SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels

Kristin Barber†, Anys Bacha§, Li Zhou†, Yinqian Zhang†, Radu Teodorescu†
†*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH*
{*barberk, zholi, yinqian, teodores*}*@cse.ohio-state.edu*
§*Department of Computer and Information Science, University of Michigan, Dearborn, MI*
bacha@umich.edu

***Abstract*—Hardware security has recently re-surfaced as a first-order concern to the confidentiality protections of computing systems. Meltdown and Spectre introduced a new class of microarchitectural exploits which leverage transient state as an attack vector, revealing fundamental security vulnerabilities of speculative execution in high-performance processors. These attacks profit from the fact that, during speculative execution, programs may execute instructions outside their legal control flows. This is used to gain access to restricted data, which is then exfiltrated through a covert channel.**

This paper proposes SpecShield, a family of microarchitectural mitigation techniques for shielding speculative data from covert channels used in transient execution attacks. Unlike prior work that has focused on closing individual covert channels used to leak sensitive information, SpecShield prevents the use of speculative data by downstream instructions until doing so is determined to be safe, thus isolating it from any covert channel. The most secure version of SpecShield eliminates transient execution attacks at a cost of 21% average performance degradation. A more aggressive version of SpecShield, which prevents the propagation of speculative data to known or probable covert channels provides only slightly relaxed security guarantees with an average of 10% performance impact.

***Keywords*-hardware security, transient execution attacks, speculative execution, covert channels**

I. Introduction

Speculative execution has been used for decades in out-of-order processors as a mechanism for hiding the latency of branch resolution and other long-latency instructions, helping to expose instruction level parallelism and increase performance. Modern processors frequently execute so-called transient instructions that are not specified by their legal control flows, due to branch misprediction, out-of-order execution, delayed exception handling, etc. When a misprediction is detected the processor discards any erroneous instructions and continues execution from a logical checkpoint. Unfortunately, the Meltdown [1] and Spectre [2] attacks have exposed fundamental vulnerabilities in how modern processors implement transient execution.

These attacks leverage transient execution to access otherwise restricted data. The first stage of these attacks uses transient instructions executed outside the legal control flow to retrieve secret data that is otherwise inaccessible to the application. This can include data belonging to a privileged process or data normally protected by the application's control flow, such as array boundary checks. Although the direct results of transient execution on the architectural (externally visible) state are rolled-back, side effects of that execution remain in the micro-architectural state of the processor and can be leaked. This insight represents the foundation for transient execution attacks.

In the second stage of these attacks, a covert channel is used to leak the secret retrieved in the first stage. Covert timing channels consist of a trojan process (transmitter) which modulates the timing of a shared system resource to reveal sensitive information to a spy process (receiver). The trojan and spy do not communicate explicitly, but covertly by observing the timing of certain events with respect to the shared resource [3]. Covert channels provide a mechanism to transfer the results of transient execution into the architectural state of the processor where they can be recovered by an attacker. Any micro-architectural structure which can be manipulated by an attacker and later observed in a repeatable and consistent fashion can be leveraged as a covert channel. Although cache-based covert channels [4], [5] were used in the Spectre and Meltdown attacks and arguably provide the highest accuracy and bandwidth, other channels are possible; including SIMD units [6] and some execution ports [7].

This paper proposes SpecShield, a mechanism for shielding speculative data from any potential covert channel. Unlike prior work [8]–[11] that has focused on closing specific covert channels, SpecShield is more general and addresses the root cause of transient execution attacks, which is the use of speculative data by dependent instructions that can leak it.

We propose three SpecShield designs with different security and performance trade-offs. Depending on the security assessment of a given micro-architecture, SpecShield can be tuned to guard against only those functional units and micro-architectural structures determined to be vulnerable to leakage. This is accomplished by controlling data-flow within the pipeline, inhibiting access to sensitive data by instructions

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00020

which utilize vulnerable micro-architectural structures until execution is outside the window of speculation. SpecShield presents a solution with more flexibility and less complexity compared to existing hardware solutions.

The most conservative version, SpecShieldSTL delays the forwarding of speculative data read from memory by a Load – and the wakeup of dependent instructions – until the Load instruction commits and is therefore guaranteed to no longer be speculative; guaranteeing speculative data can no longer be leaked. However, the performance overhead of this approach is high.

To mitigate the performance impact we develop SpecShieldERP, a more efficient design that allows the early resolution of some speculative Load instructions. An in-flight Load instruction is considered *early resolved* if all older branch instructions are resolved and not misspeculated, and all older load/store instructions have their physical addresses returned by the TLB with no exceptions or faults. The results of these Loads can therefore be safely forwarded to dependent instructions.

The third proposed design, SpecShieldERP+ relaxes some of the security guarantees in order to further reduce the performance impact. This design allows the selective forwarding of speculative data to instructions that are deemed to have low leakage risk. The propagation of speculative data stops when it reaches an instruction classified as having a high risk of being used in a covert channel. We propose a mechanism for tagging the high covert channel risk instructions at decode. We also propose allowing the list of high risk instructions to be updated in production systems through firmware patches to prevent leakage through new covert channels that might be discovered after the hardware is shipped.

SpecShieldERP and SpecShieldERP+ defeat transient execution attacks at an average performance cost of 21% and 10%, respectively.

Overall, this paper makes the following main contributions:

- Proposes a microarchitectural framework for eliminating transient execution attacks which grant access to arbitrary memory.
- Unlike prior work that has focused on closing specific covert channels, SpecShield is the first general solution addressing speculative data-flow within the pipeline.
- Implements a mechanism to flexibly tune the scope of protections, allowing for performance savings depending on the security needs of a given microarchitecture.
- Allows in-field tuning of the level of isolation for speculative data to respond to attacks discovered in the future.

The rest of this paper is organized as follows: Section II provides background on transient execution attacks. Section III discusses our threat model. Section IV presents the SpecShield design. Sections V and VI present SpecShield's evaluation. Section VII analyzes implications on security. Section VIII discusses related work and Section IX concludes.

II. Background

A. Transient Execution Attacks

Processors attempt to guess the outcome of long-latency operations as an alternative to stalling dependent instructions until the correct result is known. This is referred to as speculative (or transient) execution, and although it is often conceptually coupled with branch prediction, speculation has broader applications. Processors execute transient instructions following a multitude of instructions that may change control flow. These include branches, memory instructions that can cause address aliasing or page faults, and any other instructions that can cause exceptions. Speculative execution sometimes leads to normally unreachable code paths to be executed and restricted data to be accessed. This has conventionally been considered safe, since misspeculated state is cleaned up by the processor without any changes to the architectural (programmer-visible) state. However, transient execution attacks have found creative ways of exploiting side-effects of speculatively executed instructions to leak secret data.

Since Spectre and Meltdown were disclosed, a number of transient execution attacks have been discovered that differ in the speculative behaviors leveraged and covert timing channels used to exfiltrate the data of interest. A comprehensive survey of known transient execution attacks and defenses can be found in [12]. We briefly present Spectre-v1 as an illustrating example. Listing 1 shows the disclosure gadget from that attack. An attacker first trains the branch predictor to ensure a misspredict when $x > lenb$ by executing this branch several times with values of x which are valid. Next, the attacker selects a malicious value for x which will read outside the bounds of the array and into an unauthorized area of memory, presumably where secret data resides. Since the branch predictor has been trained to guess that x will have an in-bounds value, execution continues down the misspredicted path and a restricted value is accessed by the memory reference `b[x]`. Although secret data has been accessed, it will be cleared out of the architectural state of the system when the misspredict is identified. Before that happens, however, the secret can be leaked through a cache timing channel.

