SpecLFB: Eliminating Cache Side Channels in Speculative Executions

Xiaoyu Cheng, *School of Cyber Science and Engineering, Southeast University, Nanjing, Jiangsu, China; Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China;* Fei Tong, *School of Cyber Science and Engineering, Southeast University, Nanjing, Jiangsu, China; Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China; Purple Mountain Laboratories, Nanjing, Jiangsu, China;* Hongyu Wang, *State Key Laboratory of Power Equipment Technology, School of Electrical Engineering, Chongqing University, China; Wiscom System Co., LTD, Nanjing, China;* Zhe Zhou and Fang Jiang, *School of Cyber Science and Engineering, Southeast University, Nanjing, Jiangsu, China; Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China;* Yuxing Mao, *State Key Laboratory of Power Equipment Technology, School of Electrical Engineering, Chongqing University, China*

https://www.usenix.org/conference/usenixsecurity24/presentation/cheng-xiaoyu

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

SpecLFB: Eliminating Cache Side Channels in Speculative Executions

Xiaoyu Cheng[1,2], Fei Tong[1,2,3,*], Hongyu Wang[4,5], Zhe Zhou[1,2], Fang Jiang[1,2], Yuxing Mao[4]

[1]*School of Cyber Science and Engineering, Southeast University, Nanjing, Jiangsu, China*
[2]*Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China*
[3]*Purple Mountain Laboratories, Nanjing, Jiangsu, China*
[4]*State Key Laboratory of Power Equipment Technology,*
School of Electrical Engineering, Chongqing University, China
[5]*Wiscom System Co., LTD, Nanjing, China*
[*]*Corresponding Author: ftong@seu.edu.cn*

Abstract

Cache side-channel attacks based on speculative executions are powerful and difficult to mitigate. Existing hardware defense schemes often require additional hardware data structures, data movement operations and/or complex logical computations, resulting in excessive overhead of both processor performance and hardware resources. To this end, this paper proposes SpecLFB, which utilizes the microarchitecture component, Line-Fill-Buffer, integrated with a proposed mechanism for load security check to prevent the establishment of cache side channels in speculative executions. To ensure the correctness and immediacy of load security check, a structure called ROB unsafe mask is designed for SpecLFB to track instruction state. To further reduce processor performance overhead, SpecLFB narrows down the protection scope of unsafe speculative loads and determines the time at which they can be deprotected as early as possible. SpecLFB has been implemented in the open-source RISC-V core, SonicBOOM, as well as in Gem5. For the enhanced SonicBOOM, its register-transfer-level (RTL) code is generated, and an FPGA hardware prototype burned with the core and running a Linux-kernel-based operating system is developed. Based on the evaluations in terms of security guarantee, performance overhead, and hardware resource overhead through RTL simulation, FPGA prototype experiment, and Gem5 simulation, it shows that SpecLFB effectively defends against attacks. It leads to a hardware resource overhead of only 0.6% and the performance overhead of only 1.85% and 3.20% in the FPGA prototype experiment and Gem5 simulation, respectively.

1 Introduction

Out-of-order (OoO) and speculative execution mechanisms are the optimization techniques commonly used by modern CPUs to improve performance. In OoO execution, the CPU does not strictly execute instructions in program order, but rather executes later instructions ahead of time when conditions are met, in order to improve the utilization of processor components. Speculative mechanism refers to the processor's speculative execution of instructions, which, in coordination with pipelining and OoO execution, can significantly improve CPU performance. However, speculative execution is not guaranteed to be completely accurate, and OoO execution may result in instructions being executed before they have proper permission. Typically, these erroneously executed instructions, also known as transient instructions, change the microarchitectural state and are "rolled back" by the processor before they are "committed" to the architectural state. Therefore, programmers can only see instructions executed correctly according to the program flow at the architectural level.

However, the disclosure of side-channel attack methods such as Flush+Reload [55] and Prime+Probe [53] has shown that information can be leaked through the microarchitectural state and behavior. In early 2018, several independent security research teams disclosed serious vulnerabilities in modern CPU architectures, such as Spectre [30] and Meltdown [35] attacks, which have made a significant impact on the industry and continue to do so to this day. Among the exploited techniques for these vulnerabilities, cache side-channel attack [9, 11, 21, 22, 27, 34, 53–55] is the main approach. The Spectre attacks can leverage speculative executions to change the cache state and leak data through the cache side channels.

To mitigate speculative execution attacks, many hardware defense solutions have been proposed. Yan et al. proposed InvisiSpec [52], in which speculative loads that may cause security problems are stored in the Speculative Buffer (SB) instead of the cache. When a speculation succeeds, the speculative data is then installed back into cache. SafeSpec proposed in [28] shadows the hardware architecture used during the execution of speculative instructions, so that any changes to the microarchitectural state could be reverted if the processor's speculation is turned out to be incorrect. Both solutions make speculative execution "invisible" by clearing all side effects of speculative executions. Based on this idea, many other methods have been proposed [3, 40, 49]. However, regardless of whether speculation succeeds or not, they all require expensive re-install operations. Speculative taint tracking (STT) [57] and NDA [47] use dataflow tracing similar to taint propagation to track instructions that may lead to information leakage. These instructions are forced to be delayed until their related instructions become safe, thus preventing unsafe speculative data from being transmitted to side channels. Compared with the solution of hiding all the side-effects of speculative executions until reaching the visibility point (e.g., when all old control flow instructions have been resolved) in InvisiSpec, these solutions which selectively delay speculative executions are more efficient. However, instruction tracing and taint marking also require complex logic to compute the "root of taint", resulting in significant hardware changes and performance overhead.

Overall, most of the above hardware defense solutions require additional data structures, data movement operations and/or complex logical calculations, which can cause excessive additional performance and hardware resource overhead. These methods also lack implementation and evaluation based on real hardware prototypes, and most defense solutions are only verified through architecture simulation platforms such as Gem5 [38] rather than register-transfer-level (RTL) implementation. In addition, the types of attacks included in the security verification of these schemes are relatively homogeneous, which have always been Spectre v1 and v2 [30] with similar principles, which exploit unresolved branches in the instruction stream. However, the factors that can lead to unsafe speculation executions are diverse.

To address these issues, this paper proposes a novel hardware defense scheme, SpecLFB. It leverages the microarchitectural component Line-fill-buffer (LFB) and introduces a simple yet effective security check mechanism to LFB to prevent potentially unsafe speculative loads from being reloaded from lower-level cache into adjacent upper-level cache (e.g. from L2 cache into L1 data cache (L1D cache)), thereby preventing the establishment of cache side channels. We have implemented SpecLFB in the L1D cache of open-source RISC-V core SonicBOOM (the latest version, also called BOOMv3) [58] and both cache levels of X86 O3 CPU model in Gem5 simulator [38]. The SpecLFB-enhancing cores have been evaluated in the Verilator simulation based on the RTL code generated by Chipyard and in an FPGA hardware prototype we have developed, as well as in the Gem5 simulation. The evaluation results show that it can successfully defend against Spectre v1, v2, v4 [24] and v5 [31] attacks. In comparison with the original cores and the cores enhanced with the state-of-the-art defense schemes, the experimental results show that SpecLFB leads to a hardware resource overhead of only 0.6%. Furthermore, by running the SPEC CPU 2017 benchmark suite (SPEC2017), SpecLFB exhibits the performance overhead of only 1.85% and 3.20% in the FPGA prototype experiment and Gem5 simulation, respectively.

This paper makes the following contributions:

- We propose a novel scheme called SpecLFB to defend against cache side-channel attacks. By introducing a simple yet effective security check mechanism to LFB, the executions of unsafe speculative loads can be delayed to prevent the establishment of cache side channels from propagating sensitive data.
- To reduce processor performance overhead, SpecLFB narrows down the protection scope of unsafe speculative loads and determines the time at which they can be deprotected as early as possible.
- We fully and seamlessly implement SpecLFB in the L1D cache of the open-source RISC-V core, SonicBOOM, as well as in all cache hierarchy of Gem5 OoO processors.
- We develop an FPGA hardware prototype burned with the L1D-SpecLFB-enhancing SonicBOOM and running a Linux-kernel-based operating system.
- We evaluate SpecLFB in terms of security guarantee, performance overhead, and hardware resource overhead, through both Gem5 and RTL simulations, as well as the experiment based on the developed hardware prototype.

2 Background

2.1 Transient Execution

In modern OoO processors, instructions issued in program order are decoded into micro-operations (μops), but the execution of μops may not follow the program order. Even if an earlier μop in the instruction stream has not been completed, the CPU may execute a ready μop when certain conditions are met (i.e., the μop's operands do not have dependencies on previous μops or their values have been computed). CPU tracks all μops through the reorder buffer (ROB), and when a μop completes execution, it is retired from the ROB in program order, irreversibly modifying the CPU's architectural state. This OoO execution technique can increase the component utilization of the processor.

Conditional branches or data dependencies between instructions are generally included in the instruction stream. Theoretically, the processor without knowing the future instruction stream of a program must wait for the branch result or resolved data dependencies before continuing to execute instructions. Since the execution suspension will reduce processor performance, the processor can predict the results of conditional branches and data dependencies through some components such as branch predictors, memory disambiguators, etc., and then execute along the speculative path. If the prediction turns out to be correct, the precomputed results of the speculative execution are committed; otherwise, if the processor determines that it followed the wrong path, the instruction-related μops will be squashed and the processor will be rolled back to the last correct state and resume along the correct path.

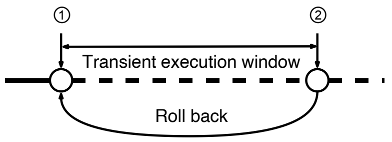


Figure 1: Transient execution flow. The point ① is when there is an unresolved branch or data dependency, and the point ② is when the processor gets the results and finds that the speculation is wrong.

