SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation

Esmaeil Mohammadian Koruyeh*, Shirin Haji Amin Shirazi*,Khaled N. Khasawneh†, Chengyu Song* and Nael Abu-Ghazaleh*
*Computer Science and Engineering Department
University of California, Riverside
{emoha004,shaji007,csong,naelag}@ucr.edu
†Electrical and Computer Engineering Department
George Mason University
{kkhasawn}@gmu.edu

Abstract*—Spectre attacks and their many subsequent variants are a new vulnerability class affecting modern CPUs. The attacks rely on the ability to misguide speculative execution, generally by exploiting the branch prediction structures, to execute a vulnerable code sequence speculatively. In this paper, we propose to use Control-Flow Integrity (CFI), a security technique used to stop control-flow hijacking attacks, on the committed path, to prevent speculative control-flow from being hijacked to launch the most dangerous variants of the Spectre attacks (Spectre-BTB and Spectre-RSB). Specifically, CFI attempts to constrain the possible targets of an indirect branch to a set of legal targets defined by a pre-calculated control-flow graph (CFG). As CFI is being adopted by commodity software (e.g., Windows and Android) and commodity hardware (e.g., Intel's CET and ARM's BTI), the CFI information becomes readily available through the hardware CFI extensions. With the CFI information, we apply CFI principles to also constrain illegal control-flow during speculative execution. Specifically, our proposed defense, SpecCFI, ensures that control flow instructions target legal destinations to constrain dangerous speculation on forward control-flow paths (indirect calls and branches). We augment this protection with a precise speculation-aware hardware stack to constrain speculation on backward control-flow edges (returns). We combine this solution with existing solutions against branch target predictor attacks (Spectre-PHT) to close all known non-vendor-specific Spectre vulnerabilities. We show that SpecCFI results in small overheads both in terms of performance and additional hardware complexity.

I. Introduction

The recent Spectre [43] attacks have demonstrated how speculative execution can be exploited to enable disclosure of secret data across isolation boundaries. Specifically, attackers can misguide the processor to speculatively execute a read instruction with an address under their control. Although the speculatively read values are not visible to programs through the architectural state, since the misspeculation effects are eventually undone, they can be communicated out using a *covert channel*. Since their introduction, a large number of attacks following the same pattern (temporary read of sensitive data through speculation, followed by disclosure of this data through a covert channel (e.g., [32], [51])) have been discovered which enable bypassing different permissions using a number of different speculation triggers [10], [13], [27], [30], [42], [45], [47], [61], [66], [70], [76]; it is clear that this is a general class of vulnerability that requires deep rethinking of processor architecture.

Since speculation is essential for the performance of modern processors, to mitigate this threat without severely restricting speculation, some solutions such as InvisiSpec [77] and SafeSpec [40] propose separating speculative data from committed data. Such an approach, rather than attempting to limit speculation, would isolate possible leakage. However, the principle has to be applied to every micro-architectural structure (e.g., cache, TLB, DRAM row buffer), and it is unclear if this approach could prevent leakage through contention, for example, using the functional unit port side-channel [7], [13], [52].

Another direction to mitigate this threat is to restrict speculation if a potentially dangerous gadget can be executed speculatively. For example, Intel and AMD suggest inserting serialization instructions like `lfence` to prevent loading potentially secret data [6], [36]. Because blindly inserting serialization instructions will have the same effect as disabling speculation, thus severely reducing performance [34], a better solution is to conditionally insert barriers. The MSVC C compiler [49], oo7 [74], and Respectre [33] use static analysis to identify dangerous gadgets and only insert `lfence` before the identified gadgets. Context-Sensitive Fencing [67] dynamically inserts serialization instructions when a load instruction operates on untrusted data (address), but only for Spectre-PHT.

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

In contrast to traditional CFI, even hardware supported proposals, whose purpose is to prevent illegal control flow within the primary architecturally visible control flow of a program, SPECCFI pushes CFI to the speculation level, where it can be used to determine whether a speculative execution path should be allowed or limited. Compared to existing solutions against Spectre-BTB and Spectre-RSB, such as the recent microcode update from Intel [36] and retpoline [69], SPECCFI introduces less performance degradation as it still allows correct speculation to proceed, while these existing solutions blindly "disable" all indirect branch prediction.