This is accomplished by using a traditional cache timing attack technique such as Flush+Reload [4] or Prime+Probe [5]. First, the cached contents of array `a` are set to a known state, where `a` is a shared resource acting as the covert channel. The secret accessed by the memory reference `b[x]` is then used to access a location in array `a` that is dependent on the restricted value, leaving a secret-dependent footprint in the cache. The attacker probes each cache line containing `a` and infers the secret value from the index exhibiting an

anomaly in access time relative to the other indices. In Flush+Reload, the secret data would correspond to the array index with the lowest access latency.

```
1    if (x < lenb)
2        y = a[b[x] * 512];
```

Listing 1: C++ Spectre-v1 disclosure gadget.

```
1       ld    r1, lenb
2       ld    r2, x
3       bge   r2, r1, done
4       ld    r3, r2(b)
5       slli  r3, r3, 9
6       ld    r4, r3(a)
7       sd    r4, y
8   done: ...
```

Listing 2: RISC assembly of Spectre-v1 disclosure gadget.

III. Threat Model

SpecShield offers protections against attack scenarios where the *access* to sensitive data and covert *leaking* of this data are both executed speculatively. This includes attacks where the opening to the speculative window is the result of:

- **Exceptions**, including Meltdown (Rogue Data Cache Load) and Foreshadow [13]
- **Branch misprediction**, including Spectre-v1 (Bounds-Check-Bypass)
- **Memory aliasing misprediction**, such as Spectre-v4 (Speculative Store Bypass) [14]
- **Control-flow hijacking**, such as Spectre-v2 (Branch Target Injection) and ret2spec [15] in cases where the data access and covert communication are both contained within the *gadget* where control-flow has been re-steered.

Figure 1 illustrates the speculative data access path (shown in red) on a diagram of an out-of-order core. We assume secret data can reside in any level of the memory hierarchy. Once speculatively loaded into the processor pipeline, a number of covert channels can be used to leak the secret data. Examples of possible covert channels are highlighted in blue.

Most prior attacks have used the cache as a covert channel and previously proposed hardware solutions [8]–[11], [16] are designed to close that channel. However, other covert channels are possible. For instance, NetSpectre [6] uses the access latency to the SIMD (AVX) unit to leak secret data, while SMoTherSpectre [7] uses contention for execution ports as a covert channel. In fact, any measurable property of a processor implementation has the potential to leak information [17]. Table I summarizes most known transient execution attacks and the type of covert channel used to exfiltrate information. While closing individual channels is generally useful to improving security, it is not the best approach for addressing transient execution attacks. Developing protection mechanisms for a particular covert channel leaves the processor vulnerable to the discovery of other practical covert channels in the future.

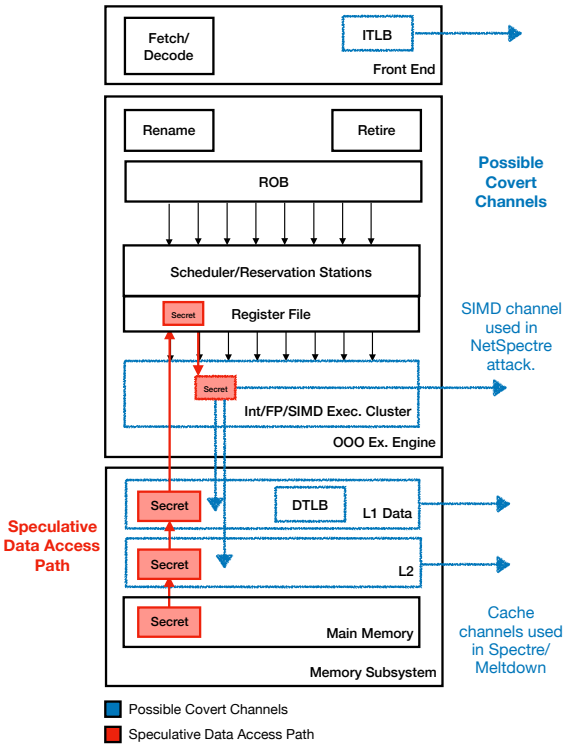

Figure 1: Overview of the threat model for speculation-based attacks in modern out-of-order processors.

Transient Execution Attack	Covert Channel Used
Spectre-v1 [2]	Cache
Spectre-v2 [2]	Cache
Meltdown (v3) [1]	Cache
Spectre-v3a [18]	Cache
Spectre-v4 (SSBD) [14]	Cache
Meltdown/SpectrePrime [19]	Coherence Messages
ret2spec [15]	Cache
LazyFP Restore [20]	Cache
Foreshadow [13]	Cache
NetSpectre [6]	AVX
SMoTherSpectre [7]	Execution Ports
MDS attacks [21] [22] [23]	Cache

Table I: Transient execution attacks and the covert channel used for data exfiltration.

A. Out of Scope for SpecShield

SpecShield protects memory-resident secrets by isolating them from speculative computations after being loaded into the pipeline. SpecShield does not consider register-resident

secrets: secret data retrieval which did not occur under misspeculation but rather represents a legal access now part of the committed state of the processor.

As such, out of scope for this work are attacks such as Spectre-v3a (Rogue System Register Read) [18] and LazyFP [20], where the data of interest resides in a restricted register. Unauthorized access to the register contents is achievable during speculative execution due to lazily enforced exceptions, similar conceptually to the Meltdown attack. These attacks have been addressed through microcode updates and enhanced clean-up routines for inter-process data during a context switch by the operating system, respectively.

Additionally, Spectre-v2 attacks with the following properties are not addressed by SpecShield: a) load of secret data into a register through legal control-flow, b) pre-loaded secret closely followed by an indirect branch vulnerable to poisoning, c) control-flow hijacking occurs through this indirect branch to a chosen gadget and d) gadget leaks register contents through some covert communication channel. Protecting against this type of attack can be done currently using other Spectre-v2 specific mitigations such as *retpolines* developed by Google [24], which address the root cause of Spectre-v2 attacks by eliminating an attacker's ability to poison the branch-target-buffer and re-steer execution in the first place.

SpecShield focuses on attack scenarios granting arbitrary access to memory. Protecting register-resident secrets would require not only the policing of speculative data, but also policing the committed state of the register file during speculative execution; this requires a different approach and we leave addressing this narrower threat to future work. Note that, by design, SpecShield allows for speculative data to be loaded into the cache. However, the presence of secret data within the memory hierarchy alone does not represent a security risk. A requirement of transient execution attacks is the ability to transfer information from the speculative domain into the architectural state of the processor. We eliminate the ability for an attacker to satisfy this requirement. Therefore, secret data residing in the cache is useless to an attacker unless there is a way in which it can be covertly leaked and decoded after the speculative window has closed.

