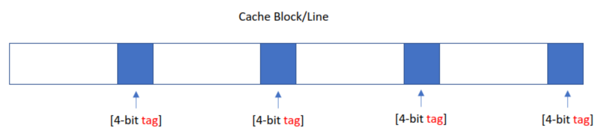


Figure 5.4: Extended cache block with MTE tag

the below cases could happen.

- *Cache Miss* If the cache misses, it acts as a requester and sends a read request to the main memory (or a lower-level cache), fetching the data and associated tag from the memory and perform the tag check logic and the cache line will be send back to the processor.
- *Cache hit* In case of a cache hit, both the data and its corresponding tag will be returned to the processor, where the tag will be verified.

Write Request - Write Back Policy

- *Cache hit* In case of a cache hit, both the data and its corresponding tag will be returned to the processor, where the tag will be verified.
- *Cache Miss* When a write miss occurs, and the cache operates in write-allocate mode, the cache fetches the block and its tag from memory, checks the tag, and writes the new data and tag to the cache, marking the line as dirty.
- *Eviction*: when a dirty cache line is chosen for eviction and the cache uses a write-back policy, the data will be written back to main memory before the line is replaced.

This approach may introduce coherency issues between the tag in memory and its counterpart in the cache. To better explain this, imagine that a cache line is dirty in cache, meaning it has been updated in the cache but not yet written back to main memory. Now, an *stg* instruction updates the tag for that same data in memory. As a result, there is a mismatch: The cache holds updated data with an old tag, while the main memory has outdated data with a new tag.

One possibility to address this issue could be synchronized updates: by allowing the *stg* instruction to also update caches directly or indirectly. For example, when updating a tag in memory, broadcast a coherency message, prompting caches to check if they have the associated data and update their local tag if needed. An alternative strategy to ensure consistency is to prompt an immediate write-back of the dirty cache line to the main memory whenever the tag gets updated by an *stg* instruction. This action, however, could have performance implications due to increased memory traffic.

Line Fill Buffer (LFB)

The Line Fill Buffer (LFB) serves various crucial roles in a CPU's memory system. It provides dedicated buffer space for data fetched due to cache misses, ensuring other instructions can proceed without delay. Furthermore, in support of out-of-order execution, the LFB can handle multiple cache misses concurrently, maximizing throughput. This ability is particularly beneficial in scenarios with frequent misses. Moreover, the LFB facilitates write coalescing, where writes to addresses currently being fetched into the buffer can be merged, thus decreasing redundant memory accesses. Additionally, when cache lines are set for replacement, they can temporarily move to the LFB before their eventual write-back, ensuring seamless data management.

The recent MDS attack [244], has demonstrated that attackers can extract data from the LFB during speculative execution. To stop such vulnerabilities, we have integrated the Memory Tagging Extension (MTE) into the LFB's design. When a memory request is missed in the L1D cache and the requisite data must be fetched from the LFB to service a load or store instruction, we introduce an additional tag check. The system compares the MTE tag stored within the LFB entry with the expected tag derived from the requesting instruction, so the data is forwarded only if there is a tag match.

Load/Store Queue

Modern out-of-order (OoO) execution microarchitectures deploy a Load-Store Queue (LSQ) to manage memory operations, specifically loads and stores. The Load-Store Queue (LSQ) streamlines the execution of load and store instructions by managing data dependencies and potential hazards. Modern processors use LSQs to facilitate out-of-order execution, ensuring the efficient flow of data within the pipeline. The LSQ helps to keep track of all the load and store operations that are in flight (i.e., issued but not yet retired). By maintaining this running list, the LSQ can identify potential data hazards, particularly those arising from the uncertain order in which memory operations might complete.

Load-Store Forwarding/Bypassing: One of the primary uses of the LSQ is to facilitate what's known as "load-store forwarding" or "load bypass." Here's how it works:

Imagine a store instruction writes data to a particular memory address. Shortly after, a load instruction seeks to read from the same address. In a basic pipeline without an LSQ, the load would need to wait until the store instruction has fully completed and the data has made its way to memory (or at least to a cache) before it can read the data. This introduces latency. With an LSQ, as soon as the data for the store instruction is available (even if it hasn't been written to memory yet), the LSQ can detect that the address of the subsequent load matches the address of the preceding store. Instead of waiting for the data to go through the memory hierarchy, the LSQ can forward the data directly from the store to the load, hence "bypassing" the usual route.

We have integrated the Memory Tagging Extension (MTE) into the Load-Store Queue (LSQ). With this integration, every new entry in the LSQ also records its associated MTE tag alongside its data. Before any data is speculatively forwarded to a subsequent instruction from an LSQ entry, a tag verification is performed to ensure that the requester's tag matches the one stored in the LSQ. For example in case of load/store bypassing, before data be forwarded from a store to a subsequent load operation, the LSQ checks the data's MTE tag against the anticipated tag for that specific address. The forwarding operation is permitted only if the tags match, ensuring that the data's integrity and source validity are preserved. In scenarios where the load/store unit potentially mispredicts a dependency, this mechanism guarantees that speculatively fetched data, especially from addresses that could be under adversarial control, cannot proceed through the pipeline without a validating tag match. Our security evaluation, as detailed in Section section 5.4, demonstrates the efficacy of this approach, showcasing its role in thwarting attacks like Spectre V4 and one variant of MDS attacks i.e *Fallout* [178] which exploit this optimization.

5.3.3 SpecASAN Implementation

Following the introduction of Arm's MTE, the Clang/LLVM compiler, starting from version 13, adapted to recognize and utilize the MTE capabilities. Similarly, the GNU C Library (glibc), as of its version 13, incorporated support for this architectural enhancement. So, our implementation requires no further modifications to the compiler. Our implementation approach for SpecASAN did not necessitate the introduction of any new instructions to the ISA. Instead, we incorporated the ARM MTE instructions, as detailed in section 5.2, into the GEM5 simulator. We further implemented the architectural modifications highlighted in subsection 5.3.2 within the GEM5 simulator's pipeline [29] to facilitate SpecASAN support.

5.4 Security Evaluation: SpecASAN against Micro-architectural attacks

In this section, we demonstrate how SpecASAN can be employed to mitigate various type of transient attacks in which lack of precise data flow information can cause ambiguities in the cpu pipeline. Attackers can leverage these ambiguities in diverse scenarios to extract sensitive data from the CPU's internal components. Through these examples, we emphasize the broad applicability of SpecASAN in addressing multiple types of transient execution attacks, as well as its ease of adoption.

In each example, we explain the source of ambiguity and data leak. we show how the SpecASAN can be adopted to provide needed data flow information to the CPU in order to overcome these ambiguities, and constrain the illegal data flow to close the vulnerability.

Spectre V1

In Example 1, we demonstrate how SpecASAN is effective at mitigating the Spectre V1 attack.Typically, a memory boundary is safeguarded by a conditional branch that checks whether a particular variable stays within permissible limits. Attackers aim to manipulate the branch predictor, seeking opportunities to leak confidential data by accessing it outside these bounds speculatively. While SpecASAN cannot directly thwart the mis-training of the predictor, it can mitigate the attack by preventing data from being loaded into the cache during speculative execution. By assigning tags to confidential memory regions and ensuring that the remainder of the memory carries different or default tags, an attacker's attempt to access out-of-bound data will trigger an SpecASAN check failure due to tag mismatch. Listing 5.5 shows an example of applying SpecASAN to prevent Spectre V1 attack.

In this example, line 4 shows a vulnerable directional branch, while line 5 indicates the memory access point. Here, an attacker might attempt to read out-of-bound data to retrieve the secret value depicted in line 1, simply by supplying a suitable index to this array speculatively. Using ARM MTE instructions, SpecASAN generates a random tag (e.g **0xa**)(line 2), and assigns this random tag to the memory address holding the secret, as shown in line 3. During each regular *load* instruction, SpecASAN verifies the tag of the pointer against the tag of the designated memory location. If the attacker supplies a value

```
// Generate random tag and insert
//it into the top 4 bits of the pointer "secret" (e.g., tag = 0xf)
1: char *secret = "password";
2: secret = (unsigned char *)insert_random_tag(secret);
// Set the tag stored in the top 4 bits of this pointer
// to the memory location to which it points.
// [secret, secret + 16) now have tag = 0xf
3: set_tag(secret);
// Accessing out of Bound
4: if ( x < array1_size) {
     y = array2[array1[x] * 256];
     // We did not set the tag for the array1 and array2
     // so both have default/different tags.
}
```

Figure 5.5: SpecASAN against Spectre V1

for x such that $x < array1\_size$, the program will proceed without interruption, given that both the pointer and the target location possess the default tags (0x0). However, if the attacker chooses an x such that $array1 + x$ accesses the memory location containing the secret, there is a tag mismatch: the pointer's tag is 0x0, whereas the memory location's tag is 0xa. This discrepancy will lead to an exception.

We note that the MTE implementation limits the size of the tag, making it possible to create tag collisions. However, this is implementation specific and future implementation could enlarge the tag space and make it possible to partition it so that a non-privileged program cannot create a tag collision.

Spectre V4 (Speculative Store Bypass)

In the second application of SpecASAN, we illustrate its use in mitigating Spectre v4. This Spectre variant leverages a CPU optimization that permits a load instruction, which may be dependent on an earlier store, to be speculatively executed before that store. The load/store unit attempts to predict whether a load instruction depends on a preceding store. If this prediction is erroneous, the load might read and possibly forward stale data to subsequent dependent micro-operations during speculation. If the attacker controls the input that determines the load address for the mis-predicted load instruction, they can access sensitive data during speculative execution (while the older store instruction waits for its operand to resolve) and subsequently leak it via a side-channel attack. In subsection 5.3.2, we outlined the necessary LSQ modifications to accommodate SpecASAN. Given that context, we present an example demonstrating how Spectre V4 can be prevented using SpecASAN.

Figure 5.6, shows a simple snippet of code in which the load instruction (line 4) is vulnerable to Speculative load bypass (Spectre V4) attack. In absence of data flow information, CPU does not have any clue about the data (sensitive/non-sensitive) that speculative load might access during speculative execution. SpecASAN can provide the hint to the CPU to ensure that it does not speculate around the dependent load when an untrusted load tries to access the sensitive data during speculation.

```
//Assume: x8 == x9 (untrusted input)
// this may cause memory ambiguity.
1: char *secret = "password"; //assume x5 contain the secret pointer
2 : irg  x5, x5 //set the pointer tag with a random tag (e.g., 0xf)
3: stg  x5, [x5]  // set the tag for memory location that secret pointer point to
...
10: str 42, [x10 + x8] // normal store that has unset tag (e.g., 0x0)
11: ldr x11, [x10 + x9] //normal load with unset tag (e.g., 0x0):
                       //assume this load specutavly bypass the dependent str
```

Figure 5.6: SPECASAN against Spectre V4

To make this more clear, consider the example given in listing 5.6. Suppose the register x5 holds the pointer directed to the secret value we aim to shield from leakage during speculative execution. Lines 2 and 3 create a random tag (for simplicity, let's assume it's 0xa) and assign this tag to both the secret pointer (x5) and the corresponding memory location (as depicted in line 3). Now let us assume that the attacker will craft x9 value in such a way that $x10 + x9$ points to the memory location that contains the secret value. If the predictor predicts that this load instruction(line 11) is not dependent on the previous store(line 10) and execute it speculatively ahead of store, Since this pointer $(x10 + x9)$ has the tag 0x0 but secret memory location has tag **0xa**, the CPU will not allow this load to access the secret and bring its value into the cache. As detailed in subsection 5.3.2, since the LSQs entry includes the tag whenever a new entry is established in this buffer, during data forwarding from store to load, the tag undergoes a check. Consequently, SpecASAN effectively thwarts this attack.

Meltdown_type attack (Meltdown, MDS and LVI)

As previously discussed in chapter 2, all variants of the original Meltdown attacks [35], including Foreshadow [240], LVI [33], as well as the recent MDS [244] attacks such as Zombieload [211] and Fallout [178], share certain attack steps. They all aim to leak data from various micro-architectural buffers, including the L1 DCache, LFB, and LSQ. By applying MTE principles and checking the tags during speculative execution, SpecASAN can mitigates these attacks.

In our third example, we delve into how SpecASAN can be leveraged to defend against MDS attack variants that target the LFB. Contrary to Spectre and the original Meltdown attacks, which aim to speculatively load sensitive data into the cache and subsequently extract the data from there, these MDS variants focus on drawing information from the Line Fill Buffer (LFB).