We also like to argue that defenses against Spectre-BTB and Spectre-RSB serve as the foundation for defense against Spectre-PHT (v1) attacks. The reason is that serialization instructions can be viewed as a special type of inline reference monitor and, therefore, it is crucial to make sure that these inserted barriers are never bypassed. However, without protections against Spectre-BTB (forward indirect branches) and Spectre-RSB (returns), attackers can easily bypass the barriers to carry out the attacks [13]. Furthermore, as demonstrated in return-oriented programming [65], by jumping to the middle of an x86 instruction, attackers can use unintended gadgets, in our case speculatively, to launch attacks. For this reason, we envision SPECCFI being combined with existing solutions against v1 attacks [19], [55], [67] to provide comprehensive protection against Spectre attacks.

The SPECCFI principle can leverage any CFI implementation (e.g., coarse-grained such as Intel's CET [38], or fine-grained such as HAFIX [21]), with small differences in implementation and leading to the enforcement of the respective version of CFI. We present our baseline design for forward edge protection in Section IV and backward edge protection in Section V. We investigate two versions of SPECCFI: SPECCFI-base that implements CFI only for speculation, and SPECCFI-full that also supports CFI for the committed control flow (i.e. conventional goal of CFI). Section VII evaluates performance and complexity of the design. We show that SPECCFI-base eliminates dangerous misspeculation (where the predicted target label does not match the destination), without impacting performance.

SPECCFI-full incurs an additional small overhead, on par with other hardware CFI implementations [20]–[22]. We also analyze the implementation complexity and find that the overhead is modest.

Although some software and hardware solutions have started to appear to defend against this class of attacks, we believe that our solution is elegant along with a number of interesting properties. We believe that it also combines well with other proposed defenses, such as SafeSpec [40] and InvisSpec [77] which limit the speculative side effects once misspeculation occurs, by limiting the opportunities for harmful speculation. Section VIII compares SPECCFI to these and other works.

In summary, the contributions of the paper include:

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

II. BACKGROUND

This section overviews some background: branch predictor structures in modern processors, Spectre attacks, and CFI.

A. *Branch prediction and Spectre attacks*

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

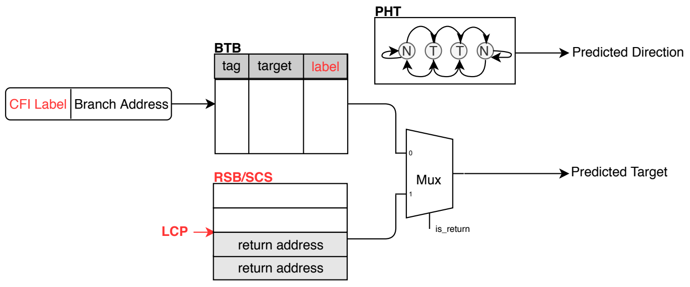


Fig. 1: Branch Predictor Unit consists of three different predictors: (1) PHT for conditional branch direction; (2) BTB for indirect branch addresses; and (3) RSB for return addresses.

Modern processors employ sophisticated predictors (shown in Figure 1) which typically consist of three components:

- Direction predictor: is responsible for predicting the direction of a conditional branch. Although a number of implementations have been studied, modern predictors typically implement a two-level context sensitive predictor [27]. The first level is a simple predictor that hashes each branch address to a direction predictor (typically a 2-bit saturating counter). This predictor is used either when a branch is not being successfully predicted or when the predictor has not been trained yet. When the predictor is trained, it typically uses a second prediction algorithm, often a variant of a gshare predictor [79], which uses the global history of a branch in addition to its address to hash to a direction predictor as before. The advantage is that the same branch can have different predictions based on the control flow path used to reach it.

TABLE I: Spectre attack variants and their targeted branch prediction components

Spectre	Element exploited
Spectre-PHT (v1) [43]	Pattern History Table (PHT)
Spectre-PHT (v1.1) [42]	Pattern History Table (PHT)
Spectre-BTB (v2) [43]	Branch Target Buffer (BTB)
Spectre-RSB (v5) [45], [47]	Return Stack Buffer (RSB)

- Target predictor: is used by indirect jump and indirect call instructions which jump to an address held in a register or a memory location, which is unknown at the front end of the pipeline. This predictor typically uses the hash of the branch address to index a cache holding the branch targets called the branch target buffer (BTB). BTBs are shared across threads on a virtual core: one value used by a process could be used by another process whose branch has a matching address in the BTB [28].
- The return address stack: Since returns are not well predicted using the BTB, and often follow strict call-return semantics, their target is predicted using a return address stack of fixed size. When a call instruction executes, the return address is pushed on this hardware stack; if overflow happens, previous entries are overwritten [45]. When a return is encountered the top of the stack is popped and used as the return target.

