Exploiting Security Dependence for Conditional Speculation Against Spectre Attacks

Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Peng Liu, Lixin Zhang, and Dan Meng

Abstract—Speculative execution side-channel vulnerabilities such as Spectre reveal that conventional architecture designs lack security consideration. This article proposes a software transparent defense framework, named as *Conditional Speculation*, against Spectre vulnerabilities found on traditional out-of-order microprocessors. It introduces the concept of *security dependence* to mark speculative memory instructions which could leak information with potential security risks. More specifically, security-dependent instructions are detected and marked with *suspect speculation* flags in the Issue Queue. All the instructions can be speculatively issued for execution in accordance with the classic out-of-order pipeline. For those instructions with *suspect speculation* flags, they are considered as *safe* instructions if their speculative execution dose not refill new cache lines with unauthorized privilege data. Otherwise, they are considered as *unsafe* instructions and thus not allowed to execute speculatively. To pursue a balance of performance and security, we investigate two filtering mechanisms, *Cache-hit-based Hazard Filter* and *Trusted Page Buffer-based Hazard Filter* to filter out false security hazards. As for true security hazards, we have two approaches to prevent them from changing cache states. One is to block all unsafe access, the other is to fetch them from lower-level caches or memory to a speculative buffer temporarily, and refill them after confirming that they are on the correct execution path. Our design philosophy is to speculatively execute *safe* instructions to maintain the performance benefits of out-of-order execution while delaying the cache updates for speculative execution of *unsafe* instructions for security consideration. We evaluate *Conditional Speculation* in terms of performance, security, and area. The experimental results show that the hardware overhead is marginal and the performance overhead is minimal.

Index Terms—Spectre vulnerabilities defense, security dependence, speculative execution side-channel vulnerabilities

1 Introduction

Speculative and out-of-order execution are fundamental techniques to exploit instruction-level parallelism (ILP) in modern high-performance processors. In typical handling of mis-speculation, the pipeline states, such as integer and floating registers, are rolled back to the fault instructions. However, some microarchitecture states, such as cache contents, are usually not reverted, since such negligence does not violate the architectural semantics. Unfortunately, recently exposed Spectre and Meltdown, which are of speculative execution side-channel vulnerabilities, have revealed the security hazards of neglecting those unrecovered microarchitectural states [1], [2], [3], [4], [5]. Attacks exploiting speculative execution vulnerabilities usually induce a victim to speculatively perform operations that would not occur during correct program execution, but when occurring it would leak the victim's confidential information via a side channel to the adversary. Speculative execution vulnerabilities become a serious threat to commodity systems since speculative execution is widely adopted in most modern microprocessors [6], [7], [8].

Industrial researchers have responded rapidly to mitigate these threats [9], [10], [11], [12]. Retpoline, proposed by Google, converts indirect jump instructions into a blocking loop that combines return instructions to avoid unsafe speculative execution [13]. Intel has provided multiple microcode updates for their products and software developers can invoke specific instructions to enable different granularities of defense mechanisms to avoid interferences with the branch predictor between applications running at different privilege levels [9]. Various isolation mechanisms, such as KAISER and Site Isolation, are developed to shut down the observable channel between security domains [14], [15]. Although they effectively ease the security tensions, most existing mitigation techniques are software-based and more or less sacrifice transparency and/or performance.

To strike a balance between security, performance, and transparency, it is essential to innovate the microarchitecture design to safeguard the speculative execution. In the fourth quarter of 2018, Intel released a series of Coffee Lake R processors with hardware-based defenses against Meltdown and Foreshadow [16]. However, other Spectre-type variants are still mitigated by microcode together with the operating system. To date, there has been no widely accepted hardware solution for defending Spectre variants. This paper focuses on the microarchitecture design innovations against the major variants of Spectre (Spectre-V1, V2, V4, and SpectrePrime).

- *Lutan Zhao, Peinan Li, Rui Hou, and Dan Meng are with the State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China. E-mail: {zhaolutan, lipeinan, hourui, mengdan}@iie.ac.cn.*
- *Lixin Zhang is with the Institute of Computing Technology, CAS, Beijing 100190, China. E-mail: texaszhang@hotmail.com.*
- *Michael C. Huang is with the University of Rochester, Rochester, NY 14627 USA. E-mail: michael.huang@rochester.edu.*
- *Peng Liu with the Pennsylvania State University, State College, PA 16801 USA. E-mail: pliu@ist.psu.edu.*

Manuscript received 10 Sept. 2019; revised 15 May 2020; accepted 18 May 2020. Date of publication 26 May 2020; date of current version 9 June 2021. (Corresponding author: Rui Hou.)
Recommended for acceptance by O. Mutlu.
Digital Object Identifier no. 10.1109/TC.2020.2997555

More specifically, our work concentrates on vulnerabilities associated with branch speculation and memory access speculation. Overall, this paper makes the following contributions:

1) We first propose the concept of *Security Dependence*. Similar to *data dependence* and *control dependence*, this kind of new dependence is used to depict the speculative instructions which leak micro-architecture information with potential security risks.
2) An effective and software transparent defense framework against Spectre, named as *Conditional Speculation*, is proposed. This framework consists of three components to *detect whether a speculative access is safe or not according to* Security Dependence *dynamically, then allow executing safe instructions as usual, and prevent the execution of unsafe parts from changing cache states*. ① *Security Hazard Detection* is introduced in the Issue Queue to identify suspected unsafe instructions with security dependence. ② Targeting at the different features of multiple side-channel attacks, *Hazard Filter* is deployed to filter out safe instructions which do not leave observable traces in channels and allow them to execute speculatively. ③ *Security Hazard Response* is employed to prevent microarchitectural states from being changed by unsafe instructions. This paper describes how *Conditional Speculation* defends against Spectre attacks based on the cache side channel.
3) Two filtering mechanisms are investigated to figure out falsely identified security hazards, with the goal of pursuing a balance of performance, security, and transparency. The first proposed *Cache-hit based Hazard Filter* targets the speculative instructions which hit the cache. Since their speculative execution will not change cache (content), they are safe. Another proposed filter, *Trusted Page Buffer based Hazard Filter* (TPBuf), identifies safe speculative instructions from another perspective. For realistic and dangerous Spectre variants that use the shared memory-based cache side channel (e.g., Flush+Reload), their speculative execution of malicious gadgets have a common feature named as S-Pattern. TPBuf is designed to capture *S-Pattern* from all speculative execution. For any speculatively executed memory instructions, it is considered safe if it does not match *S-Pattern*.
4) Two Hazard Response mechanisms are explored. The straightforward one is to block all unsafe instructions. Furthermore, inspired by InvisiSpec [17], a hardware-based buffering mechanism, named as *SPBuf*, is investigated for temporarily buffering speculative traces. It can ensure security on the one hand, and provides data for subsequent data-dependent operations, offering higher performance on the other hand. We quantify the performance impact of the locations of Speculative Buffer. Experiments show that deploying a speculative buffer in last-level cache can achieve a similar performance as deploying in all caches because *Hazard Filter*s can take advantage of the locality of the private cache even without the Speculative Buffer. This observation motivates us to propose a lightweight Speculative Buffer design. In particular, we only deploy Speculative Buffer with the last-level cache, thus avoiding the complex cache coherence modifications in the private caches.

The next section contains a brief description of Spectre. Section 3 presents the threat model. The concept of security dependence is introduced in Section 4. Section 5 introduces the framework of *Conditional Speculation*. Sections 6, 7, and 8 describe the *Hazard Detection*, *Hazard Filter*, and *Hazard Response* respectively. Section 9 evaluates the *Conditional Speculation*. Section 10 is the discussion. And Section 11 summaries related works. Section 12 concludes this paper.

2 Understanding the Spectre Attacks

Spectre attacks usually trick the processor into speculatively executing instruction sequences that should not have been executed under correct program execution. By influencing which instructions are speculatively executed, this kind of attacks is able to use a side channel to transmit/leak victim's information out. A typical Spectre attack has the following three common key steps.

2.1 Induce Victim to Incorrect Speculation

There are two major approaches in Spectre attacks to induce victim to incorrect speculation.

Branch Speculation. Through purposeful training of the branch predictors, an adversary can change the control flow to incorrect speculative execution path to access the unauthorized data [1], [4], [18], [19], [20], [21]. Some processors use static branch predictor, which makes it much easier for an attacker to construct mis-speculative execution. What's more, complete process- or thread-level isolation is rare in branch predictor for existing high-end processor cores. It makes cross-process or cross-thread attacks feasible.

Memory Speculation. Another possible approach to inducing speculative execution is *load speculation* [11]. The load instruction is usually allowed to be speculatively executed even if the address of its older store instruction is unknown. Attackers can exploit it to induce the load instruction to speculatively access confidential data illegally.

2.2 Construct a Long Timing Window for Incorrect Speculative Execution

To gather enough and stable information of the incorrect speculation, a long timing window is essential for the adversary. There are several ways to achieve this, and we introduce two classic approaches.

Delinquent Memory Accesses. The attacker can use the cache line flush instruction or other ingenious methods to evict their source operands into off-chip memory [1], [3], [4], [11]. Such delinquent memory access will hold the predicated instruction a long time in Issue Queue due to unready source operand.

Long Dependence Chain. Constructing a long data dependence chain for computing source operands can also be used to provide a longer timing window for stable speculative executions [22].

2.3 Infer Secrets From Side-Channel Leakages

During the long timing window, subsequent speculative execution might leave traces in microarchitecture which can

be observed by some side-channel methods. As a widely used method now, cache side-channel attack exploits the time difference of memory accesses to deduce whether a victim process has loaded a specific cache line or not, and then infer the offset address or execution path. There are many well-studied cache side-channel attacks, including Flush+Reload [23], Prime+Probe [24], Evict+Reload [25], Flush +Flush [26] and Evict+Time [27].

3 Threat Model

We have the following assumptions on an attacker. She can execute her codes on the same machine with the victim process without elevated privileges. And it is feasible for the attacker to induce the target branch to jump to malicious gadgets in the single or cross address space.