IV. SpecShield Design

Key to our approach is the observation that, by definition, the leakage source (covert channel) has a data-dependence on the secret value, as shown in the example in Listing 1. The access into array `a` is dependent on the secret value referenced by `b[x]`. These two memory references form the transmitter-side of the covert channel and must both be executed speculatively, before the misspeculated instructions are squashed. Therefore, if we can delay the forwarding of data to dependent instructions until we can confirm the producing instruction is no longer speculative, we can inhibit

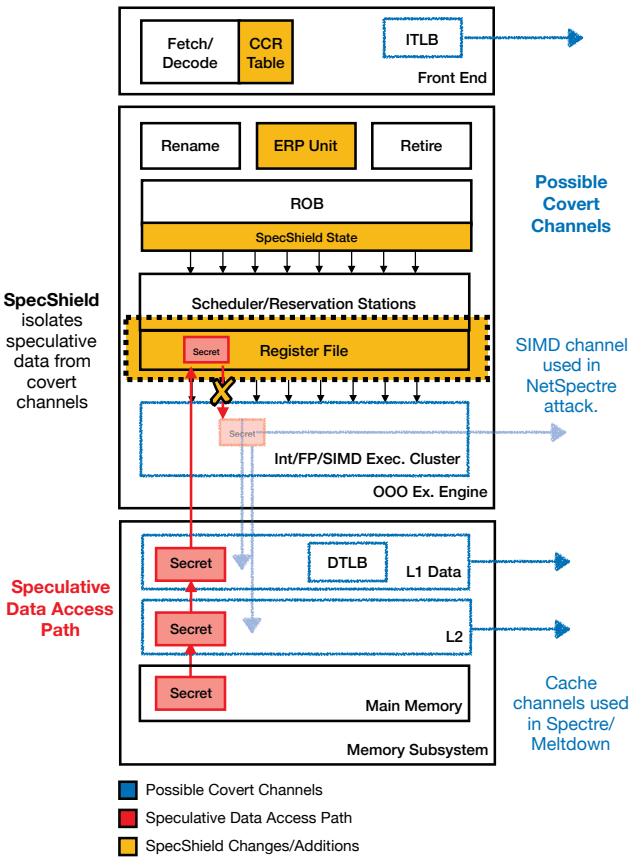

Figure 2: SpecShield isolates speculative data from covert channels used to leak secret information.

leakage and restrict the formation of the transmitter. This removes the ability to construct any covert channel that can leak this data.

To accomplish this goal we change the timing of when speculative data loaded into the processor's pipeline can be used by dependent instructions. Figure 2 shows a high-level view of how SpecShield integrates into a processor's pipeline design. SpecShield encompasses three possible designs, each with different security and performance trade-offs.

A. SpecShieldSTL: Speculative Data Stall

The most conservative SpecShield design, which we call SpecShieldSTL, delays the forwarding of data returned by a speculative load instruction until the instruction reaches the head of the reorder buffer (ROB) and is ready to commit. Figure 3 illustrates the state of the ROB for a code example that includes a `LD` instruction followed by two dependent instructions (`ADD` and `SUB`). Figure 3(a) illustrates the cycle in which the `LD` receives the requested data from memory. Normally the value of `mem(D)` would be forwarded to all dependent instructions as well as stored in physical register `r1`.

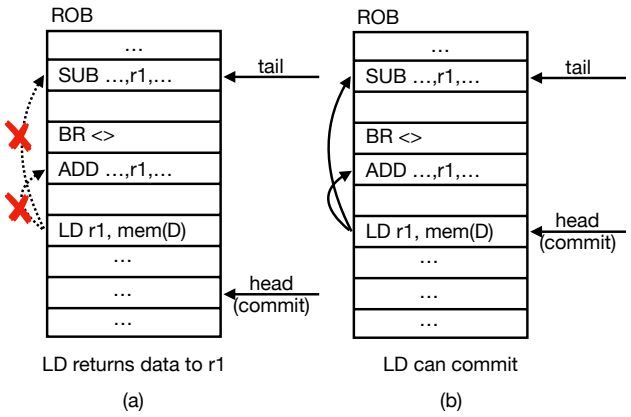

Figure 3: ROB state for SpecShieldSTL at different time snapshots. The forwarding of data loaded from memory location `mem(D)` to dependents is delayed (a) until the `LD` instruction is ready to commit (b).

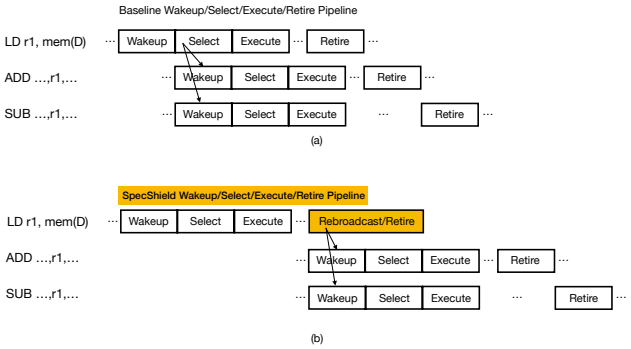

Figure 4: Baseline design (a) and SpecShieldSTL changes (b) to the timing of the Wakeup/Select pipeline to delay the broadcast of speculative `LD` results.

In SpecShieldSTL, when the `LD` data returns, the state of its ROB entry is checked. If it is not at the head of the ROB, the result of the `LD` will silently update the physical register corresponding to `r1`, but it will not broadcast the data on the result bus to dependent instructions. Since the physical register is assigned to `r1`, it will not be recycled until the `LD` instruction retires. Also, the `LD` will not be marked as complete, forcing any `r1`-dependent instructions added to the ROB after the `LD` returns to wait.

When the `LD` is ready to commit, its data is read from the physical register and broadcast to dependent instructions (Figure 3 (b)). At that point the `LD` is guaranteed to no longer be speculative. This ensures that no speculative data will be manipulated by other instructions that could leak information.

The Wakeup/Select logic in SpecShieldSTL has to be modified to consider all `LD` instructions as high (and variable) latency instructions. As a result, no `LD`-dependent instructions will be woken up when the `LD` is dispatched. Figure 4 shows the timing of the typical wakeup/select pipeline and the changes made to delay the wakeup of dependent instructions until the result of the `LD` is rebroadcast before retirement.

SpecShieldSTL is straightforward to implement in hardware requiring minimal changes to existing designs. However, delaying all instructions dependent on speculative loads by dozens or potentially hundreds of cycles leads to a significant performance impact, as will be shown in Section VI.

B. SpecShieldERP: Early Resolution

To address the performance impact of SpecShieldSTL, we develop a mechanism for early detection of non-misspeculated loads that allow their results to be forwarded earlier to dependent instructions. We call the performance optimized design SpecShieldERP. We define an Early Resolution Point (ERP) in the ROB as the most recent in-flight instruction in program order for which the following conditions are satisfied:

1) All older branch instructions (in program order) have been resolved and their actual direction is known.
2) All older load and store instructions have had their address computed, and a TLB translation has been performed.
3) No branch misspredict or memory access exception has been raised by either of the above instruction types.

By definition, all instructions between the head of the ROB and the ERP can be considered safe or "resolved" with respect to their ability to speculatively access data outside their legal control flow. Similar definitions have been proposed in previous work, with varying goals related to eager resource re-allocation in OoO processors [25]–[27].