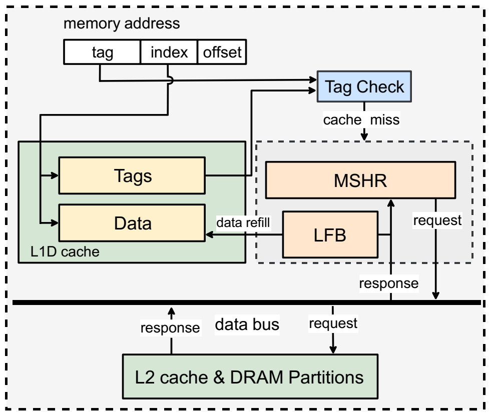


Figure 2: The working principle of LFB in L1D cache.

Therefore, these instructions that are executed out of order and whose executions are incorrectly speculated may cause the microarchitectural state of the processor to change, even though their results are never committed to the architectural state due to processor rollback. This is called transient execution, and these instructions are called transient instructions [30, 35]. As shown in Figure 1, the period of time from the beginning of the execution of a transient instruction to when the processor detects the error and begins to roll back its state is called the transient execution window.

2.2 LFB

A cache with only Data and Tag Arrays, which would not accept new load/store misses in case of cache misses, is called a blocking cache. In an OoO processor, a blocking cache prevents the load/store unit (LSU) from issuing several store or read accesses to it, thus affecting the overall processor speed. To solve this problem, modern processors [12, 16, 43, 58] have introduced Miss Status Holding Registers (MSHR) to hold the information of unresolved cache misses in order to accept more than one cache misses and implement non-blocking caches. When a cache miss occurs or data prefetching requires cache refills, MSHR will request data from the data bus and refill the requested data into cache via LFB.

As shown in Figure 2, when the data targeted by a memory access instruction is missed in the upper-level caches, the miss handling logic first checks MSHR to see if there is a pending request that matches the current one. If so, the request is merged into the same entry and no new data request needs to be issued [33]. Otherwise, a new MSHR entry and LFB entry will be reserved for this data request. When the requested cache line is fetched from the lower-level caches or memory, it is placed into LFB instead of directly being written into the upper-level caches, allowing cache eviction and refill to occur in parallel [19], so the hit time of the cache can be greatly reduced. When the cache eviction is complete, LFB is flushed into the cache arrays.

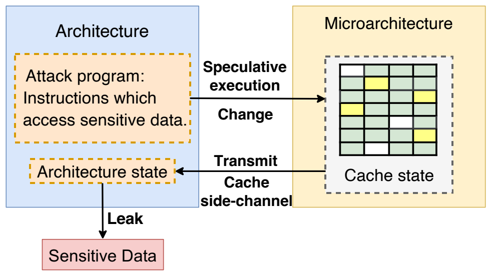


Figure 3: Speculative cache side-channel attack process.

3 Speculative Cache side-channel Attacks

This paper focuses on defending against the speculative cache side-channel attacks, which simultaneously exploit both speculative execution and cache side-channel vulnerabilities of processors. Speculative execution attack exploits the side effects of the transient instructions which are mis-speculated and destined to be squashed. The attack program speculatively executes load instructions, which leak sensitive data to the covert channel before they are squashed. A cache side-channel attack can establish cache side channels, thus providing a covert channel for the speculative execution attacks. Figure 3 shows the general process of such attacks. An attacker exploits the speculative execution mechanism to trigger transient execution of instructions, which access victim's sensitive data, causing changes in cache state. Then, the data is transmitted from the microarchitectural state to the architectural state through a cache side channel, thereby leaking the sensitive data.

3.1 Cache Side-channel Attack

A cache, consisting of SRAM, has a small capacity but is much faster than memory composed of DRAM, approaching the speed of CPU. Modern processors use a hierarchy of successively smaller but faster caches to bridge the speed gap between processor and memory. In a multi-level cache, the last-level cache (LLC) is shared by all the cores in the processor. For example, in a typical Intel processor, cache is

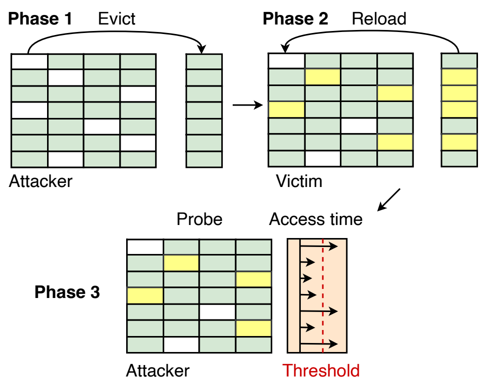


Figure 4: Side-channel attack process. In each phase, the left side of the figure represents the cache, while the right represents the evicted cache lines. Lattice in green: a normal cache line; lattice in white: the cache line at this location has been evicted by the attacker; lattice in yellow: the cache line accessed by the victim.

divided into three levels, with the L1 and L2 caches individually owned by each core, while the LLC is shared among all cores. When the processor needs data, it first checks whether a copy of the data is present in the top level of cache hierarchy, i.e., the L1 cache. If a copy is found, a cache hit occurs; otherwise a cache miss occurs, and the processor repeats the procedure to try to retrieve data from the next-level cache, and finally from the memory. Retrieving data from memory or upper-level caches closer to memory takes longer than retrieving data from upper-level caches closer to the core. The cache side-channel attack in our threat model is an attack that exploits such timing differences [30].

Attackers exploit the variation in cache state to establish a cache side channel. The attack process usually contains three phases, as shown in Figure 4. First, an attacker evicts the monitored cache lines from the cache hierarchy. Second, a victim program is executed. In the third phase, the attacker probes the time it takes for accessing the monitored cache lines. If the victim program has already accessed the monitored cache lines in the second phase, the cache lines represented by the yellow lattices will be loaded into the cache, resulting in a shorter time for the attacker to access them; otherwise, the attacker's access time will be longer. Hence, by measuring and comparing the access time, the attacker can determine whether the victim program has accessed the monitored cache line to obtain sensitive information. Based on the method of evicting the monitored cache line, cache side-channel attacks can generally be divided into two types [22]:

Flush-Based Attack: An attacker uses specialized machine instructions, such as the x86 `CLFLUSH` instruction used in Flush+Reload [55], to flush the target cache line out of the shared cache between the attacker and victim. This type of attack also includes Flush+Flush [21], Flush+Coherence [54], Reload+Refresh [9], etc.

Conflict-Based Attack: An attacker constructs a conflict set to evict the target cache line. For example, in Evict+Reload [34], the attacker forces contention in the cache set by accessing dummy data and loading it into the cache set, causing the processor to evict the target cache line due to the limited size of the cache. This type of attack also includes Evict+Time [27], Prime+Probe [53], Prime+Abort [11], etc.

3.2 Spectre Attack

Spectre attack is a representative one of speculative cache side-channel attacks. In different variants of Spectre attack, an attacker mistrains the CPU's predicted components, which causes the CPU to transiently violate program semantics by executing an instruction that would not normally execute, thereby triggering a transient execution.

In Spectre v1 [30], the attacker mistrains the pattern history table, which determines whether the conditional branch instructions (e.g., `bne` and `bnq` in RISC-V) should take the branch or not. As shown in the following code example:

```
if (x < array1_size)
    y = array2[array1[x] * L1_BLOCK_SZ_BYTES];
```

the attacker maliciously selects a value of `x` that is out of the `array1`'s range, causing `array1[x]` to be resolved to a secret byte `k` in the victim's memory, where `x` = (the address of the secret byte to be read) - (the base address of the array). Furthermore, the attacker ensures that the processor cannot immediately obtain the value of `array1_size` (because it is not in cache or requires complex computations), causing the branch condition to wait for the uncached parameter. As a result, the processor executes the instructions inside the conditional branch statement under mis-speculation, leaking the byte `k` into cache.

In Spectre v2 [30], the attacker mistrains the branch target buffer of indirect branch instructions (e.g., `jalr` in RISC-V and `MOV` in ARM) with malicious target addresses corresponding to code snippets called gadgets, which can leak sensitive data, as shown in the following code example:

```
void victimFun(uint64_t x)
{
    uint64_t y = array2[array1[x] * L1_BLOCK_SZ_BYTES];
}
```

During the training phase, the attacker makes the branch predictor mispredict a branch from the indirect branch instruction to the address of `victimFunc`. During the attack phase, similar to Spectre v1, the attacker maliciously chooses a value of `x` that is out of the `array1`'s range and delays the processor's access to the target address of the indirect branch instruction, causing the processor to speculatively execute `victimFunc` and leak the secret byte `k` into the cache.

In Spectre v4 [24], the attacker exploits the misprediction of the memory disambiguator. Modern processors adopt an optimization called speculative store bypassing to further improve performance, where the memory disambiguator is used to speculate which load instructions do not depend on any preceding store instructions. Therefore, the attacker can exploit the memory disambiguator to trigger transient executions of unsafe load instructions, allowing access to sensitive data. Considering the following code snippet:

```
ptr = secret_ptr;
ptr = general_ptr;//store operation
x = *ptr;//load operation
y = array2[x];
```

for a read-after-write (RAW) dependency operation, the attacker prevents the processor from immediately obtaining the value of `general_ptr` in the store operation `ptr = general_ptr`. Therefore, in the attack stage, the load operation `x = *ptr` should return the value of `general_ptr`. But since the value of `general_ptr` cannot be immediately obtained (because it is not in cache or requires complex computations), the processor will speculatively execute the load operation, causing `x` to take the value of `secret_ptr`, which leaks `secret_ptr` into cache.