Whenever the CPU processes a non-blocked load (i.e., a load that doesn't conflict with older stores), it first accesses the L1D cache. If the data is found within the L1D cache (a cache "hit"), the data is immediately fetched from there. If, however, there's a miss in the L1D cache, the next step is to check the Line Fill Buffer (LFB). Should there be a hit within the LFB, the required data is sourced from this buffer. On the other hand, if the data isn't found within the LFB either (another miss), a new entry is allocated within this buffer. One of the primary purposes of the LFB is to ensure that subsequent, younger loads can be serviced without delay, thereby enhancing overall performance.

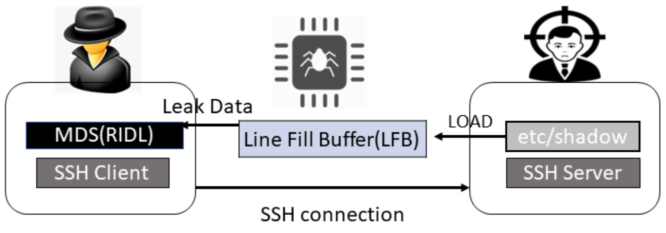


Figure 5.7: MDS-LFB Attack Flow

In the variant of the MDS attack targeting the LFB, the attacker's initial step is to force the victim's sensitive data into the LFB. Once loaded, the attacker then tries to extract or leak that data which transiently held in this buffer before being either committed to the L1D cache or replaced by other incoming data. In Figure 5.7, the attacker aims to keep the content of `etc/shadow` in-flight. This is achieved by repeatedly establishing ssh connections, increasing the likelihood of the content being loaded into the LFB. Subsequently, the attacker executes a code consisting of dependent loads that are expected to miss in the cache. The objective behind this is to capitalize on these misses and thereby increase the chances of leaking data directly from the LFB.

To prevent this MDS attack, one potential solution is to designate all load instructions as blocked loads (no entries in LFB and wait until actual data arrives). However, this approach can lead to a notable decline in performance. SpecASAN offers a more nuanced approach by differentiating between loads accessing sensitive data (tagged) and non-sensitive data. As described in 5.3.2, since the LFB entries contain tags, whenever data is set to be forwarded from this buffer, the requesting instruction's tag (here the faulty load) will be compared against the stored tag. Given that the attacker is unaware of the random tag assigned to sensitive data (`shadow` file), SpecASAN will prevents them from successfully extracting data from this buffer.

Another potential approach involves making every load instruction which has its tag set, as a blocked load in the load queue and so its corresponding data will not be in-flight. As a result, the attacker won't be able to leak sensitive data through LFB. While there is a potential for performance degradation due to these blocked loads, the impact is anticipated to be minimal since only a minor fraction of data in a typical code base is designated as sensitive.

5.4.1 Performance Evaluation and Future work

Our preliminary investigations suggest that the integration of SpecASAN into the pipeline results in a justifiable performance overhead. This is particularly notable given that not all sections of the code require tagging, but rather specific portions that are deemed sensitive or potentially vulnerable. As a next step, we are looking to solidify our understanding of this performance trade-off. To this end, we plans to execute the CPU-

17 benchmark on SpecASAN system. Through this, we aim to provide a more granular breakdown of the overhead.

Chapter 6

SafeSpec: Leakage-Free Speculation

6.1 Introduction

Speculative execution is a standard microarchitectural technique used in virtually all modern CPUs to improve performance. The recent Meltdown and Spectre attacks [95, 145, 148, 157, 167, 168, 240, 259] (we call this class of attacks *speculation attacks*) have shown that speculation can be exploited to expose information that is otherwise inaccessible. Several attack variations have been demonstrated, including arbitrary exposure of the full memory of other processes, OS kernel, hypervisor, and even SGX enclaves [46, 240] to an unprivileged attacker, making this a dangerous open attack vector on modern systems. We describe these attacks and present our threat model in Section 6.2.

Although a number of defenses and software patches have been proposed to mitigate Spectre and Meltdown [84,239], they often address only one aspect of the attack, leaving attackers with other possible variations that are still available. In addition, these patches often lead to high overheads: 10-30% reported on average, but often much higher. For example, Netflix reported 800% slowdown with the Meltdown patches on their systems [81,238]. Most of the solutions target a subset of the threat models and make assumptions that can be broken by future architectures.

In this work, we explore whether speculation can be made leakage free in a principled way, enabling CPUs to retain the performance advantages of speculation while removing the security vulnerabilities that speculation exposes. To this end, we introduce *SafeSpec*, a design principle where speculative state is stored in temporary structures that are not accessible by committed instructions. As instructions transition from being speculative to commitable, any speculative state is moved to the permanent structures. On the other hand, if a speculative instruction is squashed, the speculative side effects are canceled in place leaving no measurable side effects in the permanent structures and closing the vulnerability exploited by speculation attacks. We consider two variants that differ in when an instruction is considered safe to commit. *SafeSpec* makes no assumptions on the branch predictor behavior or on speculative execution behavior; for example, it does not prevent the attackers from mis-training or even polluting the branch predictor, nor does it prevent them from speculatively reading privileged data. Rather, SafeSpec interferes with the attacker's ability to create a covert channel using speculative data accesses to communicate illegally-accessed data out. We describe *SafeSpec* in Section 6.3.

We demonstrate the *SafeSpec* principle by building a memory hierarchy (caches and TLBs) that are free from speculation-induced leakage. In particular, we expand the load-store queues to store a pointer to a temporary associative structure that holds speculatively loaded cache lines. We also introduce a similar structure to hold speculatively loaded translation lookaside buffer (TLB) entries. We describe the design and some of the complexity-performance trade-offs in Section 6.4.

Additionally, we identify a transient type of leakage that occurs in the introduced speculative state (byproduct of *SafeSpec*) that we call *transient speculation attacks* (TSAs). We explore how to construct the shadow structures to mitigate TSAs in Section 6.5. Furthermore, Section 6.6 presents a performance, complexity and security analysis of *SafeSpec*. We also analyze the complexity of *SafeSpec* including the impact of all new structures, and demonstrate a reasonable increase in the area and power consumption. Finally, we show that *SafeSpec* stops proof-of-concept implementations of all variants of Meltdown and Spectre, as well as the new variants that we introduced.

In summary, the chapter makes the following contributions:

- We introduce the *SafeSpec* model to protect speculation by isolating speculative state from committed state.
- We identify a new class of speculative attacks (Transient Speculation Attacks) that arises in *SafeSpec*. We mitigate such attacks by sizing the shadow structures to prevent contention.
- We evaluate *SafeSpec* for caches and TLBs from a security, performance and complexity perspective.

6.2 Background and Threat Model

Speculation attacks such as Spectre, Meltdown, and their subsequent variants, exploit the fact that permissions are not checked while instructions (or subsets of instructions in the case of Meltdown) are being executed speculatively. Conventional wisdom was that this microarchitecture assumption, which allows aggressive and performance-beneficial speculation, was not dangerous since the effects are simply undone once misspeculation is discovered (or once an exception is raised in the case of Meltdown). The attacks showed that, the secret values that are speculatively read can be communicated through a side channel opening this dangerous and previously unknown class of vulnerabilities.

Spectre and Meltdown attacks differ only in how they trigger speculation. Meltdown attacks exploit speculation within a single instruction that will eventually fail due to permission checks, or processor faults. Before they fail, illegal accesses are executed speculatively and communicated through the side channel. In contrast, Spectre attacks manipulate the branch prediction structures to cause the speculative execution of code that will read the secret data and communicate it.

```
if (offset < array1_size)
    y = array2[array1[offset] * 64];
```

The code snippet above demonstrates Spectre variant 1 of the attack. In this code, the attacker mistrains the branch in the if statement to be always taken. To launch the attack, the code is executed with a large offset, that makes the access to array1 read into the kernel address space. This access is performed speculatively since the branch has been trained to be predicted taken. Then, resulting value is used to perform an access into `array2`. As we discussed above, accesses into the `array2` leaving a footprint in the cache for array2 that can be detected by the attacker (using a standard side channel attack such as Flush and Reload [87].

Given the large number of variants that have been discovered, it is unlikely that simple defenses that target each variant individually would provide principled protection from this class of attacks. SafeSpec is general and applicable to different micro-architectural structures. However, as a demonstration, our prototype implementation only protects caches and TLBs to explore concretely the implications and complications that result from *SafeSpec*. Therefore, we further assume that other covert channels, including the ones through the branch predictor, memory bus and DRAM buffers are out-of-scope for the current work, but will be addressed using similar principles by future work. Similarly, we only consider a system with a single core. Thus, speculation attacks against the cache coherence and memory consistency model states [237] are also left for future work.

6.3 ***SafeSpec*: Leakage-free Speculation**

SafeSpec is a principled approach to secure processors against speculation attacks while retaining the ability to carry out speculative execution to benefit from its performance. The general principle (shown in Figure 6.1) addresses the problem at the root by introducing shadow state to separate state that is produced speculatively without affecting the primary structures of the processor (which we call committed state). For example, if a speculative load instruction causes a load of a cache line, instead of loading that cache line

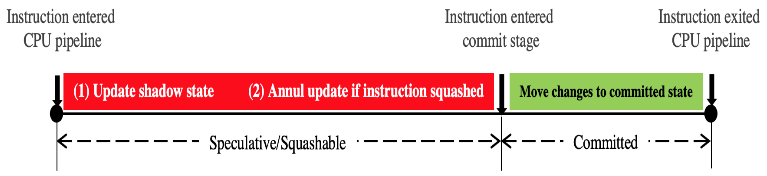

Instruction entered CPU pipeline
Instruction entered commit stage
Instruction exited CPU pipeline
(1) Update shadow state
(2) Annul update if instruction squashed
Move changes to committed state
Speculative/Squashable
Committed

Figure 6.1: SafeSpec overview

into the processor caches, we hold the line in a temporary structure. If the load instruction is later squashed, these effects are removed in place, leaving no changes to the cache from the misspeculated instructions, and closing the vulnerability. Alternatively, if the instruction commits, the cache line is moved from the temporary structure to the L1 cache. While is simple in principle, a number of questions relating to its security, complexity and performance have to be resolved.

When to move state from speculative to committed. There are two options available to decide when to move state from the shadow to the committed state. In the first variation, which we call *wait-for-branch* (WFB), we can assume an instruction to be no longer speculative when all the branches (more generally, all predictions) it is dependent on have been resolved. WFB stops all variants of spectre which depend on mistraining the branch predictor/return stack buffer; none of the mis-speculated instructions moves to the committed state. However, it does not prevent Meltdown which relies on speculation within a single instruction.The second variation *wait-for-commit* (WFC) waits until the instruction commits before moving its effects to the committed state, and therefore also prevents Meltdown.

Shadow state organization and size: If the shadow state structures are too small, then either speculative state is replaced (causing a loss of an update to the committed state if this data were to be committed later), or the instruction has to stall until there is room in the speculative structure before it issues. From a performance perspective, the organization and size of the shadow structure should be designed such that the structures can hold the speculative state generated by speculation as measured across typical workloads. However, we will show that security considerations introduce more stringent requirements on the speculative state.

Mitigating Transient Speculation Attacks: *SafeSpec* by construction prevents speculative values from affecting the state of committed structures, which is the pathway used to communicate data covertly in the published speculation attacks. However, it does not create isolation between instructions that are in the speculative state. This creates a possibility for a new variant of attacks which we call *transient speculation attacks (TSAs).* In particular, since most instructions that commit start in the speculative state, there is a window of time where they can share the speculative state with misspeculated instructions before they are squashed. If we are not careful, it is possible to create a covert channel in this period to communicate the sensitive data from the mis-speculated branch to the branch that will be committed, allowing the data to be exfiltrated. The attack is illustrated in Figure 6.2 and we discuss how to mitigate TSA attacks in Section 6.5.