Figure 5 shows two snapshots of the ROB content and the position of the ERP. In Figure 5(a), the ERP is below the `BR<c1>` instruction. This means that all Branch and Load instructions between the ERP and the ROB head have been resolved and are neither misspeculated nor have they raised an exception. The `BR<c1>` instruction, on the other hand, has not been resolved, thus preventing the ERP from moving upwards.

SpecShieldERP allows the immediate forwarding of results for all the Load instructions that are older than the ERP and can therefore be considered safe. The result of the `LD r1, mem(A)` instruction can immediately be forwarded to its dependent instructions, which in this example is the `ADD` instruction. This allows Load results to be forwarded much earlier than in SpecShieldSTL, which restricts all Loads to wait until commit. All other Loads that are above the ERP (e.g `LD r2, mem(B)` in Figure 5) will continue to delay the forwarding of their results until they reach the ERP.

The ERP is updated every cycle by a dedicated ERP Unit (Figure 2) by checking the state of all Branch and

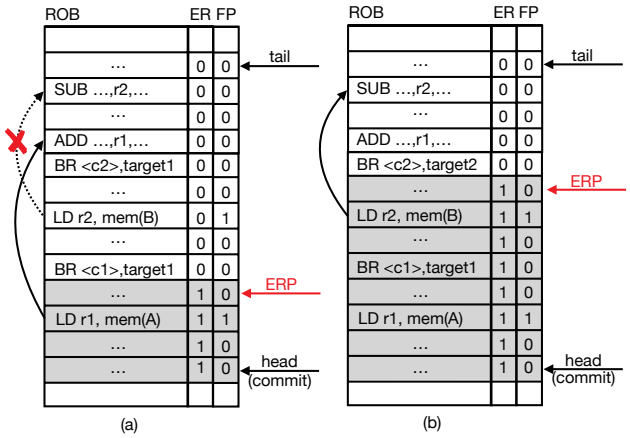

Figure 5: (a) SpecShieldERP immediately forwards the result of the `LD r1` instruction, which is behind the ERP, and delays forwarding the result of `LD r2, mem(B)`. (b) When the `LD r2` instruction is "early resolved" its result stored in `r2` is forwarded to dependents.

Load/Store instructions between the ERP and the ROB tail. The ERP is moved to the instruction below the oldest unresolved Branch or Load/Store instruction in the ROB. Figure 5(b) shows that after the `BR<c1>` instruction has been resolved and correctly predicted and the `LD r2, mem(B)` instruction has completed the address translation phase with no exceptions, the ERP moves up. The ERP will be updated to point to the instruction below `BR<c2>`, which, in this example, is the oldest unresolved Branch in the ROB. When the `LD r2, mem(B)` returns with the `mem(B)` data, it can be forwarded freely to all dependent instructions (in this example the `SUB` instruction).

A new ER status bit associated with each ROB entry indicates whether the instruction has been "early resolved" or not. When a Load returns the ER bit is checked. If it is set the result is immediately broadcast on the result bus to all dependents. If it is not set, the destination register of the `LD` is silently updated. A "forward pending" (FP) bit will be set in that Load's ROB entry. When the Load is early resolved the FP bit is checked. If it is set, the result of the Load will be rebroadcast to dependent instructions.

Even though SpecShieldERP is significantly less conservative compared to SpecShieldSTL it does not significantly relax the security properties of SpecShieldSTL. By allowing only data loaded by instructions within the Early Resolution window to be forwarded to dependents, and serializing address computation and permissions check with data access, SpecShieldERP prevents any speculative data from being accessed by any other instruction in the pipeline.

1) Other Exceptions and Interrupts: While it is possible for the instructions now considered safe (below the ERP) to experience other types of exceptions or interrupts, prior work has shown these to be ineffective in facilitating transient execution attacks [12]. Per our threat model, it is sufficient to consider only branch status and memory-related exceptions in defining the ERP. This is because no other exception provides transient access to memory/cache data outside the legal control flow of the application. Canella et. al [12] report finding no traces of transient execution past traps and aborts. Divide-by-zero raises an exception but the result register is set to '0' and no real values are leaked. Results of unaligned memory accesses and segmentation faults never reach transient execution. Likewise, transient execution was not successful following an invalid op-code, because exceptions raised early in the pipeline are handled immediately before an entry is even created in the ROB for the faulting instruction.

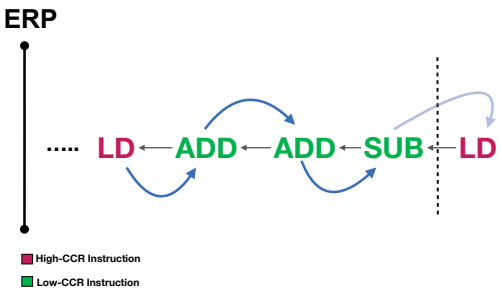

Figure 6: Illustration showing how the flow of data between instructions in a dependency chain is policed under SpecShieldERP+.

C. SpecShieldERP+: Aggressive Early Resolution

The last design we explore, called SpecShieldERP+, relaxes some of the security guarantees of SpecShieldERP by allowing speculative data to be accessed by select instructions that cannot be used as covert channels. The goal is to reduce performance impact by limiting the number of delayed forwards of speculative data. SpecShieldERP+ is designed to be used in systems for which we can prove certain covert channels are not viable or systems that use other methods to close select covert channels (e.g. the cache).

Unlike SpecShieldERP, SpecShieldERP+ allows results from instructions that have not yet passed the ERP to selectively forward their results to consumers that we deem to have low Covert Channel Risk (CCR). The prototype used in our evaluation has been configured to have a high CCR list which includes Loads, Stores and Branches–effectively restricting the forwarding of data to these instruction classes until the producer of that data is behind the ERP. This would mitigate any attack that uses the cache as a covert channel or requires control of branches (virtually all known attacks). Depending on the architecture, type of instructions supported, etc., a different CCR list could be constructed.

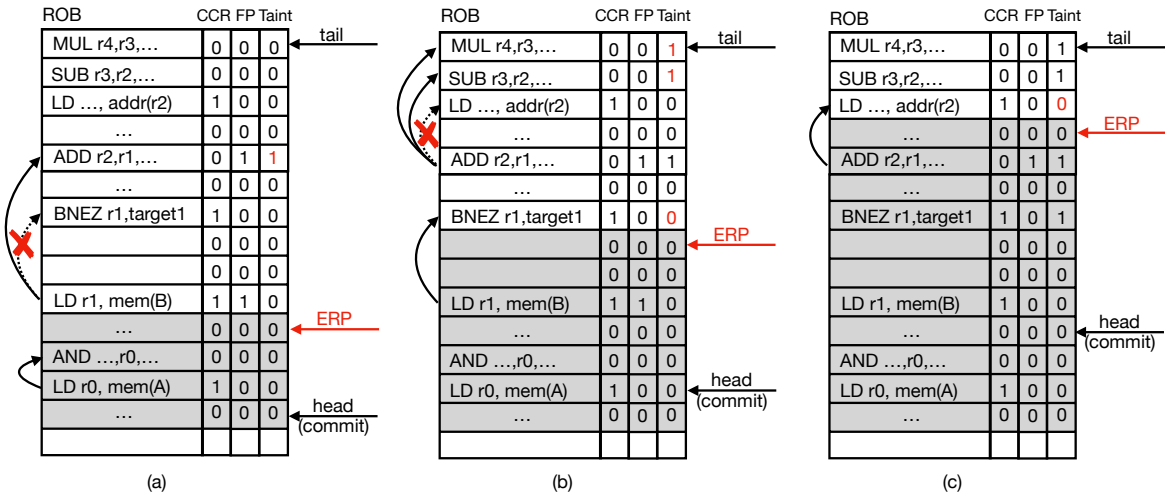

Figure 7: The ROB state at different time snapshots for a SpecShieldERP+ example.

