Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks

Peinan Li†, Lutan Zhao†, Rui Hou†*, Lixin Zhang‡ and Dan Meng†
†*State Key Laboratory of Information Security, Institute of Information Engineering, CAS and University of Chinese Academy of Sciences. Email: {lipeinan, zhaolutan, hourui, mengdan}@iie.ac.cn*
‡*Institute of Computing Technology, CAS. Email: zhanglixin@ict.ac.cn*

***Abstract*—Speculative execution side-channel vulnerabilities such as Spectre reveal that conventional architecture designs lack security consideration. This paper proposes a software transparent defense mechanism, 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 risk. 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 will not refill new cache lines with unauthorized privilege data. Otherwise, they are considered as *unsafe* instructions and thus not allowed to execute speculatively. To reduce the performance impact from not executing *unsafe* instructions speculatively, we investigate two filtering mechanisms, *Cache-hit based Hazard Filter* and *Trusted Page Buffer based Hazard Filter* to filter out false security hazards. Our design philosophy is to speculatively execute *safe* instructions to maintain the performance benefits of out-of-order execution while blocking the 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.**

***Keywords*-Spectre vulnerabilities defense; Security dependence; Speculative execution side-channel vulnerabilities;**

I. 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 micro-architectural 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 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].

Corresponding author: Rui Hou(hourui@iie.ac.cn)

Industrial researchers have responded rapidly to mitigate these threats [8], [9], [10], [11]. Retpoline, proposed by Google, converts indirect jump instructions into a blocking loop that combines return instructions to avoid unsafe speculative execution [12]. 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 [13]. Various isolation mechanisms, such as KAISER and Site Isolation, are developed to shutdown the observable channel between security domains [14]. 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. On March 15, 2018, Intel reported that its newly redesigned processors released later in 2018 will protect against Meltdown and related Spectre vulnerabilities (especially, Meltdown and Spectre-V2) [13]. 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). More specifically, our work concentrates on vulnerabilities associated with branch speculation and memory access speculation. Overall, this paper makes the following contributions:

- 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 risk.
- An effective and software transparent defense mechanism against Spectre vulnerability, named as *Conditional Speculation*, is proposed for generic microprocessors. Specially, *Security Hazard Detection* is introduced in

the Issue Queue to identify suspected unsafe instructions with security dependence. Once the real hazards are confirmed at the execution stage, those unsafe speculative execution will be terminated and discarded using existing re-execution and speculation recovery mechanisms.

- 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 proposed *Cache-hit based Hazard Filter* targets at 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 our targeted Spectre variants that use the shared memory (e.g, Flush+Reload) based cache side channel and steal memory page information, their speculative execution of malicious gadgets have a common feature named as *S-Pattern*. TPBuf is designed to capture *S-Pattern* from all speculative executions. For any speculative executed memory instructions, it is considered as safe if it does not match the *S-Pattern*.

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. And Section 5 introduces the mechanism of *Conditional Speculation*. Section 6 is the evaluation. Section 7 is the discussion. And Section 8 contains related works. Section 9 concludes this paper.

II. 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 are able to use a side channel to transmit/leak victim's information out. The typical Spectre attacks have three common key steps listed as below.

A. Induce victim to incorrect speculative execution path

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

Branch speculation. Branch prediction is one of major speculative execution techniques [1], [2], [4], [5], [15]. Through purposeful training of the branch predictors, an adversary can change the control flow to incorrect speculative execution path to access the unauthorized data. Some processors use static branch predictor, which makes it much easier for 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* [4], [10]. Load instruction is usually allowed to be speculatively executed even if the address of its older store instruction is unknown. Attackers can exploit such design to induce the load instruction to speculatively access sensitive data illegally.

B. Construct a long time 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 cache line flush instruction or other ingenious methods to evict their source operands into off-chip memory [1], [2], [4], [5], [10]. 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 longer timing window for stable speculative executions [16].

C. Infer secrets from side-channel information leakages

The incorrect speculative execution might leave traces 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 [17], [18], Prime+Probe[19], Evict+Reload[20], Flush +Flush [21] and Evict+Time [22].

III. Threat Model

We have following assumptions on attacker. She can execute her codes on the same machine with victim process without elevated privileges. And it is possible for the attacker to know the source codes and address layout of the victim process/thread.

This paper aims at a large class of representative, realistic, and dangerous Spectre attacks. They steal victim's memory contents instead of the value of registers (Note that stealing memory contents is perhaps more dangerous than stealing register values), and they rely on the shared memory to construct the cache side channel between the attacker and the victim to leak the information (Note that this is more realistic and efficient than other forms of side channels). We define the above defense scope primarily for the following reasons:

1) The community has not yet found a way to enumerate all the possible side channels. In theory, many choices are possible for the side-channel component. Thus it should be noted that Spectre does not restrict itself 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 known Spectre variants.

2) There are two major types of methods to construct the cache side channel: shared memory based approach and eviction set based approach. Shared memory based approach, such as Flush+Reload, Flush+Flush and Evict+Reload, takes advantage of shared pages between the attacker and the victim. Eviction set based approach, such as Prime+Probe and Evict+Time, monitors the states of a eviction cache set carefully selected by the adversary without sharing any contents. Since it is much more convenient to monitor the fine-grained cache status, shared memory based approach has higher resolution [23]. Furthermore, shared memory based approach does not suffer from false positives, complex processing for detecting access and frequently interrupts [24]. Consequently, it is not only more powerful but also more dangerous, which is widely employed in existing Spectre variants.

Note that while we limit our discussion to this threat model in this paper, the basic architecture can be adjusted for expanded threat model. We leave that as future work.

IV. Security Dependence and Defense Strategy

A. Definition of 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. In this paper, we focus 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 in this paper – even though its speculative execution definitely leaks *some* information into the universe.