Filtering Delayed Side Effects: One of the issues with *SafeSpec* occurs when an instruction is squashed in the middle of its execution. If the instruction has already initiated a high latency operation such as a read from memory, we have to ensure that the response from memory can be discarded after it is received. We handle this situation by discarding values received if there is no matching transaction. However, it may also be desirable to filter these transactions lower in the system, such that the committed transactions commit

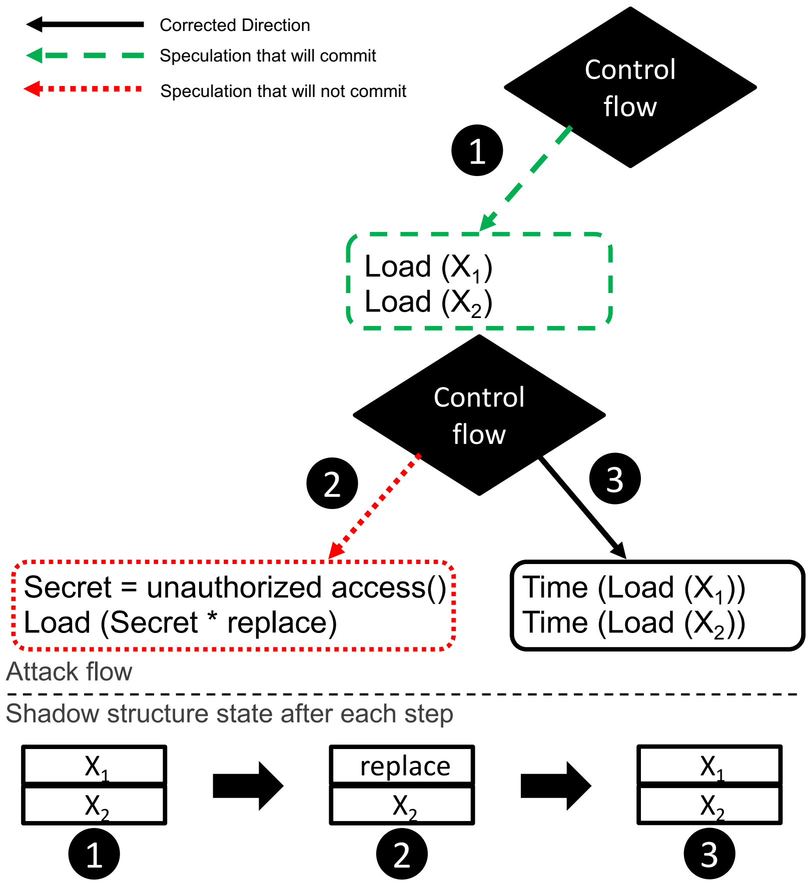


Figure 6.2: Transient speculation attack (TSA)

directly, and the squashed ones are cancelled in place. To control the size of this filter, we include a branch id with the transactions and track operations at the branch granularity. The filter can also be used to mark committed branches so that memory responses corresponding to them are committed directly.

6.4 *SafeSpec* for Caches and TLBs

To demonstrate the *SafeSpec* principle, we implemented it to protect CPU caches and TLBs from leakage during speculative execution. To provide full protection, all speculatively updated structures should follow the SafeSpec principle. We chose the CPU caches because they are easily exploitable targets for covert communication and the ones used in the Spectre/Meltdown attacks.

To protect from speculative covert channels that occur during memory accesses, and following the *SafeSpec* principles, we need to add shadow state to protect the following structures.

Data caches: this is the covert channel used in all three Meltdown/Spectre variants. We add a shadow structure to hold the cache lines that have been fetched speculatively. The structure is associatively-filled lookup table (filled associatively, but accessed as a lookup-table). In the Load/Store queue, we point speculative loads that have received their data to a corresponding entry in this table. Speculative instructions in the *same execution branch as the load that fetched a shadow cache line* that accesses this cache line can use the value from the shadow structure. If an instruction commits (depending on WFB or WFC), the cache line is moved from the shadow structure to the caches. If the instruction is squashed, the shadow structure entry is marked as available. Thus, not even the cache replacement algorithm state is affected by the speculative data that does not commit.

Instruction caches: we built variants of Meltdown/spectre using the instruction cache that replaces data dependent array access with dependent branches to a location in an array to disclose the data through the i-cache, illustrating that it must be protected as well [136]. To develop this attack variant, we had to overcome branch predictor behavior: data dependent branches were using the branch predictor, rather than the secret data. Thus, we had to initialize the branch target buffer (BTB) to a third location, and then introduce sufficient delay in the pipeline for the data dependent branch such that it has time to register the data dependent location in the i-cache.

TLBs: we also conjectured that the TLBs may be used as a covert channel vector. Given recent attacks such as Foreshadow [240] and TLBleed [80] which directly target the page translation behavior for speculation attacks, its critical to protect these structures.

To implement *SafeSpec* for the data cache, we add an associatively-filled lookup table to hold speculatively read cache lines. It is important to note that memory consistency models, such as Total Store Order (TSO) semantics of the x86-64, often ensure that store side-effects appear in order; in other words, the cache is not updated until the store commits, making stores robust to speculation attacks. We augment the load store queue with a pointer to the shadow cache line for load operations that are speculative. Any instruction dependent on the speculative load reads the cache line from the shadow structure. Once the load instruction commits, the shadow cache line is written to the caches according to the inclusion policy of the caches (in our case, since the caches are inclusive, it is written to

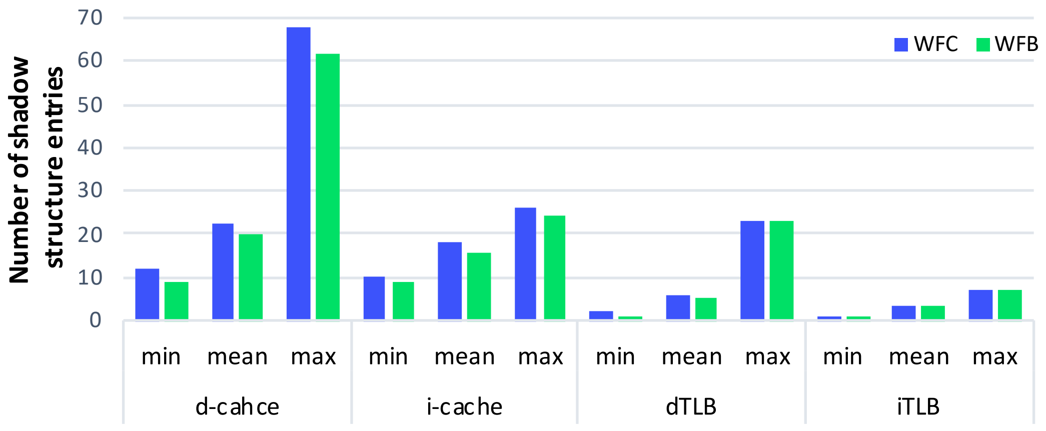


Figure 6.3: Shadow structure size that fits 99.99% of accesses

all levels of the cache) and freed in the shadow structure. If the load is squashed, the value is freed in the shadow structure. For the i-cache and the TLBs, we create similar shadow structures, and augment the reorder buffer (ROB) with pointers to the shadow state entries if the instruction is speculative and the cache line (or TLB entry) were fetched speculatively.

From a performance perspective, the structures should be sized such that they accommodate the speculative state needed by representative workloads. If the shadow structures are full, we could either drop some of the shadow state (leading to loss of updates to the committed state with performance, rather than correctness implications), or block until there is space in the shadow state before issuing an instruction (also with performance implications). We will see later that the constraints introduced by security requirements to eliminate TSAs are more stringent than those required by performance. Figures 6.3 show the distribution of the size of the speculative state sampled over time for the SPEC 2017 benchmarks. The shadow d-cache for 3 of our benchmarks grows occasionally to almost the maximum possible size (bound by the size of the load-store queue). A shadow i-cache with about 25 cache lines is sufficient for all of the benchmarks. In addition, less than 10 entries are sufficient for speculative iTLB misses, but some benchmarks require more dTLB entries (up to 25). Given that the overhead of supporting WFC is small, we elect to support WFC to get the increased protection to cover Meltdown.

6.5 Transient Speculation Attacks

The *SafeSpec* principle prevents direct side-channel leakage from the speculative state to the committed state, closing all known speculation attacks. However, although the committed instructions and the speculative instructions eventually reside in separate structures, creating the separation and closing the channel, eventually committed instructions can start out as speculative. During this window, the eventually committed instructions share the shadow state with any speculative instructions that will be squashed. If the shadow structures are not designed carefully, covert channels can be created during this transient window to communicate sensitive data (which can only be read by a mis-speculated path) to an instruction pathway that will be committed such that the leakage results are visible to the program. It is important to emphasize that these attacks (which we call Transient Speculation Attacks, or TSAs) are substantially more difficult than Spectre/Meltdown because there is only a limited window of speculation in which the malicious Trojan code must not only read sensitive data, but also create measurable contention to the spy before either of their predicate branches commits.

TSAs are possible only if the shadow structures are shared and sized such that they enable contention. Consider an example where we size the TLB shadow structures based on typical program behavior. Since programs do not have many pending TLB misses within a speculation window, it stands to reason to size these structures to be small. In the rare case when the shadow structures are full, we may handle this by either discarding updates or by blocking the issue of requests when there is no room in the shadow structure. Either of these behaviors provides potential for a covert channel. Consider that the Trojan fills the structures with TLB misses if it wants to communicate a 1. If updates are discarded, a spy can detect a communication if its TLB accesses are not committed (they were discarded). Alternatively, if we block TLB accesses when the structures are full, the spy can detect a communication of 1 if its TLB accesses are delayed causing a longer TLB miss time. The attack is illustrated in Figure 6.2.

To prevent TSAs through the shadow structures, we elect to provision them for the worst case scenario to make sure that transient contention cannot be created within a speculation window. This approach guarantees that no contention on the shadow structures is possible, at the cost of provisioning fairly large associative structures. We believe that with some more analysis, or with some detection defense that detects an attack when the shadow structures grow abnormally large, this worst case provisioning can be substantially relaxed without introducing leakage.

Table 6.1: Configuration of the Simulated architecture

Parameter	**Configuration**
CPU	6-way issue, 96 Issue Queue entries, out-of-order,
	no SMT, 72 Load Queue entries, 56 Store Queue entries,
	224 ROB entries, 64 iTLB entries, 64 dTLB entries,
	commit up to 6 Micro-Ops/cycle
Private L1 i-/d-Cache	32 KB, 8-way, 64B line, 4 cycle hit
Shared L2 Cache	256 KB, 4-way, 64B line, 12 cycle hit
Shared L3 Cache	2 MB, 16-way, 64B line, 44 cycle hit

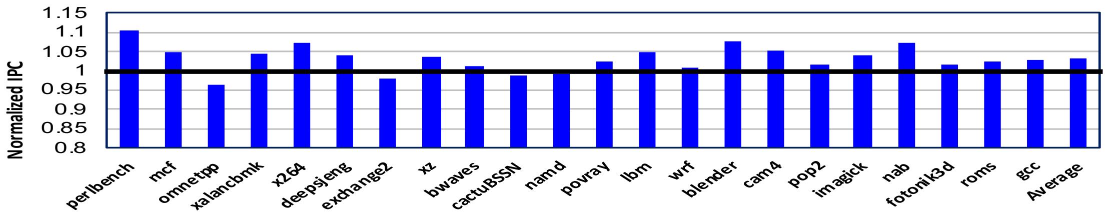


Figure 6.4: Relative performance to non-secure OoO execution

6.6 Evaluation

We conduct experiments with MARSSx86 [193], which is a cycle-accurate full-system simulator of out-of-order x86 cores. We configured the CPU and cache models of MARSSx86 to simulate the Intel Skylake processor as shown in Table 6.1.