This paper aims at a large class of representative Spectre-type attacks. They steal victims memory contents instead of the value of registers (Note that stealing memory contents is perhaps more dangerous than stealing register values). We define the defense scope primarily for the following reasons. The community has not yet found a way to enumerate all the possible side channels. Many choices are possible for the side-channel component, such as the cache hierarchy, AVX units [28] and ports [29]. Thus it should be noted that Spectre-type attacks do not restrict themselves to cache side channel only. However, identifying the existence of a side channel is only the first, small step towards mounting highly successful attacks. Compared with other side channels, cache side channel seems much more mature and efficient and has been widely used in most of the documented Spectre variants. Note that while we limit our discussion to this threat model in this paper, the basic architecture can still work for an expanded threat model.

4 Security Dependence

Security is a complex issue. In this paper, we focus on the type of problems caused by side-channel vulnerabilities exploited by Spectre. These are essentially micro-architecture information leakages due to mis-speculation. To help capture the problems caused by unsafe speculative execution, we introduce the concept of *Security Dependence*.

Instruction j is security-dependent on Instruction i with respect to leakage channel c if both conditions hold:

- i precedes j in program order.
- If j is speculatively executed ahead of i, j will leakage information into channel c.

Note that since leakage happens in a variety of channels, security dependence is defined with respect to the particular channel. Given the above definition, Table 1 summarizes the major security dependence in Spectre vulnerabilities. In this paper, we focus on how to defend against the first six variants based on cache (content) side channels. In other words, if j does not change cache content, then it does not have a security dependence with respect to cache content channel and we consider it to be safe. We can find that security dependence comes from two situations, including memory-memory speculation and branch-memory speculation.

TABLE 1
Security Dependence in Spectre Variants(Instr: Instruction, Mem: Memory Access, and br: Branch Instruction)

Variants	Instr i	Instr j	Channel c[1]
Spectre V1 [1]	conditional br	mem	cache
Spectre V1.1 [19]	indirect br	mem	cache
Spectre V2 [1]	indirect br	mem	cache
Spectre V4 [11]	mem	mem	cache
SpectrePrime [4]	conditional br	mem	cache
SpectreRSB [20]	return	mem	cache
ret2spec [21]	return	mem	cache
NetSpectre [28]	conditional br	mem/AVX	cache/AVX
SMoTherSpectre [29]	indirect br	conditional br	port contention

[1]*Side channel* c *listed in this table is the method used in corresponding documented papers.*

Security Dependence Under Memory-Memory Speculation. One example is the Proof-of-Concept (PoC) X86 assembly code piece of Spectre V4 in Listing 1, which exploits speculative store bypass (also named as load speculation) [11]. Instruction 1 ($i1$ hereafter) is a store operation, and $i2$ is a load operation. Assuming these two instructions access the same memory address, there is a RAW dependence actually. The attacker might construct an environment in which $i1$ is pending in the issue queue for the unprepared source register (*rdi*), $i2$ is speculatively launched and get forwarded stale sensitive data. Once the address of $i1$ is known, the load mis-peculation occurs and the incorrect execution results of $i2$ need to be discarded. However, the sensitive data related cache line has already been refilled into L1 cache by $i2$ and $i4$, such information leakage allows attackers to infer the sensitive data. Therefore, we say both $i2$ and $i4$ are security dependent on $i1$.

Listing 1. PoC Code piece of Variant 4: Speculative Store Bypass

```
1  mov [rdi+rcx],al          ;unsolved store
2  movzx r8,byte [rsi+rcx]   ;unsafe load
3  shl r8,byte 0xc           ;shifted as a index
4  mov eax,[rdx+r8]          ;dependent load
```

Listing 2. PoC Code piece of Variant 1: Bounds Check Bypass

```
1 Loop:
2   mov rdi, -0x8(rbp)
3   mov 0x200a54(rip), eax
4   mov eax, eax
5   cmp -0x8(rbp), rax   ;cache miss
6   jbe 40063f <Loop>    ;unresolved branch
7   mov -0x8(rbp), rax   ;unsafe load
8   add 0x601080, rax    ;indirect index
```

Security Dependence Under Branch-Memory Speculation. In case of Spectre V1 in Listing 2. In attackers' well-designed environment, the branch $i6$ stays in the issue phase, and the $i7$ is speculatively executed ahead of time to access unauthorized sensitive data. As with the previous scenario, the cache contents are changed in such mis-speculation and the sensitive data might be inferred out via cache side channel. According to our definition, $i7$ is security dependent on $i6$.

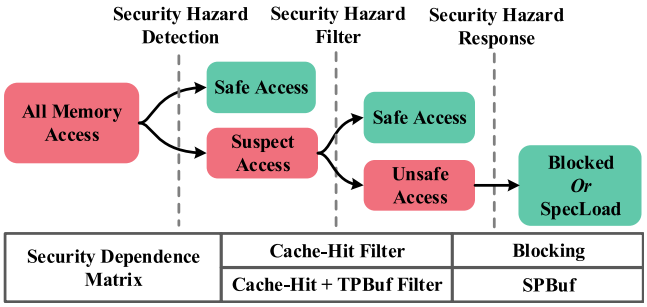


Fig. 1. The framework of Conditional Speculation (*red shadow indicates a risk of leaking sensitive information, while green shadow means that no sensitive trace remains in caches*).

5 Conditional Speculation Framework

Speculation and cache side-channel information leakages are critical factors for Spectre attacks. A straightforward defense policy is to prohibit any speculation of memory instructions at the cost of severe performance slowdown. Actually, not all speculative memory access instructions pose a risk of leaking information. Therefore, we propose a defense framework, named as *Conditional Speculation*, which *detects whether a speculative access is safe or not according to security dependence dynamically, then allows executing safe instructions as usual, and prevents the execution of unsafe parts from changing cache states*. The framework of *Conditional Speculation* is summarized in Fig. 1 with three major components. Guided by this framework, we propose a generic design of core microarchitecture, as shown in Fig. 2.

Security Hazard Detection is integrated into the Issue Queue to identify the security dependent memory access instructions and assigns each instruction with a *suspect speculation* flag to indicate they POSSIBLY change cache contents due to mis-speculation.

Security Hazard Filters are integrated into the load store queue (LSQ) and L1 DCache to identify whether instructions tagged with *suspect speculation* are safe or not. With the goal of pursuing a balance of performance and security, we propose two filters in LSQ and L1 DCache to remove false security hazards. The proposed *Cache-hit based Hazard Filter* targets the instructions which hit in L1 DCache without any cache (content) side-channel information leakage. From another perspective, *Trusted Page Buffer based Hazard Filter* (TPBuf Filter) identifies more safe speculative instructions. Since Spectre variants that use the shared memory-based cache side channel, in which secret data and memory region used for constructing side-channel usually located at different memory pages, their speculative execution of malicious gadgets has a common feature (*S-Pattern*) described in Section 7.2. It is considered safe if a suspicious access does not match *S-Pattern*. The instructions that survive the filters are allowed to be speculated normally, thereby obtaining better performance in the context of security.

Security Hazard Response handles the remaining *unsafe* instructions. We investigate two approaches to prevent them from changing cache states. One approach is blocking all unsafe accesses (*Blocking*), which sends back a signal from L1 Dcache to Issue Queue that the re-issue logic is applied to the memory instruction until its security dependence is resolved. The *Blocking* approach requires only minor modifications to the processor but might have negative impacts on performance. The other method, named as *SPBuf*, makes unsafe accesses invisible to caches hierarchy and allows them to load data speculatively to subsequent data-dependent instructions. In *Conditional Speculation*, we find that deploying Speculative Buffers only in last-level cache can achieve a similar performance as deploying in all level caches for the benefits of Hazard Filter. Thus we deploy the buffer in the last level cache only, which avoids modifying cache coherence logic in private caches for SPBuf.

6 Security Hazard Detection

We design a *Hazard Detection* logic based on bit matrices as shown in Fig. 3, which zooms in the Issue stage in Fig. 2. Bit-matrix is a popular way used by some commodity processors to track data dependence and age information [30], [31]. With a security detection module, the security dependence becomes one of the factors to determine whether an instruction to be issued, together with data dependence and age information.

Matrix Organization. Assuming that the Issue Queue (IQ) has N items, the security dependence matrix will contain a register array of NxN bits. The number of read ports of this matrix is equal to the dispatch width, and the number of write ports is equal to the issue width. This matrix is indexed by IQPos (Issue Queue Position) of each instruction. Given any Instruction X, IQPos_X denotes its location in the IQ. If the value of the Matrix[IQPos_X, IQPos_Y] is 1,

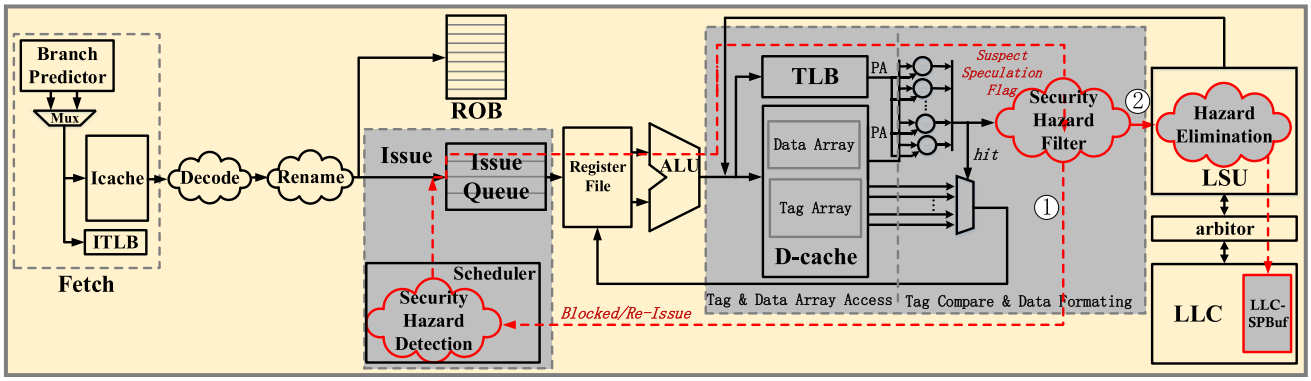


Fig. 2. The microarchitecture overview of conditional speculation mechanism.