Instructions are tagged with a CCR bit starting at the Decode stage, using a firmware-programmable CCR Table (Figure 1). The content of the table can be updated after the processor is deployed if new vulnerabilities are discovered.

Figure 6 illustrates how data flow among instructions in a dependency chain policed with SpecShieldERP+. In this illustration, gray arrows represent dependencies, blue arrows represent data forwarding and the dashed-line represents where the flow of data must stop, assuming that all instructions in this chain have *not* currently reached the ERP. In SpecShieldERP+, because we allow data forwarding to any non-CCR dependents, the initial `LD` in this chain can immediately forward to the `ADD` instruction. Forwarding continues along the chain, until we encounter the first dependent high-CCR instruction (second `LD`), at which point result forwarding is halted at the immediate predecessor of the CCR instruction (the `SUB`). Only when the `SUB` reaches the ERP will it forward its result to the dependent `LD` instruction.

Keeping track of instruction sources requires constructing the forward execution slice for each Load instruction. One option to implement this is to adopt a hardware-based program slice tracking mechanism similar to that proposed by Carlson et al. [28] for backward slices. The hardware overhead for this solution however is nontrivial. We instead choose a simpler approach for SpecShieldERP+, starting from the observation that we do not need to keep track of the source of the speculative data precisely. It is instead sufficient to know if an instruction has received speculative data or not. If it has, it is prohibited from forwarding its results to high CCR instructions until it is early resolved or committed. We use a simple form of taint propagation [29]–[31] to keep track of which instructions handle speculative data. Data produced by a speculative load is considered tainted. An instruction that has at least one tainted operand also becomes tainted and produces a tainted output. In order to keep track of that state we add a "Taint" bit to each entry in the ROB.

Figure 7 helps illustrate with an example how SpecShield-ERP+ works. Figure 7(a) shows an instruction sequence in the ROB in which the ERP is below a Load instruction. The oldest Load in the ROB (`LD r0, mem(A)`) is already marked as "early resolved." When its data returns it can immediately be broadcast and its dependent instruction (the `AND`) woken up. The `LD r1, mem(B)` instruction, on the other hand, is not resolved yet. In SpecShieldERP the result of the load would only be forwarded to dependents when early resolved. In SpecShieldERP+ however, we allow the immediate forwarding of a load's data to any low-CCR dependent instructions. When the data for the `LD r1, mem(B)` returns, the result is broadcast to all dependent instructions. Since the `LD` is speculative and has not yet reached the ERP, a Taint signal is broadcast along with the data and the source address of the `LD`. Since the data is "tainted", only dependent instructions with their CCR bit unset will update their source operand with the result of the `LD`. Any dependent instruction with a CCR bit set, indicating a high covert channel risk, will ignore the tainted result for now. In the example in Figure 7(a), the result of the `LD` is forwarded to the `ADD` instruction but not the `BNEZ`. The `ADD` instruction is also marked as tainted by setting its Taint bit, to indicate it has received speculative data. The "forward pending" (FP) bit of the `LD` is set to indicate that the result of the `LD` might still need to be forwarded to some dependent instructions when it is safe to do so. The result of the `LD` is stored in a register, but only forwarded to low CCR dependents.

When the `ADD` instruction finishes execution (Figure 7(b)), its Taint bit is checked. If it is set, the result is broadcast along with a Taint signal to its low CCR dependents

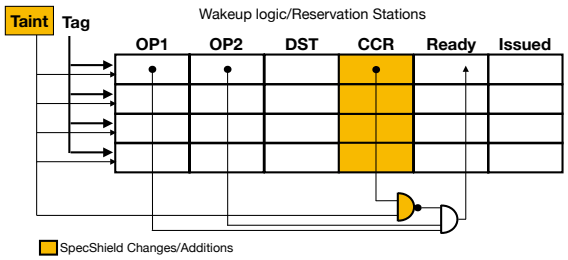

Figure 8: The Wakeup/Reservation Station logic with SpecShield support.

(the `MUL` and `SUB`), and ignored by the high CCR ones (the `LD`). The FP bit for the `ADD`, as well as the Taint bits of the `ADD` low-CCR dependents (the `MUL` and `SUB`) are all set. When the ERP is updated and the `LD r1, mem(B)` is early resolved, its FP bit is checked. Since it had been marked with "forward pending", its result now needs to be broadcast to any high CCR dependents – in this case the `BNEZ`. This is accomplished by re-broadcasting the tag of the `LD`, this time with the Taint signal unset. Since the taint signal is unset all remaining dependent (high CCR) instructions are woken up and allowed to access the result of the `LD` from the register file. Note that, since the `LD` is early resolved and its Taint bit is unset the `BNEZ` is not marked as tainted.

Figure 7(c) shows that when the `ADD` is early resolved its FP bit is checked and, since it is set, the result is forwarded to the dependent `LD addr(r2)` instruction.

SpecShieldERP+ requires simple modifications to the wakeup/select logic, as shown in Figure 8. In particular, each reservation station entry has an extra CCR field that mirrors the one in the ROB and is set when the instruction is renamed. Waking up dependent instructions involves broadcasting the destination tags of the currently selected instructions. Our design also broadcasts Taint signals to indicate the taint state of the selected instructions. The wakeup logic for each reservation station entry, in addition to checking if the tag is a match for any of the source operands, also checks the Taint signal as well as the CCR bit of the instruction. If Taint="1" and CCR="1" the instruction is not woken up, even if both input operands are available.

D. Multicore Impact

SpecShield is implemented within the processor core and its effects are not visible to the cache hierarchy. Unlike prior work [8], [9], SpecShield does not require modifications that impact the cache hierarchy or the cache coherence protocol. This makes SpecShield much easier to implement in hardware.

V. Methodology

We used the gem5 [32] cycle-level simulator in full-system mode, running a Linux operating system and modified out-of-order CPU model to implement the three SpecShield designs summarized in Table III. Our simulations were run with an Ubuntu 14.04 disk image, Linux v4.18.7 kernel and x86 ISA. Table II shows the CPU and cache parameters used. SPEC2006 workloads [33] were used in the evaluation with the *reference* input set. Results correspond to simulations of the 10 most representative regions of 500 million instructions with an additional 100K instruction warm-up phase, chosen using the SimPoint methodology [34].

CPU Architecture			
CPU Clock	2GHz	LQ, SQ Entries	32
L1 ICache	32KB (4-way)	IQ Entries	64
L1 DCache	32KB (8-way)	BTB Entries	4096
L2 Cache	2MB (16-way)	dTLB Entries	64
Issue Width	8	iTLB Entries	64
ROB Entries	192	FP Registers	256
Branch Predictor	LTAGE	Int Registers	256

Table II: Architectural configuration parameters.