Spectre Attacks Spectre attacks exploit the branch and aliasing predictors to fool them to access unauthorized data speculatively [13], [15], [30], [42], [43], [45], [47]. The main properties that the attack exploits in speculative execution are: (1) lazy permission checks on speculation: while instructions are being executed speculatively, the processor will not check the permissions until the commit stage; (2) Speculative instructions have unintended side-effects on micro-architectural states even if they do not get committed; and (3) attackers can deliberately mislead execution into attacker-intended gadgets by mistraining branch predictors, and use the previous property to leak sensitive information. Specifically, an attacker selected gadget is executed speculatively to perform unauthorized access and leak the value through a side-channel [10], [34], [43]. Based on the prediction structure being attacked, variants of the Spectre attacks that are addressed in this work are shown in Table I. Mitigations for other variants of the Spectre attacks as well as variants of the Meltdown attack have been discussed in detail by Canella et al. [15].

B. Control-flow Integrity

Control-flow integrity (CFI) [4], [57] is a state-of-the-art solution to mitigate control-flow hijacking attacks. In such attacks, attackers corrupt/overwrite control data (i.e., data that controls indirect control transfer, function pointers and return addresses for instance) to divert the victim program's execution to carry out attacker-chosen logic, for example, to enable malware or open a backdoor. CFI prevents such attacks by enforcing a basic safety property: *software execution must follow only legal paths within a control-flow graph (CFG) determined ahead of time* [4]. Hence, a CFI mechanism always consists of two components: one that computes the CFG of the program and one that regulates the control transfer while it is executing.

Constructing CFG. The security guarantee of a CFI mechanism directly depends on the accuracy of the CFG, which can be constructed through static or dynamic analysis. Coarse-grain CFI mechanisms [81], [82] generate the CFG using static analysis: any address-taken function can be a legitimate target for any indirect call; any address taken basic block can be a legit target for any indirect jump; and the address of the next instruction after any call can be a legit target for a return. Although coarse-grained CFI can eliminate most illegal control transfer targets, follow-up research has shown that the CFG used is too permissive/inaccurate that it still allows attacks [17], [29]. Fine-grained CFI solutions improve the accuracy of the CFG by incorporating type information [53], [58], [68], [72], [75]. Unfortunately, the CFG may still allow illegal control transfers [16], [26]. More recently, researchers have proposed utilizing run-time information to further improve the precision of the CFG [24], [54], [71], which can even achieve perfect accuracy [31] (i.e., one possible target per indirect control transfer site).

Regulating control-flow. Once the CFG is calculated, legitimate control transfers can be grouped into equivalence sets. Within the same set, control-flow can be transferred from any source location (e.g., a call site or return site) to any target location (e.g., target function or call site). By assigning each equivalence set a unique ID/label, run-time control-flow can be regulated with a simple check—source label must match destination label. Such checks can be implemented using either software or hardware. Some hardware extensions only support a single label [11], [38], [39] thus can only enforce coarse-grained CFI. Others support multiple labels [20], [22] and fine-grained CFI. Some hardware extensions also include a shadow stack to enforce unique return target [20], [21], [38], [39].

Adoption. Because of its effectiveness against control-flow hijacking attacks, CFI has been adopted by both commodity software and hardware. Tice et al. [68] introduced forward-edge CFI to LLVM and GCC in 2014. Android adopted this implementation in 8.1 to protect its media stack and extended the protection in Android 9 to more components and the OS kernel. Microsoft introduced its own CFI implementation, control-flow guard in Visual Studio 2015 and has been utilizing it to protect important OS components, including the web browser. In Windows 10 (V1730), Microsoft extended the protection to the OS kernel and hypervisor (Hyper-V). On the hardware side, Intel introduced Control-flow Enforcement Technology (CET) [38] and ARM introduced a similar mechanism, Branch Target Indicators (BTI), in ARMv8.5-A [11].