In Spectre v5 [31], also known as SpectreRSB, the attacker exploits the return stack buffer (RSB), which is a hardware stack used to track the return addresses of previous call instructions. When encountering a `ret` instruction, the processor speculatively uses the top address of the RSB as the return address, as accessing the software stack is slower. Therefore, the attacker can exploit the RSB to supply CPU with a malicious return address corresponding to a gadget (the same as in Spectre v2), resulting in speculative execution of the gadget, thereby allowing the gadget to leak sensitive information via a cache side channel.

4 Threat Model

We focus on eliminating cache side channels in speculative execution established by flush- and conflict-based cache side-channel attacks, which require pre-evicting the cache lines to be leaked. Overall, we make the following assumptions:

- Attackers are allowed to run arbitrary code before and during the victim's executions to affect the victim's speculation, for example, by altering the state of RSB, branch predictor, and memory disambiguator.
- Attackers are aware of the cache index method and replacement strategy, enabling them to evict target cache lines from the cache.
- Attackers can locate gadgets within the victim's executable memory space.
- All these assumptions hold under the condition that the processor hardware and operating system work correctly.

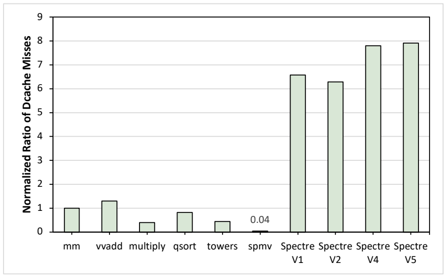


Figure 5: Normalized ratio of L1D cache misses.

We do not consider other speculative side channels, such as TLB [20], port contention [4,7], etc. Speculative side channels based on TLB can be mitigated through alternative secure TLB architectures, e.g., the one proposed by Deng et al. [14]. Speculative side channels based port contention can be mitigated by turning off SMT or by employing the defense scheme proposed in [32]. We neither consider non-speculative side channels [10, 15], nor physical side channels [29, 44] (which are generally noisier, requiring longer time for attackers to observe the effects).

5 Speculative Loads Protected in SpecLFB

In modern OoO processors, to reduce hardware cost and complexity, all instructions are considered as speculative executions until they reach the head of ROB, and any load of these instructions initiating memory reads is a speculative load [52]. Transient execution allows speculative loads to load data into cache, which may cause security issues. Although the processor can handle exceptions or squash these loads after validating the speculation, the cache state in the microarchitecture has already been changed, allowing attackers to establish cache side channels to leak sensitive information. To prevent the establishment of cache side channels, the speculative loads, which are potentially exploited by attackers, should be protected to prevent them from changing the cache state.

Based on the principle of speculative cache side-channel attacks (cf. Section 3), the cache lines that are exploited to leak data by the attacker must be evicted in the first phase before they can be refilled by the speculative loads of the victim, thereby causing cache misses in the second phase and then creating observable timing differences in the third phase, as shown in Figure 4. Figure 5 shows that the ratio of data cache misses during the executions of Spectre v1, v2, v4, and v5 attack programs in SonicBOOM is significantly higher than that of the normal programs from the benchmarks provided by the RISC-V toolchain [1]. Furthermore, in the simulations of attacks on SonicBOOM using Chipyard, the output log indicates that all the unsafe loads exploited by the attacker in the victim code generate cache misses. Therefore, to defense against such attacks, the speculative loads causing cache misses are considered unsafe and should be protected by delaying their execution. In this paper, we use MUSL to denote an unsafe speculative load that causes cache miss.

MUSLs can be confirmed as safe without needing to reach the head of ROB. Delaying the execution of MUSLs until they reach the head of ROB is overly conservative. Take the MUSLs in Spectre v1 for example. They can be safe as soon as the branch resolves in a correct prediction. This is the earliest point when the data returned by speculative loads is no longer considered unsafe. Therefore, to save performance as much as possible, it is necessary for processors to confirm MUSLs as safe as early as possible.

When processors can confirm MUSLs as safe depends on different speculation sources. According to [51], there are mainly the following sources:

- Control flow prediction. It predicts the execution path that a program will follow through the branch prediction unit (BPU), which may generate speculative loads in unresolved control flows.

- Address speculation. It predicts the data dependencies between loads and stores when the physical address is not fully available, which may generate speculative loads in an unresolved memory access order.

- Value prediction (VP). It predicts the result of a μop based on the execution history allowing dependent instructions to continue their execution without waiting for the result to become available, which may generate speculative loads in an unresolved value.

- Other exceptions. Additionally, we also consider other exceptions that may lead to speculative execution and correspond to speculative loads in other exceptions.

We have run in the SonicBOOM core the programs of RISC-V benchmarks [1] to validate the types of speculative loads. Since the current version of SonicBOOM does not implement VP, Figure 6 only shows the normalized ratio of the speculative loads resulting from the other three sources, where one can find that most speculative loads result from the unresolved control flow and memory access order.

Based on the above classification and analysis, we further precisely elaborate the MUSLs to be protected as follows:

1. *MUSLs in an Unresolved Control Flow*

   When the branch condition or target address of control instructions, such as branches and jumps, is unknown or has been predicted but not yet verified, the speculative loads that follow these unresolved branch instructions are the MUSLs to be protected. When the conditions and addresses can be verified by the processor, the MUSLs can be transformed into safe loads, if the speculative execution is in the correct path. Otherwise, they will be squashed at the end of the transient execution window.

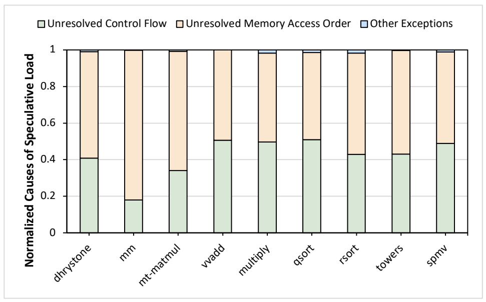


Figure 6: Normalized ratio of speculative loads. These statistics only consider the oldest reason that caused the speculative load since there might be multiple reasons.

2. *MUSLs in an Unresolved Memory Access Order*

   Two scenarios are discussed based on two different reasons causing such MUSLs, respectively.

   Memory Dependency. During memory access, waiting for the results of stores with unresolved addresses increases the processor's latency. To further improve performance, the memory disambiguator of the processor is used to predict which loads do not depend on any previous stores. When it determines that a load has no such RAW dependencies, the load will be speculatively executed. Therefore, the speculative loads causing cache misses that follow the stores are the MUSLs to be protected. Such a MUSL can be transformed into a safe load, if it is later discovered that the store which the MUSL follows points to a different address from that of the MUSL. Otherwise, the MUSL will be squashed at the end of the transient execution window.

   Memory Consistency Model. The memory consistency model specifies the order in which a core executes memory operations and is observed by other cores in a shared memory system. When a memory operation is committed, it exposes itself to the memory hierarchy and other cores in the system. Therefore, those instructions that are not committed in the program order must ensure that they comply with the defined memory consistency model [6]. Read operations can read data from memory and are executed out of order before reaching the ROB head, indicating that some loads will be speculatively executed before earlier loads and stores in the ROB. However, for a memory consistency model that does not allow store-load or load-load reordering operations, the speculative loads may be squashed due to violating the model. For example, in the total-store-order model, which does not allow store-store or load-load reordering, if the execution of an older load is delayed, a younger load may be executed speculatively. The speculative load that generates cache misses in this scenario is a MUSL to be protected. If an invalidation is received when the older load has finished its execution, the MUSL will be squashed. Otherwise, the MUSL can become safe.

3. *MUSLs in an Unresolved Value*

   With VP, the processor first determines whether the result of a μop is predictable based on the execution history. If it is considered to be predictable, the processor may execute the dependent instruction using the predicted source operand value in parallel with the instruction which generates the required result. Therefore, the speculative loads causing cache misses generated by these instructions are the MUSLs to be protected. Such a MUSL can be transformed into a safe load, if the prediction is validated to be correct. Otherwise, the MUSL will be squashed at the end of the transient execution window.

   Remark: To the best of our knowledge, both Gem5 and SonicBOOM at their current versions where we implement SpecLFB do not support VP. Because the ROB unsafe mask in our design (as shown in Section 6.2) has to be set according to the specific microarchitecture which supports VP, we have no practical implementation of SpecLFB which considers VP in either Gem5 or SonicBOOM.

4. *MUSLs in Other Exceptions*

   For the instructions that can cause other exceptions, such as unauthorized data access, memory operations with unknown addresses, arithmetic operations with overflow, etc., they will not be squashed until the processor can check for the exceptions. Therefore, the loads causing cache misses in the subsequent instructions that are speculatively executed during this period are the MUSLs to be protected. If any exception exists after the processor checks, the MUSLs will be squashed. Otherwise, the MUSLs can be transformed into safe loads.

6 SpecLFB Design

6.1 Design Overview

In non-blocking caches, MSHR is an essential microarchitecture component between different levels of cache hierarchy and memory. The lower-level cache is requested by MSHRs to retrieve the missing data of MUSLs. Therefore, SpecLFB can be reused with the same defense mechanism in the LFBs of different cache levels. For the convenience of illustration, we use the L1D cache as an example scenario. Generally, the response data is refilled into the L1D cache via LFB. While in a processor that adopts the proposed SpecLFB, as shown in Figure 7, the response data must pass through a security check mechanism introduced to LFB. Only when the security check mechanism determines that it is safe, can the data be refilled into the L1D cache, thus preventing the execution of MUSLs from altering the cache state. To ensure the correctness and immediacy of the security check, a set of judgments are required, such as whether the response data is loaded by MUSLs, or whether MUSLs can be converted into safe loads to continue refilling response data into the L1D cache. This is essential to further avoid the scenarios where safe data are misjudged as the data loaded by MUSLs or where the data loaded by MUSLs that have been converted into safe loads still fail in passing through the security check, which results in longer execution delays of the related instructions and degraded processor performance.