Design	**Description**
Baseline	Conventional OoO forwarding
FENCE	Serialization after every branch
SpecShieldSTL	No load forwards until it is ready to commit
SpecShieldERP	No load forwards until it reaches ERP
SpecShieldERP+	No instruction forwards to high CCR consumers until it reaches ERP

Table III: Designs evaluated.

VI. Evaluation

In this section we examine the performance impact of SpecShield and analyze the implications of some of the design trade-offs made in the three variants.

Figure 9 shows the relative performance of each SpecShield design, summarized in Table III with respect to a baseline using conventional OoO execution. We also include the performance overhead of an implementation which serializes execution after every branch. This approach follows the conservative recommendation by hardware manufacturers [35] to protect against transient attacks with the use of explicit serializing instructions in the application to constrain speculation.

A. Serialization

Serialization is a coarse-grained way to explicitly restrict and constrain the amount of speculation a processor is able to perform. When a serializing instruction is dispatched from the issue queue into the pipeline, no other instructions may begin execution until the serializing instruction has committed (reached the ROB head). Forcing the pipeline to serialize on every branch can result in low resource utilization and significantly degrade performance. This is evident from Figure 9, where the slowdown for this approach (labeled

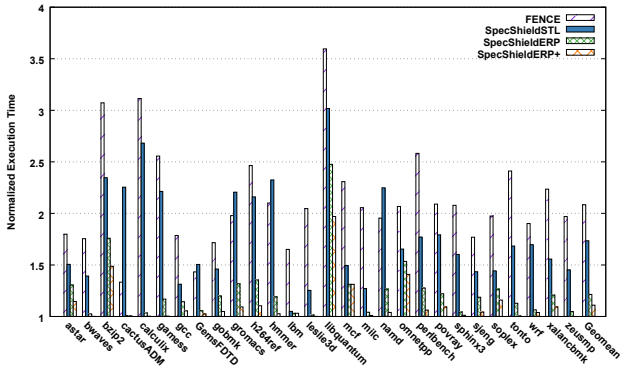

Figure 9: Performance impact of SpecShieldSTL, SpecShieldERP and SpecShieldERP+ relative to an unsecured baseline.

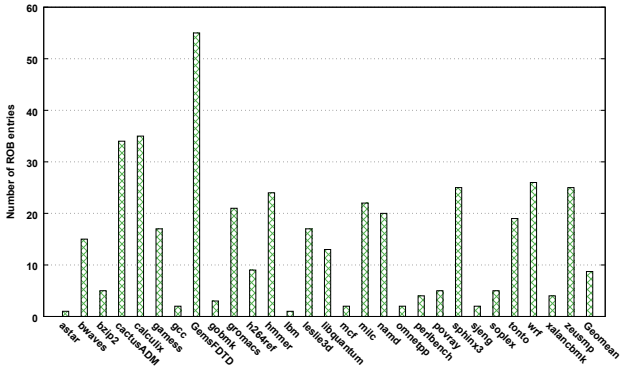

Figure 10: Average number of ROB entries from ERP to commit.

FENCE) is shown to be 2.08× on average. Serialization has a higher performance cost than all SpecShield variants, including the very conservative SpecShieldSTL. FENCE does outperform SpecShieldSTL in a few cases: namely *cactusADM*, *GemsFDTD*, *gromacs*, *hmmer* and *namd*. This can be explained by an application's ratio of load instructions (delayed by SpecShieldSTL) to branch instructions (serialized by FENCE). For applications that have many more loads than branches, serialization may be a better choice than SpecShieldSTL.

B. SpecShieldSTL

SpecShieldSTL has an average performance overhead of 73% which, although quite large, is almost 4× lower than FENCE serialization. SpecShieldSTL is very conservative in forwarding data from loads to dependent instructions. Each load instruction must wait until it reaches commit before it is permitted to forward its data, which could leave dependent instructions stalled for dozens or even hundreds of cycles.

The performance hit is worse for benchmarks that have relatively low miss rates such as *calculix*, *hmmer* or *gromacs*, which have less than 10 misses per 1K instructions. Since for these applications most loads are hits, delaying their resolution to commit has the highest performance impact. The slowdown for the three applications is 2.68×, 2.32× and 2.2×, respectively.

Applications with high miss rates (especially LLC misses) such as *lbm* or *milc* exhibit a lower performance impact because load misses already stall the pipeline considerably. In many cases, a missing load will reach the head of the ROB before its data returns, therefore suffering no additional stall in SpecShieldSTL. In addition to being memory bound, *lbm* also frequently stalls the pipeline due to contention in the load/store queue. The frequent pipeline stalls hide the delays caused by SpecShieldSTL.

C. SpecShieldERP

The SpecShieldERP design eliminates most of the overhead of SpecShieldSTL by loosening the traditional definition governing when an instruction is no longer speculative for the purpose of our threat model. Recall that the early resolution point in the ROB is defined such that all older branches are resolved and all older memory instructions have had their physical address retrieved from the TLB with no faults or exceptions.

1) Impact of Early Resolution: In order to evaluate the opportunity created by the "early resolution" of instructions, we measure the distance in number of instructions from the ERP to the head of the ROB (commit point). This data is shown in Figure 10 as a per benchmark average.

This metric allows us to quantify the opportunity for earlier forwarding by measuring the window of instructions which can take advantage of SpecShieldERP and SpecShieldERP+. In other words, it is a measure of how much earlier load results can be used, compared to SpecShieldSTL. The *calculix* and *GemsFDTD* benchmarks illustrate well how "early resolution" can translate into performance improvement. *calculix* and *GemsFDTD* have respective slowdowns of 2.6× and 1.50× under SpecShieldSTL. These benchmarks also have relatively large ERP-to-commit distances of 55 and 35 instructions, respectively. This can be interpreted to mean that on average, 55 (35) of the instructions in the ROB have reached the ERP and would be allowed to forward their results with SpecShieldERP(+)–resulting in a potentially significant performance improvement. This is corroborated by the results in Figure 9, where we see that the runtimes for *calculix* and *GemsFTDT* with SpecShieldERP are now only 5% and 3% longer than the baseline, respectively.

On the other hand, some benchmarks, such as *astar*, *gcc* and *mcf*, have short ERP-to-commit distances of only 1-2 instructions. As a result, they benefit less from SpecShieldERP compared to other benchmarks, but still exhibit a reduction

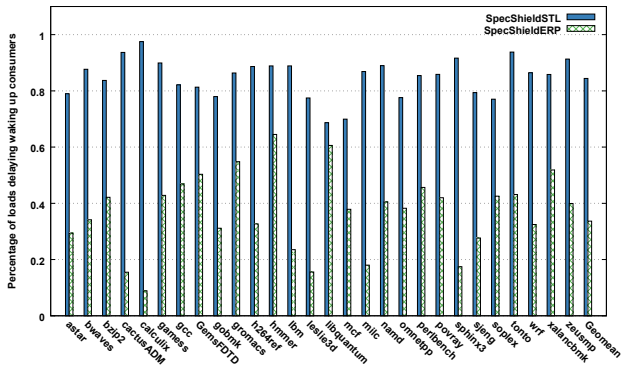

Figure 11: Percentage of load instructions that have not reached commit (early resolution) when requested data returns and will therefore delay wakeup of dependent instructions in SpecShieldSTL and SpecShieldERP.