Given the above definition, Table I summarizes the major security dependence in Spectre vulnerabilities. The security dependence in Spectre vulnerabilities come from two situations, including memory-memory speculation and branch-memory speculation.

Table I: Security dependence in Spectre vulnerabilities

Variants	Instruction *i*	Instruction *j*
Spectre V1	conditional branch	memory access
Spectre V2	indirect branch	memory access
Spectre V4	memory access	memory access
SpectrePrime	conditional branch	memory access

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) [25]. Instruction 1 ($i1$ hereafter) is a store operation, and $i4$ is a load operation. Assuming these two instructions access the same memory address, there is a RAW dependence. The attacker might construct an environment in which $i1$ is pending in the issue queue and $i4$ is speculatively launched for accessing unauthorized sensitive data. Once the address of $i1$ is known, the load mis-peculation occurs and the incorrect execution results of $i4$ need to be discarded. However, the sensitive data have already been refilled into L1 cache, such information leakage allows attacker to infer the sensitive data. Therefore we say $i4$ is security dependent on $i1$.

```
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 1: PoC Code piece of Variant 4: Speculative Store 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
9   ...
```

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

Security dependence under branch-memory speculation: In case of Spectre V1 in Listing 2. In attacker's 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 attack. According to our definition, $i7$ is security dependent on $i6$.

B. Defense Strategy

Speculation and cache side-channel information leakages are critical factors for a successful Spectre attack. A straightforward defense policy is to prohibit any speculation of

memory instructions. However, speculation is a fundamental performance boost technique in modern microprocessors. Disabling speculative execution has severe negative performance impacts and is not a practical option in most cases. In addition, not all speculative memory access instructions pose risk of leaking important information. We therefore propose a defense strategy that **supports speculative execution but is capable of blocking speculative instructions with security dependence**. This approach can lead to a desirable trade-off between performance and security. In the meantime, we also strive to keep it practical by maintaining acceptable implementation cost and complexity.

V. Conditional Speculation Mechanism

A. Design Overview

This section proposes a generic design of core microarchitecture enforcing security dependence as shown in Figure 1. The *security hazard detection module* is integrated into the issue queue to identify the security dependent memory access instructions. These instructions are tagged with the *suspect speculation* flags, which indicate they POSSIBLY change cache contents due to mis-speculation. Then they will be speculatively issued for execution once their data dependence are cleared.

For those instructions with *suspect speculation* flags, they are considered as *safe* instructions if their speculative execution WILL NOT refill new cache lines with unauthorized privilege. Otherwise, they are *unsafe* instructions. Mentioned in the above section, our design philosophy is to speculatively execute *safe* instructions to maintain the performance benefits of out-of-order execution while blocking the speculative execution of *unsafe* instructions for security consideration. We propose two filtering mechanisms to figure out false security hazards and to decide whether the instruction with *suspect speculation* flag is safe or not, with the goal of pursuing a balance of performance and security. The proposed *Cache-hit based Hazard Filter* targets the speculative instructions which hit the cache without any cache (content) side-channel information leakage. And *Trusted Page Buffer based Hazard Filter* identifies safe speculative instructions from another perspective. For our targeted Spectre variants that use the shared memory based cache side channel and steal memory page information, their speculative execution of malicious gadgets have a common feature named as *S-Pattern*. TPBuf is designed to capture *S-Pattern* from all speculative executions. For any speculative executed memory instructions, it is considered as safe if it does not match the *S-Pattern*. The instructions that survive the filtering are allowed to be aggressively speculated, thereby obtaining better performance in the context of security.

B. Security Hazard Detection in Issue Queue

We design the security detection logic based on bit matrices as shown in Figure 2. Bit-matrix is a popular way used by some commodity processors to track data dependence and age information [26], [27], [28]. Conventionally, the data dependence matrix and age matrix together can determine the instruction(s) to be issued. With security detection module, the security dependence matrix also must determine if an instruction to be issued has any security dependence.

Matrix organization: Security dependence matrix needs to efficiently support both row and column access. Assuming that the Issue Queue has N items, the security dependence matrix will contain a register array of NxN bits. It is indexed by IQPos (Issue Queue Position). 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. Given any Instruction X, IQPos_X denotes its location in the Issue Queue. If the value of the Matrix[IQPos_X, IQPos_Y] is 1, 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 Issue Queue, one entry is allocated with the index IQPos_X. For each Instruction Y which is valid in the Issue Queue at this moment, Matrix[IQPos_X,IQPos_Y] is computed according to the following formula.

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

This formula is based on the following logic to determine the security dependence between instructions. First, if Y is valid before X is dispatched into Issue Queue, it means Y precedes X. Second, for Spectre variants, we check only if a memory instruction is security-dependent on previous branch or memory instructions. Third, if preceding branch or memory instructions are still waiting in the Issue Queue when a memory instruction is issued, this memory instruction will be considered to have security dependence.

Hazard detection: Figure 2 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 one instruction is selected to be issued and a security hazard is detected, it will be tagged with *suspect speculation* flag.

Dependence clearance: After one 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