6.6.1 Performance Analysis

The first experiment measures the performance of compared to the baseline processor under conservative condition. In particular, we consider the shadow state access time to be equivalent to the access time of the L1 cache (4 cycles), when it is substantially

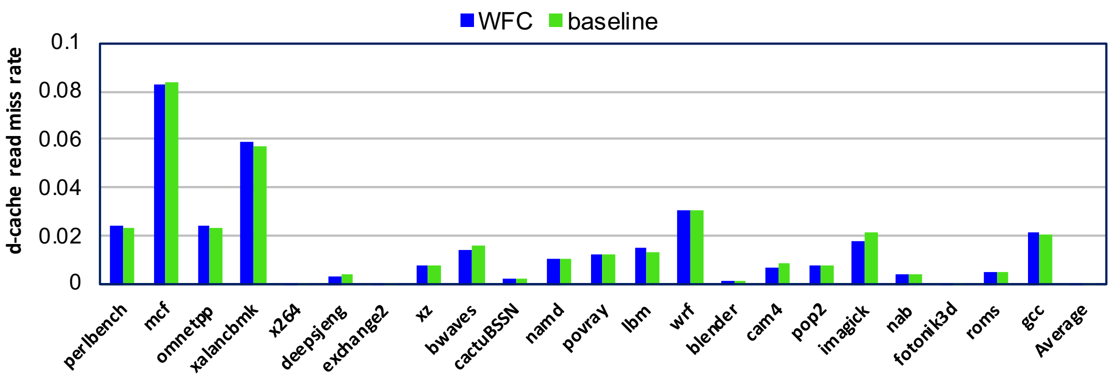


Figure 6.5: d-cache read miss rates including shadow d-cache

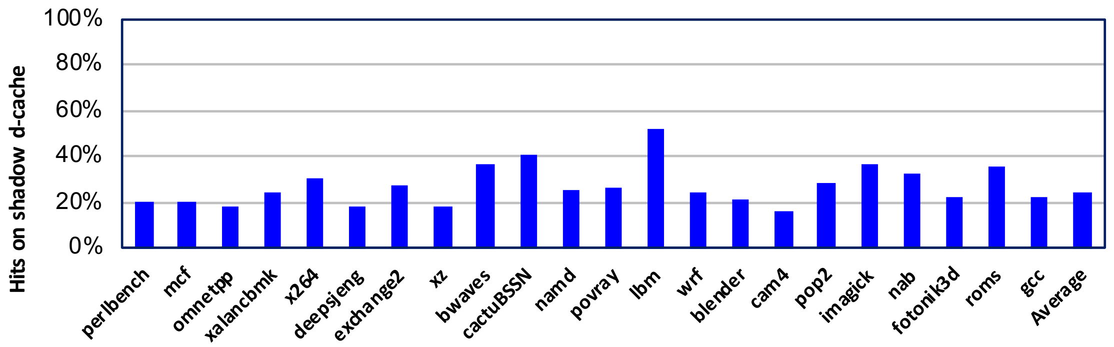


Figure 6.6: Percentage of hits on shadow d-cache

smaller, and accessed as a lookup table. Figure 6.4, shows the IPC values for all SPEC2017 benchmarks. We see a small improvement in performance with a geometric mean of about 3%. We believe that this advantage results from a combination of effects including the larger effective cache size and avoiding polluting the cache with wrong path speculative state.

To gain more insight into the observed performance, Figure 6.5 shows the miss rate on read operations in the d-cache. There is little difference in behavior between SafeSpec and the baseline with respect to the data accesses. Figure 6.6 shows the percentage of the reads that hit the shadow structures.

The i-cache behavior is significantly different than the d-cache. Figure 6.7 shows the miss rate on the i-cache. For the i-cache, there are more substantial differences between WFC and the baseline. Some outlier behavior such as Pop2 and imagick where the percentage of i-cache misses drops significantly could be due to the larger size of the shadow structures expanding the effective size of the cache reducing conflict and capacity misses. Moreover, we see in Figure 6.8 that most of the hits occur in the shadow i-cache structure reflecting the high spatial locality of the access patterns in the i-cache; in other words, while a cache line is still speculative, several instructions execute from the same cache line. In contrast, the d-cache has less spatial locality, resulting in fewer accesses hitting the shadow state. We note that the cache miss rates are combined for all instructions (i.e., we do not exclude instructions that are squashed); therefore, many of these hits in the shadow structures may not end up being productive.

To understand the benefits of the shadow structure in filtering misspeculated accesses, Figure 6.9 shows the percentage of the shadow state that ends up being committed for the i-cache and the d-cache. We observe that a substantially higher percentage of the d-cache state ends up being committed, perhaps due to the fact that speculative loads are issued later in the pipeline making them more likely to commit. For both the d-cache and especially the i-cache, the shadow structure filters a large number of misspeculated accesses that are squashed without cluttering the caches.

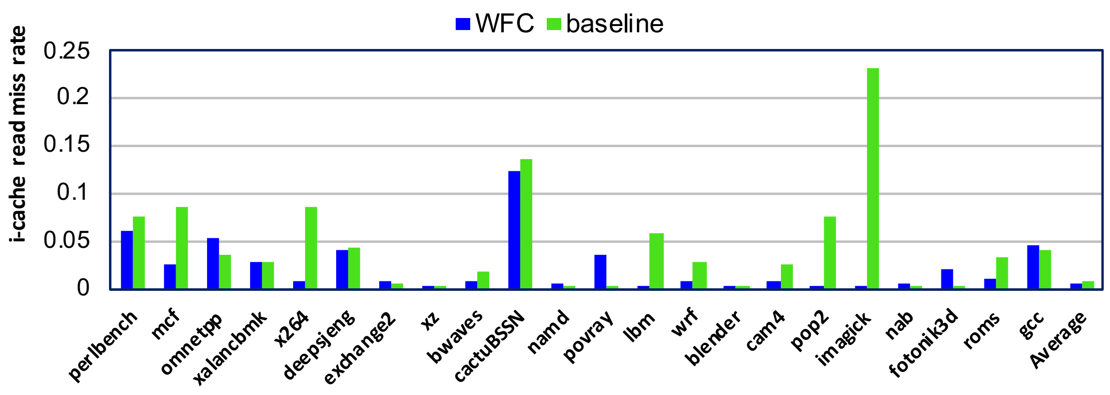


Figure 6.7: i-cache miss rate including the shadow i-cache

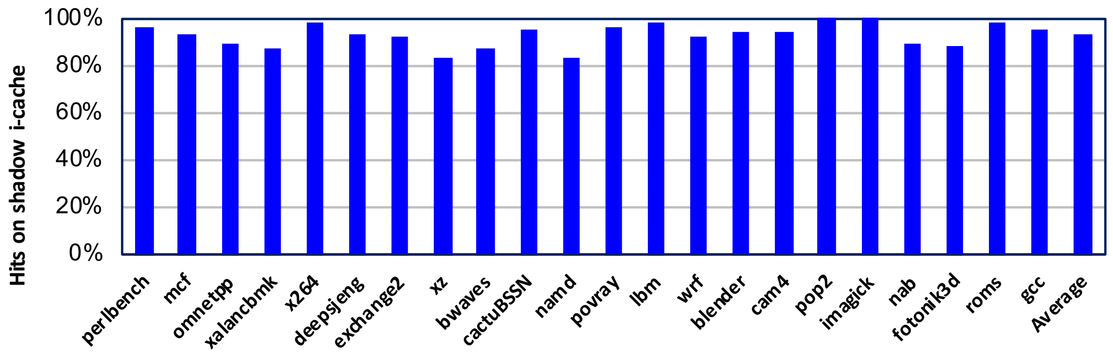


Figure 6.8: Percentage of hits on shadow i-cache

6.6.2 Security Analysis

Table 6.2 shows that both WFC and WFB close Spectre attacks, but only WFC is guaranteed to also stop Meltdown attacks. We evaluated our proof of concept code implementing Spectre in the simulator and found indeed that the attack fails under both WFC and WFB models. We evaluated the protection coverage for Spectre-style attacks targeting structures other than the d-cache (i-cache, iTLB, and dTLB). All three side

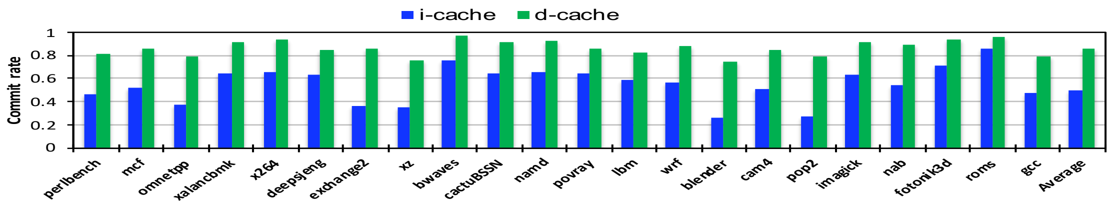


Figure 6.9: Commit rate of shadow state

Table 6.2: Security Analysis of Meltdown/Spectre

Spectre	**WFC**	**WFB**	**Meltdown**	**WFC**	**WFB**
Spectre-PHT [141, 145]	✓	✓	Meltdown [157]	✓	✗
Spectre-BTB [145]	✓	✓	Foreshadow [240, 259]	✓	✗
Spectre-STL [94]	✓	✗	Variant 3a [21]	✓	✗
Spectre-RSB [148, 167]	✓	✓	Lazy FP [229]	✓	✗
			Variant 1.2 [141]	✓	✗

channels were closed. We tested proof of concept code for the i-cache and a transient attack through the d-cache and observed that the attack fails on the *SafeSpec* protected CPU. We could not get TLB-based attacks working in the simulator, perhaps because of the large delays of page walks, or due to the limitations of the MarSSx86 models of the TLBs.

6.6.3 Hardware overhead

introduces hardware overheads to the CPU pipeline due to the addition of the shadow structures. We compared the hardware overhead for two different sizes for the shadow structures; 1) Secure: shadow structure size equal to the maximum speculative state during speculation; and 2) SafeSpec with WFC: shadow structure sizes were optimized based on 99.99% speculative state size for SPEC2017 benchmarks using the WFC implementation.

Table 6.3: SafeSpec hardware overhead at 40nm.

	Power (mW)	**Power** (%)	**Area** (mm^2)	**Area** (%)
Secure	290.27	26.4	9.79	17
WFC	35.14	3	1.17	2

We report the area, power, and access time values, as well as a percentage compared to the Skylake CPU L1 cache configuration (shown in Table 6.1), using CACTI v5.3 [224] in Table 6.3. The results show that the area overhead is tolerable for the secure design, making the design highly practical.

6.7 Discussion, Limitations and Future Directions

SafeSpec is a principled approach for protecting systems from speculation attacks by preventing crossover leakage from speculative instructions that will eventually be squashed to permanent structures where they could be visible to attackers through a side channel. By preventing this leakage, we close the covert channel that is exploited by recent speculation attacks such as Meltdown and Spectre. This general principle should be applied to all speculatively modified state within a CPU.

SafeSpec requires a deep redesign of the CPU to separate out the speculative state from the permanent state. It also has implications on security: we identified a form of transient side channels that occur through the shadow structures. The goal of this work is to establish the *SafeSpec* principle by protecting the CPU caches and TLBs. We recognize that other structures affected by speculative instructions must also be protected using this principle or otherwise the attackers will switch to using them. Future work should look at protecting the branch predictor, DRAM buffers, account for prefetchers, as well as other structures.

Another limitation of the current work is that we do not support multi-threaded workloads. Addressing this limitation involves two considerations. The more straightforward consideration is how to preserve the semantics of protocols such as cache coherence, memory consistency models, atomic operations, and transactional memory. We believe that these continue to operate in the same way by treating the speculative state to be part of the state of the caches. The second issue is significantly more difficult: these protocols themselves can be used to communicate speculative side-effects as has been recently shown by the MeltdownPrime attack [237]. Designing leakage-free protocols is a difficult problem that deserves separate and complete treatment and therefore we elected to leave supporting multi-threaded workloads to future work.

We identified the problem of transient covert channels that occur while instructions that will eventually commit share the shadow structures with speculative instructions that will not. In the window while both set of instructions are speculative, they share the shadow state creating the potential for covert communication. To prevent covert communication, one approach is to size the shadow structures for the worst case contention level and make them fully associative. This worst case size is bound by the size of the load-store queue for the d-cache and dTLB, or the size of the reorder buffer for the i-cache and the iTLBs. While this pessimistic approach guarantees no potential for leakage, more careful analysis can show that a much smaller size will suffice given the transient nature of the exposure.

We also characterized the size of the shadow state created by normal program execution and showed that it is substantially smaller than the worst case. Thus, we expected these large shadow structures to be mostly unused providing opportunities for dynamically resizing them for energy efficiency. In addition, it is possible to use abnormal growth of the structures as an indicator of a possible attack and introduce mitigations to stop the attacks. This can also be explored in future work.

Speculation attacks challenge the foundation of out-of-order microarchitectures which have been the key building blocks of computer systems in the last several decades. Since these attacks are very new and most of the proposed defenses are at the software/-firmware levels, CPU manufacturers and microarchitecture researchers face an open challenge of how to redesign speculative out-of-order processors to be immune to speculation attacks. This work represents a first step in this direction that we hope will spur future research in this area.