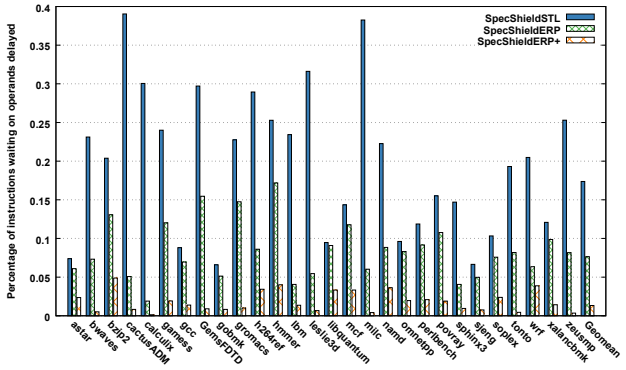

Figure 12: Percentage of instructions impacted by delayed wakeup due to dependencies on speculative data.

in overhead of 20%, 17% and 18%, respectively.

2) Performance Improvement: Overall, the performance benefits of SpecShieldERP are substantial, reducing the performance penalty to an average of 21% across all applications we examine, compared to 73% for SpecShieldSTL.

3) Impact of Delayed Wakeup: Figure 11 shows how many loads delayed waking up their consumers after having their data request satisfied by the memory system but had not yet reached a point at which they can safely broadcast their results – this is commit for SpecShieldSTL and the early resolution point in SpecShieldERP. For SpecShieldSTL, we can see that many loads must delay forwarding data to dependents – as much as 97% for *calculix* and 84% on average. The percentage of loads for SpecShieldERP leading to stalls of dependents ranges from 8% for *calculix* to 64% for *hmmer*, with an average of 33%. The reduction due to SpecShieldERP is substantial, and helps explain the performance improvement. The percentage of delayed loads is still high however, a good motivation for not blocking the wake-up of all dependent instructions, as it is done in SpecShieldERP+.

D. SpecShieldERP+

Figure 9 also shows SpecShieldERP+, which is our most optimized design and seeks to improve upon SpecShieldERP by eliminating unnecessary delayed wake-ups of consumer instructions that can be classified as having a low covert channel risk (CCR).

1) Performance Improvement: The performance improvement of SpecShieldERP+ is significant, with now only a 10% average penalty over the baseline, as Figure 9 shows. SpecShieldERP+ outperforms SpecShieldERP across all applications. This is because the only instructions that experience an additional delay in receiving data are the high-CCR instructions (loads and branches). This limits the number of stalled instructions reducing performance impact.

2) Benefits of Selective Early Wakeup: Figure 12 shows the number of instructions delayed waiting on speculative operands for each SpecShield design. As expected, SpecShieldSTL leads to the most delayed instructions, an average of 17% of the total number of instructions. The most dramatic improvements are observed for SpecShieldERP+, which reduces the fraction of stalled instructions to less than 1.3% on average. This is because, by allowing early forwarding of speculative data to low CCR instructions, we allow additional instructions to execute without delay while stalling only high CCR instructions.

E. Performance/Coverage Summary

For completeness, Table IV gives a summary of mitigation overheads for existing software and hardware solutions, as well as the channels each protects against leakage. Descriptions of each of these defenses can be found in Section VIII. We report average overheads for the software-based defenses listed in Table IV from compiling SPEC2006 C applications using Intel ICC (v19.0.3.206) with automated `lfence` injection enabled [36], as well as CLANG (v.9.0.0) with the Speculative Load Hardening [37] mitigation enabled.

	Defense	Overhead	Benchmarks	Channels Protected
SW	LFENCE [35]	144%	SPEC2006	All
	SLH [37]	108%	SPEC2006	Cache
HW	Invisispec [8]	22-78%	SPEC2006	Cache
	SafeSpec [9]	-3%	SPEC2017	Cache, TLB
	DAWG [10]	1-15%	PARSEC	Cache
	CS Fencing [38]	8-48%	SPEC2006	Cache
	Cond. Spec. [11]	7-53%	SPEC2006	Cache
	Select Delay [16]	11-46%	SPEC2006	Cache
	SpecShieldSTL	73%	SPEC2006	All
	SpecShieldERP	21%	SPEC2006	All
	SpecShieldERP+	10%	SPEC2006	Flexible

Table IV: Overhead of existing and proposed mitigation solutions and channels they protect from leakage.

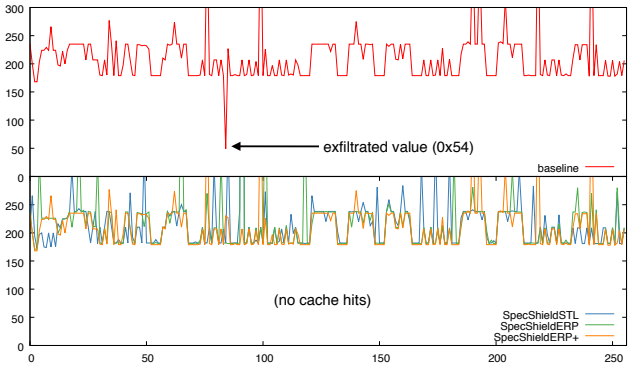

Figure 13: Access latency for each index of array `a` in Listing 1, showing baseline (top) and SpecShieldSTL, ERP and ERP+ (bottom).

VII. Security Analysis

In this section we analyze the security guarantees offered by SpecShield and discuss the security/performance trade-offs.

A. Empirical Results

To analyze the effectiveness of SpecShield in preventing leakage through covert channels we simulated proof-of-concept Spectre-v1 code in gem5, similar to the code shown in Listing 1. Flush+Reload was used to recover the secret. Figure 13 shows the empirical results averaged over 100 trials. In theory, the secret value will have the lowest access latency of any index in the array, enabling us to decode its value. The secret value is one byte wide, therefore the array must have 256 indices to represent every possible value. The top of Figure 13 shows the unsecure baseline. We can see that there is only one index with an access latency less than 175 cycles; indeed, this index corresponds to the correct secret value extracted from the victim process (`0x54`).

However, none of the SpecShield variants exhibits such an outlier, as shown in the bottom of Figure 13. This shows that our mitigation techniques are effective in shielding speculative data from the covert channel.

B. Discussion of Isolation Properties

As Figure 2 shows, SpecShield isolates data retrieved from the memory hierarchy through speculative load instructions from covert channel leakage. The degree of this isolation varies with the design. However, all SpecShield designs provide well defined security guarantees with respect to that isolation.

SpecShieldSTL delays the use of data loaded by speculative instructions until Load instructions are committed. Data cannot be accessed by any other instruction until the Load retires. Any attack that attempts to speculatively load data from an illegal memory address or from an illegitimate control flow path will not be able to leak the speculative data. Exception handling, permissions checks and branch resolution for each load and all older instructions is performed before data use is allowed. As a result, SpecShieldSTL guarantees that any data which is the result of a speculative load instruction is isolated from any other instruction until it is safe.