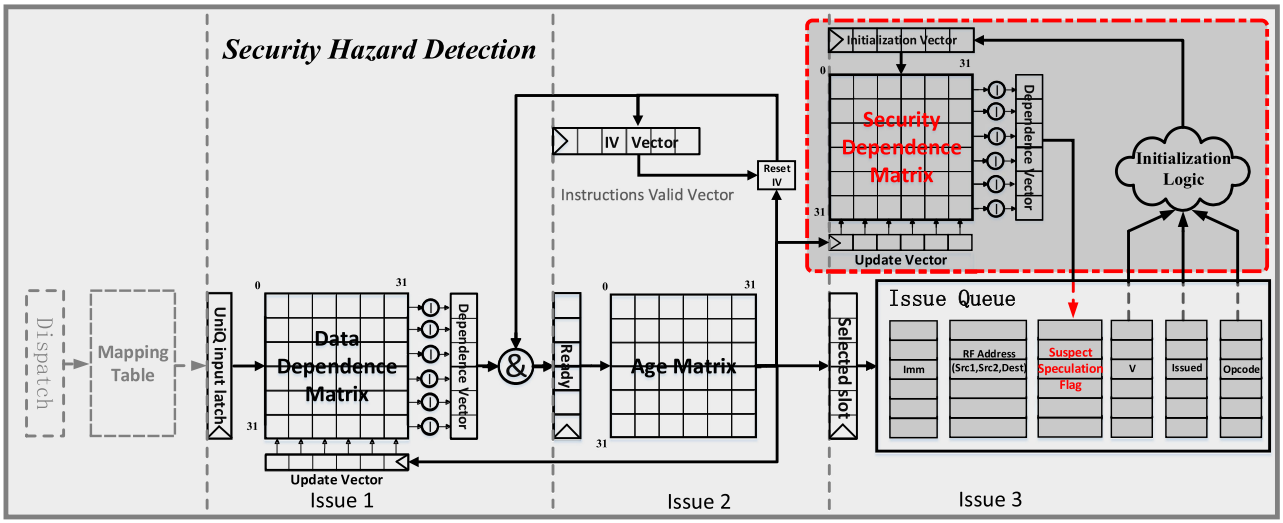


Fig. 3. Security hazard detection based on security dependence matrix.

X has security dependence on Y. Otherwise, it means there is no security dependence between them.

Matrix Initialization. When the new instruction X is dispatched into the IQ, one row is allocated with the index IQPos_X. For each Instruction Y which is valid in the IQ at this moment, Matrix[IQPos_X, IQPos_Y] determines the security dependence between instructions according to the following formula. There are three conditions: ①If Y is valid and precedes X, which means Y is dispatched into the IQ before X. ② According to our threat model, we check if a memory instruction is security-dependent on the previous branch or memory instructions. ③ If the preceding branch or memory instructions are still waiting in the IQ when a memory instruction is issued, this memory instruction will be considered to have security dependence.

$$\begin{aligned} Matrix[X,Y] =& (IssueQ[X].opcode == MEM) \\ & \&(IssueQ[Y].opcode == MEMorBR) \\ & \&IssueQ[Y].valid \\ & \&!IssueQ[Y].issued. \end{aligned}$$

Hazard Detection. Fig. 3 illustrates the three-stage options of the Issue Queue. At the 1st stage, the data dependence matrix generates a dependence vector. At the 2nd stage, this vector is then sent to the age matrix to select the oldest ready instruction to be issued. At the 3rd stage, for those instructions selected to be issued, the security dependence matrix is queried to get their security dependence, and then the states of corresponding entries of Issue Queue are updated. In particular, bits in each row of the security dependence matrix are processed by OR operation and the result demonstrates whether there is a potential security hazard. When an instruction is selected to be issued and a security hazard is detected, it will be tagged with a *suspect speculation* flag.

Matrix Clearance. After an instruction X is issued, the corresponding bit in *Update Vector Register* will be set as 0. The column of security dependence matrix indexed by IQPos_X will be reset at the next cycle. Such operation means that the security dependence between corresponding instructions and X is cleared.

7 Security Hazard Filter(s)

According to the security dependence under branch-memory and memory-memory speculation, a large number of memory access instructions are justified as suspect by *Hazard Detection* module. Employing a conservative approach to execute these instructions causes significant performance slowdown. However, some speculative memory accesses are safe because of no observable traces left. Kinds of filters are investigated to figure out these safe instructions and allow them to be executed speculatively. Focus on Spectre variants based on cache side channels, we propose two filters, *Cache-hit based Hazard Filter* and *Trusted Pages Buffer based Hazard Filter*.

7.1 Cache-Hit Based Hazard Filter

When a memory instruction is speculatively issued to the memory access pipeline, it is tagged with its *suspect speculation* flag. If the suspect memory access hits in L1 DCache, it will continue as a normal memory instruction. However, if it encounters a miss in L1 DCache, the missing request will be marked as unsafe and the memory access will be sent to *Hazard Response* module. This design requires only minimal changes in the L1 DCache control logic.

Secure Update for Cache Replacement Logic. It should be noted that if speculative accesses that hit L1 DCache under update cache replacement metadata (e,g, LRU bits), secret information is possible to be observed [32]. For example, an attacker can train the LRU bits of given sets, then carefully induce the victim to change the LRU bits speculatively, then figure out which sets have been accessed, and finally infer sensitive data. To prevent such attacks, we propose two secure update policies and evaluate them in Section 9.3.

1) *No update policy* skips LRU updates for speculative accesses that hit the L1 DCache. For speculative accesses that eventually become non-speculative, not updating LRU bits can diminish the effectiveness of L1 DCache replacement policy.
2) *Delayed update policy* sets a *pending LRU update* tag when a speculative access hits in the L1 DCache and performs the actual LRU update when the access reaches

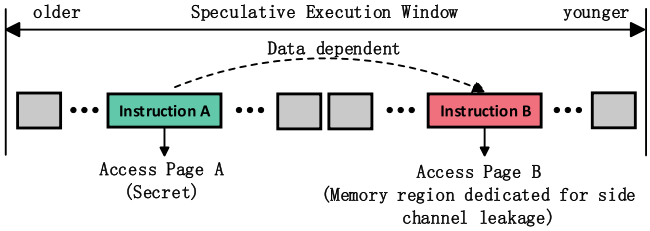


Fig. 4. Typical instruction flow of malicious gadget for shared memory (e.g., Flush+Reload) based Spectre attack.

the head of the ROB (or becomes non-speculative) and the corresponding LRU array is not being used by accesses from the load/store pipelines.

7.2 Trusted Pages Buffer Based Hazard Filter

Trusted Pages Buffer based Hazard Filter aims at the Spectre variants based on cache side channels with shared pages, which is realistic and widely used in existing Spectre variants. The specific analysis is described in Section 9.2. These variants obey a common feature, named as *S-Pattern* in this paper. By dynamically identifying more safe instructions based on *Cache-hit Filter*, this filter gets better performance.

1) S-Pattern

As shown in Fig. 4, the speculative execution of malicious gadgets can be concluded into a common feature. In particular, it is observed that the malicious speculative execution flow always contains two special memory instructions (A and B). These two instructions have the following usages and behaviors.

1) A is used to speculatively access sensitive data. And B speculatively accesses the memory region shared with the attacker, which is used for building the cache side-channel between the victim and the attacker. Since secret data and shared memory regions usually locate at different memory pages, these two instructions access different pages. For example, confidential pages and non-secure pages are tagged with different flags in SGX and TrustZone. Therefore, if an attacker needs a shared non-secure page to construct a side channel, it always locates on a different page from secrets.
2) B is data-dependent on A. The result of A is used to calculate the index of the shared memory region. Such well-designed dependence is also another important point for the attacker to infer the secret values.
3) In order to build a cache side-channel, the attacker needs to first flush the specific shared memory data. Then, the induced speculative execution of B has a cache miss and thus reloads the cache line into L1 DCache. This change in state information can be perceived by the attacker through the cache side channel. Thus the cache miss of B is essential to leak sensitive information over the cache side channel.

TABLE 2
Filter Strategy for one Incoming Request

Query Result	Decision
There is at least one valid entry whose request accesses different memory pages, and this request is in Writeback status.	UnSafe
Others	Safe

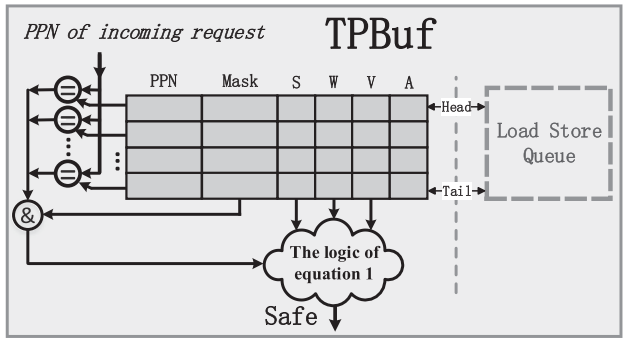


Fig. 5. The microarchitecture design of TPBuf.

Motivated by the aforementioned observation, we call the above common characteristic behavior as *S-pattern*. Specifically, if the instruction sequence of speculative execution is observed to have the following characteristics, we consider this sequence of speculative instruction has *S-pattern* behavior.

1) There are at least two instructions (A and B) that separately access different memory pages.
2) Instruction B has data dependence on instruction A.
3) Instruction B results in an L1 DCache miss.

Although the malicious gadgets of Spectre attacks are featured as *S-Pattern* behaviors, it should be noted that the instruction flow with *S-Pattern* is not necessarily a Spectre attack. For example, instruction A may read secret data from registers, and the instruction B may propagate transmits the secrets to attackers. While such leakage is important to address, it is clearly less dangerous than leaking from cache hierarchy. Therefore, we consider such leakage out of scope.

2) Microarchitecture implementation of TPBuf

TPBuf is designed to capture memory access behaviors with *S-Pattern* from all speculative executions. It records all the on-the-fly speculative memory access requests and tracks their execution status (e.g., whether the requested cache line is refilled or not). When a new memory request which misses in the L1 DCache, TPBuf compares its page address with its history records. And it decides whether this new speculative instruction is safe based on the logic described in Table 2.

The microarchitecture of TPBuf is shown in Fig. 5. One main design principle is to utilize the existing logics as much as possible to reduce the complexity of implementation. TPBuf is placed close to the Load Store Queue (LSQ) and its entries have a 1:1 mapping with the entries of LSQ. The allocation, commit and squash of TPBuf's entries are operated along with the movement of the LSQ's Head and Tail pointers. Besides, TPBuf covers all on-the-fly speculative memory instructions in the speculative execution window. In order to prevent the attacker from speculatively accessing unauthorized data directly and then spreading the data to his own memory space, the access address must be checked

and get physical page number (PPN) using TLB first. TPBuf records and uses the PPN as the tag of each entry (PPN is part of the physical address which is stored in PAddr). In addition, each TPBuf entry stores a mask and a number of status bits. TPBuf detects the *S-pattern* and passes the results to Cache-hit filter which decides whether a suspect speculative miss request is safe. In this way, the original memory consistency model and cache coherence are unaffected.