6.8 Concluding Remarks

We presented a general principle for supporting speculative execution in a way that makes out-of-order processors immune to speculation-based attacks. The principle relies on leaving speculative state in shadow structures, and only committing this state once the instructions that generate them are guaranteed to commit. Thus, side-effects of mis-speculation are hidden from the primary structures of the CPU, closing the vulnerability. We demonstrated the principle to protecting caches and TLBs of the CPU. Our design completely closes all published attacks, as well as new variants that we developed to leak through the i-cache or the TLBs. We showed that careful design is needed to prevent a form of leakage that can arise while instructions share the speculative state. We mitigate this leakage by sizing the speculative state conservatively. Constructed this way, transient attacks also become impractical. The performance of the *SafeSpec* CPU was actually slightly higher than an unmodified CPU, despite conservative estimates on the shadow state. We believe that the presented design represents a first step of many towards a principled protection of speculative execution. To provide complete protection, other microarchitectural states that can be updated speculatively should use the same principle.

Chapter 7

Intel Optane Side-channel Analysis

7.1 Introduction

The advancement in data-intensive and high performance computing, e.g. Large scale machine learning and large-scale graph analytics workloads, has increased the demands for more efficient and scalable memory systems. As a result, the cost of memory (DRAM) is becoming an important concern in data centers and other high performance computing facilities dealing with large scale data analysis. Besides the cost of the memory, keeping the leakage power in tolerable limits and bridging the bandwidth gap between processor and memory are two major challenges that need to be addressed in new memory designs. Several promising Non-Volatile Memories (NVMs) such as, Spin-Transfer Torque RAM (STTRAM) [152], Phase Change Memory (PCM) [263], Resistive RAM (ReRAM) [262], and Ferroelectric RAM (FeRAM) [214] are being studied to address the mentioned challenges and offer high density and zero leakage power. Emerging NVMs memory can be integrated in different levels of memory hierarchy from caches to main memory. Due to promising aspects, emerging NVMs are already being commercialized by industries e.g., Everspin (MRAM) [7], Adesto (ReRAM) [6], Intel/Micron (PCM) [3], and Cypress (FeRAM) [8]. Intel's 3D Xpoint memory [3] is a recent example of NVM's adoption as a cache for Solid State Drives.

The unique characteristics of these emerging memories may lead to security and privacy issues that need to be investigated. Most of these new emerging memories have high write current that can be potentially exploited to launch fault injection [134] attack, Denial-of-Service (DoS) attack, information leakage attack, and Rowhammer attack (e.g. Rowhammer attack on STTRAM [133]). The other common vulnerability among these types of memory is asymmetric read/write current which could potentially lead to side channel attack and more specifically power analysis side channel attack [43, 119].

Other potential vulnerabilities in these types of memory include the timing side channel attacks [**?**]. In order to improve the efficiency of reads and writes in NVM memories, many architecture-level performance optimizations have been studied which can lead to timing side channel attack. For example, [98] shows that accessing different regions of Multi-Level Cell (MLC) PMC have different latency which may lead to a timing side channel attack. Also, when NVM memory is placed in memory hierarchy and interacting with other parts of the memory system, it can potentially create a time side channel attack.

In this chapter, we investigate the potential for timing covert and side channels in Intel's new NVMe memory, known as Intel Optane. When Optane operates in memory mode, the system's original DRAM serves as the Last Level Cache (LLC), while Optane assumes the role of the system's main memory. In summary, the contributions of the chapter include:

- We present our reverse engineering result for Intel Optane memory and discuss the Cache properties and components on DRAM cache and Intel Optane memory controller.
- We present different timing covert and side channel attacks based on the level of memory hirarcy which data will access as well as cache line state in the DRAM cache.

7.2 Background

Intel Optane memory is a transformative technology, bridging the gap between traditional storage and main memory. Its technical configuration offers two primary operational modes:

`APP Direct Mode:` In this setup, Optane acts as byte-addressable persistent memory. This allows systems to engage with Optane just as they would with RAM, at the byte level. However, unlike traditional RAM, Optane retains this data even after power shutdown. This combination brings together the rapid access speeds of RAM with the persistence of storage drives.

`Memory Mode:` nder this mode, Optane serves as the system's primary memory. In parallel, the system's existing DRAM functions as the Last Level Cache (LLC). This structural change maximizes the system's ability to access data, considerably speeding up the processing of extensive datasets.

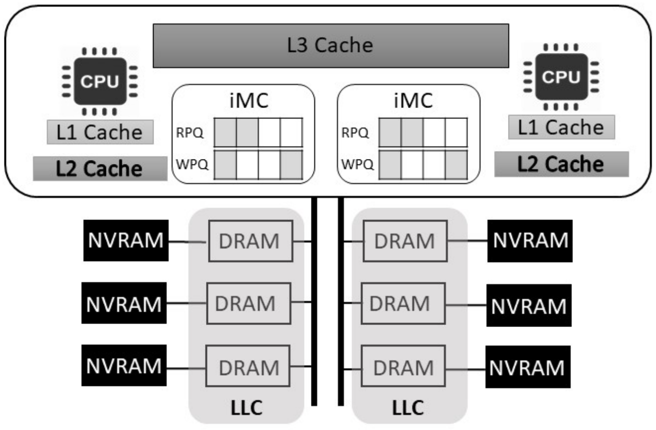


Figure 7.1: Optane Memory-Mode Architecture Overview

One of Optane's major advantages is its cost-effectiveness for enhancing memory capacity. Systems can now have more memory without the typical high costs associated with large DRAM installations. Recognizing these benefits, industry giants like VMware and Microsoft Azure have integrated Intel Optane into their systems.

A standout advantage of Intel Optane is its capacity expansion. Systems equipped with Optane can handle more data in memory, leading to faster processing times, especially beneficial for tasks involving large datasets. This enhancement in memory accessibility and processing speed, without the prohibitive expenses traditionally associated with significant DRAM expansions, makes Optane a sought-after choice. A testament to its efficacy and potential is its adoption by major industry players like VMware and Microsoft Azure.

In the remainder of this chapter, we will concentrate on the unique properties of Optane memory when it functions as the main system memory. Figure 7.1 illustrates Optane's setup in this mode. As highlighted earlier, in this configuration, the conventional DRAM assumes the role of the Last Level Cache (LLC). When the CPU needs to access a piece of data, it first checks the DRAM. If the required data is found there (a cache hit), it's accessed swiftly. If not (a cache miss), the system fetches the data from the Optane memory. Once fetched, this data might also be cached to the DRAM for quicker subsequent accesses. Since DRAM in this mode acts as a cache, the data within it is transient. It's worth noting that in Memory Mode, the persistence property of Optane is not utilized. Data in both the DRAM and Optane is treated as volatile, meaning all will be lost in case of a power outage or system reboot.

While Optane is fast compared to traditional storage solutions, DRAM is still quicker. Hence, using DRAM as a cache ensures that frequently accessed data can be available at the fastest speeds possible. Additionally, Optane's robust endurance ensures that even with frequent read/write cycles, its longevity isn't easily compromised.

7.3 Reverse Engineering the Intel Optane in Memory-mode

In this section, we detail our findings on Intel Optane's memory-mode on both DRAM cache and Optane memory controller, derived from our reverse engineering efforts and available resources. We delve into its cache properties, internal buffers, prefetcher units.The various components of the DRAM cache, which we will elaborate on in the subsequent sections, are shown in Figure 7.2.

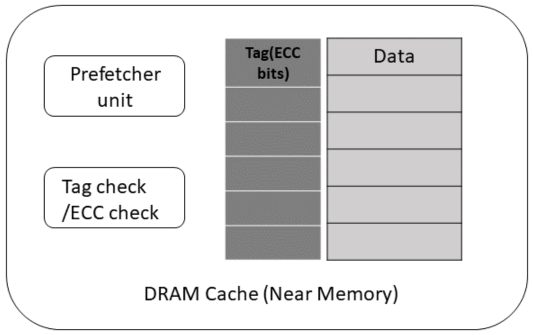


Figure 7.2: Optane DRAM cache

7.3.1 Properties of DRAM Cache (Near memory)

In this unique memory configuration, the DRAM and Intel Optane have distinctive roles, functioning in a tiered memory hierarchy. The DRAM, given its faster access times and proximity to the CPU, is deemed the *near memory.* Its primary task is to serve as a cache, absorbing frequent memory requests, especially those that do not find a match in the Last Level Cache (LLC) of the CPU. When a memory request is made and it misses the LLC, the system's next immediate lookup is in this *near memory* DRAM cache.

Our reverse engineering reveals that the DRAM cache, which operates as a direct mapped cache, acts as a memory-side cache for Optane. Whenever a memory request misses in the CPU's Level 3 cache, it first searches the DRAM cache. In this configuration, the DRAM cache operates as a `direct mapped write-back` cache, with an access granularity set at 64 bytes. So, whenever two cache lines map to the same cache entry, the older line is evicted and written to the Optane memory, while the new line takes its place.

To store the tags for each cache line, the DRAM cache repurposes the Error Correction Code (ECC) generator/checker units. Traditionally, these units are tasked with generating and verifying ECCs to ensure error-free data reading and writing to/from DRAM. Every DIMM in this system incorporates DRAM ECC by adding a supplementary DRAM module. This provides 8 bytes of ECC for every 64 bytes of data. Given that DRAM utilizes only 20 of the 64 available ECC bits for error correction, there remains ample space. This surplus is used to store a segment of the Physical address as a tag, in addition to the requisite metadata. Table 7.1 summarize the DRAM cache properties.

Table 7.1: Properties of DRAM when acting as cache for Optane memory

Cache Size	**Number of Sets**	**Number of Ways**	**Line Size**	**Write Policy**
Available DRAM Size	Direct Mapped Cache	1	64 bytes	Write Back

DRAM Cache Addressing and Cache size

Within Intel Optane's memory mode configuration, the addressing procedure for the DRAM cache is directly derived from the system's physical address. Below, we outline the specifics of this addressing mechanism:

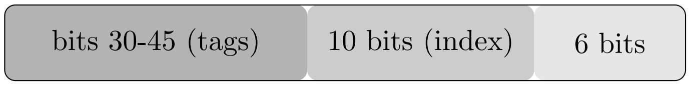
1. **Offset to Cache Line:** The lowest six bits of the address act as the offset for the cache line.

2. **Cache Line Location:** Depending on the DRAM cache's size and its ratio to the overall Optane memory, a specific portion of the physical address is employed as the index to pinpoint the cache line's exact location.

3. **Tag and Metadata Storage:** The remaining address bits are repurposed within the ECC bits, storing both the tag and the relevant metadata for the corresponding cache line.

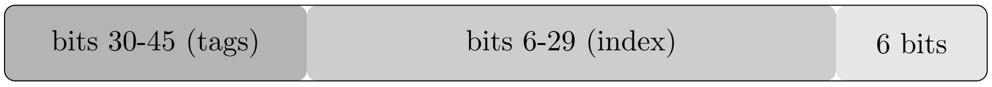
To provide a clearer understanding, consider the following address configurations:

- **32-bit Address System:**



This configuration results in a cache size of $2^{10} \times 2^{6}$ bytes, which equates to 64 KB.

- **46-bit Address System:**



Leading to a cache size of $2^{23} \times 2^{6}$ bytes, or 512 MB.

Prefetcher

Our analysis unveils a sophisticated prefetching mechanism at work within the DRAM cache. Upon detecting a cache miss in the DRAM cache, the prefetcher activates. Instead of merely fetching the single missing cache line, the prefetcher proactively retrieves a larger chunk of data. Specifically, it fetches 256 bytes, equivalent to 4 cache lines, from the Optane memory.

7.3.2 Optane controller

The Intel Optane memory controller, when configured in memory mode, serves as a bridge between the memory system and the underlying 3D XPoint technology. Here's a more in-depth look into its components and operations in this mode:

Data Cache: In memory mode, the data cache within the Optane memory controller primarily holds frequently accessed data blocks. Due to the inherent latency differences between DRAM and 3D XPoint, this cache becomes essential for faster data retrievals, especially for commonly accessed data.

Persistent Write Buffer: The persistent write buffer in memory mode serves as an intermediary zone for incoming write operations. Due to the write latency of 3D XPoint, especially for small, random writes, data is initially staged in this buffer. It aggregates multiple writes until an optimal batch size is achieved, usually in the order of a few kilobytes to several megabytes, to ensure efficient and sequential writes to the 3D XPoint cells.