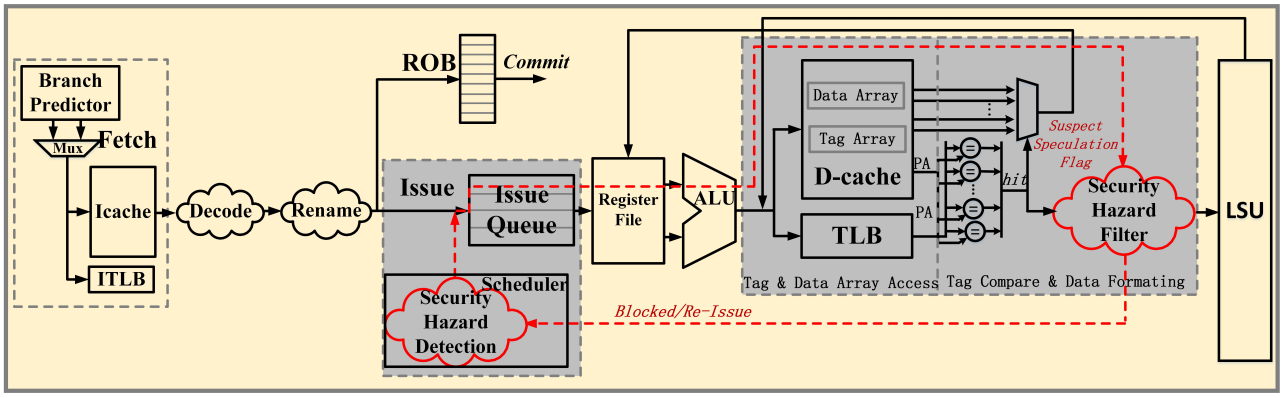

Figure 1: The microarchitecture overview of Conditional Speculation mechanism

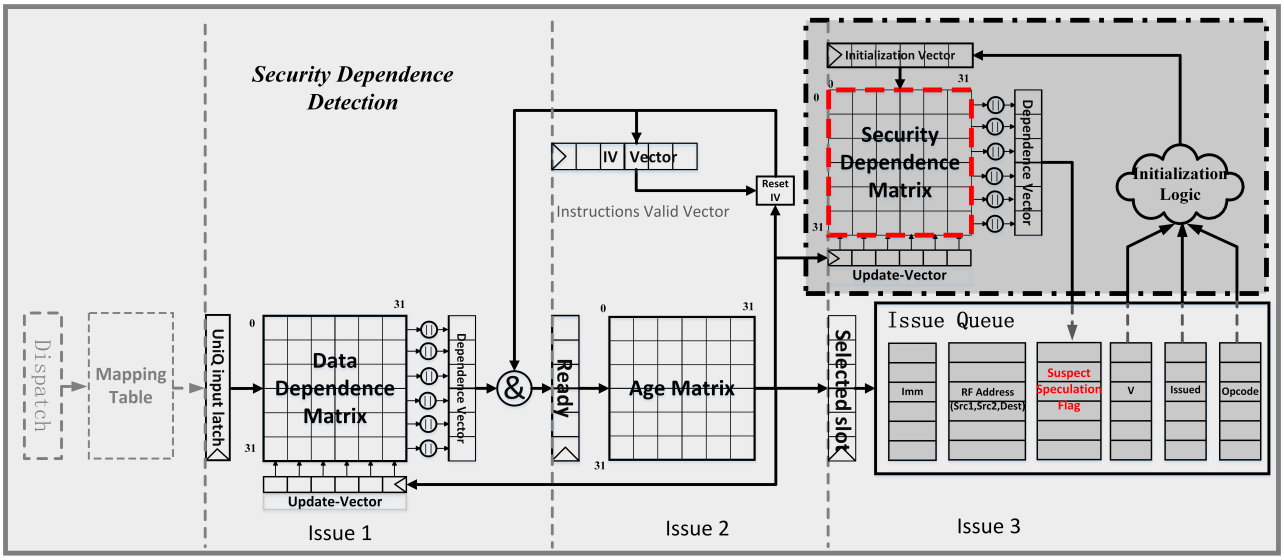

Figure 2: Security Hazard Detection based on Security Dependence Matrix

means that the security dependence between corresponding instructions and X are cleared.

C. Cache-hit based Hazard Filter

A straightforward approach of hazard elimination is to simply block all instructions tagged with *suspect speculation* flags in the issue queue. Such a policy obviously would cause performance degradation. Meanwhile, only memory requests resulting in a cache miss will change cache content, and most of applications exhibit good temporal and spatial locality. Motivated by these two observations, *Cache-hit based Hazard Filter* is proposed to reduce the performance impacts from conservatively not executing instructions with a *suspect speculation* flag.

For a memory instruction tagged with a *suspect speculation* flag, it will be speculatively issued to the memory access pipeline. If the speculative memory access hits L1 cache, its execution will continue as a normal memory instruction. However if it encounters a miss in L1 cache, the missing request will be discarded. A signal is sent back from L1 cache to Issue Queue that the re-issue logic

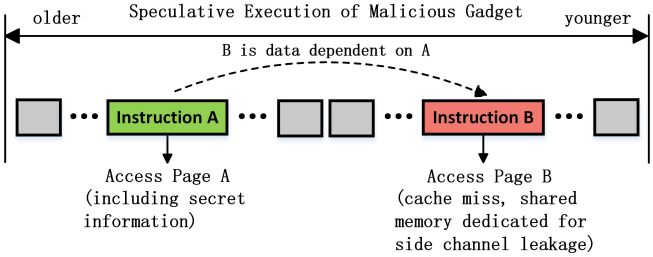

Figure 3: Typical instruction flow of malicious gadget for shared memory (e.g, Flush+Reload) based Spectre attack.

should be applied to the memory instruction until its security dependence is resolved. These blocked instructions wait in the issue queue for the security dependence to clear before being re-issued. This design requires only minimal changes in the L1 cache control logic: a miss request with *suspect speculation* flag will not be processed.

D. Trusted Pages Buffer based Hazard Filter

(1) Motivation

The *Cache-hit based Hazard Filter* only considers speculative memory instructions that hit in L1 DCache as safe and allows them to be speculatively executed. For applications with high L1 DCache miss rates, it will not be able to recover majority of the benefits of speculative execution. For those applications, we propose a new hazard filter *Trusted Page Buffer (TPBuf)* to detect safe speculation for instructions that miss in L1 DCache. TPBuf is based on the observation that not all speculative cache misses can be exploited to construct speculative side channels.