Allocation. When memory access instructions are allocated in LSQ, they also are allocated in TPBuf and *A* bit is set. And *Mask* is generated according to *A* bits in TPBuf. It indicates which memory instructions in TPBuf are older than the new entry in the program order.

Update. The *S* bit is updated with the suspect speculation flag attached with the memory instruction. When the *PPN* is recorded in TPBuf, the *V* bit is set. The *W* is set when data fetched by the memory instruction becomes available to other instructions.

Detection. When an incoming request enters TPbuf, the TPBuf compares its *PPN* with the *PPN* of existing entries and then generates an address-match vector(*Match*). These vectors, including *Match*,*V*, *W*, and *S*, are used as inputs of the logic of equation 1 to determine whether the requests are safe. Specially, '|' means *reduction OR*, which operates OR on all of the bits in a vector to generate 1-bit output.

$$safe = \ !\ (\ |(\ \mathbf{V}\ \&\ \mathbf{W}\&\ \mathbf{S}\ \&\ \mathbf{Match})). \tag{1}$$

8 Security Hazard Response

To handle the unsafe memory accesses, two *Hazard Response* mechanisms are explored. *Blocking Hazard Response*: A straightforward scheme is to block unsafe speculative accesses and bounce them back to Issue Queue where they wait for the clearance of security dependence before being re-issued. *SPBuf Hazard Response*: Inspired by InvisiSpec, we propose the SPBuf Hazard Response. InvisiSpec [17] is a prior mitigation against Spectre variants based on cache side channels. InvisiSpec considers all suspect instructions as unsafe and employs *SpecLoad*s to retrieve data into *Speculative Buffer*s deployed next to the private and shared last-level caches temporarily, and does not update the caches until the accesses become safe. Different from it, *Conditional Speculation* identifies most safe instructions by *Hazard Filter*s and allows them to execute normally. For the remaining unsafe instructions, we propose lightweight *Speculative Buffer*s and *SpecLoad*s. This section mainly focuses on the two lightweight features of *SPBuf Hazard Response* in detail.

1) First lightweight feature: Deploying Speculative Buffer only in Last-Level Cache

We quantify the performance impacts of deploying *Speculative Buffer*s in different level caches. Experiments show that deploying a *Speculative Buffer* in LLC can achieve a similar performance as deploying in all caches because *Hazard Filter*s can take advantage of the locality of the private caches even without *Speculative Buffer*. Therefore, we deploy a Speculative Buffer in LLC (LLC-SPBuf) only, which reduces the complexity to support cache coherence and maintain memory consistency for private caches.

Similar to TPBuf, the LLC-SPBuf has the same size as the Load Queue. LLC-SPBuf is indexed with the index of Load Queue, and each entry consists of the valid flag, cache line, and corresponding physical address. For a single-core processor, when a *SpecLoad* misses in LLC, it first checks whether its cache line has been loaded in LLC-SPBuf by earlier *SpecLoad*s. If it hits in LLC-SPBuf (same physical address), the cache line will be copied to its entry and responded to the private caches. Otherwise, the request is sent to the main memory and copies the responded cache line to the LLC-SPBuf. When the unsafe access is committed eventually, an *Update* request is sent to invalidate all entries with the same physical address and copy the cache line from LLC-SPBuf to LLC directly, eliminating the traffic overhead of reloading data from main memory. In this paper, we primarily focus on the effect of Conditional Speculation in single-core processors at the performance. As for the implementation details for multi-core processors, such as how to support memory consistency models, cache coherence and so on, we can make a reference to InvisiSpec.

2) Second lightweight feature: Transforming Validation into Exposure

SpecLoad is a type of memory requests which is introduced to inform the cache hierarchy that this unsafe access only reads data back to the pipeline and does not change any cache state, such as coherence state, LRU. Therefore, the pipeline may fail to receive if there is any invalidations directed to the line loaded by *SpecLoads*. To avoid this violation of memory consistency, *Validation* and *Exposure* are proposed in InvisiSpec. Compared to *Validation*, *Exposure* has low overhead because a load instruction can be committed as soon as the *Exposure* is issued, whereas the instruction cannot be committed until the completion of a *Validation*. In *Conditional Speculation*, there are few SpecLoads for the benefits of *Hazard Filter*, so we can transform the *Validation* to *Exposure* by scheduling the issue of SpecLoads as following. Consider TSO first, *SpecLoads* are initiated when all of the earlier loads accessing the same cache line in LSQ have initiated their *SpecLoads*. As a result, TSO would not require squashing the load on the reception of invalidation to the line it loaded. The *SpecLoad* just needs *Exposure*, not *Validation*. Now consider RC, only speculative loads that read when there is at least an earlier fence in the LSQ will be squashed by an invalidation to the line. Hence, their *SpecLoads* are initiated when the earlier fence is committed.

When the instruction is committed normally, an *Exposure* is issued to refill corresponding cache lines to the caches and update cache states. When a *SpecLoad* request is initialized for the instruction and sent to the lower caches, a new entry is allocated in the miss handler (MSHR)s. Traditionally, requests accessing the same cache line in MSHR can be merged to a single request and sent to the lower cache hierarchy. As for an incoming *SpecLoad* request, it can be merged into an earlier *SpecLoad* request to the same cache line, but cannot be merged to common requests for the consideration of security. Similarly, an *Exposure* request can be merged to earlier *Exposure* requests.

9 Evaluation

9.1 Methodology

We evaluate *Conditional Speculation* on a cycle-accurate simulator (Gem5) and a real hardware in terms of security,

TABLE 3
Simulator Configurations

Parameter	Configuration
Architecture	ALPHA at 2.5 *GHz*
Core	4-way out-of-order, 15 stages, 32 Load Queue entries, 24 Store Queue entries, 192 ROB entries, 64 Issue Queue entries, Tournament predictor, 4096 BTB entries
ITLB/DTLB	64 entries
L1 ICache	64 KB, 4-way, 64B line, 2 cycle hit
L1 DCache	64 KB, 4-way, 64B line, 2 cycle hit
L2 Cache	2 MB, 16-way, 64B line, 10 cycle hit
L3 Cache	8 MB, 32-way, 64B line, 60 cycle hit
Memory	8 GB, 192 cycle latency

performance, and area cost. We simulate an Out-of-Order processor with *Conditional Speculation* to evaluate the features in terms of performance on the complexity of Out-of-Order core. Table 3 lists the key parameters of the simulated processor. To corroborate the validity of the results, we build an FPGA prototype based on open-source Berkeley Out-of-Order RISC-V processor (BOOM) [33]. The most important part of *Conditional Speculation*, including *Cache-hit Filter based Hazard Filter* and *Trusted Pages Buffer based Hazard Filter*, are evaluated in this prototype.

Security features are evaluated by analyzing Proof-of-Concept (PoC) codes. For Gem5, we adopt SPEC CPU 2006 benchmarks with reference input size for performance evaluation. The simulator is warmed up for one billion instructions, and then run another billion instructions in the cycle-accurate mode. For real hardware, the BOOM processor runs all the SPEC CPU 2006 benchmark suite with train input size. Finally, the *Security Dependence Matrix*, *TPBuf* and *SPBuf* is implemented using Register-Transfer Level (RTL) code and then synthesized and implemented with SMIC $40nm$ technology for area and timing evaluations.

Different mechanisms of *Conditional Speculation* are named in the following abbreviations:

Baseline. Base out-of-order processor with the configuration listed in Table 3, without any defense mechanism.

Naive Policy. Simply considers all the security-dependent memory accesses as *unsafe* and block their execution.

[F]-Blocking. Conditional Speculation with *Blocking Hazard Response*. 'F' means the mechanism of *Hazard Filter*, with 'CF' indicating *Cache-hit Filter* and 'CTF' indicating *Cache-hit Filter* and *TPBuf Filter* working together.

[F]-SPBuf-[C]. Conditional Speculation with *SPBuf Hazard Response*. 'F' means the *Hazard Filter* and 'C' means which level cache is deployed with *Speculative Buffer* ('ALC' for all level caches and 'LLC' for last level cache respectively). For example, 'CF-SPBuf-ALC' employs *Cache-hit Filter* and deploys *Speculative Buffer* in all level caches.

9.2 Security Analysis

Hazard Detection is the initial detection of security dependence and only tags a suspect flag for each instruction, the security of *Conditional Speculation* depends on the combination of *Hazard Filter* and *Hazard Response* mechanisms. Table 4 summarizes the security analysis. In this table, Spectre variants based on cache side channels are divided into six typical scenarios classified by different combinations of cache side-channel attacks and page sharing mode.

TABLE 4
Security Analysis

Attack Classification	CF		CTF	
	Blocking	SPBuf	Blocking	SPBuf
Flush+Reload, share pages	✓	✓	✓	✓
Flush+Flush, share pages	✓	✓	✓	✓
Evict+Reload, share pages	✓	✓	✓	✓
Prime+Probe, share pages	✓	✓	✓	✓
Prime+Probe, no shared pages	✓	✓	×	×
Evict+Time, no shared pages	✓	✓	×	×

Hazard Filter. *Cache-hit Filter* (CF) does not load data for speculative memory accesses that miss in L1 DCache, which is the prerequisite of observing secrets from cache side channels. *Cache-hit Filter and TPBuf Filter* (CTF) identifies the unsafe speculative memory accesses according to *S-Pattern*, which is the common feature of cache side-channel attacks based on shared data. Therefore, *CF* succeeds in defending against all six types of variants and *CTF* can defend against the first four types in Table 4. It should be noted that shared pages based channels are widely used in PoC codes of existing Spectre variants, including "Flush +Reload, share data" based variants (Spectre V1, V1.1, V2, V4), and "Prime+Probe, share data" based SpectrePrime.

Hazard Response. Both *Blocking* and *SPBuf* prevent unsafe accesses from leaving sensitive traces on caches, so they do not influence the security for their defense ability of *Hazard Filter*. Furthermore, it should be noted that the *Blocking* mechanism can additionally mitigate side channels through the memory system (e.g., DRAM contention).