In order to facilitate the judgment in the security check mechanism, an ROB unsafe mask is designed as shown in Figure 7. The bits of the mask correspond to the entries in the ROB with a one-to-one mapping, which is set and changed according to the `unsafe` parameter of μop and the various information and exceptions returned by execution units (EU) and LSU. Parameter `unsafe` is a boolean variable. If an instruction meets one of the following three conditions: 1) it uses the load queue, 2) it uses the store queue (excluding a fence instruction that isolates the previous and subsequent store operations to ensure sequential execution), or 3) it is a branch/jump instruction, then the `unsafe` parameters of the instruction's μops are set to true. If the value of a bit in the ROB unsafe mask is 1, it means that the instructions in the ROB entry to which the bit corresponds are currently in an unsafe state, i.e., these instructions may generate MUSLs; otherwise, they are in a safe state, i.e., they will not generate MUSLs. After LFB obtains the missing data requested by the L1D cache, it judges whether the data can pass the security check mechanism according to the ROB unsafe mask bit corresponding to its related instruction. Only a mask bit of 0 can it pass the check and continue refilling the data into the L1D cache. By exploiting SpecLFB to delay the execution of MUSLs, the processor can eliminate the need for additional data structures to hide the cache lines fetched by speculative loads and data movement operations. Additionally, since the cache line already exists in the L1 upon a cache hit, no additional coherence mechanism is required.

6.2 Detailed Design

6.2.1 ROB unsafe mask

The ROB unsafe mask has the same number of bits as the number of entries in ROB, and a one-to-one mapping exists between the ROB entries and the mask bits. The design details of the ROB unsafe mask vary depending on the specific processor architecture. As shown in Figure 8, the ROB entries in SonicBOOM are divided into multiple banks, and each row (corresponding to a small cell in Figure 8) of a bank stores the information of one instruction [46]. Since the addresses of the instructions dispatched to the ROB are always contiguous, they are stored in different banks of each entry in the ROB in

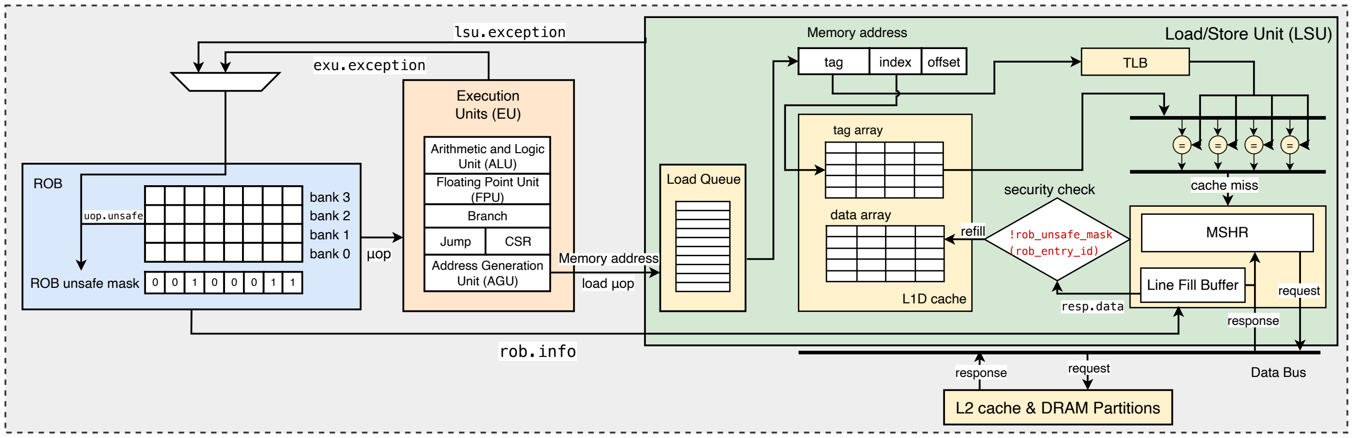


Figure 7: Design overview of SpecLFB.

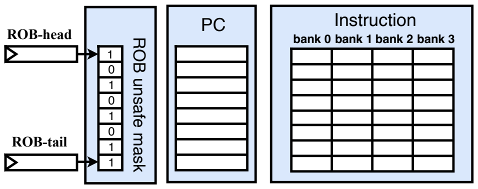


Figure 8: ROB unsafe mask structure.

the order of arrival, and only the addresses of the instructions in bank 0 of each entry are stored to save hardware overhead. Branch instructions or jump instructions are dispatched separately to an entry to ensure the continuity of instruction addresses in each entry of the ROB. Therefore, the value of each bit in the ROB unsafe mask is not only determined by the instruction in bank 0, but also jointly determined by the instructions in all banks of that entry. Once there is a possibility that any instruction in the entry may generate MUSLs, the corresponding bit in the mask for that entry is set to 1.

Let M and N denote the number of entries and that of banks in the ROB, respectively. Let r_i represent the ith ($i \in [0, M-1]$) bit of the ROB unsafe mask, and let b_{ij} represent the security state of the instruction stored in the jth ($j \in [0, N-1]$) bank of the ith entry of the ROB. If the corresponding instruction is possible to generate MUSLs, i.e., it is unsafe, then $b_{ij} = 1$; otherwise, $b_{ij} = 0$. Then, r_i is

$$r_i = b_{i1} \,||\, b_{i2} \,||\, ... \,||\, b_{ij} \,||\, ... \,||\, b_{i(N-1)} \,. \tag{1}$$

6.2.2 Rules for updating ROB unsafe mask

For an instruction stored in the jth bank of the ith entry in the ROB, if the value of the `unsafe` parameter of the instruction's μop is true, the corresponding b_{ij} is initialized to 1. The load of the instruction resulting in cache miss is considered as a MUSL temporarily, so that the data loaded by the MUSL cannot pass through the security check mechanism when being refilled into the L1D cache through the LFB. Based on the elaboration of MUSLs in Section 5, the update rules for corresponding b_{ij} of the instruction are as follows:

- For the instructions in unresolved control flow, according to the information returned by BPU, SpecLFB traverses ROB in each cycle to update b_{ij} depending on whether the instructions are still under the control flow. Specifically, if there are no older control instructions that are unresolved before the instructions, the b_{ij} of the instruction should be updated to 0; otherwise, they should be set to 1. In practice, general processors would provide interfaces for obtaining more detailed control-flow-related information. For example, in SonicBOOM, each μop has a `br_mask` parameter to record which speculated branches it belongs to. When traversing ROB, SpecLFB checks if the branch the current μop belongs to has been resolved. If so, the b_{ij} of the instruction should be updated to 0.

- For the instructions in unresolved memory order, SpecLFB tracks data dependencies through the load queue (LDQ) and store queue (STQ) in the LSU. When placing a memory access instruction (denoted as $\mathcal{A}$) into the LDQ/SDQ, SpecLFB compares it with the instructions in every entry of the LDQ/STQ for checking data dependencies. If a data dependency is not found with an older store/load instruction, for example, there is no older store instruction that accesses the same memory address as instruction $\mathcal{A}$, the b_{ij} of the instruction should be updated to 0. SpecLFB updates the data dependencies between instructions when an instruction is dequeued from the LDQ/SDQ.

- For the instructions related to other exceptions, once the ROB receives exception information from backend units such as execution units and LSU, the b_{ij} of the instruction and younger instructions cannot be updated to 0 until the exception has been handled.

When all b_{ij} of the ith ROB entry are 0, its r_i can be updated to 0. Then, the data loaded by the MUSLs in the instructions of that ROB entry can pass the security check of the LFB and be refilled into the L1D cache. While SpecLFB promptly updates the b_{ij} of instructions when the correctness of speculation has been verified, the delays introduced by traversing the ROB queue and the STQ/LDQ to confirm the safety of related instructions are inevitable. Nevertheless, these delays are still relatively shorter compared to other solutions. For STT, unsafe speculative instructions are allowed to continue their execution when the youngest tainted root of the instructions reaches a visibility point, where there are the oldest unsafe speculative instructions. In addition to the delay for computing the instruction's tainted root and traversing the ROB, there is also a delay caused by the fact that the visibility point could be updated only when the related instruction is committed. For SEE-RV, it uses the point-of-no-return pointer in the ROB to track the oldest unsafe speculative instruction. This approach helps avoid the delay for computing the instruction's tainted root. However, it causes the instructions to be delayed until all older speculative instructions become safe. This delay only occurs for the instructions in unresolved control flow in SpecLFB. Furthermore, during the waiting period for SpecLFB to update the ROB unsafe mask, the MSHR can simultaneously request data of MUSLs from the lower-level caches. As soon as the protection is disabled, the requested data can be refilled directly from the LFB into the L1D cache instead of the lower-level caches or memory, and thus without causing too much delay. In the case of a processor validating incorrect speculation, the relevant instructions have been squashed from the pipeline, and the relevant data in LFB will become invalid before changing the state of the L1D cache. The processing of these data only needs to follow the original rollback mechanism of processors.