Based on the threat model defined in Section 3, we focus on Spectre variants that utilize the shared memory (e.g, Flush+Reload) paradigm and illegally access memory page information. As shown in Figure 3, the speculative execution of malicious gadget is featured as a common memory access pattern in our targeted Spectre attacks. In particular, it is observed that the malicious speculative execution flow always contain two special memory instructions (A and B). These two instructions have following usages and behaviors.

1) A is used to speculatively access sensitive data. And B speculatively accesses the memory region shared by attacker, which is used for building the cache side-channel between victim and attacker. Since secret data and memory region used for side-channel usually locate at different memory pages, these two instructions access different pages.
2) 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.
3) B is data-dependent on A. The result of A is used to calculate the index of shared memory region. Such well-designed dependence is also another important point for attacker to infer the secret values.

Motivated by 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) which separately access different memory pages.
2) Instruction B results in an L1 DCache miss.
3) Instruction B has data dependence on instruction A.
4) There may be multiple instructions (computation, memory or other kind of instructions) between A and B.

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 security reasons, we try to prevent the formation of speculative instruction flows with *S-Pattern* at the micro-architecture level. While ensuring security, such mechanism also naturally leads to possible performance losses. In Section 6, we evaluate the performance of this strategy in detail and analyze the proportion of *S-Pattern* in normal programs like SPEC CPU 2006.

Table II: 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

(2) Overview of TPBuf design

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 track their execution status (e.g., whether the requested cache line is refilled or not). When a new memory request which misses 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 II.

(3) Microarchitecture of TPBuf

The microarchitecture of TPBuf is shown in Figure 4. One main design principle is to utilize the existing logics as much as possible to reduce the complexity of implementation, such as avoiding TPBuf to be the new timing critical path inside the core pipeline. 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. In addition, 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. 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 or not a suspect speculative miss request should be blocked. In this way, the original memory consistency model and cache coherence are unaffected. Lookup of TPBuf is shown in Figure 4.

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.

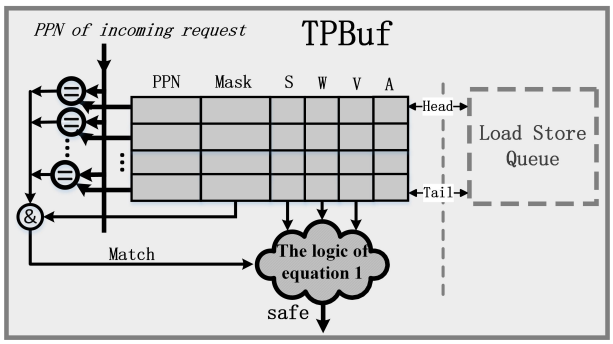

Figure 4: The microarchitecture design of TPBuf (*PPN:physical page number Mask:indicates program order S:suspect speculation W:writeback V:address is valid A:entry is allocated*)

It indicates which memory instructions in TPBuf are older than the new entry in 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 become available to others 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.

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

VI. Evaluation

A. Methodology

We have used the cycle-accurate simulator Gem5 [29] to model a generic high-performance out-of-order processor. We simulate the mechanisms of *Conditional Speculation* to evaluate the features in terms of security, performance and area cost. Table III lists the key parameters of the simulated processor.

Security features are evaluated through analyzing Proof-of-Concept (PoC) codes. We adopt SPEC CPU 2006 benchmarks with reference input size for performance evaluation [30]. The simulator is warmed up for one billion instructions, and then run another billion instructions in the cycle accurate mode. Finally, the *Security Dependence Matrix* is implemented using Register-Transfer Level (RTL) code and then synthesized and implemented with SMIC $40nm$ technology for area and timing evaluations.

Experiment environments with different mechanisms of *Conditional Speculation* are named in the following abbreviations:

Table III: Simulator Configurations.

Parameter	Configuration
ISA	ALPHA
Frequency	2.5*GHz*
Processor type	4-way out-of-order
Pipeline	15 stages
Commit	Up to 4 instructions/cycle
ROB	192 entries
LDQ	32 entries
STQ	24 entries
Issue Queue	64 entries
ITLB/DTLB	64 entries
L1 ICache	64KB, 4-way, 64B line, 2 cycle hit
L1 DCache	64KB, 4-way, 64B line, 2 cycle hit
L2 Cache	2MB, 16-way, 64B line, 10 cycle hit
L3 Cache	8MB, 32-way, 64B line, 60 cycle hit
Memory	8GB, 192 cycle latency

- **Origin:** Base out-of-order processor with the configuration listed in Table III, without any security dependence modules.
- **Baseline:** Baseline mechanism of *Conditional Speculation*, which simply considers all the security-dependent memory accesses as *unsafe*.
- **Cache-hit Filter:** The mechanism of *Conditional Speculation* with *Cache-hit based Hazard Filter*.
- **Cache-hit Filter + TPBuf Filter:** The mechanism of *Conditional Speculation* with *Cache-hit based Hazard Filter* and *Trusted Pages Buffer based Hazard Filter* working together.

B. Security Analysis

Table IV summarizes the security analysis of the proposed three defense mechanisms. The major known Spectre variants can be divided into six typical scenarios based on different combinations of cache side-channel attacks and page sharing mode. The first four are in the scope of our threat model, and they are our main defensive goals. Furthermore, we analyze two more scenarios, including Prime+Probe and Evict+Time.

In the cases of *Baseline* and *Cache-hit Filter*, speculative memory accesses are not allowed to change the cache state (content). Thus those two mechanisms can ensure the security of speculative execution. For *Cache-hit Filter + TPBuf Filter*, it is able to block the unsafe speculation for the first four scenarios via dynamically identifying the *S-Pattern*. However, *S-Pattern* is not designed for cache side-channel attacks which are based on non-shared pages. Therefore, *Cache-hit Filter + TPBuf Filter* can not provide secure solution for the last two scenarios listed in Table IV. It should be noted that all published PoC codes of existing Spectre attacks construct side channels based on shared memory pages, and they can be grouped to: "Flush+Reload, share data" (Spectre V1, V1.x V2, V4), and "Prime+Probe,

Table IV: Security analysis of three mechanisms of conditional speculation.