In summary, both *Blocking* and *SPBuf* based Hazard Response mechanisms do not influence the defense ability for cache side channels. As for *Hazard Filter* mechanisms, *CF* defenses against all Spectre variants based on cache side channels, and *TPBuf Filter* is able to prevent widely used variants exploiting shared pages, but fails to block variants that don't need shared data.

9.3 Performance Evaluation on Simulator

Fig. 6 compares the performance impacts of three different mechanisms of *Conditional Speculation*. Not surprisingly, the *Naive Policy* causes the largest performance degradation (54.6 percent performance degradation on average, and the worst case is 146.8 percent for hammer). In contrast, *CF-Blocking* provides a certain degree of relaxation. By dynamically identifying false security hazard, it allows the memory instructions that hit L1 DCache to be executed. Such a filter significantly improves the performance (on average reduce performance degradation from 54.6 to 13.2 percent). In particular, *CF-Blocking* recognizes 86.3 percent speculative accesses as safe due to the high L1 DCache hit rate for SPEC

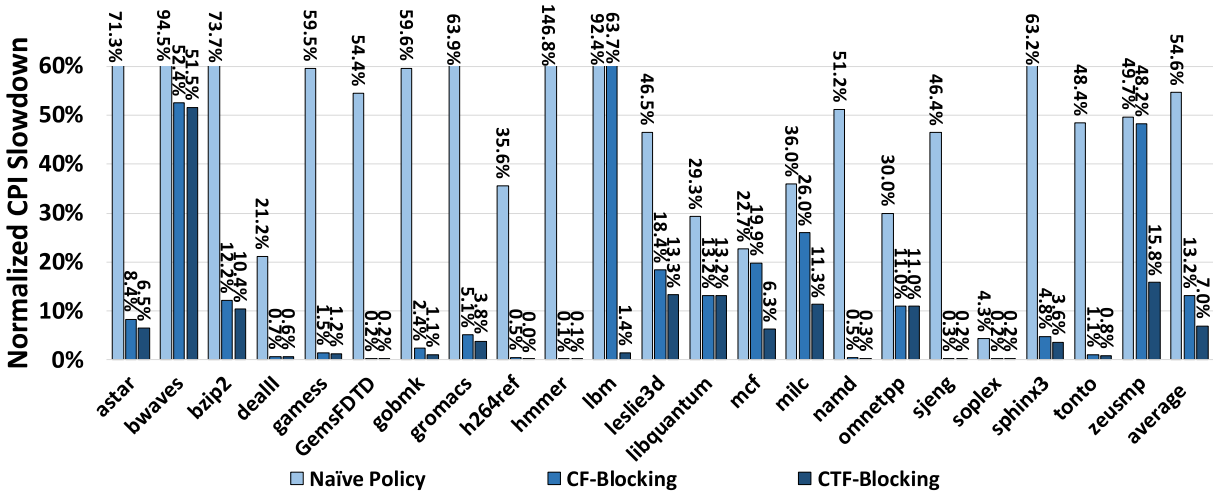


Fig. 6. Performance evaluation for blocking policy (All the values in this figure are normalized to *Baseline*).

TABLE 5
Dependence Analysis for Branch-Memory Dependence only and Memory-Memory Dependence

Benchmark	Branch-Memory Dependence only				Branch-Memory and Memory-Memory Dependence	
	Performance Overhead	Unresolved Branch Rate	Unsafe Memory Access Rate	Misprediction Rate in Issue Stage	Performance Overhead	Resolving Time Ratio
astar	65.5%	16.7%	27.2%	8.5%	71.3%	2.3
bwaves	0.7%	0.7%	2.6%	0.1%	94.5%	1.7
bzip2	53.5%	7.8%	24.8%	5.2%	73.7%	2.6
dealII	3.7%	6.2%	8.1%	1.1%	21.2%	2.7
gamess	20.7%	5.6%	14.4%	2.1%	59.5%	1.8
GemsFDTD	40.8%	16.4%	22.9%	1.8%	54.4%	0.7
gobmk	43.6%	13.0%	25.0%	8.3%	59.6%	2.8
gromacs	16.7%	3.4%	8.7%	6.5%	63.9%	3.0
h264ref	9.7%	6.7%	15.4%	3.1%	35.6%	1.6
hmmer	1.4%	2.3%	12.8%	0.3%	146.9%	3.6
lbm	53.5%	1.0%	7.9%	0.4%	92.5%	184.3
leslie3d	0.5%	1.2%	9.1%	0.9%	46.5%	2.2
libquantum	22.4%	8.5%	28.4%	0.0%	29.3%	15.2
mcf	9.2%	20.4%	35.5%	2.1%	22.7%	92.1
milc	2.1%	1.0%	10.3%	0.1%	36.0%	4.3
namd	32.9%	11.0%	21.6%	2.8%	51.2%	1.1
omnetpp	27.4%	26.7%	19.7%	1.8%	30.0%	2.7
sjeng	30.8%	15.0%	25.1%	7.3%	46.4%	2.4
soplex	4.0%	9.3%	13.6%	4.1%	4.3%	9.9
sphinx3	45.0%	10.7%	25.4%	3.0%	63.1%	3.3
zeusmp	0.1%	4.5%	7.8%	0.3%	49.7%	2.3
Average	23.1%	9.0%	17.4%	2.8%	54.9%	8.3

CPU benchmarks. *CTF-Blocking* gets further performance improvements. *S-Pattern* depicts the memory access pattern with malicious behaviors in Spectre attacks. For any speculative instruction which misses L1 DCache, if it does not match S-pattern, it is considered safe and can still be speculatively executed. It can be observed that *CTF-Blocking* further reduces the performance overhead to 7.0 percent on average.

1) Performance overhead for Naive Policy

The security dependence comes from two major situations, branch-memory speculation and memory-memory speculation. In order to have a better understanding on the performance loss, we make in-depth analysis as below.

We first model a *branch-memory* dependence matrix which recognizes speculative memory accesses dependent on branch instructions as unsafe. It introduces 23.1 percent performance degradation on average. As expected, the more branch instructions exist, the more speculative memory accesses will be tagged as unsafe. In addition, high misprediction rate might further reduce the performance. As shown in Table 5, the worst case of astar (65.5 percent overhead) has a high branch misprediction rate (8.5 percent). Besides, 16.7 percent instructions are unresolved branch instructions when they are allocated in the Issue Queue, and 27.2 percent memory instructions are marked as unsafe.

TABLE 6
Filter Analysis (Blocked Rate Means the Proportion of Blocked Speculative Memory Accesses in Correct Path)

	Baseline	Naive Policy	CF-Blocking		CTF-Blocking		CTF-SPBuf-ALC
Benchmark	L1 Hit Rate	Blocked Rate	Blocked Rate	L1 Hit Rate of Speculative Access	Blocked Rate	S-Pattern Mismatch Rate	L2 Hit Rate of USL
astar	94.4%	74.6%	3.3%	90.4%	2.2%	14.5%	>99.9%
bwaves	81.3%	73.0%	5.6%	90.3%	5.5%	1.5%	60.8%
bzip2	96.7%	77.8%	1.6%	95.5%	1.3%	5.0%	95.6%
dealII	97.3%	58.7%	0.1%	**99.4%**	0.1%	15.5%	98.8%
gamess	96.0%	75.0%	0.5%	98.8%	0.4%	10.8%	>99.9%
GemsFDTD	>**99.9%**	79.1%	<**0.1%**	99.9%	<0.1%	0.2%	>99.9%
gobmk	95.3%	72.5%	1.6%	96.3%	0.2%	**39.4%**	96.5%
gromacs	93.8%	71.4%	2.1%	94.8%	1.1%	19.0%	97.4%
h264ref	99.1%	62.5%	0.3%	98.3%	<0.1%	**47.0%**	99.6%
hmmer	97.9%	65.4%	0.3%	**99.4%**	0.3%	2.1%	>99.9%
lbm	**61.8%**	65.9%	**15.8%**	**60.7%**	**0.3%**	**86.2%**	80.9%
leslie3d	95.1%	85.3%	1.6%	96.5%	1.2%	17.2%	65.11%
libquantum	**79.6%**	88.4%	1.6%	95.2%	1.6%	<**0.1%**	77.2%
mcf	**73.9%**	65.2%	**9.3%**	**75.1%**	**3.2%**	**32.6%**	80.6%
milc	**66.2%**	77.9%	**13.0%**	**67.6%**	9.2%	6.3%	76.9%
namd	97.5%	77.4%	0.2%	**99.6%**	0.1%	**31.9%**	97.8%
omnetpp	92.9%	76.7%	4.4%	78.2%	4.1%	0.8%	99.9%
sjeng	99.4%	78.1%	<0.1%	99.7%	<0.1%	11.9%	85.0%
soplex	84.9%	71.0%	3.3%	82.1%	3.3%	0.3%	>99.9%
sphinx3	97.9%	77.4%	0.3%	96.6%	0.2%	13.1%	76.5%
zeusmp	**55.3%**	67.0%	**15.0%**	**61.5%**	**3.9%**	**26.9%**	79.64%
Average	88.4%	**73.3%**	**3.8%**	86.3%	**1.8%**	18.2%	87.9%

After appending the *memory-memory* dependence, the proportion of unsafe speculative memory accesses increases. More seriously, other instructions that are dependent on these unsafe operations have to be blocked in the issue queue. Experiments show that some cases are particularly sensitive to memory-memory security dependence. For example, the performance overhead of lbm increases from 53.5 to 92.4 percent, and it takes more than 180 times longer to resolve a branch than the *Baseline* case. As depicted in Table 6, the Naive policy will block almost 73.3 percent speculative memory accesses on correct execution path.