SpecShieldERP allows forwarding of data loaded by speculative instructions only when these instructions are older than the ERP, which means they (and any older loads) did not cause exceptions and are not part of misspredicted control flow. While these instructions can still be squashed by other exceptions, traps or aborts, transient execution has been shown to halt after the faulting instruction or return dummy values where no actual information is leaked for these cases [12]. Therefore, the security guarantees of SpecShieldERP are the same as SpecShieldSTL, because only safe loads are allowed to forward data to dependent instructions.

SpecShieldERP+ relaxes the security guarantees of SpecShieldERP by allowing speculative data to be used by instructions that are deemed to have a low covert channel risk (CCR). As long as the classification of instructions based on their CCR is accurate, SpecShieldERP+ is as secure as SpecShieldERP. The potential additional vulnerability of SpecShieldERP+ comes from the possibility that new covert channels will be discovered for instructions classified as having a low CCR. We manage this increased risk by allowing firmware updates to reclassify instructions as high CCR by updating the CCR table of a processor if new vulnerabilities are discovered.

In the worst case, adding to the list of instructions which must wait for data to become non-speculative with respect to our definition could cause SpecShieldERP+ to degenerate into SpecShieldERP. However, note that the performance overhead of SpecShieldERP is already quite reasonable at an average of 21%. SpecShieldERP+ allows the opportunity to remove constraints imposed on forwarding data when they are known to be unnecessary for a given micro-architecture. This reduces performance impact further.

VIII. Related Work

A. Software Mitigation Solutions

Various software solutions have been deployed in an effort to mitigate the three initial Spectre and Meltdown variants [36], [39]–[41]. A proposed Spectre-v1 (bounds-check-bypass) mitigation is the insertion of a serializing instruction (`lfence`) after any vulnerable section [36]. Speculative Load Hardening (SLH) [37] examined the use of code transformations that inject data-dependencies and masking operations around conditional branches as a mitigation to Spectre-v1. Other work focused on mitigating the Branch Target Injection variant of Spectre through a hybrid approach that combined microcode and system software [36]

to allow flushing and disabling the branch predictor at runtime. Turner et al. [24] considered a compiler-level steering scheme that is also known as a return trampoline as a way of isolating indirect branches. The kernel protection table isolation (KPTI) solution was introduced as a mitigation for Rogue Data Cache Load (Meltdown) [42].

B. Hardware Mitigation Solutions

Invisispec [8] develops a defense mechanism for the data cache hierarchy, by removing observable side-effects from the channel. This is accomplished by placing data from transient loads into a temporary buffer and bypassing the data cache until it can be determined the load was not misspeculated. This requires modifications to system components, such as the cache coherence protocol, in order to detect and revert execution in the event of missed invalidations for data in the temporary buffer. Conditional Speculation [11] again is an approach to protect the data cache from leakage, blocking memory requests in the issue queue until they are known to not be misspeculated. Sakalis et al. [16] take a similar approach, blocking transient loads from execution, but also incorporate value prediction in order to hide miss latency. SafeSpec [9] proposes that separate shadow structures for processor components be used during transient execution. Transient state of instructions remain in these structures until the outcome of the prediction is known. During a misprediciton, transient results remain invisible to the main CPU structures. The prototype evaluates only shadowed set-associative structures such as the ITLB, DTLB and caches. Context-Sensitive Fencing [38] explores automatically injecting fences into the instruction stream when necessary, focusing on preventing leakage from the data cache. Barber et al. [43] proposed an initial idea for a covert channel agnostic defense by delaying the use of speculative data. The overheads for these countermeasures are summarized in Table IV.

SpecShield takes a fundamentally different approach than these prior works. SpecShieldERP blocks speculative data from being used by any instruction. This removes the ability to construct any covert channel to transfer this data into the architectural state. SpecShieldERP+ offers the flexibility to selectively guard against known or potential channels. SpecShield is implemented within the processor pipeline, requiring no changes to the cache hierarchy, coherence protocol or memory consistency.

IX. Conclusion

Transient execution attacks have revealed fundamental weaknesses in how modern processors handle speculative data. This paper presents a first step towards isolating speculative data from any potential covert channels that could be used to leak secret data during speculation. SpecShield prevents the use of speculative data by other instructions until doing so is determined to be safe. The most secure version of the design, SpecShieldERP, eliminates transient attacks at a cost of 21% average performance degradation. The optimized version, SpecShieldERP+, which prevents the propagation of speculative data to known or probable covert channels, provides only slightly relaxed security guarantees with a 10% performance impact. We also presented a mechanism for trading off performance for security, by allowing designers to easily choose which instructions are considered to have high covert channel risk.

Acknowledgment

The authors would like to thank the anonymous reviewers for their great feedback. This work was funded in part by the Air Force Research Laboratory and a gift from Intel Corporation.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P 2019 : 40th IEEE Symposium on Security and Privacy*, 2019.

[3] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering covert timing channels on shared processor hardware," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 216–228.

[4] Y. Yarom and K. E. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," *USENIX Security Symposium*, vol. 2013, pp. 719–732, 2014.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, pp. 1–20, 2006.

[6] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network." *arXiv preprint arXiv:1807.10535*, 2018.

[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[8] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[9] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC '19 Proceedings of the 56th Annual Design Automation Conference 2019*, 2019.

[10] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 2018, 2018, pp. 974–987.

[11] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional Speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 264–276.

[12] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses." *arXiv preprint arXiv:1811.05441*, 2018.

[13] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

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

[15] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS 2018 : The 25th ACM Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[16] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Sjlander, "Efficient invisible speculative execution through selective delay and value prediction," in *ISCA '19 Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 723–735.

[17] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

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

[19] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 947–960.

[20] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels." *arXiv preprint arXiv:1806.07480*, 2018.

[21] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," *arXiv:1905.05726*, 2019.

[22] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[23] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading kernel writes from user space," *arXiv preprint arXiv:1905.12701*, 2019.

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

[25] G. B. Bell and M. H. Lipasti, "Deconstructing commit," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2004*, 2004, pp. 68–77.

[26] J. F. Martnez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002, pp. 3–14.

[27] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *Proceedings of the 26th annual international symposium on Microarchitecture*, 1993, pp. 202–213.

[28] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 272–284.

[29] G. E. Suh, J. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, vol. 39, no. 11, 2004, pp. 85–96.

[30] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004, pp. 243–254.

[31] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, vol. 44, no. 3, 2009, pp. 109–120.

[32] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

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

[34] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction-level Parallelism*, vol. 7, 2005.

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

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

[37] C. Carruth, “RFC: Speculative load hardening (a spectre variant #1 mitigation),” 2018, https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.

[38] T. Mohammadkazem, A. Venkat, and D. Tullsen, “Context-Sensitive Fencing: Securing speculative execution via microcode customization,” in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[39] Mitre, “CVE-2017-5753,” Mitre, 2017, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753.

[40] ——, “CVE-2017-5715,” Mitre, 2017, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715.

[41] ——, “CVE-2017-5754,” Mitre, 2017, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754.

[42] 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.

[43] K. Barber, L. Zhou, A. Bacha, Y. Zhang, and R. Teodorescu, “Isolating speculative data to prevent transient execution attacks,” *IEEE Computer Architecture Letters*, 2019.