Attack Classification	Baseline	Cache-hit Filter	Cache-hit + TPBuf Filter
Flush+Reload, share data	✓	✓	✓
Flush+Flush, share data	✓	✓	✓
Evict+Reload, share data	✓	✓	✓
Prime+Probe, share data	✓	✓	✓
Prime+Probe, no shared data	✓	✓	×
Evict+Time, no shared data	✓	✓	×

share data" (SpectrePrime). In summary, our mechanism can defend those known Spectre attack methods.

C. Performance Evaluation

Figure 5 compares the performance impacts of three different mechanisms of *Conditional Speculation*. The *Baseline* mechanism blocks all speculative memory accesses for security consideration. Not surprisingly, such conservative policy causes the largest performance degradation (53.6% performance degradation on average, and the worst case is 146.8% for hammer). In contrast, *Cache-hit Filter* provides a certain degree of relaxation. By dynamically identifying false security hazards, 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 53.6% to 12.8%). In particular, *Cache-hit Filter* recognizes 89.6% speculative accesses as safe due to the high L1 DCache hit rate for SPEC CPU benchmarks. *Cache-hit Filter + TPBuf Filter* 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 as safe and can still be speculatively executed. It can be observed that *Cache-hit Filter + TPBuf Filter* further reduces the performance overhead to 6.8% on average.

(1) Performance overhead for conservative blocking policy

The security dependence comes from two major situations, including 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 23.0% 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. For the worst case of astar (65.5% overhead), which has a high branch misprediction rate (8.5%). Besides, 16.7% instructions are unresolved branch instructions when they are allocated in the Issue Queue, and 27.2% memory instructions are marked as unsafe.

After appending the *memory-memory* dependence, the proportion of unsafe speculative memory accesses is increased. 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%, and it takes more than 150 times longer to resolve a branch and pending speculative memory accesses than the *Origin* case. As depicted in Table V, the Baseline policy will block almost 73.6% speculative memory accesses in correct execution path.

(2) Performance gain for two kinds of filters

Cache-hit Filter: This filter exploits locality of memory access behaviors of normal workloads. Compared to *Baseline* method, *Cache-hit Filter* improves the performance by 26.6% on average as shown in Figure 5. Take GemsFDTD in Table V as an example, the cache hit rate is more than 99.9%, and then only 0.1% speculative memory accesses are recognized as unsafe. In case of lbm, milc and zeusmp, they have higher L1 DCache miss rate. Thus their performance improvements brought by *Cache-hit Filter* are low. Such analysis is demonstrated in Figure 5. Most of programs in SPEC CPU 2006 have high cache hit rates, this filter on average successfully recognizes 89.6% speculative accesses as safe, and blocks only 3.6% speculative memory accesses in the correct execution path.

Cache-hit Filter + TPBuf Filter: Besides exploiting the locality of memory access, the performance improvements of this mechanism is 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 rated, 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 when TPBuf Filter and Cache-hit Filter are combined. For instance, lbm has a lower L1 DCache hit rate (61.8%), and there are 86.2% speculative accesses mismatching S-Pattern. In this case, *Cache-hit Filter + TPBuf Filter* captures those safe speculations and brings 38.1% performance improvement in comparison with *Cache-hit Filter*. Another interesting case is libquantum. Although it has also a lower

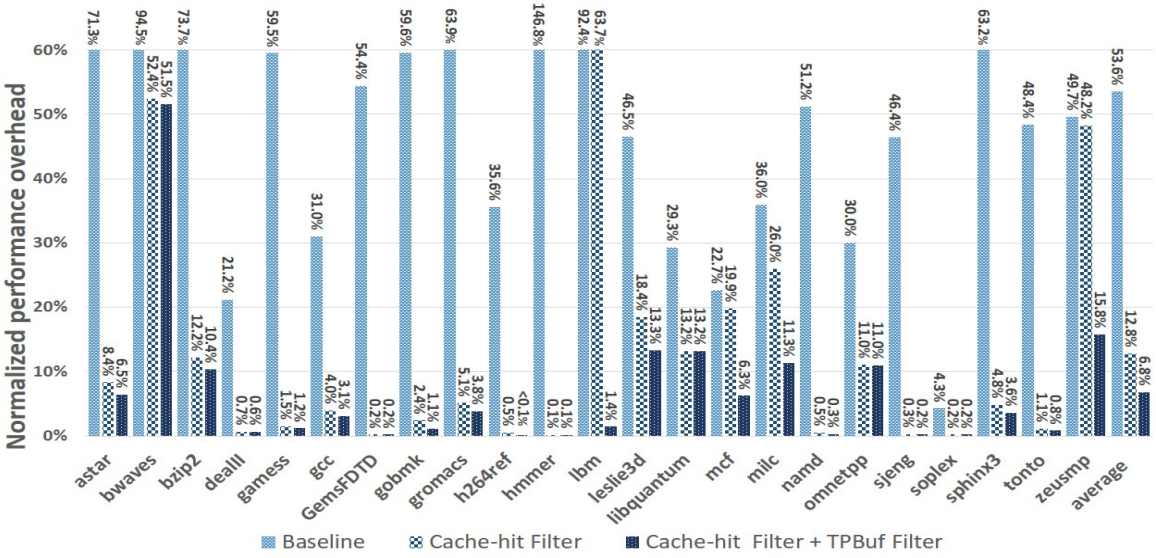

Figure 5: Performance evaluation (All the values in this figure are normalized to *Origin*).

Table V: Filter Analysis (Blocked Rate means the proportion of blocked speculative memory accesses in the correct execution path).