Address Indirection Table: This table is pivotal for the efficient functioning of the Optane in memory mode. It manages wear-leveling by dynamically mapping logical addresses, as perceived by the system, to the physical addresses within the Optane module. As data gets written and re-written, the address indirection table ensures it is spread across the 3D XPoint cells. Such dynamic remapping, which remains transparent to both the OS and the applications, guarantees that no specific cell endures excessive wear, extending the module's operational lifespan.

7.4 Timing Channels

In this section, we introduce potential timing-based covert and side channels inspired by the results outlined in section 7.3. We first explore a timing channel that exploits the access time difference across various levels of the Optane memory hierarchy, as detailed in subsection 7.4.1. Subsequently, we discuss another channel inherent to Optane's memory-mode, focusing on the nature of access within its memory hierarchy subsection 7.4.2. For each identified channel, we will present both covert and side channel attacks.

7.4.1 Timing channel 1

In the first scenario, attacker tries to exploit the timing difference between accessing data in near memory (DRAM) and accessing data in Optane memory which serves as the system memory. We assume that the CPU caches (L1, L2, L3) are not vulnerable to timing channels and we use the DRAM cache to leak the data. If a cache line is not found in any level of the CPU caches, the system then checks the DRAM cache. If there's a hit in the DRAM cache, the data is retrieved from there. However, if there's a miss, the request proceeds to the Optane memory controller. Should the desired cache line be located in the Optane side cache, the data is relayed back to the DRAM cache. If not, it is fetched directly from the Optane memory (Far memory) Figure 7.3.

Our findings indicate a notable disparity in access times across the memory hierarchy. Specifically, if a data access in the DRAM cache takes X cycles (indicative of a DRAM cache hit), accessing the data from the Optane side cache requires approximately

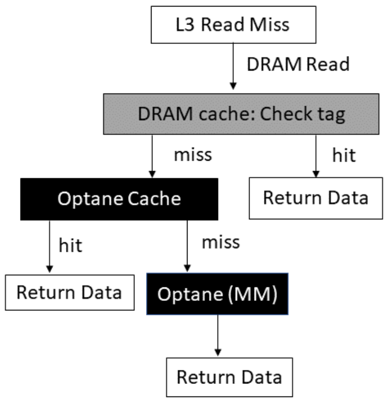


Figure 7.3: Optane Memory Access

1.5 to 2 times X cycles. Further, if the data must be fetched directly from the Optane memory (far memory), the time taken escalates to 3 times X cycles. Given these differences in access times, it's feasible for attackers to exploit these timing differences and establish timing-based covert or side channels.

Covert Channel:

To create a covert/side-channel attack based on this timing differences, an attacker needs to flush the cache line from all levels of CPU caches before evicting the cache line from the DRAM cache. An attacker may do this by using the *flush* instruction, which is available for all programs with no special privileges, or priming a very large array to make sure the cached data will be evicted from all levels of the caches. Another approach to bypass the CPU cache hierarchy is to use *non-temporal* instructions.

Create Eviction set:

To create the eviction set, we use `huge page` (2 MB pages) which provides greater control over physical addresses. This makes it easier to locate addresses that conflict with a particular target address, but even with this control, the hugeness of the DRAM cache presents challenges. So to find the Eviction set, we access a pair of memory addresses and measure the access time. By repeatedly accessing two specific memory addresses and measuring the time it takes, we can infer whether they conflict in the cache or not. If accessing one address (let's say Address A) and then another (Address B) causes the data in Address A to be evicted from the cache (due to them being in a conflict set), the subsequent access to Address A would be slower (since it's now a cache miss). This increased latency is an indication of conflict. So to create the eviction set we follow below steps.

i. **Flush Target Address**: Flush a target address from the cache to ensure it's not already cached.

ii. **Baseline Access Time**: Access the target address and measure this access time to set a baseline.

iii. **Access Potential Conflicting Address**: Access another memory address which is potentially conflicting with the target.

iv. **Re-access Target**: Access the target address again and measure its access time.

v. **Check for Conflict**:

$$\text{if Re-access time} > \text{Baseline time significantly, then}$$

The two addresses are likely part of the same conflict set in the cache.

Covert Channel: At a high level, the sender transmits bits by evicting cache lines from the DRAM cache. To transmit a '1', sender needs to evict a cache line from all levels of the CPU cache as well as the DRAM cache. To do so, an attacker needs to pre-calculate a set of cache lines that cause eviction of targeted cache lines from the CPU caches and mapped to the same entry in the DRAM cache as described above. Since the DRAM cache is a direct-mapped cache, access to this set of lines causes eviction of the cache line to the Optane memory. When the receiver probes the targeted cache line in the DRAM cache, it has to be fetched from the Optane main memory which takes significantly longer. In the case of sending a '0' the sender does nothing, so accessing the targeted cache line will be in one level of the cache hierarchy. This way, the receiver is able to distinguish between receiving '0' or '1'. In case the sender and receiver share the memory, e.g using a shared library, an attacker would be able to use an unprivileged *flush* instruction followed by accessing a memory address that is mapped to the same cache line as the targeted line. As a result, the targeted line will be evicted out from the DRAM cache and will be written to the Optane memory.

So in summary the sender and receiver protocol would be as follows: before any data transmission begins, both the sender and receiver synchronize on a specific set of physical addresses (PA) with identical indices.

Sender's Protocol

- **Transmitting a '1'**: The sender's goal is to create a scenario where the target data has to be fetched from the slower Optane memory, which requires longer access time. To accomplish this:

   1. The sender first evicts a particular cache line from all CPU cache levels and the DRAM cache.
   2. Accessing the eviction set, the sender ensures the target cache line moves to the Optane memory.

- **Transmitting a '0'**: In contrast, the sender refrains from any activity, leaving the data in its current cache state.

Receiver's Protocol

1. The receiver occupies a previously agreed-upon communication set.
2. They then flush this set from the cpu caches and subsequently access it, keeping track of the time taken for the operation. A lengthened access time, indicating a "miss" in the DRAM cache (because data had to be fetched from Optane memory), is interpreted as a '1'. Conversely, a normal access time, or "hit", is recognized as a '0'.

Side-channel attack 1 : No shared memory: The primary goal of this attack is to spy on memory accesses made by a victim process. The attacker doesn't share memory with the victim, but the DRAM cache is shared between the two processes and attacker aim to learn access pattern of victim's process. This attacks involves following steps

Step 1: Initialization:

- *Flush the Target Address*: Ensure that the intended target address isn't in the CPU cache.

- *Access the Conflict Set Address*: Access a memory location that has the potential to conflict with the victim's memory accesses in the DRAM cache (monitored line).

Step 2: Monitor the Victim:

- *Wait for the Victim to Execute*: Wait for the victim to carry out its operations, potentially accessing its conflicting memory location.

Step 3: Probe for Information:

- *Flush and Re-access the Target*: Flush the previously accessed target address from the cache, then access it again, noting the time taken.
- *Deduce from Timing*:
  - *Slow Access (Miss)*: A slower access indicates that the victim accessed a conflicting memory location, evicting the attacker's data from the DRAM cache and the requested line needs to be fetch from Optane memory which takes longer.
  - *Fast Access (Hit)*: A faster access implies the victim did not access the conflicting location.

Side-channel 2: shared memory:

This attack aims to covertly monitor keypress events within the 'gedit' editor, leveraging the GTK library's behavior. The steps of the attack are detailed below.

Offline Phase: Cache Template Attack - During this initial phase, a cache template style attack is executed to record the cache access patterns corresponding to different keypress events in ‘gedit‘. This forms the foundation by providing a profile of cache access patterns that are indicative of various keypress events.

Creation of the Conflict Set - With the acquired data from the offline phase, a conflict set is devised. This set comprises specific memory addresses anticipated to be conflicted with the cache line related to pressed keys.

Continuous Monitoring - This is the active phase of the attack. Firstly, the addresses related to key press events will be flushed from all level of CPU cache hirarchy by issuing *flush* instruction. Following this, the conflict set address is accessed to evict these cache lines from the DRAM cache as well. After that, the monitored set is flushed, accessed again, and the time taken for access is measured.

Fast Access (Cache Hit): This indicates a potential keypress event by victim.

Slow Access (Cache Miss): This suggests the absence of a keypress event.

Employing this continuous monitoring approach, the attacker can derive keypress events by merely observing cache timings, enabling them to subtly keylog activities on the GTK library.

7.4.2 Timing channel 2

In the earlier discussion (section 7.3), we highlighted that when Optane operates in memory mode, the accompanying DRAM functions as a *Direct-mapped cache*, utilizing a write-back policy. In this section, we delve into the varied access types to the DRAM cache, particularly focusing on the state of the cache line—whether it’s ’dirty’ (modified)

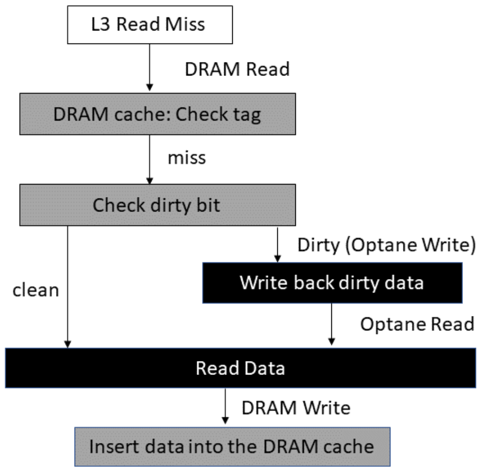


Figure 7.4: Optane DRAM cache access flow

or 'clean' (unmodified). We aim to show how a potential adversary could establish a covert channel by leveraging the differential time intervals associated with these access types.

Data Request from the CPU's Last Level Cache (L3)

If the requested data is not located in the CPU's Last Level Cache (L3), a load request is made to the Integrated Memory Controller (IMC) to retrieve the data, either from the DRAM cache or the Optane memory.

Depending on the status and existence of a cache line in the DRAM cache, read requests are categorized into:

DRAM cache Hit: The address requested by the LLC (L3) is found within the DRAM cache and data will be returned.

Dirty Miss: If the requested address is not present in the DRAM cache, the access is a miss and the requested data should be obtained from the Optane memory. As mentioned before, the DRAM cache is a direct-mapped cache so a miss in this cache implies that this cache set contains data for another address. If the already resident data in this set has been modified since its insertion, the data needs to be written back to the Optane memory. This type of access is known as a dirty miss.

Clean Miss: In case the requested address is not presented in the DRAM cache and the existing data is unmodified since its insertion, there is no need to be written back to the main memory. This is known as a clean miss.

Write Requests Originating from L3 Cache

The L3 can initiate write requests in two scenarios:

1. When a dirty cache line must be evicted and written back to the DRAM cache.
2. In instances of a non-temporal store operation.

The course of action on receiving a write request is:

cache Hit: If the write request's specified address is in the DRAM cache: The memory controller checks the cache line's tag by issuing a read request to the DRAM cache. If the tag is confirmed to be present, the controller updates the cache line with a write operation.

Dirty Miss: In case of a miss in DRAM cache:

- The controller issues a read request to verify the cache line's tag (DRAM cache Read). in this case the tag is not found in the cache and the cache line needs to be fetched from Optane.
- If the current line in the DRAM cache which the desired address (conflicting line) is dirty, the system writes it back to the Optane memory (Optane Write), reads the new

data from the Optane memory (Optane Read), and then writes this new data back to the DRAM cache (DRAM cache write) line.

Clean Miss

- The controller issues a read request to verify the cache line's tag (DRAM cache Read). in this case the tag is not found in the cache and the cache line needs to be fetched from Optane.

- In the event of a clean miss, the conflicting line has not been updated since the time of its insertion. Consequently, this line will not be written back to the Optane memory. Instead, the new line will be read from the Optane memory and then written back to the DRAM cache.

7.4.3 Covert channel

An adversary could potentially exploit the timing differences based on the state of the line in the DRAM cache to build a covert channel. By deliberately inducing dirty misses or clean misses and measuring the time it takes to access data, the adversary can communicate information. Based on the information presented in the last section as shown in Figure 7.4, For write requests that miss in the DRAM cache when the cache line is dirty the following access happen: (i) *Read the Tag*: DRAM cache Read, (ii) *Write the Dirty Line*: Optane Write, (iii) *Read New Line*: Optane Read, (iv) *Update Cache*: DRAM cache Write.

For a clean miss we will have following accesses: (i) *Read the Tag*: DRAM cache Read, (ii) *Fetch New Line*: Optane Read, (iii) *Update Cache*: DRAM cache Write.