III. SPECCFI SYSTEM MODEL

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

A. Threat Model

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

Meltdown style attacks [46], [50], [61], [62], [70], [73] are outside the threat model since they occur due to speculation on the value to be used within the execution of the same instruction; privileged kernel memory [46], L1 cache contents [70], fill buffer [62], in-flight data in modern CPUs (for example: Re-Order Buffer and Line Fill Buffers) [73], and store buffer [50], [61]. Moreover, misspeculation through the direction predictor (which leads to Spectre-PHT) does not result in a control flow violation, since both conditional branch directions are legal control flow paths. Luckily, existing works have already developed protections against Spectre-PHT, primarily by limiting speculation around conditional branches that can lead to dangerous misspeculation [19], [33], [49], [67], [74]. Similarly, Spectre-STL is out-of-scope but can be mitigated by disabling speculative store bypass [5], [9], [36]. To the best of our knowledge, SpecCFI is the first hardware design that targets the more dangerous Spectre-BTB and Spectre-RSB attacks even when they use different side-channels (e.g., contention-based side-channel in SMT processors [13]).

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

B. Instruction Set Architecture (ISA) Extension

Most hardware CFI extensions [11], [20]–[22], [38] use target labeling to enforce forward-edge CFI, and a shadow stack to enforce backward-edge CFI. Without the loss of generality, we assume two modifications to the ISA to inform the hardware of the labels from the CFG analysis:

- Extending the indirect `jmp` and `call` instructions to include CFI labels. For coarse-grained CFI enforcement (e.g., Intel CET [38] and ARM BTI [11]), the label at jump and call sites can be omitted.
- Adding a new instruction to mark legitimate indirect branch targets with corresponding labels. For coarse-grained CFI enforcement, the label can be omitted (e.g., the case of Intel CET) or collapsed to two labels: one for jump targets and the other for call targets (e.g., the case of ARM BTI).

The shadow stack is generally transparent to the program and will not be directly manipulated. However, certain language features such as exception handling, `setjmp/longjmp`, require manipulation of the shadow stack. To support these features, additional instructions are needed, but since they do not interact with SpecCFI, we omit their details. The Intel CET specifications [38] provide an example of such instructions. Table II summarizes required ISA changes.

TABLE II: ISA Extensions to support CFI.

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

C. Compiler Modification

SpecCFI relies on the compiler to mark valid indirect control transfer targets with labels. Fortunately, because these required modifications are the same as CFI, they are already available as part of commodity compilers. For example, both LLVM and GCC include support for (1) software-enforced fine-grained forward-edge CFI [68], (2) Intel CET, and (3) ARM BTI. Therefore, SpecCFI requires little or no modifications to the compilers. SpecCFI is compatible with any label based CFI implementation.

IV. Forward-Edge Defense

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

A. Preventing Spectre-BTB (within the same address space)

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

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

Since only committed indirect branches update the BTB, possible targets that may be used by attackers are limited to gadgets starting with a `cfi_lbl` instruction with an identical label to that of the `call/jmp` instructions label. Note that a label may be shared by multiple locations in the code in CFI, and misspeculation among these locations is still possible (i.e., control flow bending [17]); as known from CFI solutions, this set is much smaller than the potential targets set without CFI.

B. Preventing Spectre-BTB (cross-address-spaces)

`0x09: load rax, 0x25`
`0x10: call *rax, L1`
`...`
`0x25: cfi_lbl L1`
`0x26: add rbx,1`
 	
(Attacker)
	`0x09: load rax, 0x50`
	`0x10: call *rax,L1`
	`...`
	`0x25: load rbx,[secret]`
	`0x50: cfi_lbl L1`
	
	(Victim)

Fig. 2: Example attack across address spaces

Storing CFI labels in BTB entries mitigates attacks within the same address space, but not those across address spaces, when attackers pollute the globally shared BTB from another program. In this case, if attackers know the label used by the victim program (e.g., through offline analysis), they can craft an entry in the BTB with the same label and bypass the protection. Consider the example in Figure 2. The attacker inserts L1 and 0x25 in the 0x10 index of BTB, by selecting the label and location of a branch. When the CPU context switches to the victim space, the victim call at location 0x10 is indexed to BTB and uses the BTB entry, inserted by the attacker to predict its target. Since the label matches, the CPU continues speculative execution of the malicious gadget from 0x25, and the attacker successfully redirects the control flow and executes the malicious gadget to reveal the secret.