Benchmark	Origin	Baseline	Cache-hit Filter		Cache-hit Filter + TPBuf Filter	
	L1 Hit Rate	Blocked Rate	Blocked Rate	Cache Hit Rate of Speculative Memory Access	Blocked Rate	S-Pattern Mismatch Rate
astar	94.4%	74.6%	3.3%	90.4%	2.2%	14.5%
bwaves	81.3%	73.0%	5.6%	90.3%	5.5%	1.5%
bzip2	96.7%	77.8%	1.6%	95.5%	1.3%	5.0%
dealII	97.3%	58.7%	0.1%	**99.4%**	0.1%	15.5%
gamess	96.0%	75.0%	0.5%	98.8%	0.4%	10.8%
gcc	96.2%	79.1%	0.4%	95.3%	0.2%	18.8%
GemsFDTD	>**99.9%**	79.1%	<**0.1%**	99.9%	<0.1%	0.2%
gobmk	95.3%	72.5%	1.6%	96.3%	0.2%	**39.4%**
gromacs	93.8%	71.4%	2.1%	94.8%	1.1%	19.0%
h264ref	99.1%	62.5%	0.3%	98.3%	<0.1%	**47.0%**
hmmer	97.9%	65.4%	0.3%	**99.4%**	0.3%	2.1%
lbm	**61.8%**	65.9%	**15.8%**	**60.7%**	**0.3%**	**86.2%**
leslie3d	95.1%	85.3%	1.6%	96.5%	1.2%	17.2%
libquantum	**79.6%**	88.4%	1.6%	95.2%	1.6%	<**0.1%**
mcf	**73.9%**	65.2%	**9.3%**	**75.1%**	**3.2%**	**32.6%**
milc	**66.2%**	77.9%	**13.0%**	**67.6%**	9.2%	6.3%
namd	97.5%	77.4%	0.2%	**99.6%**	0.1%	**31.9%**
omnetpp	92.9%	76.7%	4.4%	78.2%	4.1%	0.8%
sjeng	99.4%	78.1%	<0.1%	99.7%	<0.1%	11.9%
soplex	84.9%	71.0%	3.3%	82.1%	3.3%	0.3%
sphinx3	97.9%	77.4%	0.3%	96.6%	0.2%	13.1%
zeusmp	**55.3%**	67.0%	**15.0%**	**61.5%**	**3.9%**	**26.9%**
Average	88.7%	**73.6%**	**3.6%**	89.6%	**1.7%**	18.2%

cache hit rate (79.6%), more than 99.9% accesses belong to *S-Pattern*. These operations are considered unsafe and are blocked for speculation. Thus the performance benefit from *Cache-hit Filter + TPBuf Filter* is limited for libquantum. On average, this mechanism improves the performance by 5.3% in contrast with *Cache-hit Filter*.

D. Sensitivity Analysis on the Complexity of Out-of-Order Core

Three typical processor cores with different complexities are simulated for sensitivity evaluation. We choose A57-like configuration for the scenario of mobile processor, Core I7-like configuration for desktop processor and Xeon E5

Table VI: Parameter Sensitivity Analysis.

Benchmark	A57-like			I7-like			Xeon-like		
	Baseline	Cache-hit Filter	Cache-hit Filter + TPBuf Filter	Baseline	Cache-hit Filter	Cache-hit Filter + TPBuf Filter	Baseline	Cache-hit Filter	Cache-hit Filter + TPBuf Filter
astar	46.0%	7.2%	5.5%	49.0%	9.8%	8.2%	53.8%	11.2%	9.2%
bwaves	89.6%	42.7%	41.8%	87.4%	51.8%	51.6%	88.7%	53.1%	52.5%
bzip2	43.3%	12.3%	9.3%	69.7%	21.0%	19.7%	85.8%	28.0%	22.3%
dealII	40.4%	0.7%	0.2%	18.0%	0.5%	0.7%	22.6%	0.9%	1.3%
gamess	25.9%	1.5%	1.4%	53.3%	2.2%	1.4%	61.4%	2.5%	1.7%
gcc	23.3%	2.6%	1.8%	25.2%	3.9%	2.7%	25.8%	4.4%	3.0%
GemsFDTD	32.6%	0.6%	0.6%	44.6%	0.5%	0.3%	53.1%	-0.2%	-0.6%
gobmk	36.0%	2.2%	1.2%	36.2%	3.7%	1.8%	40.4%	4.2%	2.0%
gromacs	43.7%	4.6%	5.5%	52.6%	7.8%	5.8%	55.4%	9.0%	7.0%
h264ref	19.5%	0.5%	0.1%	31.0%	0.7%	0.3%	37.7%	0.7%	0.3%
hmmer	109.4%	1.2%	1.1%	127.7%	1.7%	1.6%	156.0%	3.7%	3.6%
lbm	72.3%	47.8%	0.7%	74.4%	53.3%	1.1%	73.1%	47.8%	1.1%
leslie3d	45.6%	16.6%	12.9%	40.0%	21.6%	14.8%	38.0%	19.0%	13.1%
libquantum	38.7%	10.4%	10.4%	25.5%	13.4%	13.4%	26.7%	14.2%	13.8%
mcf	16.0%	13.5%	3.6%	24.0%	19.7%	4.7%	25.1%	23.1%	5.0%
milc	35.6%	21.7%	10.4%	31.9%	23.9%	8.7%	32.0%	24.1%	10.1%
namd	37.7%	1.2%	0.6%	42.3%	1.4%	0.7%	50.0%	1.5%	0.8%
omnetpp	22.4%	8.4%	8.4%	52.5%	40.2%	40.0%	62.5%	45.8%	44.9%
sjeng	30.0%	0.4%	0.2%	32.2%	0.2%	0.2%	35.1%	0.3%	0.2%
soplex	2.6%	0.1%	0.1%	2.3%	0.2%	0.2%	3.1%	0.2%	0.2%
sphinx3	49.2%	4.2%	2.5%	52.4%	8.4%	5.3%	58.4%	8.8%	5.5%
zeusmp	44.1%	42.5%	14.4%	46.7%	45.9%	14.9%	47.1%	46.4%	15.0%
Average	41.1%	11.0%	6.0%	46.3%	15.1%	9.0%	51.4%	15.9%	9.6%

V4-like configuration for server processor. First, for our proposed three mechanisms, the same trend can be observed for different processor platforms. Secondly, as processors become more complex, the performance overhead of our mechanisms has increased to a certain extent. For *Cache Hit Filter + TPBuf filter*, the performance loss on A57-like processor platform is 6% and 9.6% on Xeon platform.