6.3 SpecLFB Security Analysis

SpecLFB keeps track of speculative loads through ROB unsafe masks. Initially, all speculative loads are marked as unsafe because their correctness is not yet verified. The ROB unsafe mask ensures that speculative loads are considered safe only when specific conditions are met (e.g., control flow resolution, memory access order resolved, exception handling completed), which is thoroughly determined based on the processor's speculation sources (cf. Section 5) and the working principle of microarchitecture. The ROB unsafe masks currently designed do not consider MUSLs in VP, making the tracking of unsafe speculative loads accurate only on processors without VP implementation. For processors that implement VP, the related update rules of ROB unsafe masks need to be set in accordance with their specific implementation.

Take Spectre attack based on evict+reload as an example. Figure 9 shows how SpecLFB blocks the establishment of cache side channels, where each lattice denotes a cache line.

- In phase one, the attacker first evicts the specified cache lines from the cache.

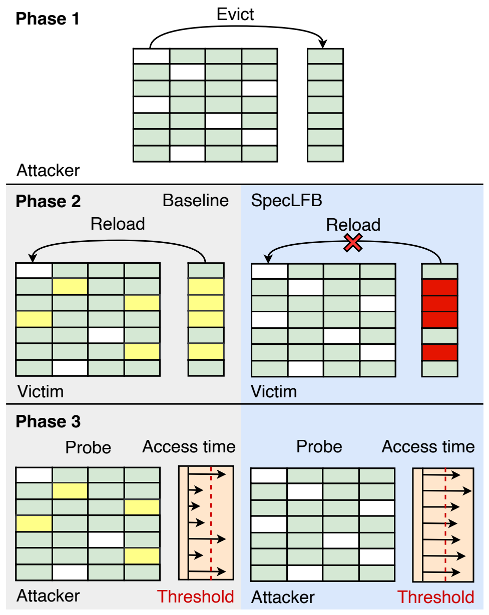


Figure 9: The process of Evict+Reload attacks in Baseline and SpecLFB. The red lattices represent the cache lines accessed by the victim speculatively. For more details of the figure please refer to Figure 4.

- In phase two, the victim runs the program under the conditions created by the attacker. For example, in Spectre v1 and v2, the attacker mistrains the branch predictor to speculate along a wrong path, thereby triggering a transient execution. The cache lines represented by the yellow lattices that have already been evicted in phase one but need to be accessed during the transient execution will be reloaded into the cache in phase three. However, in the processors that adopt the SpecLFB scheme, these cache lines represented by the red lattices will be regarded as data loaded by MUSLs. During the transient execution period (i.e., waiting for the processor to verify speculation), these cache lines cannot pass through the security check to be refilled into the cache.

- In phase three, the attacker probes the time it takes to execute a read at the memory address corresponding to the cache line evicted in phase one. The cache lines that have been accessed by the victim in phase two are reloaded into the cache, resulting in a significantly shorter access time for the corresponding yellow lattices than the cache lines that are not in the cache (represented by the white lattices). In the processors that adopt the SpecLFB scheme, the cache lines that have been accessed by the victim in phase two are not refilled into the cache. This makes it impossible for the attacker to determine which cache lines have been accessed by measuring the access time, thus preventing the leakage of sensitive data.

7 Experimental Setup

We use Chipyard v1.8.0 to generate RTL for each of the three SonicBOOM-based processors shown in Table 1, and then run the simulations through Verilator v4.210. The original SonicBOOM is selected as the unsafe baseline. For the comparison in this paper, we have also implemented SSE-RV [39] in the latest SonicBOOM[1]. As introduced in Section 10, SSE-RV is a hardware-level taint tracking protection mechanism that improves upon STT [57]. Similar to our approach, it prevents the propagation of sensitive data by delaying speculative instructions. Then by running Vivado, a Xilinx software suite for hardware design synthesis and analysis, the three SonicBOOM cores have been successfully synthesized with their hardware resource overheads obtained.

Besides, to achieve a more realistic evaluation in terms of processor performance, we have built a hardware prototype based on Xilinx EK-KC-705 FPGA [2] platform (with a clock frequency of 50MHz) burned with the above three SonicBOOM cores. The prototype framework is shown in Figure 10. The common parameters of the three SonicBOOMs are shown in Table 2. Then a Linux-kernel-based operating system, Debian, is booted in the prototype, based on which SPEC2017 has been run for a comprehensive evaluation of the three processors' performance.

Due to the fact that SonicBOOM's L2 cache uses an open-source simple cache component which is beyond the scope of the SonicBOOM core that we can modify, we evaluate the performance of SpecLFB applied in all cache levels based on the SpecLFB's Gem5 [38] implementation, where SpecLFB is applied to both levels of the cache in the O3 CPU model in Gem5. Additionally, we compare the performance of SpecLFB to the state-of-the-art defense, STT [57], which was also implemented in Gem5. Table 1 provides the configurations of the simulated processors in Gem5, and the parameter settings for each component of Gem5 are detailed in Table 2.

To evaluate the effectiveness of the proposed defense scheme, we cross-compile proof-of-concept (PoC) programs of Spectre attacks both for the RISC-V instruction set architecture (ISA), including Spectre v1, v2, v4, and v5, and the X86 ISA, including Spectre v1 and v4. The programs of the RISC-V ISA have been run in both the Chipyard simulator and the FPGA hardware prototype, while the programs of the X86 ISA have been run in Gem5 simulator.

[1]SSE-RV was originally implemented in an earlier version of SonicBOOM, which has some differences in handling 32-bit instructions and cannot be adapted to Chipyard v1.8.0.

Table 1: Processor configurations.

Processor	SonicBOOM Configurations	Gem5 Configurations
Baseline	Original SonicBOOM	Original O3 CPU
SSE-RV	SonicBOOM enhanced with SSE-RV	
STT		O3 CPU enhanced with STT
SpecLFB	SonicBOOM enhanced with SpecLFB	O3 CPU enhanced with SpecLFB

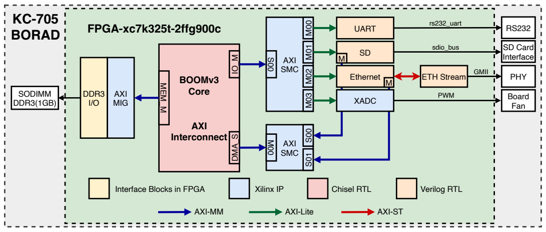


Figure 10: Prototype framework.

Table 2: Gem5 and SonicBOOM-FPGA parameters.

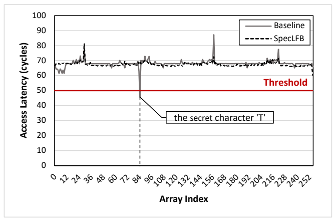
Parameter	SonicBOOM-FPGA	Gem5
ISA	RV64	X86-64
Frequency	FPGA@50MHz	simulate@2GHz
Processor type	2-decode 4-issue MediumBoom O3CPU	8-decode 8-issue DerivO3CPU
ROB/LDQ/STQ	64/16/16 entries	192/32/32 entries
L1I Cache	16KB, 4-way, 64B line	32KB, 8-way, 64B line, 4 MSHRs
L1D Cache	16KB, 4-way, 64B line, 2 MSHRs	32KB, 8-way, 64B line, 4 MSHRs
L2 Cache	512KB, 16-way, 64B line	2MB, 16-way, 64B line, 20 MSHRs

8 Evaluation

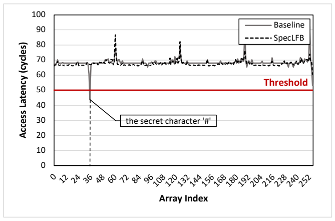
8.1 Attack Replication

To replicate the Spectre attacks introduced in Section 3.2, we adopt the Flush+Reload [55] and Evict+Reload [34] techniques to establish a cache side channel in X86 and RISC-V cores, respectively. In a practical attack scenario, the attacker can treat the content of any memory address they want to attack as an address which is essentially a numerical value in their own process, and actively access it in the code. Specifically, we construct a 256-element array (i.e., `array2` introduced in Section 3.2) in the attacker's code (since only one byte can be exploited at a time, with a value between 0 and 255). Then a corresponding array element is loaded into the L1D cache based on the value of the content at the attack address. For example, if the value of the content at the address is 0, the attack code will load the first element of the array into the Dcache. In this way, by detecting which element of the array is in the Dcache, the value of the content at the attack address can be determined.

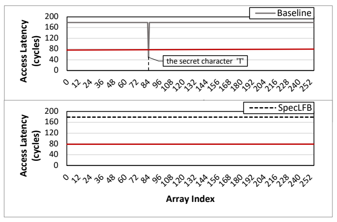
Loading memory data into cache is relatively straightforward, but it is not as easy for cache evict operations in the



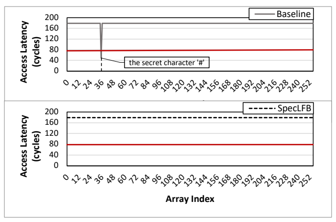
(a) Spectre v1 in Chipyard



(b) Spectre v4 in Chipyard



(c) Spectre v1 in Gem5



(d) Spectre v4 in Gem5

Figure 11: Access latency measured in the Spectre attacks through cache-based side channel.

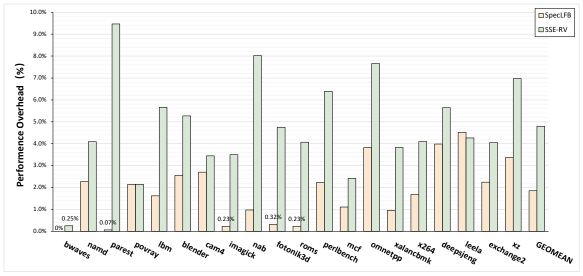


Figure 12: The performance overhead comparison between SpecLFB and SSE-RV [39] both running the selected workloads contained in SPEC2017.