To prevent cross-address-space attacks, one possibility is to randomize the mapping of addresses to the BTB (e.g., similar to the CASESAR solution for caches [59]) to make it difficult for attackers to guess the label or the location associated with the target branch. However, as this approach only provides probabilistic guarantees against attacks, we decided to use an alternative implementation that avoids using labels in the BTB. Specifically, our implementation enforce the CFI check by ensuring that the first speculatively executed instruction after an indirect branch is a legal `cfi_lbl` instruction with a matching label, guaranteeing that the speculation target is a legal target in the program's Control Flow Graph. We note that this is the standard implementation of hardware acceleration of CFI. However, since we are using CFI to constrain speculation (not just the committed instructions), this approach requires pushing the check earlier in the pipeline to the decode stage of the first instruction on the speculative path. However, as our experimental analysis shows, this change results in negligible impact on performance legal speculation is not delayed.

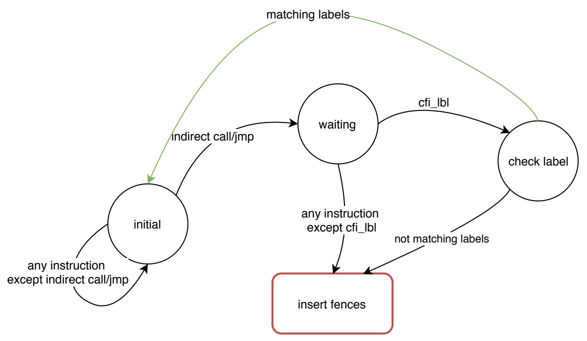


Fig. 3: State machine for forward edge protection

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

The state machine implementing the check in the decode stage of the pipeline is shown in Figure 3. Starting at the initial state, any indirect `call/jmp` instruction in the decode stage sets the `CFI_REG` register with its own CFI label and causes the CPU to wait for a `cfi_lbl` instruction. The decode stage makes sure that the next instruction is a `cfi_lbl` instruction. This restricts potential gadgets to those starting with a `cfi_lbl` instruction. Moreover, the CPU will confirm that the `CFI_REG` value and the label of the `cfi_lbl` instruction are equal. In this way, potential gadgets are further restricted to those with a matching label. When the instruction following the `call/jmp` is not a `cfi_lbl` instruction or when the label of the `cfi_lbl` instruction does not match the label of the `call/jmp`, an `lfence` micro-op is inserted into the pipeline to guarantee prevent execution from the wrong speculative path.

C. Enforcing CFI for Committed Instructions

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

V. Backward-Edge Defense

The backward-edge defense component of SpecCFI protects misspeculation on return instructions. Return instructions typically obtain their predicted addresses from a hardware stack called the Return Stack Buffer (RSB). The RSB has been shown to be vulnerable to a range of Spectre attacks [45], [47]. To provide protection for the backward-edge, hardware CFI proposals use a Shadow Call Stack (SCS), which is protected from normal memory reads and writes, and can only be manipulated through special instructions [38]. Similar to RSB, the SCS is used to retain the return addresses of previously executed calls. The differences are: (1) SCS is in memory, so it is saved and restored across context-switch; while RSB is a special cache in the CPU and its content is shared across different context. (2) SCS is only used for CFI enforcement and its size is configurable; while RSB is only used for speculation, and since misspeculation was thought to be only a performance problem, RSB is a best effort structure that is not maintained precisely and has a limited size.

A. Combined Speculation-consistent RSB/SCS: Overview

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

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

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

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

B. Misprediction Recovery

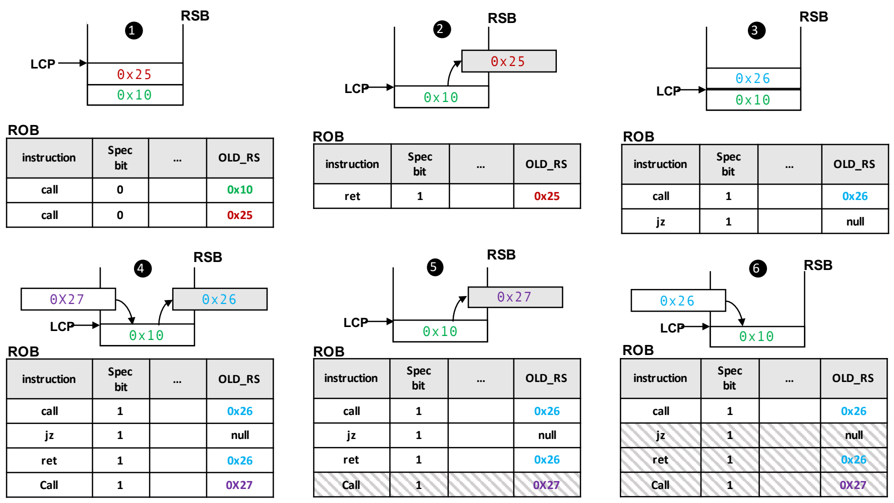
Every `ret` instruction utilizes the RSB/SCS to predict its jump target. Since the state of RSB/SCS is modified by speculative call and ret instructions, in case of misspeculation, the CPU has to recover the correct state of the structure.