E. Hardware Overhead Evaluation

The security dependence matrix is implemented at Register-Transfer Level (RTL). Based on SMIC 40nm 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 $0.05mm^2$ on average, which is only 3.5% of a 4-way 32KB Cache. Synthesizing with TT corner using Design Compiler, the timing of the critical path is only increased by 1.4%.

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 $0.00079mm^2$ on average, which is about 0.055% of a 4-way 32KB 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.

VII. Discussion

A. Secure Update for Cache Replacement Logic

It should be noted that speculative accesses that hit L1 DCache under *Conditional Speculation* may still leak information via updating the cache replacement metadata (e,g, LRU bits) speculatively. Such kind of leakage might be exploited by adversaries [31], [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 a *no update policy*. This policy skips LRU updates for speculative accesses that hit L1 DCache. For speculative accesses that eventually become non-speculative, not updating LRU bits can diminish the effectiveness of L1 DCache replacement policy. To understand the performance implication of this policy, we have evaluated it on top of *Cache-hit Filter + TPBuf Filter* using the same set of benchmarks in Figure 5. The results show that it introduces 0.71% performance degradation.

We have also evaluated a *delayed update policy* that sets a *pending LRU update* tag when a speculative accesses hit L1 DCache and performs the actual LRU update when the access reaches the head of the LSQ (i.e., becomes non-speculative) and the corresponding LRU array is not being

used by accesses from the load/store pipelines. For the same set of benchmarks in Figure 5, our experiments show that this policy improves *no update policy* by 0.26%. Considering the complexity of this policy, we believe that *no update policy* is the better option due to its simplicity. Further optimization of *no update policy* is possible where every bit of performance is desired and is part of future work.

B. Extend Conditional Speculation beyond DCache Side Channel

As explained in our threat model, this paper aims at a large class of Spectre attacks which rely on the shared memory to construct the cache side channel between the attacker and the victim. However, the microarchitecture 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 new kind of dedicated filter for the new side channel.

One case study is the extension to ICache based side channel. The *Conditional Speculation* plus similar *ICache-hit filter* can be applied to ICache based side channel. As long as there is a unresolved branch instruction in pipeline, the NPC (next PC) in fetch stage is marked as unsafe status. If the requests of the unsafe NPCs hit in L1 ICache, the instructions will be fetched to pipeline. Otherwise the ICache access will be stalled until its preceding security-dependent branch instructions are resolved. It is one of our ongoing work to evaluate its performance impact of *ICache-hit filter*.

VIII. Related Work

Software-based Mitigations: *LFENCE* instruction can be employed to mitigate speculative load operations in critical sensitive gadgets [33]. Furthermore, *oo7* is proposed as binary analysis framework to check and fix code snippets using less fences [34]. In term of V2, Intel has provided many microcode updates to prevent speculative execution side channel. *IBRS* and *IBPB* are proposed to avoid the malicious branch training in same or different logic processors [13]. *SMEP* can be used to prevent speculative execution from user to kernel, thereby avoiding observing kernel data. As for V4, SSBD stalls speculative loads before calculating the addresses of older stores. *Retpoline*, proposed by Google, can transfer indirect branch and jump to secure return operations and stall aggressive memory accesses [12]. It is noted that software-based mitigations need code modification and recompilation. Furthermore, V1.1 and V1.2 can bypass the *LFENCE* defense [4], [35]. As a comparison, this paper focuses on the microarchitecture design innovations against the major variants of Spectre.

Hardware-based Mitigations: SafeSpec holds speculative refilled data for caches and TLBs in shadow structures [36]. When the instructions turn to be safe, the data will be moved to architectural caches. However the current SafeSpec does not consider cache coherence, it cannot defense the attacks employing store operations in multithreaded workloads, such as SpectrePrime. InvisiSpec proposes speculative buffer to store speculative refilled data, which employs the similar principle [37]. And it also provides delicate mechanisms to handle cache coherence and consistency. Different from their perspective of undoing the micro-architecture changes caused by mis-speculation, our work firstly propose the concept of *security dependence*, and our design philosophy is to speculatively execute safe instructions to maintain the performance benefits of out-of-order execution while blocking unsafe speculative accesses for security consideration. Therefore, SafeSpec and InvisiSpec are naturally orthogonal to our approach, and can be combined together for further performance improvement.

Cache Side-Channel Defenses: Existing cache side-channel defenses can be classified into two categories: noise interfere and resource partition. Typical noise interferes contain fuzzing time, random address mapping and obfuscating program critical etc [38], [39], [31]. Resource partition has two primary methods: cache partition and time partition. Since cache partition is usually based on last-level cache, Spectre attacks still have the possibility to steal secrets [40], [41], [42], [43]. Time partition eliminates the time difference of memory access but it can't prevent attacker speculatively load sensitive information [44], [45]. In essence, *Conditional Speculation* can cooperate with existing cache side-channel defenses.

IX. Conclusion

Speculative execution vulnerabilities have become a serious security threat to commodity computers because speculative execution is one of the fundamental optimization technology in high performance processors. *Conditional Speculation* is a software transparent and effective hardware based mechanism to mitigate existing Spectre attacks.