2) Performance overhead from secure update for Cache replacement logic

To understand the performance implication of *secure update policy*, we evaluated them on top of *CTF-Blocking* using the same set of benchmarks in Fig. 6. The results show that *no update policy* introduces 0.71 percent performance degradation. For *delayed update policy*, our experiments show that it improves *no update policy* by 0.26 percent. There is little difference in the performance impact of the two policies. *Considering the complexity of this policy, we believe that* no update policy *is the better option due to its simplicity and this policy is used for the following experiments.*

3) Performance gain from Filters

Cache-Hit Filter (CF). This filter exploits the locality of memory access behaviors of normal workloads. Compared to *Naive Policy*, *CF* improves the performance by 26.7 percent on average as shown in Fig. 6. Take GemsFDTD in Table 6 as an example, the cache hit rate is more than 99.9 percent, and then only 0.1 percent speculative memory accesses are recognized as unsafe. In case of lbm, milc and zeusmp, they have higher L1 DCache miss rates. Thus their performance improvements brought by *CF* are low. Such analysis is demonstrated in Fig. 6. Most of the programs in SPEC CPU 2006 have high cache hit rates, this filter on average successfully recognizes 86.3 percent speculative accesses as safe, and blocks only 3.8 percent speculative memory accesses on the correct execution path.

Cache-Hit Filter and TPBuf Filter (CTF): Besides exploiting the locality of memory access, the performance improvements are also related to the proportion of cache misses which do not match the S-Pattern. For most cases with high cache hit rates (such as dealII, hmmer and namd), there is little space for optimization. But for applications with low speculative cache hit rates, high S-Pattern mismatch rate denotes that there are large proportions of safe speculative cache misses, which can be recognized by TPBuf Filter and executed speculatively as normal. Therefore, significant performance improvements can be achieved after introducing TPBuf Filter to Cache-hit Filter. For instance, lbm has a lower L1 DCache hit rate (61.8 percent), and there are 86.2 percent speculative accesses mismatching S-Pattern. In this case, *CTF* captures those safe speculations and brings 38.1 percent performance improvement in comparison with *CF*. Another interesting case is libquantum. Although it has also a lower cache hit rate (79.6 percent), more than 99.9 percent accesses belong to *S-Pattern*. These operations are considered unsafe and are blocked for speculation. Thus the performance benefit from *CTF* is limited for libquantum. On average, this mechanism improves the performance by 5.4 percent in contrast with *CF*.

4)] Performance gain from SPBuf

Fig. 7 shows *Conditional Speculation* configured with SPBufs has a further performance improvement. *Speculative Buffer* in LSU and LLC, there are two major reasons for performance improvement, one is spatial locality in L2, and the other is to shorten the time to access main memory. As demonstrated in Table 6, the majority of cases, such as astar, gamess, and hmmer, have excellent locality (more than 99.9

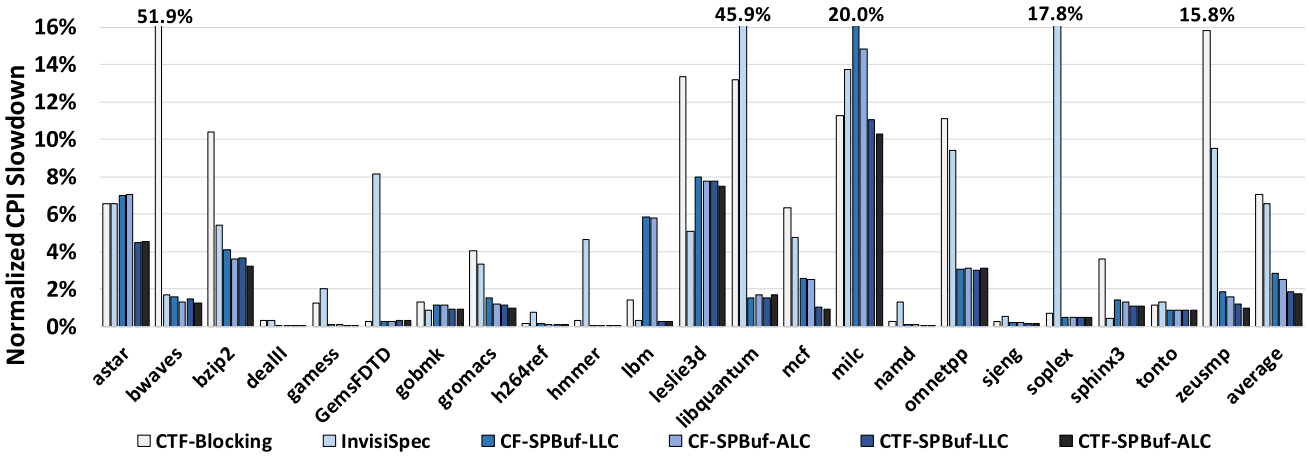


Fig. 7. Performance evaluation of SPBuf (Each case is normalized to its *baseline*).

percent) for *SpecLoads* in L2, which succeeds in providing data to pipeline and accelerating subsequent execution. For cases with low spatial locality, the mechanism helps to shorten the waiting time to refill data from memory, which will improve performance effectively as well. For example, astar, libquantum and zeusmp have large amounts of SpecLoad requests to memory and there are good performance improvements as shown in Fig. 7. Although milc has poor locality in caches as well, there are too many consecutive *SpecLoads* accessing different cache lines, then the miss handler will always be blocked and cannot handle other SpecLoad requests. Therefore, there is less improvement for milc. Shown in Fig. 7, this mechanism significantly reduces the performance overhead to 1.7 percent on average.

In addition, we find that there is a small difference in performance impact between *CF-SPBuf-LLC* and *CF-SPBuf-ALC* as shown in Fig. 7. This shows that deploying Speculative Buffers in private caches does not significantly improve performance because most of the performance improvement of Speculative Buffer in private cache comes from the locality of the private caches, while the locality has been fully utilized by *Cache-hit Filter* in *Conditional Speculation*. *Considering the complexity of deploying SPBuf in private caches, we believe that deploying Speculative Buffer in LLC only is the better option due to its efficiency.*

5) Breakdown of performance benefits from different components

We analyze in detail the percentage of each mechanism for performance gains, which is shown in Fig. 8. In conclusion, compared to Naive Policy blocking all unsafe speculative loads, the SPBuf mechanism significantly improves performance by 34.1 percent. On average, the improvement rates of Cache-hit Filter, TPBuf Filter, and SPBuf are 26.7, 4.1, and 3.3 percent respectively. Most cases have a high L1 hit rate, therefore, Cache-hit Filter is the major contributes the most. For the reason that the characteristics of dynamic execution vary from different cases, the effect of the other two mechanisms is not quite similar. In lbm, mcf, milc, and zeusmp, many unsafe speculative loads do not match S-Pattern, so there is a significant performance boost introduced by TPBuf Filter. For the cases of bwaves, libquantum, and zeusmp, SPBuf dramatically reduces the waiting time for loading data for unsafe speculative loads, resulting in great performance improvements.

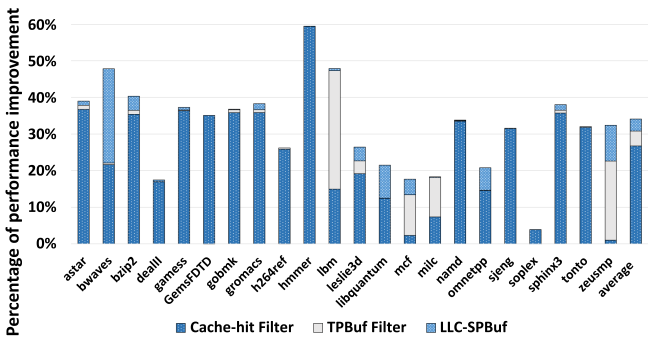


Fig. 8. Performance improvement of three different mechanisms normalized to Naive Policy.

6) Comparison with InvisiSpec

InvisiSpec [17] is a prior work for employing speculative buffer to block Spectre attacks. Based on the same simulation configuration, we compare the performance overhead of *Conditional Speculation* with InvisiSpec. It is noted that we used open-source code implemented by InvisiSpec, which does not have L3 Cache. In our evaluation, InvisiSpec is configured to defeat existing Spectre attacks through cache side channel, as known as InvisiSpec-Spectre. As shown in Fig. 7, the average performance loss of InvisiSpec is 6.5 percent, which is close to *CTF-Blocking*. For the cases of bwaves, bzip2, leslie3d and sphinx3 that benefit less from the *Hazard Filters*, InvisiSpec performs better than *CTF-Blocking*. However, for the cases that have high cache hit rates, such as GemsFDTD, hmmer, and soplex, *CTF-Blocking* reaches lower performance overhead. What's more, all configurations of *Conditional Speculation* with *SPBuf Hazard Response* (*CF-SPBuf-LLC*, *CF-SPBuf-ALC*, *CTF-SPBuf-LLC*, *CTF-SPBuf-ALC*) have better performance than InvisiSpec. Especially for libquantum that has nether good spatial locality nor high S-pattern mismatch rate, *Conditional Speculation* with *SPBuf Hazard Response* further improves the performance.

9.4 Evaluation on Hardware

1) Performance evaluation on RISC-V FPGA platform

Conditional Speculation with *CTF-Blocking* has been implemented on the open-source RISC-V-based Berkeley Out-of-Order Machine (BOOM) processor to observe

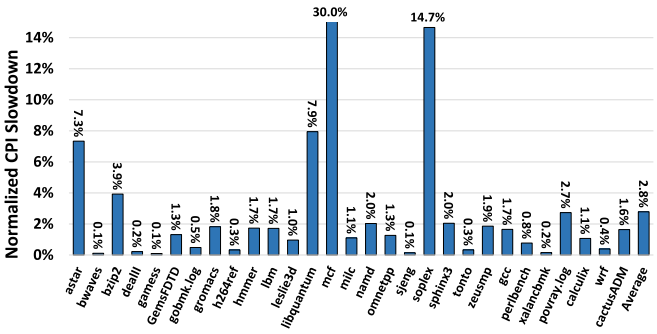


Fig. 9. Performance evaluation on RISC-V FPGA platform.

performance impacts and its defense ability in the real hardware. In our experiments, this processor is configured with a 4-way 10-stage pipeline and a two-level cache hierarchy (L1D/L1I Cache: 32 KB, L2 Cache: 1 MB). And there are 16 entries in the LSQ, 48 entries in the ROB and 20,16,10 entries for memory, integer and float point respectively in the issue queue.

All the benchmarks of SPEC CPU 2006 with train input size run on the Xilinx VC709 FPGA platform at the frequency of 50 *MHz*. For the software transparency of *Conditional Speculation*, the benchmark cases can run successfully without any modification. As shown in Fig. 9, the performance overhead of *CTF-Blocking* on FPGA platform is 2.8 percent on average. Different from simulating a part of the programs in Gem5, the entire programs of benchmarks are simulated in experiment on the FPGA, resulting in the inconsistency of results. However, both of them demonstrate that *Conditional Speculation* does introduce only minimal performance overhead. Finally, we port the previously disclosed Spectre V1 PoC [1] to the processor to evaluate the effectiveness of *Conditional Speculation* against the real-world attack. And experiments show that this mechanism succeeds to defend against this attack.