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

C. RSB/SCS Work Flow

To clarify how this structure works, we step through the example code sample presented in Figure Figure 5.

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



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

instruction	Spec bit	...	OLD_RS
ret	1		0x25

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

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

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

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

Fig. 4: Example of the operation of the combined RSB/SCS

```
0x09:       call Function1;
0x10:
0x24:       Function1:
                   call Function2;
0x25:              call Function3;
0x26:              call Function4;
0x27:
0x36:       Function2:
                   ret;
0x74:       Function3:
                   jz 0x86;
0x86:       ret;
```

Fig. 5: Code sample to illustrate the operation of RSB/SCS

back to the RSB/SCS to reset the state to the previous state before the misspeculation.

D. *Preventing RSB Poisoning*

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

VI. Security Analysis

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

A. *Guarantees against Branch Target Injection*

Branch target injection attacks target two prediction components: the branch target buffer (BTB) and the return stack buffer (RSB). Similar to CFI, SpecCFI does not prevent such injections: we assume attackers can still insert arbitrary targets into the BTB, for example by executing branches inside their own protection domain [28]. What SpecCFI guarantees is that if the injected target is not a valid indirect control transfer target in the victim protection domain, then the injected prediction target will not be executed speculatively, i.e., they cannot speculatively execute arbitrary code gadgets. For RSB, SpecCFI essentially converts it into a precise shadow call stack (SCS) and maintains it across context switches, such that both in-address-space injection and cross-address-space injection are no longer possible.

Impact of Imprecise CFG: One weakness of static CFG construction is imprecision, leading to having multiple possible targets with the same label. This ambiguity may still allow attackers to launch attacks using permitted function-level gadgets [14], [16], [26], [60]. Since SpecCFI also relies on the CFI analysis to provide valid targets for forward-edge indirect control transfer, it also inherits the same limitation: *mis-prediction is still possible to any of the targets sharing a valid label.* Since SpecCFI is compatible with any label based CFI, it can benefit from improvements in CFI systems that are increasing the precision in tracking the legal control flow.

B. *Incorporating Defense against Spectre-PHT*

SpecCFI on its own can only mitigate Spectre-BTB and Spectre-RSB attacks. In this subsection, we discuss how SpecCFI can be (and *should* be) combined with Spectre-PHT defenses to complete the defense against known Spectre variants. In particular, to defend against Spectre-PHT attacks, researchers have proposed code analysis techniques [33], [49], [74] to (1) identify dangerous code gadgets that can be used to leak information and (2) conditionally insert serialization instructions (e.g., `lfence`) to prevent these dangerous code gadgets from being executed speculatively. One tricky part of such analysis is that, although on the committed path, direct control transfer is always correct; during speculation, even direct control transfer can be wrong. As a simple example, consider a direct call behind a conditional branch: if the prediction on the conditional branch is wrong, then the following direct call is also wrong. For this reason, when analyzing the code to identify potential dangerous gadgets for Spectre-like attacks, one must perform inter-procedural analysis (for both direct and indirect calls) to account for gadgets that may span across function calls. The unique opportunity here is that, if the static analysis to identify and eliminate Spectre gadgets uses the same CFG for CFI enforcement, then malicious gadgets at the beginning of function should already be eliminated. As a result, when combined with such defenses, even if SPECCFI allows misspeculation due to imprecise CFG, the wrong target cannot be used to launch attacks, because the gadgets have already been eliminated.