Similar to *Data Dependence* and *Control Dependence*, we first propose the concept of *Security Dependence*. This new dependence aims at recognizing the speculative instructions which have potential risk to leak micro-architecture information. *Security Hazard Detection* is introduced in the Issue Queue to identify unsafe speculation instructions based on security dependence. Once the security hazards in speculative execution are confirmed to be unsafe, they will be terminated and discarded via leveraging existing re-execution and speculation recovery mechanisms.

Simply blocking speculative memory accesses severely downgrades the performance. With the goal of pursuing a balance of performance and security, two filtering mechanisms are investigated to figure out false security hazard-

s. The proposed *Cache-hit Filter* aims at the speculative instructions which hit the cache. Since their speculative execution will not change cache (content), they are safe. And *TPBuf Filter* identifies safe speculative instructions from another perspective. For our targeted Spectre variants that use the shared memory based cache side channel and steal memory page information, their speculative execution of malicious gadgets have a common feature named as *S-Pattern*. *TPBuf Filter* is designed to capture *S-Pattern* from all speculative executions. For any speculative executed memory instruction, it is considered as safe if it does not match the *S-Pattern*. With safe instructions being allowed to execute speculatively as normal, *Conditional Speculation* remains the performance benefits of speculative execution.

We evaluate the mechanism of *Conditional Speculation* in terms of performance, security and area. The results show that the hardware overhead is marginal and the performance overhead is minimal.

ACKNOWLEDGMENT

This work was supported by National Science Foundation of China for Excellent Young Scholars under grant No. 61522212, and the Strategic Priority Research Program of Chinese Academy of Sciences under grant No. XDC02000000. We would like to thank Prof Michael C. Huang for his help in paper writing. We thanks Prof Peng Liu for polishing this paper. And we appreciate the technical discussions with Fengkai Yuan, Wei Song, Wenhao Wang and Xiaoxin Li. We also wish to thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[2] P. Z. Jann Horn, "Reading privileged memory with a side-channel," https://cryptome.org/2018/01/spectre-meltdown.pdf, 2018.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[4] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[5] C. Trippel, D. Lustig, and M. Martonosi, "Meltdown-Prime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols," *arXiv preprint arXiv:1802.03802*, 2018.

[6] M. Hill, "A Primer on the Meltdown & Spectre Hardware Security Design Flaws and their Important Implications," *Computer Architecture Today*, 2018.

[7] R. B. S. S. T. S. Mark D.Hill, Paul Kocher, "On the Implications of the Meltdown & Spectre Design Flaws," *ISCA 2018 Panel*, 2018.

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

[9] AMD, "Software techniques for managing speculation on amd processors," https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018.

[10] A. Technology, "Speculative store bypass disable," https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf, 2018-05-21.

[11] R. Grisenthwaite, "Cache speculation side-channels," https://armkeil.blob.core.windows.net/developer/Files/pdf/Cache_Speculation_Side-channels.pdf, 2018.

[12] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://www.reddit.com/r/cpp/comments/7o3oad/retpoline_a_software_construct_for_preventing/, 2018.

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

[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*. Springer, 2017, pp. 161–176.

[15] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution," *arXiv preprint arXiv:1802.09085*, 2018.

[16] G. Maisuradze and C. Rossow, "Speculose: Analyzing the Security Implications of Speculative Execution in CPUs," *arXiv preprint arXiv:1801.04084*, 2018.

[17] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–Bringing access-based cache attacks on AES to practice," in *2011 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2011, pp. 490–505.

[18] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *Usenix Conference on Security Symposium*, 2014, pp. 719–732.

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

[20] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." in *USENIX Security Symposium*, 2015, pp. 897–912.

[21] D. Gruss, C. Maurice, and K. Wagner, "Flush+Flush: A Stealthier Last-Level Cache Attack," *Computer Science*, 2015.

[22] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[23] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 51–67. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[24] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[25] swiat, "Fanalysis and mitigation of speculative store bypass," https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/, 2018.

[26] B. Sinharoy, J. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira *et al.*, "IBM POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2–1, 2015.

[27] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," in *Proceedings of the 34th annual IEEE/ACM international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 225–236.

[28] A. Henstrom, "Scheduling operations using a dependency matrix," Apr. 29 2003, uS Patent 6,557,095.

[29] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[30] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[31] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *2017 IEEE/ACM 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 347–360.

[32] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," 2018.

[33] Intel, "Mitigation overview for potential side channel cache exploits in linux," https://software.intel.com/sites/default/files/Intel_Mitigation_Overview_for_Potential_Side-Channel_Cache_Exploits_Linux_white_paper.pdf, 2018.

[34] I. G. T. M. A. R. Guanhua Wang, Sudipta Chattopadhyay, "oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis," *arXiv preprint arXiv:1807.05843*, 2018.

[35] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," *arXiv preprint arXiv:1807.07940*, 2018.

[36] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," *arXiv preprint arXiv:1806.05179*, 2018.

[37] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *Proceedings of the 51th International Symposium on Microarchitecture*, ser. MICRO'18, 2018.

[38] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 203–215.

[39] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital Side-Channels through Obfuscated Execution." in *USENIX Security Symposium*, 2015, pp. 431–446.

[40] F. Liu, Q. Ge, Y. IEEE/ACM, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[41] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 74.

[42] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice-a low-cost high-speed address translation mechanism," in *17th Annual International Symposium on Computer Architecture, 1990. Proceedings*. IEEE, 1990, pp. 355–363.

[43] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 89–102.

[44] M. Godfrey and M. Zulkernine, "A server-side solution to cache-based side-channel attacks in the cloud," in *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013, pp. 163–170.

[45] V. Varadarajan, T. Ristenpart, and M. M. Swift, "Scheduler-based Defenses against Cross-VM Side-channels," in *USENIX Security Symposium*, 2014, pp. 687–702.