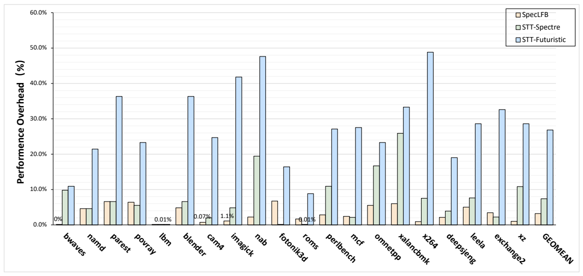


Figure 13: The performance overhead comparison between SpecLFB and STT [57] both running the selected programs contained in SPEC2017.

RISC-V as in the x86 ISA, since the current RISC-V ISA does not provide any instruction for achieving direct flush operations of cache lines like the `clflush` instruction in the x86 ISA. Therefore, we achieve an L1D cache evicting operation at a specified cache address by loading dummy data into the target location in the L1D cache to cover the original content, which is based on the working principle of Evict+Reload, for the replication of the attacks in RISC-V core.

8.2 SpecLFB Security Evaluation

Since Spectre v1 and v4 are associated with the MUSLs in unresolved control flow and those in unresolved memory access order, respectively, we choose them as two representative attacks to evaluate SpecLFB Security. Figure 11 shows the attacker's access latency of the elements in `array2` of Spectre v1 and v4 (cf. Section 3.2) during the third phase of the side-channel attack. Based on the measurement of the access latency taken for accessing data upon cache hits and cache misses, we set the latency threshold to 50 cycles for SonicBOOM in Chipyard and 80 cycles for the O3 CPU model in Gem5. After triggering a transient execution in the victim, the attacker probes the elements of `array2` and reports the access latency. As shown in Figure 11, for Baseline, only the accesses to the element of `array2` corresponding to the secret character hit in the cache with a time less than 50 cycles, while all the other accesses to the elements of `array2` cause cache misses; therefore, the attacker is able to obtain the secret character. In contrast, for SpecLFB, the access latency of the elements in `array2` is evenly distributed, without significant differences that fall below the threshold, as SpecLFB does not allow the data loaded by MUSLs to change the cache state until it is validated by the processor. After the speculation has been validated as incorrect, the element of `array2` corresponding to the secret character in LFB becomes invalid without being reloaded into the cache. Therefore, the attacker is unable to obtain the secret data.

A successful attack is defined as leaking one character during a PoC program run. In both Chipyard and Gem5, we run each of the evaluated attacks 100 times. The length of the secret string randomly generated exceeds 100 characters at each run. In the processors without any defense mechanism, the success rate of leaking the secret value is 100%. With our defense mechanism, the success rates in Chipyard and Gem5 drop to below 0.01% and to 0, respectively. Due to the software noise, three secret characters are detected by chance out of 40,000 attack attempts in Chipyard. Also for the FPGA hardware prototypes, all four attacks work successfully in the original SonicBOOM but fail to work in the SonicBOOM enhanced with SpecLFB. In summary, our mechanism provides strong and effective protection against these attacks.

8.3 Performance Evaluation

We run SPEC2017 in the FPGA-based hardware prototype and Gem5 to evaluate our mechanism. In the FPGA-based hardware prototype, performance overhead for a processor running a workload is defined as the workload's execution time normalized to that of the Baseline (as shown in Table 1), while in Gem5, it is defined as the IPC (Instructions per Clock Cycle) of the workload normalized to that of the Baseline. Figure 12 and Figure 13 show the comparison of the performance overhead between SpecLFB and SSE-RV/STT both running the selected workloads contained in SPEC2017, where the last pair of bars "GEOMEAN" in the figures shows their average performance overhead over all the executed programs.

Evaluation based on Hardware prototype. As shown in Figure 12, the average overhead of SpecLFB and SSE-RV is 1.85% and 4.8%, respectively. Compared to SSE-RV, SpecLFB has the advantage that only the instructions generating MUSLs will be delayed during the L1D cache refill stage. For the instructions becoming safe, SpecLFB can release its protection at an earlier point and the data related to the delayed instructions has already been in the LFB, which allows the delayed instructions to access the data more quickly when they continue to be executed. Therefore, for the programs with limited cache capacity demands, such as "imagick" and "parest", which can be executed well without regular accesses to the memory [23] resulting in few cache misses, the corresponding performance overhead of SpecLFB is much smaller. In contrast, SSE-RV incurs a larger performance overhead for the programs with a large number of memory access operations, such as "parest".

Evaluation based on Gem5. We compare the IPC performance of SpecLFB with that of STT in two protection modes, i.e., STT-Spectre (STT-*Sp*) and STT-Furistic (STT-*Fu*). As shown in Figure 13, STT-*Sp*, which aims to defend against Spectre-type attacks, incurs an average overhead of 7.36%. However, as indicated in Table 3, STT-*Sp* only considers control-flow prediction, while STT-*Fu* considers all speculative sources, resulting in a significantly higher average overhead of 26.82%. In contrast, when considering more speculative sources compared to STT-*Sp*, SpecLFB introduces only a 3.20% average overhead. The reason for such an overhead difference is similar to SSE-RV. On one hand, STT conservatively delays all unsafe speculative loads, whereas SpecLFB only delays those that cause cache misses. On the other hand, the relatively complex tracking logic and visibility point of STT cause more delays, which further delay the execution of unsafe speculative instructions. Additionally, SpecLFB allows the delayed instructions to access data earlier after the protection is disabled. However, it cannot be denied that SpecLFB achieves some performance improvements compared with STT and SEE-RV by narrowing down the defense scope. For protected speculative loads, SpecLFB specifically protects speculative loads causing cache misses, while STT-*Sp* and STT-*Fu*, as well as SEE-RV also consider speculative loads causing cache hits. For blocked covert channels, SpecLFB specifically blocks covert channels through cache, while STT-*Fu* is able to block any covert channel.

FPGA resources utilization. Based on the same core parameter configurations, SoC bus, and peripheral devices, we synthesize the resources usage of the three processors using Vivado 2021.2 based on the FPGA hardware prototype. As shown in Table 4, SpecLFB only adds less than 1% of resources compared to the baseline, and consumes fewer resources than SSE-RV, especially in terms of LUTs (Look-Up-Tables) and FFs (Flip-Flops). LUTs are primarily used to implement logic circuit functionality, and FFs are sequential logic elements used for storing and transferring binary data in digital circuits on FPGAs. The high LUT utilization of SSE-RV is mainly due to the additional logic in the taint initialization and propagation stages, which is more complex than the security mechanism of SpecLFB. The high FF utilization is due to the taint file, propagation queue, and intermediate registers, while SpecLFB only adds intermediate registers related to the ROB unsafe mask. Therefore, SpecLFB provides a lower additional hardware resource overhead while addressing security issues. The recent transient execution attack defense work ProSpeCT [13] also implements its defense hardware prototype on the open-source OoO RISC-V processor Proteus [8] and deploys it in FPGA for resource evaluation. As shown in Table 4, when the size of BOOM and Proteus are quite different, SpecLFB has advantages in terms of both the absolute number and relative proportion of additional resource overhead. ProSpeCT is proposed as a universal secure speculative processor model encompassing the security semantics of all current speculative mechanisms. The ProSpeCT model is implemented as a Proteus plugin, with some additional modifications in the base processor. Such plugin implementation cannot constrain resource overhead tailored to the specific processor implementation, which causes the hardware resource overhead to be large and sensitive to the specific implementation of the processor and the processor size. SpecLFB only adds specified logic to the micro-architecture already existing in the processor, and thus has obvious resource overhead advantages.

Table 3: Comparison of STT [57], SpecLFB and SSE-RV [39].

Scheme	Performance overhead	Evaluation Method	Speculation			
			Value Prediction	Control Flow	Memory Access	Exception
STT-*Sp*	8.5%	Gem5	✗	✓	✗	✗
STT-*Fu*	14.5%		✓	✓	✓	✓
SpecLFB	3.20%		✗	✓	✓	✓
	1.85%	RTL-FPGA				
SEE-RV	4.8%		✗	✓	✓	✗

9 Limitations and Discussions

Speculative accesses resulting in cache hits. SpecLFB defends against speculative cache side-channel attacks by preventing cache lines speculatively loaded by the victim under

Table 4: FPGA resources utilization.

Scheme	Core	Device	LUTs	FFs
Baseline	SonicBOOM	Xc7a325T	169,463	93,994
SSE-RV	SonicBOOM	Xc7a325T	172,538 (+1.81%)	94,567 (+0.61%)
SpecLFB	SonicBOOM	Xc7a325T	170,765 (+0.77%)	94,283 (+0.31%)
Baseline	Proteus-O3	Xc7a35T	16,847	11,913
ProSpeCT	Proteus-O3	Xc7a35T	19,728 (+17.1%)	12,600 (+5.8%)

cache misses from being refilled into the cache. However, we cannot ignore the threat of speculative cache side-channel attacks where attackers may not need to evict the cache lines they want to leak in the first phase. This could lead to cache hits rather than cache misses when the victim executes speculative loads. For example, Xiong et al. [50] proposed a cache side-channel attack based on Least-Recently-Used (LRU) strategy, which does not require pre-evicting cache lines to be leaked. In its first phase, the attacker brings cache lines they want to monitor into the cache. In the second phase, the victim performs speculative accesses, leading to cache hits. In the third phase, the attacker needs to access a cache line that is not in the cache. The least-recently-accessed cache line is replaced based on the LRU strategy. Finally, the attacker determines whether the victim has speculatively loaded the monitored cache line based on whether it has been replaced.