2)] Hardware overhead evaluation

The security dependence matrix is implemented at Register-Transfer Level (RTL). Based on SMIC 40 nm technology, we use Synopsys ASIC design flow and tools to assess the timing and area cost for such matrix and related control logic. For the issue queue with 64 entries, the additional area occupied by this matrix is 0.05 mm^2 on average, which is only 3.5 percent of a 4-way 32 KB Cache. Synthesizing with TT corner using Design Compiler, the timing of the critical path is only increased by 1.4 percent.

In the implementation of *Cache-hit Filter*, the RTL level modification is marginal, as it only needs to check the safe flag. For the *TPBuf* implementation, TPBuf is placed close to the Load Store Queue (LSQ) and its entries have a 1:1 mapping with the entries of LSQ. The additional area occupied by TPBuf is 0.00079 mm^2 on average, which is about 0.055 percent of a 4-way 32 KB Cache. Compared to the complexity of store-load forwarding and ordering-failure detection in LSQ, TPBuf only involves PPN address comparison logic and does not introduce new critical paths.

We implement the Speculative Buffer with a 1 KB SRAM. It takes up 0.017 mm^2 on average, which is about 1.2 percent of a 4-way 32 KB Cache. In general, its critical path is less than L1D Cache. So it doesn't have an impact on critical paths.

10 Discussion

10.1 Extend Conditional Speculation Beyond DCache Side Channel

As explained in our threat model, this paper aims at Spectre attacks based on cache side channel. However, the micro-architecture states are far beyond cache contents, which can also include physical registers, various queues, TLB and ICache etc. Any other future unknown side channel might be fundamentally different from all the known side channels. Hence, we do not argue that our defense method can effectively cope with all the unknown covert channels. It is worth pointing out that this is a good research problem. We need to combine the idea of *Conditional Speculation* with a new kind of dedicated filter for the new side channel.

One case study is to extend conditional speculation to defend the Spectre variants based on port-contention side channel, such as SMoTherSpectre [29]. The key of this attack is that attackers lure a victim process to speculatively execute a specific gadget. The gadget occupies different execution ports according to the jump direction of the conditional branch instruction, which is data-dependent on the secret. Therefore, the attackers can infer the outcome of conditional branch through port-contention side channel and finally steal the secrets. To defend such kind of attacks, *Conditional Speculation* marks all these speculative conditional branch instructions as *unsafe*, and blocks them in the issue stage until they become non-speculative. Since the speculative execution of conditional branch instructions are blocked, it is only possible for the branch predictor instead of the secret to cause the port contention, and then the attacker can't infer the secret by timing the port competition in the speculative window. In terms of performance, it is noted that *Conditional Speculation* delays the execution of speculative conditional branch, but their subsequent instructions are still allowed to be speculatively executed as normal. So it won't have a significant impact on performance. Even so, a hazard filter can be introduced to further reduce performance loss. One possible approach is to use the compiler to identify the gadgets that might be used to construct the port-contention side channel and mark their dependent conditional branch as unsafe. Compiler will insert instructions or attach additional tag to initialize the specific filter. With this hazard filter equipped, *Conditional Speculation* blocks only the marked conditional branch, and other conditional branches that are not marked are still allowed to execute normally. In summary, SMoTherSpectre can be defended by our extended *Conditional Speculation* effectively.

10.2 Extend Conditional Speculation Beyond Spectre

Naturally, there exists security dependence between the instruction which triggers the transient execution and the instruction which may leak secrets by any side channel. In addition to the two scenarios mentioned, security dependence covers both Meltdown-type and MDS-type attacks, as listed in Table 7. In Meltdown [2] and L1TF [34] attacks, a memory instruction (instruction *i*) that does not resolve the permission check speculatively access sensitive data, after which a memory instruction (instruction *j*) may disclose sensitive information through side channels. Furthermore,

TABLE 7
Security Dependence in Meltdown-Type and MDS-Type Variants (Instr: instruction, mem: memory access, and reg: register)

	Variants	Instr i	Instr j	Channel c
Meltdown	Meltdown [2]	unprivileged mem	mem	cache
	Meltdown V3a [38]	unprivileged reg	mem	cache
	Meltdown-RW [19]	unprivileged write	mem	cache
	Lazy FP [39]	FPU or SIMD reg	mem	cache
	ForeShadow [34]	unprivileged mem	mem	cache
MDS	ZombieLoad [35]	mem	mem	cache
	RIDL [36]	mem	mem	cache
	Fallout [37]	mem	mem	cache

aggressive forward from stale data is a kind of speculation as well in MDS attacks, such as ZombieLoad [35], RIDL [36] and Fallout [37]. In this scenario, the instruction which gets the forwarded data acts as the instruction i, and the subsequent instruction acts as the instruction j.

Conditional Speculation is a defense strategy established on security dependence. Therefore, this defense mechanism can extend to these scenarios. Take ZombieLoad as an example, it exploits the aggressive forwarding mechanism from the line fill buffer in Intel processors, where the processor forward data to subsequent loads even when the located page has been released. As a result, stale data may be forwarded to the subsequent loads. Security dependence can be established between the load which accepts forwarded data and subsequent memory accesses. A *Security Hazard Detection* can be deployed in line fill buffer to mark the forwarded loads as suspect instructions. Because this forwarded data may convey secrets, TPBuf Filter can be applied to detect S-Pattern between this operation and the first subsequent loads.

11 Related Work

Many works are proposed to defend against Spectre attacks and they can be classified into software-based and hardware-based mitigation.

Software-Based Mitigations. Serializing instructions, such as LFENCE, can be inserted into critical sensitive gadgets by manually or compiler to mitigate V1 [40], [41]. In terms of V2, Intel has provided many microcode updates, such as *IBRS* and *IBPB* to avoid malicious branch training across protection domains [9]. As for V4, SSBD stalls speculative loads before calculating the addresses of older stores. *Retpoline*, proposed by Google, transfers indirect branches and jumps to secure instruction sequences to stall aggressive memory accesses [13]. Some researchers propose that a software developer indicates the branches capable of leaking information and the processor avoids predicting them for protection [42]. It is noted that software-based mitigation need code modification and recompilation. Furthermore, they are possible to be bypassed. For example, *LFENCE* defense can be bypassed by Spectre V1.1 and V1.2 [3], [20].

Hardware-Based Mitigations. Existing defense mechanisms can be grouped into the four categories:

① *Prevent malicious training.* BRB [43] allocates separate branch predictor tables for different process to defend against training across process. SPECCFI [44] embeds Control Flow Integrity (CFI) principles into the branch prediction decisions to constrain dangerous speculation. These works are primarily targeted at specific Spectre variants. In comparison, *Conditional Speculation* is a more general defense framework.

② *Prevent speculative execution from leaving traces in caches.* Typical examples are InvisiSpec [17] and SafeSpec [45]. They employ *dedicated buffers* to hold speculatively refilled data temporarily and update architectural caches when they are predicted correctly. Instead of buffering, CleanupSpec [46] allows caches to refill speculative accesses and then undo the changes of cache-states if they were on an incorrect execution path. Besides, DAWG [47] provides a strict cache partition for different security domains to prevent leaving traces of speculative execution on caches across security domains. These works primarily aim at addressing Spectre attacks based on cache side channels.

③ *Restrict the speculative execution of memory access instructions.* Second category conservatively considers all speculative accesses as suspicious, but actually only speculative instructions which access secrets-dependent data and change the cache states are malicious. Instead of pessimistic blocking, SpectreGuard [48] identifies confidential instructions by software (OS/library API) and Context-Sensitive Fencing [49] exploits taint tracking to identify potentially unsafe execution patterns. As a hardware solution with software-transparent, *Conditional Speculation* uses *Hazard Filter* to identify safe accesses. And inspired by InvisiSpec, the remaining few unsafe accesses that previously should have been blocked are allowed to execute speculatively. Similar to the *Cache-hit Filter*, Selective Delay [50] also considers speculative accesses that hit in L1 DCache as safe. The difference is that it employs a value prediction mechanism to reduce the performance overhead caused by blocking unsafe accesses.

④ *Block the propagation of secrets.* The works of the category two and three primarily mitigate Spectre attacks based on cache side channels. researchers pointed out that transient execution attacks require transferring secrets from the speculative domain into the architectural states of the processor no matter what kind of side channel is used to disclose sensitive information. Thus NDA [51], SpecShield [52], and STT [53] are proposed to break this requirement. These mechanisms assume that secrets need to be propagated speculatively at least once before information is leaked to a side channel. They prevent the use of potential secrets from suspicious accesses by downstream instructions and can mitigate information leakage through different side channels. Though current *Conditional Speculation* are designed to defeat cache-based Spectre attacks, new filters can be deployed to defend against other side channels. And *Conditional Speculation* can also introduce a secret data dependence matrix to track the propagation of secrets, further filtering out safe instructions that are independent on secrets and allowing them to execute normally.

12 Conclusion

An effective and software transparent hardware-assisted framework, named as *Conditional Speculation*, is proposed to mitigate Spectre variants based on cache side channel. This framework consists of three components: *Security Hazard Detection*, *Security Hazard Filter* and *Security Hazard Response*. As for the *Security Hazard Detection*, a bit-matrix is introduced in the Issue Queue to identify suspected memory instructions with security dependence. Then, *Security Hazard Filter*s are deployed to pick out safe instructions which do not leave observable traces on cache side channels and

allows them to execute normally. In this paper, we investigate two *Security Hazard Filters*: *Cache-hit Filter* aims at the safe instructions which hit the cache because they will not change cache (content). *TPBuf Filter* identifies safe instructions from a common feature of variants based on shared memory, named as *S-Pattern*. Cache-hit Filter succeeds to defense against all Spectre variants based on cache side channel. With a compromise of security, TPBuf Filter aims at the dangerous and widely-used attacks based on shared pages and gets a better performance. *Security Hazard Response* is employed to handle unsafe instructions determined by the first two components. *Blocking* unsafe execution is an easy-to-implement method, but sacrifices the performance. *SPBuf* mechanism supports retrieving data to the pipeline without changing cache contents, and eliminates the pipeline stalls of subsequent execution. Specifically, we find that it is unnecessary to deploy SPBuf in private caches with *Hazard Filter* identifying most of safe instructions, and deploying SPBuf in last level cache is efficient and less complex.