Given the time difference due to one fewer write to the Optane memory, an attacker would be able to create a covert channel as described below.

The covert channel exploits timing differences based on the state of the line in the DRAM cache. Specifically, the timing distinction between a clean miss and a dirty miss is utilized to convey information covertly. The time taken to process a dirty miss is longer than that of a clean miss due to the additional steps involved in handling the dirty state of the cache line.

Trojan (Sender):

Send "1" The Trojan induces a dirty miss by writing to a specific cache set, making the cache line dirty. It then either flushes this cache line or uses a non-temporal store.

Send "0" The Trojan accesses a conflicting address to cause a miss the next time the original data is accessed, resulting typically in a clean miss.

Spy (Receiver): The Spy infers the data sent by the Trojan based on the time taken to access data. First, it flushes the cache, then accesses the conflicting address, which results in a cache miss. The type of miss and the access time reveal the transmitted bit.

Clean Miss Detected by a shorter access time, inferring a received bit as '0'.

Dirty Miss Detected by a longer access time due to the need for writing back data, inferring the received bit as '1'.

By deliberately manipulating and observing cache behavior, the Trojan and the Spy are communicating covertly. The Trojan manipulates the cache state to send a bit, and the Spy observes the cache's behavior (timing) to infer that bit.

7.5 Concluding Remarks

In this chapter, we did an in-depth exploration of the unique memory configuration involving the DRAM and Intel Optane. They each play distinctive roles in a tiered memory hierarchy, with DRAM serving as the 'near memory' or memory side cache, especially those that miss in the CPU's LLC. On the other hand, Optane serves as the main memory of the system, stepping in whenever the DRAM cache does not fulfill the memory requests.Our detailed reverse engineering efforts have revealed key insights into the operation of the DRAM cache, identifying it as a direct-mapped write-back cache. Additionally, we shed light on different components of the DRAM cache, such as the prefetcher and tag check mechanism. In the realm of the Optane memory controller, we identified components like the AIT, write buffer, and the Optane side internal cache.

From these observations, we introduced two types of timing channels. These are based on the access to the different levels of memory hierarchy as well as cache line state. For each timing channel, we've shown how adversaries are able to create a covert channel, and where possible, construct a side channel to leak data.

A promising direction for future research is delving deeper into reverse engineering the AIT. By understanding this component more precisely, we may unveil new possibilities of attacks based on its structure and functionality.

Chapter 8

Concluding Remarks

In the face of escalating cybersecurity threats, the landscape of attacks on computing systems has grown exponentially, with attackers demonstrating increased motivation and sophistication. Recent incidents, including the Meltdown and Spectre attacks, have underscored that even the hardware and architecture that underpins computing systems can serve as exploitable attack vectors, compromising the security of software and data. This dissertation embarked on an extensive exploration of the boundary between hardware and software in the realm of computer security, focusing onto hardware-originated attacks while also investigating architectural support for strengthening system and software security.

The first chapter of this dissertation introduced SpectreRSB, a novel Spectre attack developed to target the return stack buffer, a critical component employed in optimizing the execution of return instructions on modern CPUs. By demonstrating the viability of these attacks, this research highlights the pressing need for comprehensive defenses against Spectre vulnerabilities.

In response to the challenges posed by Spectre attacks, this dissertation has also introduced an innovative defense mechanisms. SpecCFI emerges as a new CPU design principle that harnesses program analysis to secure processors against Spectre attacks while still maintaining the benefits of speculative execution. By proposing Speculative Execution Regulation (SER), a versatile defense approach, this research advances the concept of maintaining security invariants even during speculative execution, mitigating a wide range of transient execution attacks.

Furthermore, the dissertation has proposed SafeSpec, a pivotal defense strategy that strives to make speculation leakage-free in a principled manner. By containing speculative state within temporary shadow structures that remain inaccessible to committed instructions, SafeSpec effectively retains the advantages of speculation while neutralizing its associated vulnerabilities.

Finally, our research highlighted two timing channel vulnerabilities in Intel Optane's memory hierarchy. These channels are shaped by distinct memory behaviors, particularly based on accessing data at different levels of the memory hierarchy and cache line states. Our analysis demonstrated how adversaries might exploit these channels for covert data transmission and, in certain scenarios, launch side channel attacks.

Bibliography

[1] Arm. https://www.arm.com.

[2] Altera de2-115 development and education board. https://www.altera.com/solutions/partners/partner-profile/terasic-inc-/board/altera-de2-115-development-and-education-board.html#overview, 2010.

[3] Intel and Micron Produce Breakthrough Memory Technology, 2015. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/#gs.5irpfz.

[4] Spec cpu2017 documentation. https://www.spec.org/cpu2017/Docs, 2017.

[5] Test suite extensions. https://llvm.org/docs/Proposals/TestSuite.html, 2019.

[6] Adesto Touts ReRAM for Automotive, 2021. https://www.eetimes.com/adesto-touts-reram-for-automotive/.

[7] Automotive Temperature Range MRAM, 2021. https://www.everspin.com/file/882/download.

[8] Ferroelectric RAM (FeRAM) – Instant non-volatile memory, 2021. https://www.cypress.com/products/f-ram-nonvolatile-ferroelectric-ram.

[9] Optane™ PMem, 2021. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[11] ADVANCED MICRO DEVICES, INC. Amd64 technology: Speculative store bypass disable. https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf, 2018.

[12] ADVANCED MICRO DEVICES, INC. Software techniques for managing speculation on amd processors. https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018.

[13] Jaeguk Ahn, Cheolgyu Jin, Jiho Kim, Minsoo Rhu, Yunsi Fei, David Kaeli, and John Kim. Trident: A hybrid correlation-collision gpu cache timing attack for aes key recovery. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 332–344, 2021.

[14] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 565–577, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

[16] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. Technical report, 2018. Available from `https://eprint.iacr.org/2018/1060.pdf`.

[17] Osman Aleksander. The ao486 project. `https://github.com/alfikpl/ao486`, 2014.

[18] Microsoft Security Response Center (Saar Amar). An armful of CHERIs. `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`, 2022.

[19] AMD. Amd crossfire guide for direct3d® 11 applications, 2017.

[20] ARM. Cache speculative side-channels. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[21] ARM. Vulnerability of speculative processors to cache timing side-channel mechanism. `https://developer.arm.com/support/security-update`, 2018.

[22] ARM. Armv8.5-a memory tagging extension (white paper). `https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjcuLXjl7j2AhXLWMOKHW6qCsgQFnoECAwQAQ&url=https%3A%2F%2Fdeveloper.arm.com%2F-%2Fmedia%2FArm%2520Developer%2520Community%2FPDF%2FArm_Memory_Tagging_Extension_Whitepaper.pdf&usg=AOvVaw12l-TyFWLy8JrjhIa_uwHl`, 2019.

[23] ARM Limited. Arm® a64 instruction set architecture (00bet9), 2018.

[24] Steve Bannister. Memory tagging extension: Enhancing memory safety through architecture. `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety`, 2019.

[25] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently breaking ASLR in the cloud. In *Proceedings of the 9th USENIX Conference on Offensive Technologies (WOOT)*, 2015.

[26] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[27] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, volume 41, page 46, 2005.

[28] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. *arXiv preprint arXiv:1903.01843*, 2019.

[29] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. In *ACM SIGARCH Computer Architecture News*, 2011.

[30] Erich Bloch. The engineering design of the Stretch computer. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, 1959.

[31] Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[32] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 667–684, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[33] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[34] Nathan Burow, Scott A Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. Control-flow integrity: Precision, security, and performance. *arXiv preprint arXiv:1602.04056*, 2016.

[35] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, 2019.

[36] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of defenses against transient-execution attacks. In *GLSVLSI '20: Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020.

[37] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[38] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[39] Chandler Carruth. Mitigating speculative attacks in crypto. `https://github.com/HACS-workshop/spectre-mitigations/blob/master/crypto_guidelines.md`, 2018.

[40] Chandler Carruth. Rfc: Speculative load hardening (a spectre variant 1 mitigation). `https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html`, 2018.

[41] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)*, 2006.

[42] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009.

[43] Abhishek Chakraborty, Ankit Mondal, and Ankur Srivastava. Correlation power analysis attack against stt-mram based cyptosystems. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 171–171, 2017.

[44] Anupama Chandrasekhar, Gang Chen, Po-Yu Chen, Wei-Yu Chen, Junjie Gu, Peng Guo, Shruthi Hebbur Prasanna Kumar, Guei-Yuan Lueh, Pankaj Mistry, Wei Pan, Thomas Raoux, and Konrad Trifunovic. Igc: The open source intel graphics compiler. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 254–265. IEEE Press, 2019.

[45] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[46] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[47] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[48] Long Cheng, Hans Liljestrand, Thomas Nyman, Yu Tsung Lee, Danfeng Yao, Trent Jaeger, and N. Asokan. Exploitation techniques and defenses for data-oriented attacks. In *arXiv:1902.08359*, 2019.

[49] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[50] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[51] Clang. Hardware-assisted addresssanitizer (hwasan). `https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html`.

[52] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[53] Patrick Cozzi. WebGL Overview, Khronos Group, 2018. `https://www.khronos.org/webgl/`.

[54] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. In *Design Automation Conference (DAC)*, 2015.

[55] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Design Automation Conference (DAC)*, 2014.

[56] Erik P. DeBenedictis. It's time to redefine moore's law again. *Computer*, 50(2):72–75, 2017.

[57] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[58] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[59] Matthew Dillon. Clarifying the spectre mitigations. `http://lists.dragonflybsd.org/pipermail/users/2018-January/335637.html`, 2018.

[60] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *USENIX Security*, 2017.

[61] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2019.

[62] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side-channel attacks. In *ACM Transactions on Architecture and Code Optimization (TACO)*, January 2012.

[63] D.Page. Partitioned cache architecture as a side-channel defense mechanism. In *Crypt. ePrint Arch.*, 2005.

[64] T. Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59(9), June 2002. Accessed online April 2018 from `http://www.phrack.org/phrack/59/p59-0x09`.

[65] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 972–984, 2021.

[66] Dag Arne Osvik Eran Tromer and Adi Shamir. Efficient cache attacks on aes, and countermeasures. In *Journal of Cryptology*, pages 667–684, 2009.

[67] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[68] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[69] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[70] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *Proc. IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2016.

[71] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization*, 13(1):10, 2016.

[72] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs. an optimization guide for assembly programmers and compiler makers (2016). *URL http://www. agner. org/optimize/microarchitecture. pdf.*

[73] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 357–372, 2018.

[74] Yiwen Gao, Hailong Zhang, Wei Cheng, Yongbin Zhou, and Yuchen Cao. Electromagnetic analysis of gpu-based aes implementation. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[75] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[76] V. GEORGE, T. PIAZZA, and H. JIANG. Technology insight: Intel next generation microarchitecture codename ivy bridge, September 2011. Available online from `http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf`.

[77] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[78] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[79] Thomas R Eisenbarth Gorka Irazoqui and Berk Sunar profile. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 353–364, 2016.

[80] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *USENIX Security Symposium*, 2018.

[81] Brendan Gregg. KPTI/KAISER meltdown initial performance regressions, 2018. Accessed online April 2018 at `http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html`.

[82] Matthew Gretton-Dann. Arm a-profile architecture developments 2018: Armv8.5-a. `https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a`, 2018.

[83] Daniel Gruss. Transient-execution attacks and defenses. ..., ...(...):..., 2020.

[84] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176, 2017.

[85] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 368–379, 2016.

[86] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 300–321. Springer, 2016.

[87] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[88] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[89] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *ACM on Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[90] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[91] Amira Guesmi, Ihsen Alouani, Khaled N. Khasawneh, Mouna Baklouti, Tarek Frikha, Mohamed Abid, and Nael Abu-Ghazaleh. Defensive approximation: Securing cnns using approximate computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 990–1003, New York, NY, USA, 2021. Association for Computing Machinery.

[92] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[93] Wenjian HE, Wei Zhang, Sharad Sinha, and Sanjeev Das. Igpu leak: An information leakage vulnerability on intel integrated gpu. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 56–61. IEEE Press, 2020.

[94] J. Horn. speculative execution, variant 4: speculative store by-pass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[95] Jann Horn. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, 2018.

[96] Jann Horn. Speculative execution, variant 4: Speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[97] Project Zero (Jann Horn). Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, 2018.