Leakage through speculative loads under cache hits is out of the current defense scope of SpecLFB. However, we do not consider it a limitation of using LFB to ensure speculative execution security, as LFB can still be utilized to defend against such attacks. Since attackers need to manipulate the cache state, which often involves cache eviction and replacement, to directly or indirectly leak the speculative data, cache misses are inevitable in speculative cache side-channel attacks. For example, the third phase of the attack in [50] requires accessing a cache line that is not previously in the cache, triggering a cache replacement. Therefore, as long as the LFB can identify the loads that are exploited by the attacker to manipulate cache states, regardless of whether these loads are speculative or not, further defensive measures can be taken. Specifically, to defend against the attack described in [50], the cache entries where the cache lines accessed by unsafe speculative load instructions reside will be marked in the second phase. This marking needs to be removed when unsafe speculative loads become safe. The unsafe speculative load instructions can still be traced using SpecLFB's ROB unsafe mask. In such attack scenario, cache manipulation from the attacker does not have to happen speculatively, so interception cannot follow SpecLFB. Therefore, in the refill process of the third phase, if the unsafe speculative load has completed execution without causing a cache miss, the cache lines in the marked cache entries will be replaced through LFB with the highest priority and the marking will be removed. Otherwise, the logic used by SpecLFB to intercept speculative loads that cause cache misses will be reused. As a result, the original refill purely according to LRU in the third phase of the attack will be prevented. To the best of our knowledge, in Gem5 and SonicBOOM, LFB can specify the location in the cache where cache lines are refilled. For the speculative loads that become safe, the cache lines they accessed may be replaced from the cache by this solution, potentially affecting the memory access latency of dependent instructions. Additionally, because LFB cannot intercept the data loaded by speculative instructions under cache hits, it does not prevent speculative changes to cache metadata. Instead, it prevents attackers from detecting such changes, such as the third phase of the attack described in [50]. This solution prevents attackers from changing the cache occupancy state during the probe phase, such as eviction and replacement, making it more challenging for them to carry out such attacks.

Other speculative side channels. Besides cache side channels, there are other speculative side channels, such as TLB [20], FPU [18], port contention [7], etc., also posing a threat to processors in high-security scenarios. There have been defense mechanisms [37, 57] comprehensively considering all speculative sources, which prevent speculative results from leaving the CPU core and leaking into side channels. However, since they provide aggressive protection against all side channels, their significant performance overhead makes them unsuitable for being deployed in real-world processors. Although the cache is not dedicated to speculative side-channel attacks, it remains the most practical method of data exfiltration and is an example attack in Spectre [30]. SpecLFB restricts defense scope only to cache-based attacks to achieve a trade-off between performance overhead and defense scope. Besides SpecLFB, we will consider using other existing microarchitectures of CPU in future work to specifically prevent secrets from being leaked through other side channels, thereby achieving more defense capabilities of the system.

Limitations of the implementation and experimental verification. First, we currently do not have a practical implementation of SpecLFB that considers the speculation source VP. However, it is possible to be implemented in the processor where VP has been specifically supported. In such a processor, the SpecLFB' ROB unsafe mask could be set based on the elaboration of MUSLs in an unresolved value and the working principle of VP in the processor to provide protection, which we leave as a future work. Second, due to the specific defense scope of SpecLFB, we have not directly compared it with the solutions that have entirely the same defense scope in performance evaluation. Third, due to the cache constraints imposed by SonicBOOM, we further utilize Gem5 rather than RTL to evaluate the performance impact of SpecLFB when applied to all cache levels. As the RTL-FPGA based evaluation is obviously more accurate, which is also strong evidence of the practicality of SpecLFB in the real world, we also leave it as a future work to implement the SpecLFB applied to all cache levels in hardware prototype.

10 Related work

Existing mechanisms for mitigating speculative cache side-channel attacks can be classified into four types as shown below in detail, including 1) limiting the execution of speculative instructions, 2) making the results of unsafe speculative executions invisible to the microarchitectural state, 3) delaying the executions of unsafe speculative instructions, and 4) reducing the accuracy of the covert channel.

Limiting the execution of speculative instructions. Intel and AMD recommend using serialization instructions, e.g., `lfence`, in a branch [25, 48]. ARM introduces a full data synchronization barrier and instruction synchronization barrier that can be used to prevent speculation [5]. However, serializing every branch would be equivalent to disabling branch speculation entirely, severely degrading processor performance [25]. To counter the attackers' mistraining of speculation mechanisms, both Intel and AMD extended their ISAs with a mechanism to control indirect branches [26], but it can only target the Spectre variants based on branch poisoning. oo7 [45] is a method of statically analyzing binary programs, using taint analysis, address analysis, and modeling of speculative execution to check for potentially vulnerable code patterns. It mitigates speculative execution attacks by inserting fence instructions. Such a software mitigation relying on static analysis of a specific problem typically requires recompiling the program and has a low coverage.

Making the results of unsafe speculative executions invisible to the microarchitectural state. InvisiSpec [52] adds an SB to store the speculative data leading to security issues. If the speculation is incorrect, the data stored in SB becomes invalid. Otherwise, the speculative data will be re-installed into cache. As the access to SB does not participate in cache coherence protocols, InvisiSpec needs to reload its data for validation at the instruction committing stage, which results in significant performance overhead due to double memory access. Similarly, SafeSpec [28] stores speculated data in a fully associative shadow structure of the load/store queue, enabling recovery of any changes to the microarchitectural state, but the same data must be present multiple times in the shadow structure to prevent side-channel leakage, resulting in a high invalidation cost. MuonTrap [3] avoids double memory access by storing the result of speculative cache access in an L0 filter cache. However, the data movement caused by the re-installing operation still degrades processor performance. Therefore, CleanupSpec [40] allows the data to be installed into the cache during the speculative execution, and the replaced data is stored into a newly added data structure. The replaced data will be re-installed into the cache hierarchy to roll back the cache state only when speculation fails. ReversiSpec [49] proposes a comprehensive cache coherence protocol that takes into account speculative cache access. Compared to InvisiSpec, the SB only stores data when it is not in the cache, resulting in less data movement when a speculative load becomes safe. Overall, in comparison with the above mechanisms, SpecLFB does not require additional data structures to hide the cache lines fetched by the speculative loads or the data movement caused by the re-installing operations.

Delaying the executions of unsafe speculative instructions. Sakalis et al. [41] proposed the delay-on-Miss mechanism, which aims to delay the execution of speculated loads rather than hide the side effects of those speculative loads that have already been executed. Then they adopted a value predictor to improve the mechanism, which predicts the value of L1 cache misses to continue the speculations rather than delaying them, further improving performance in cases of low prediction success rates. However, the predictor also causes significant hardware overhead. STT [57] proposes a taint tracking technique, which marks physical registers to track the transient execution flow after a speculative memory access. Both NDA [47] and STT use data flow tracking similar to taint propagation to track instructions that may cause information leakage. These instructions are forced to be delayed until their related instructions become safe. SDO [56] added a safe VP mechanism based on STT to reduce overhead caused by the delay and tracking analysis. SSE-RV [39], based on STT, uses existing ROB pointers in SonicBOOM to track speculated instructions, simplifying the tracking analysis logic. However, it still requires a large taint file as a shadow structure for the PRF to implement tainting, and unnecessary instructions may still be incorrectly identified as unsafe instructions, which increases the delay of execution. In contrast, SpecLFB requires a smaller range of speculated instructions to be delayed and a shorter delay for each instruction, with minimal changes to existing hardware.

Reducing the accuracy of the covert channel. Reducing the precision of cache side channels can also mitigate cache side-channel attacks. Therefore, many web browsers lower the accuracy of timers in JavaScript by adding jitter or even removing some timers. However, Schwarz et al. [42] have shown that timers can be constructed in many different ways, and it is not practical to remove all of them. Disruptive Prefetch [17] and Prefender [36] add noise to cache side channels by utilizing the content brought into the cache by data prefetchers, but they can only achieve secure enhancement without fully defending against the attacks.

11 Conclusion

This paper proposed SpecLFB, a novel mechanism for defending against speculative cache side-channel attacks. It introduces a simple yet effective security check mechanism to the LFB, delaying the execution of unsafe speculative loads and preventing sensitive data leakage through cache side channels. To minimize the overhead of both processor performance and hardware resources, SpecLFB limits the scope of unsafe speculative loads that need to be delayed to MUSLs, which denotes unsafe speculative loads that cause cache misses. In addition, SpecLFB transforms them into safe loads and continues their execution as early as possible through the LFB unsafe security check mechanism based on the ROB unsafe mask we designed. We have implemented SpecLFB both in SonicBOOM and Gem5 O3 processor. Based on the evaluations through RTL simulation, FPGA hardware prototype experiment, and Gem5 simulation, SpecLFB has been shown to effectively block the establishment of cache side channels in Spectre attacks. It leads to a hardware resource overhead of only 0.6% and the performance overhead of just 1.85% in the FPGA hardware prototype experiment and 3.20% in the Gem5 simulation.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China (No. 61971131), in part by "Zhishan" Scholars Programs of Southeast University, and in part by the Ministry of Educations Key Lab for Computer Network and Information Integration, Southeast University, Nanjing, China.

References

[1] *Benchmarks for RISC-V Processor*. https://github.com/riscv-software-src/riscv-tests.

[2] *Xilinix EK-KC-705 FPGA*. https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[3] AINSWORTH, S., AND JONES, T. M. MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *2020 ACM/IEEE 47th International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 132–144.

[4] ALDAYA, A. C., BRUMLEY, B. B., UL HASSAN, S., GARCÍA, C. P., AND TUVERI, N. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (S&P)*, IEEE, pp. 870–887.

[5] ARM. *Cache Speculation Side-Channels*. Addison-Wesley, 2018. https://developer.arm.com/support/arm-security-updates.