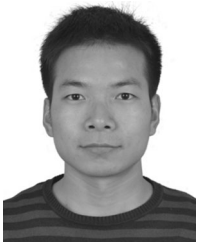
Acknowledgments

This work was supported in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDC02000000 and Beijing Municipal Science & Technology Commission (Project Number: Z191100007119011).

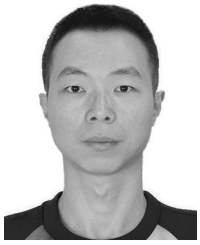
References

[1] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proc. 40th IEEE Symp. Security Privacy*, 2019, pp. 1–19.
[2] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 973–990.
[3] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proc. DARPA Inf. Survivability Conf. Expo.*, 2000, pp. 119–129.
[4] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *CoRR*, abs/1802.03802, 2018, pp. 1–11.
[5] S. Islam *et al.*, "SPOILER: Speculative load hazards boost rowhammer and cache attacks," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 621–637.
[6] M. Hill, "A Primer on the meltdown & spectre hardware security design flaws and their important implications," *Comput. Architecture Today*, 2018. Available: https://cra.org/crn/2018/03/a-primer-on-the-meltdown-spectre-hardware-security-design-flaws-and-their-important-implications/
[7] M. D. Hill, P. Kocher, R. B.Lee, S. Sethumadhavan, and T. Sherwood, "On the implications of the meltdown & spectre design flaws," in *Proc. Int. Symp. Comput. Architecture Panel*, 2018, pp. 1–38.
[8] C. Canella *et al.*, "A systematic evaluation of transient execution attacks and defenses," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 249–266.
[9] Intel, "Intel analysis of speculative execution side channels (Rev.4.0)," Jul. 2018. Available: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf
[10] AMD, "Software techniques for managing speculation on AMD processors (Rev.1.24.18)," 2018. Available: https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf
[11] J. Horn, "speculative execution, variants 4: speculative store bypass," 2018. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528
[12] ARM, "Cache speculation side-channels (Ver.2.4)," Oct. 2018. Available: Downloads/Cache_Speculation_Side-channels-v2.4.pdf
[13] Intel, "Retpoline: A branch target injection mitigation (Rev.003)," Jun. 2018. Available: https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf
[14] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *International Symposium on Engineering Secure Software and Systems*. Berlin, Germany: Springer, 2017.
[15] M. Bynens, "Untrusted code mitigations," 2018. [Online]. Available: https://v8.dev/docs/untrusted-code-mitigations
[16] Intel, "Engineering new protections into hardware," 2018. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/eng ineering-new-protections-into-hardware.html?wapkw=spectre+mitigation
[17] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proc. 51th Int. Symp. Microarchitecture*, 2018, pp. 428–441.
[18] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution," in *Proc. IEEE Eur. Symp. Security Privacy*, 2019, pp. 142–157.
[19] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *CoRR*, vol. abs/1807.03757, pp. 1–12, 2018.
[20] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop Offensive Technol.*, 2018, pp. 1–12.
[21] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 2109–2122.
[22] G. Maisuradze and C. Rossow, "Speculose: Analyzing the Security Implications of Speculative Execution in CPUs," *CoRR*, vol. abs/1801.04084, 2018.
[23] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. Usenix Conf. Secur. Symp.*, 2014, pp. 719–732.
[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 605–622.
[25] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. USENIX Secur. Symp.*, 2015, pp. 897–912.
[26] D. Gruss, C. Maurice, and K. Wagner, "Flush+Flush: A stealthier last-level cache attack," *CoRR*, abs/1511.04594, 2015.
[27] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers Track at the RSA Conference*. Berlin, Germany: Springer, 2006, pp. 1–20.
[28] M. Schwarz *et al.*, "NetSpectre: Read Arbitrary Memory over Network," *Comput. Secur.—ESORICS 2019*, pp. 279–299, 2019.
[29] A. Bhattacharyya *et al.*, "Smotherspectre: Exploiting speculative execution through port contention," *CoRR*, abs/1903.01843, 2019.
[30] B. Sinharoy *et al.*, "IBM POWER8 processor core microarchitecture," *IBM J. Res. Develop.*, vol. 59, no. 1, pp. 2:1–2:21, Jan./Feb. 2015.
[31] A. Henstrom, "Scheduling operations using a dependency matrix," US Patent 6 557 095, Apr. 29, 2003.
[32] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *Proc. IEEE/ACM 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 347–360.
[33] K. Asanović *et al.*, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Dept., Univ. California, Berkeley, Apr. 2016.
[34] J. Van Bulck *et al.*, "FORESHADOW: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Secur. Symp*, 2018, pp. 991–1008.
[35] M. Schwarz *et al.*, "ZombieLoad: Cross-privilege-boundary data sampling," *CCS*, 2019, pp. 1–16.
[36] S. van Schaik *et al.*, "RIDL: Rogue in-flight data load," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 88–105.
[37] M. Minkin *et al.*, "Fallout: Reading kernel writes from user space," *CoRR*, abs/1905.12701, 2019.
[38] ARM, "Vulnerability of speculative processors to cache timing side-channel mechanism," 2018. [Online]. Available: https://developer.arm.com/support/security-update
[39] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *CoRR*, abs/1806.07480, pp. 1–6, 2018.

[40] Intel, "Mitigation overview for potential side channel cache exploits in linux (Rev.2.0)," May 2018. Available: https://software.intel.com/security-software-guidance/api-app/sites/default/files/Intel_Mitigation_Overview_for_Potential_Side-Channel_Cache_Exploits_Linux_white_paper.pdf
[41] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via binary analysis," *CoRR*, abs/1807.05843, pp.1–16, 2018.
[42] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, 2018, pp. 693–707.
[43] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: Mitigating branch predictor side-channels," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2019, pp. 466–477.
[44] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SPECCFI: Mitigating spectre attacks using CFI informed speculation," *CoRR*, abs/1906.01345, pp. 1–15, 2019.
[45] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 60:1–60:6.
[46] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An undo approach to safe speculation," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 73–86.
[47] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2018, pp. 974–987.
[48] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," in *Proc. 56th ACM/IEEE Des. Autom. Conf.*, 2019, pp. 1–6.
[49] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 395–410.
[50] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proc. 46th Int. Symp. Comput. Architecture*, 2019, pp. 723–735.
[51] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 572–586.
[52] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding speculative data from microarchitectural covert channels," in *Proc. 28th Int. Conf. Parallel Architectures Compilation Techn.*, 2019, pp. 151–164.
[53] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 954–968.

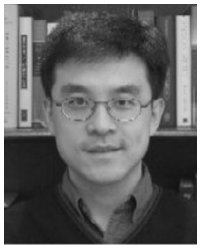


Lutan Zhao received the BE degree from the School of Electrical Engineering and Automation, Henan Polytechnic University, in 2014, and the ME degree from the School of Electronic Engineering, University of Electronic Science and Technology of China, in 2017. He is working toward the doctoral degree with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. His current research interests include computer architecture and hardware security.



Peinan Li received the BE degree from the School of Computer Information and Technology, Shanxi University, in 2014, and the ME degree in computer technology from the Harbin University of Science and Technology, co-educated with the Institute of Automation, Chinese Academy of Sciences, in 2017. He is working toward the doctoral degree with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. His current research interests include computer architecture and hardware security.

Rui Hou received the BS and MS degrees in computer architecture from the Harbin Institute of Technology, China, in 2001 and 2003, respectively, and the PhD degree in computer architecture from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2007. He is a professor and vice director of State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. He has published more than 40 papers in international conferences and journals, and got more than 50 patents. His current research interests include computer architecture, processor security, data center server architecture and, AI security.



Michael C. Huang (Member, IEEE) received the BS degree in computer science and engineering from Tsinghua University, Beijing, in 1994, the MS and the PhD degrees in computer science from the University of Illinois at Urbana-Champaign, in 1999 and 2002, respectively. From 1994 to 1997, he was a lead architect in building a 32-processor hierarchical shared-memory multiprocessor research prototype. He joined the faculty of the Electrical and Computer Engineering department, in 2002. He spent 2010 on sabbatical at IBM T. J. Watson Research Center working on future POWER processor concept development. His research interests include various aspects of high-performance computer architecture such as processor microarchitecture, communication and memory substrate, reliability, and energy-efficient and complexity-effective design. His is particularly interested in addressing emerging issues and exploring new capabilities in the underlying device, circuit, and manufacturing technology. He is a recipient of the NSF CAREER Award and an IBM Faculty Award. He is a member of ACM.

Peng Liu received the BS and MS degrees from the University of Science and Technology of China, and the PhD degree from George Mason University, in 1999. He is a professor of Information Sciences and Technology, founding director of the Center for Cyber Security, Information Privacy, and Trust, and founding director of the Cyber Security Lab at Penn State University. His research interests include the areas of computer and network security. He has published a monograph and more than 220 refereed technical papers. His research has been sponsored by NSF, ARO, AFOSR, DARPA, DHS, DOE, AFRL, NSA, TTC, CISCO, and HP. He has served on more than 90 program committees and reviewed papers for numerous journals. He is a recipient of the DOE Early Career Principle Investigator Award. He has co-led the effort to make Penn State a NSA-certified National Center of Excellence in Information Assurance Education and Research. He has advised or co-advised around 20 PhD dissertations to completion.

Lixin Zhang (Senior Member, IEEE) received the BS degree in computer science from Fudan University, in 1993, and the PhD degree in computer science from the University of Utah, in 2001. He is a professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His main research areas include computer architecture, data center computing, high performance computing, advanced memory systems, and workload characterization. He was previously a research staff member with IBM Austin Research Lab and a master inventor of IBM.

Dan Meng received the BS, MS, and PhD degrees from the Harbin Institute of Technology, Harbin, Heilongjiang, China. He is the director of Institute of Information Engineering, Chinese Academy of Sciences, and the dean of the School of Cyber Security, University of Chinese Academy of Science, Beijing, China. His research interests include high performance computer architecture and cyber security.