[98] Morteza Hoseinzadeh, Mohammad Arjomand, and Hamid Sarbazi-Azad. Reducing access latency of mlc pcms through line striping. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 277–288, 2014.

[99] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[100] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 385–399, New York, NY, USA, 2020. Association for Computing Machinery.

[101] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[102] Open Source Security Inc. Respectre: The state of the art in spectre defenses. `https://www.grsecurity.net/respectre_announce.php`, 2018.

[103] INTEL. Control-flow enforcement technology (CET). `https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html`.

[104] Intel. Getting the most from opencl™ 1.2: How to increase performance by minimizing buffer copies on intel® processor graphics, 2014.

[105] Intel. Opencl 2.0 shared virtual memory overview, 2014.

[106] Intel. Intel processor graphics gen9 architecture, 2015.

[107] Intel. Intel analysis of speculative execution side channels. `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf`, 2018.

[108] Intel. Retpoline: A branch target injection mitigation. `https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`, 2018.

[109] Intel. Speculative execution side channel mitigations. `https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf`, 2018.

[110] INTEL. Deep dive: Intel analysis of microarchitectural data sampling. `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling`, 2019.

[111] Intel. Intel graphics compiler, 2019.

[112] Intel. Intel processor graphics gen11 architecture, 2019.

[113] Intel. Intel® open source hd graphics and intel iris™ plus graphics programmer's reference manual: Volume 5: Memory views, 2019.

[114] Intel Corp. Speculative store bypass bug cve, 2018. CVE 2018-3639. Accessed online from `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639`.

[115] Intel Corporation. Intel® 64 and ia-32 architectures optimization reference manual. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, 2016.

[116] Intel Corporation. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2017.

[117] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[118] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design*, pages 629–636. IEEE, 2015.

[119] Anirudh Iyengar, Swaroop Ghosh, Nitin Rathi, and Helia Naeimi. Side channel attacks on sttram and low-overhead countermeasures. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 141–146, 2016.

[120] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, page 25–36, New York, NY, USA, 2007. Association for Computing Machinery.

[121] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019.

[122] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[123] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[124] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a gpu. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA'16, pages 394–405, Barcelona Spain, March 2016. IEEE.

[125] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on gpus. In *Proceedings of the on Great Lakes Symposium on VLSI*, VLSI'17, pages 167–172, 2017.

[126] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[127] Mengxuan Wang Juan Fang and Zelin Wei. A memory scheduling strategy for eliminating memory access interference in heterogeneous system. In *The Journal of Supercomputing*, volume 76, page 3129–3154, 2020.

[128] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[129] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: relaxed inclusion caches for mitigating LLC side-channel attacks. In *Design Automation Conference (DAC)*, 2017.

[130] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[131] V. Kemerlis, G. Portokalidis, and A. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *USENIX Security Symposium*, 2012.

[132] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, San Diego, CA, 2014. USENIX Association.

[133] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Analysis of row hammer attack on sttram. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 75–82, 2018.

[134] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Fault injection attacks on emerging non-volatile memory and countermeasures. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[135] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *arXiv preprint arXiv:1806.05179*, 2018.

[136] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Design Automation Conference (DAC)*, 2019.

[137] S. K. Khatamifard, L. Wang, S. Köse, and U. R. Karpuzcu. A new class of covert channels exploiting power management vulnerabilities. *IEEE Computer Architecture Letters*, 17(2):201–204, 2018.

[138] khronos group. OpenCL Overview, Khronos Group, 2018. `https://www.khronos.org/opencl/`.

[139] Hyojong Kim, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. Coda: Enabling co-location of computation and data for multiple gpu systems. *ACM Trans. Archit. Code Optim.*, 15(3), sep 2018.

[140] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. 2018.

[141] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[142] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. In *Computing Research Repository (CoRR)*, 2018.

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

[144] Paul Kocher. Spectre mitigations in microsoft's c/c++ compiler. `MicrosoftCompilerSpectreMitigation.html`, 2018.

[145] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv e-prints arXiv:1801.01203*, 2018.

[146] J. Kong, O. Aclicmez, J. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proc. of CSAW Workshop, held with CCS'08*, October 2008.

[147] J. Kong, O. Aclicmez, J. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Int. Symp. on High Performance Comp. Architecture (HPCA)*, February 2009.

[148] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[149] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[150] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCFI: Mitigating spectre attacks using CFI informed speculation. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[151] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[152] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2013.

[153] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[154] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, 2017.

[155] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. Poisonivy: Safe speculation for secure memory. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[156] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association.

[157] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (Security)*, 2018.

[158] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[159] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP), San Jose, CA, US*, 2015.

[160] S. Liu, Y. Wei, J. Chi, F. H. Shezan, and Y. Tian. Side channel attacks in computation offloading systems with gpu virtualization. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 156–161, 2019.

[161] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient Non-Observability. In *USENIX Security Symposium*, 2021.

[162] H. Lu. [patch 0/5] x86: Cve-2017-5715, Spectre bug disclosure. `https://gcc.gnu.org/ml/gcc-patches/2018-01/msg00422.html`, 2018.

[163] Tingting Lu and Junfeng Wang. Data-flow bending: On the effectiveness of data-flow integrity. In *Computers Security*, 2019.

[164] Chao Luo, Yunsi Fei, and David Kaeli. Side-channel timing attack of rsa on a gpu. *ACM Transactions on Architecture and Code Optimization*, 16(3), August 2019.

[165] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. Side-channel power analysis of a gpu aes implementation. In *33rd IEEE International Conference on Computer Design*, ICCD'15, 2015.

[166] A. LUTOMIRSKI. x86/fpu: Hard-disable lazy fpu mode. `https://lkml.org/lkml/2018/6/14/509`, 2018.

[167] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[168] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the security implications of speculative execution in CPUs. *arXiv preprint arXiv:1801.04084*, 2018.

[169] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 118–129, 2012.

[170] Manel Martínez-Ramón, Arjun Gupta, José Luis Rojo-Álvarez, and Christos Christodoulou. *Machine Learning Applications in Electromagnetics and Antenna Array Processing.* Artech House, 2021.

[171] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[172] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[173] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[174] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[175] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[176] Microsoft. Spectre mitigations in msvc. `https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/`, 2018.

[177] microsoft. Microsoft azure, 2019.

[178] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. Fallout: Reading kernel writes from user space. 2019.

[179] O. Mutlu, Y. N. Patt, H. Kim, and D. N. Armstrong. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54:1556–1571, 2005.

[180] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 129–140. IEEE, 2003.

[181] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[182] Hoda Naghibijouybari, Khaled Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpus. In *Proc. International Symposium on Microarchitecture (MICRO)*, pages 354–366, 2017.

[183] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2139–2153, New York, NY, USA, 2018. Association for Computing Machinery.

[184] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. (mis)managed: A novel tlb-based covert channel on gpus. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 872–885, New York, NY, USA, 2021. Association for Computing Machinery.

[185] Ben Niu and Gang Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[186] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[187] Ben Niu and Gang Tan. Per-input control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[188] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[189] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery.

[190] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[191] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[192] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security*, 2013.

[193] A. Patel, F. Afram, and K. Ghose. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *Proc. of QUF*, 2011.

[194] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[195] PAX team. Future of pax. `https://pax.grsecurity.net/docs/pax-future.txt`, 2002.

[196] PAX team. RAP: RIP ROP. `https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`, 2015.

[197] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[198] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *USENIX Security Symposium*, 2016.

[199] Filip Pizlo. What spectre and meltdown mean for webkit. `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/`, 2018.

[200] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[201] The Chromium Projects. Site isolation. `http://www.chromium.org/Home/chromium-security/site-isolation`.

[202] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proc. IEEE/ACM International Symposium on Microarchitecture (Micro)*, 2018.

[203] Toktam Ramezanifarkhani and Mohammadreza Razzazi. Principles of data flow integrity: Specification and enforcement. In *J. Inf. Sci. Eng., vol. 31, no. 2, pp. 529–546*, 2015.

[204] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[205] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1090, 2021.

[206] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[207] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. Ghost loads: What is the cost of invisible speculation? In *Proceedings of the 16th ACM International Conference on Computing Frontiers (CF)*, 2019.

[208] Mohammad Hossein Samavatian, Saikat Majumdar, Kristin Barber, and R. Teodorescu. Hasi: Hardware-accelerated stochastic inference, a defense against adversarial machine learning attacks. *ArXiv*, abs/2106.05825, 2021.

[209] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[210] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725*, 2019.

[211] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data

sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[212] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *arXiv preprint arXiv:1905.09100*, 2019.

[213] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.

[214] James F Scott and Carlos A Paz De Araujo. Ferroelectric memories. *Science*, 246(4936):1400–1405, 1989.

[215] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[216] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC)*, 2012.

[217] Kostya Serebryany. ARM memory tagging extension and how it improves C/C++ memory safety. *login Usenix Mag.*, 44(2), 2019.

[218] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves C/C++ memory safety. 2018.

[219] Andre Seznec. TAGE-SC-L branch predictors. In *Proc. of the 4th Championship on Branch Prediction (`http://www.jilp.org/cbp2014/`)*, 2014. Accessed online April 2018 from, `https://hal.inria.fr/hal-01086920/document`.

[220] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[221] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2015.

[222] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2019.

[223] Navin Shenoy. Firmware updates and initial performance data for data center systems, 2018. Accessed online April 2018 at `https://newsroom.intel.com/news/firmware-updates-and-initial-performance-data-for-data-center-systems/`.

[224] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An integrated cache timing, power, and area model, 2001. Technical Report 2001/2, Compaq Computer Corporation.

[225] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 639–656, 2019.

[226] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[227] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[228] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[229] J. Stecklina and T. Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[230] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2004.

[231] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[232] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning, 2018.

[233] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[234] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 213–226, 2018.

[235] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security*, 2014.

[236] Top-500. Top-500 supercomputer list, 2018.

[237] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[238] Liam Tung. Linux meltdown patch: 'up to 800 percent cpu overhead', netflix tests show, February 2018. ZDNet article, accessed online April 2018 at `https://www.zdnet.com/article/linux-meltdown-patch-up-to-800-percent-cpu-overhead-netflix-tests-show/`.

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

[240] Jo Van B., M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium (Security)*, 2018.

[241] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[242] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[243] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[244] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue inflight data load. In *IEEE Symposium on Security and Privacy (Oakland)*, May 2019.

[245] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A framework for reverse engineering hardware page table caches. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, 2017.

[246] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys (CSUR)*, 53(2):1–33, 2020.

[247] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[248] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *arXiv preprint arXiv:1807.05843*, 2018.

[249] Hua Wang, Yao Guo, and Xiangqun Chen. Fpvalidator: validating type equivalence of function pointers on the fly. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[250] Xin Wang and Wei Zhang. An efficient profiling-based side-channel attack on graphics processing units. In Kim-Kwang Raymond Choo, Thomas H. Morris, and Gilbert L. Peterson, editors, *National Cyber Summit (NCS) Research Track*, pages 126–139, Cham, 2020. Springer International Publishing.

[251] Yao Wang and G. Edward Suh. Timing channel protection for a shared memory controller. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[252] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Proc. International Symposium on Microarchitecture (MICRO)*, December 2008.

[253] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference (ACSAC)*, 2006.

[254] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[255] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter Neumann. Capability hardware enhanced risc instructions (CHERI). `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`.

[256] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[257] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks. Technical report, University of Cambridge, Computer Laboratory, 2018.

[258] Junyi Wei, Yicheng Zhangy, Zhe Zhou, Zhou Liy, and Mohammad Abdullah Al Faruque. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[259] O. Weisse, J. Van, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, 2018.

[260] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[261] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. In *arXiv:1912.11523*, 2020.

[262] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[263] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[264] Henry Wong. Store-to-load forwarding and memory disambiguation in x86 processors. `http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/`, 2014.

[265] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[266] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[267] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 19–35, 2016.

[268] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: Mitigating contention based side and covert channel attacks on gpus. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 497–509, New York, NY, USA, 2019. ACM.

[269] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[270] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 2019*

IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[271] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[272] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020. USENIX Association, August 2020.

[273] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2018.

[274] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[275] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.

[276] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[277] Tse-Yu Yeh and Yale N Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 124–134, 1992.

[278] Michael K. Reiter Yinqian Zhang, Ari Juels and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[279] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. of ISPASS*, 2007.

[280] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery.

[281] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Defending virtual function tables' integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2015.