[6] BELL, G. B., AND LIPASTI, M. H. Deconstructing commit. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004*, IEEE, pp. 68–77.

[7] BHATTACHARYYA, A., SANDULESCU, A., NEUGSCHWANDTNER, M., SORNIOTTI, A., FALSAFI, B., PAYER, M., AND KURMUS, A. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 785–800.

[8] BOGNAR, M., NOORMAN, J., AND PIESSENS, F. Proteus: An extensible risc-v core for hardware extensions. In *RISC-V Summit Europe, Date: 2023/06/05-2023/06/09, Location: Barcelona, Spain.*

[9] BRIONGOS, S., MALAGÓN, P., MOYA, J. M., AND EISENBARTH, T. Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *2020 USENIX Security Symposium (USENIX Security)*, pp. 1967–1984.

[10] BRUMLEY, B. B., AND TUVERI, N. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security* (2011), Springer, pp. 355–371.

[11] CANELLA, C., VAN BULCK, J., SCHWARZ, M., LIPP, M., VON BERG, B., ORTNER, P., PIESSENS, F., EVTYUSHKIN, D., AND GRUSS, D. A systematic evaluation of transient execution attacks and defenses. In *2019 USENIX Security Symposium (USENIX Security)*, pp. 249–266.

[12] CHEN, C., XIANG, X., LIU, C., SHANG, Y., GUO, R., LIU, D., LU, Y., HAO, Z., LUO, J., CHEN, Z., ET AL. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product. In *2020 ACM/IEEE 47th International Symposium on Computer Architecture (ISCA)*, pp. 52–64.

[13] DANIEL, L.-A., BOGNAR, M., NOORMAN, J., BARDIN, S., REZK, T., AND PIESSENS, F. Prospect: Provably secure speculation for the constant-time policy.

[14] DENG, S., XIONG, W., AND SZEFER, J. Secure TLBs. In *2019 ACM/IEEE the 46th International Symposium on Computer Architecture (ISCA)*, pp. 346–359.

[15] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), IEEE, pp. 1–13.

[16] FOG, A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering 2* (2012).

[17] FUCHS, A., AND LEE, R. B. Disruptive prefetching: impact on side-channel attacks and cache designs. In *2015 8th ACM International Systems and Storage Conference*, pp. 1–12.

[18] FUSTOS, J., AND YUN, H. Spectrerewind: A framework for leaking secrets to past instructions. *ArXiv e-prints* (2020).

[19] GONZALEZ, A., KORPAN, B., ZHAO, J., YOUNIS, E., AND ASANOVIC, K. Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In *2019 Third Workshop on Computer Architecture Research with RISC-V (CARRV).*

[20] GRAS, B., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *2018 27th USENIX Security Symposium (USENIX Security)*, pp. 955–972.

[21] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, Springer, pp. 279–299.

[22] GUO, Y., ZIGERELLI, A., ZHANG, Y., AND YANG, J. Adversarial Prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (S&P)*, IEEE, pp. 1458–1473.

[23] HASSAN, M., PARK, C. H., AND BLACK-SCHAFFER, D. A reusable characterization of the memory system behavior of spec2017 and spec2006. *ACM Transactions on Architecture and Code Optimization (TACO) 18*, 2 (2021), 1–20.

[24] HORN, AND JANN. *Speculative Execution, variant 4: Speculative store bypass, 2018.* https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[25] INTEL. *Intel Analysis of Speculative Execution Side Channels*. Addison-Wesley, 2018. https://software.intel.com/security-software-guidance.

[26] INTEL. *Speculative Execution Side Channel Mitigations*. 2018. https://software.intel.com/en-us/download/speculative-execution-sidechannel-mitigations.

[27] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! a fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17*, Springer, pp. 299–319.

[28] KHASAWNEH, K. N., KORUYEH, E. M., SONG, C., EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 ACM/IEEE 56th Design Automation Conference (DAC)*, IEEE, pp. 1–6.

[29] KHATAMIFARD, S. K., WANG, L., DAS, A., KOSE, S., AND KARPUZCU, U. R. Powert channels: A novel class of covert communicationexploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 291–303.

[30] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., ET AL. Spectre attacks: Exploiting speculative execution. *Communications of the ACM 63*, 7 (2020), 93–101.

[31] KORUYEH, E. M., KHASAWNEH, K. N., SONG, C., AND ABU-GHAZALEH, N. B. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *2018 USENIX Security Symposium (USENIX Security)*.

[32] KORUYEH, E. M., SHIRAZI, S. H. A., KHASAWNEH, K. N., SONG, C., AND ABU-GHAZALEH, N. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 39–53.

[33] LI, C., SONG, S. L., DAI, H., SIDELNIK, A., HARI, S. K. S., AND ZHOU, H. Locality-driven dynamic gpu cache bypassing. In *2015 29th ACM on International Conference on Supercomputing*, pp. 67–77.

[34] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. In *2016 USENIX Security Symposium (USENIX Security)*, pp. 549–564.

[35] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., ET AL. Meltdown: Reading kernel memory from user space. *Communications of the ACM 63*, 6 (2020), 46–56.

[36] LIT, L., HUANG, J., FENG, L., AND WANG, Z. PREFENDER: A Prefetching Defender against Cache Side Channel Attacks as A Pretender. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, pp. 1509–1514.

[37] LOUGHLIN, K., NEAL, I., MA, J., TSAI, E., WEISSE, O., NARAYANASAMY, S., AND KASIKCI, B. {DOLMA}: Securing speculation with the principle of transient {Non-Observability}. In *2021 30th USENIX Security Symposium (USENIX Security)*, pp. 1397–1414.

[38] LOWE-POWER, J., AHMAD, A. M., AKRAM, A., ALIAN, M., AMSLINGER, R., ANDREOZZI, M., ARMEJACH, A., ASMUSSEN, N., BECKMANN, B., BHARADWAJ, S., ET AL. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[39] SABBAGH, M., AND FEI, Y. Secure speculative execution via risc-v open hardware design. In *2021 Fifth Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[40] SAILESHWAR, G., AND QURESHI, M. K. CleanupSpec: An" undo" approach to safe speculation. In *2019 ACM/IEEE 52th International Symposium on Microarchitecture (MICRO)*, pp. 73–86.

[41] SAKALIS, C., KAXIRAS, S., ROS, A., JIMBOREAN, A., AND SJÄLANDER, M. Efficient invisible speculative execution through selective delay and value prediction. In *2019 ACM/IEEE the 46th International Symposium on Computer Architecture (ISCA)*, pp. 723–735.

[42] SCHWARZ, M., MAURICE, C., GRUSS, D., AND MANGARD, S. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography and Data Security: 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers 21*, Springer, pp. 247–267.

[43] SEAL, D., WALL, S., AND SLOSS, A. N. *ARM Architecture Reference Manual*, 2nd ed. Addison-Wesley, 2000. https://developer.arm.com/architectures/cpu-architecture/armv7-architecture-reference-manual.

[44] VELINOV, A., MILEVA, A., AND STOJANOV, D. Power consumption analysis of the new covert channels in coap. *International Journal On Advances in Security 12*, 1 & 2 (2019), 42–52.

[45] WANG, G., CHATTOPADHYAY, S., GOTOVCHITS, I., MITRA, T., AND ROYCHOUDHURY, A. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering 47*, 11 (2019), 2504–2519.

[46] WATERMAN, A., AND LAB., T. R. *BOOM v3 Processor Documentatio*. 2023. https://github.com/riscv-boom/doc.

[47] WEISSE, O., NEAL, I., LOUGHLIN, K., WENISCH, T. F., AND KASIKCI, B. NDA: Preventing speculative execution attacks at their source. In *2019 ACM/IEEE 52th International Symposium on Microarchitecture*, pp. 572–586.

[48] WU, C.-F., CHUNG, I.-H., LEE, C.-Y., AND CHEN, W.-K. Software techniques for managing speculation on amd processors. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, pp. 17–24.

[49] WU, Y., AND QIAN, X. ReversSpec: Reversible coherence protocol for defending transient attacks. *arXiv preprint arXiv:2006.16535* (2020).

[50] XIONG, W., KATZENBEISSER, S., AND SZEFER, J. Leaking information through cache lru states in commercial processors and secure caches. *IEEE Transactions on Computers 70*, 4 (2021), 511–523.

[51] XIONG, W., AND SZEFER, J. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys (CSUR) 54*, 3 (2021), 1–36.

[52] YAN, M., CHOI, J., SKARLATOS, D., MORRISON, A., FLETCHER, C., AND TORRELLAS, J. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *2018 ACM/IEEE 51th International Symposium on Microarchitecture (MICRO)*, pp. 428–441.

[53] YAN, M., SPRABERY, R., GOPIREDDY, B., FLETCHER, C., CAMPBELL, R., AND TORRELLAS, J. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (S&P)*, IEEE, pp. 888–904.

[54] YAO, F., DOROSLOVACKI, M., AND VENKATARAMANI, G. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 168–179.

[55] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *2014 USENIX Security Symposium (USENIX Security)*, pp. 719–732.

[56] YU, J., MANTRI, N., TORRELLAS, J., MORRISON, A., AND FLETCHER, C. W. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 707–720.

[57] YU, J., YAN, M., KHYZHA, A., MORRISON, A., TORRELLAS, J., AND FLETCHER, C. W. Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data. In *2019 ACM/IEEE 52th International Symposium on Microarchitecture (MICRO)*, pp. 954–968.

[58] ZHAO, J., KORPAN, B., GONZALEZ, A., AND ASANOVIC, K. SonicBoom: The 3rd generation berkeley out-of-order machine. In *2020 Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 5.