At the same time, defenses against Spectre-PHT attacks have to use SPECCFI-like techniques to be sound. The reason is the same reason inline reference monitors like Software Fault Isolation [48], [78] have to enforce some control-flow regulation—if attackers can hijack the control-flow to arbitrary locations, then they can easily bypass the inserted checks and bypass the protection. This is especially dangerous to variable length ISA like x86 where attackers can jump to the middle of an instruction to find unintended instructions forming exploitable gadgets. Similarly, SPECCFI provides the same runtime guarantee to Spectre-PHT defenses: by enforcing that even speculative control-flow cannot deviate from the CFG used in static analysis, *the code being analyzed and instrumented will be the same as that executed.*

C. Comparison to Intel CET

A few days before the submission of this paper, Intel published a new specification of its CET [38] extensions. The new specification includes a paragraph (section 3.8) indicating their plans to include a check that an indirect branch executed speculatively targets a legal `Branch_end` target. Intel suggested this solution, which is essentially the configuration of SPECCFI using CET as the CFI implementation, concurrently with our work.

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

- Backward edge protection using the speculation aware shadow stack. While Intel CET uses a shadow stack to protect the backward edge for committed instructions, the specifications describe no plans to use it for limiting speculation. It is not trivial to extend the shadow stack to track the speculative state, as we describe in Section V.
- Generalized CFI protection and limiting control flow bending. CET only enforces that control flow (whether committed or, in the new specifications, speculative) happens to the start of a legal basic block. As a result, it allows arbitrary control flow bending [16], which does not meaningfully restrict the attack opportunities. In contrast, SPECCFI admits any CFI implementation, which can substantially shrink the control bending attack possibilities. Specifically, from a given indirect control flow instruction, only the gadgets with matching CFI label are reachable. State-of-the-art CFI systems such as PathArmor/Context Sensitive CFI can be supported [71] substantially limiting the control flow opportunities. In particular, we intend to explore supporting uCFI [31] in our future work, leaving no control flow bending opportunities available.

TABLE III: Configuration of the simulated CPU

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

VII. PERFORMANCE AND COMPLEXITY EVALUATION

In this section, we evaluate SPECCFI in terms of performance and hardware complexity. All performance experiments were conducted using the MARSSx86 (Micro Architectural and System Simulator for x86) [56], a widely used cycle accurate simulator. MARSSx86 is built using PTLsim [80] and does a full system simulation (including the OS) on top of the QEMU [12] emulator. First, we configured MARSSx86 to simulate an Intel Skylake processor; configurations are shown in Table III. We then integrated SPECCFI into the simulator to model all new operations realistically and in full details, in order to retain hardware faithful cycle accurate modeling of the extended processor pipeline.

A. Performance Evaluation

We use the SPEC2017 benchmarks [2] for evaluation, which is a standard benchmark suite used to evaluate the impact of processor modification on a range of representative applications that exhibit a range of different behaviors. All benchmarks were compiled using an LLVM compiler that is modified to mark valid indirect control transfer targets with labels. Unfortunately, since there is no official LLVM front-end for FORTRAN [3], we were not able to compile 8 out of the 23 SPEC2017 benchmarks as they contain FORTRAN code.

One option to prevent Spectre attacks is to insert fences to stop speculation around indirect control flow instructions.

In order to evaluate SpecCFI performance, we compare it against the following design points:

- Baseline: this is the case of an unmodified unprotected machine. Specifically, we compile and run the SPEC2017 benchmarks using unmodified version of LLVM compiler and MARSSx86 simulator. In all of our experiments, we use the Instructions committed Per Cycle (IPC), a common metric for evaluating the performance of processors, to report performance. The IPC values of the defenses are normalized to this baseline implementation without defenses; thus, a higher normalized value than 1 indicates better than baseline performance.
- *Retpoline-style software fencing*: we implement a system adding fences to indirect branches using software. The compiler is modified to substitute all the indirect branches and return instructions with a sequence of instructions which ensure that the target of the branches are resolved before any following instruction that might touch the cache (i.e, load) are issued. For protecting the forward edges (i.e. indirect call and jumps) This is done by converting each indirect call to the three following instructions: ❶ a `load` preparing the value of the target register/memory, ❷ an `lfence` making sure that no future load is issued before the branch is resolved and ❸ the actual call to the address specified in the target register. Taking the same approach for securing backward edges (i.e. returns) we substitute any `ret` instruction with a sequence of ❶ a `pop` from top of the software stack to the target register, ❷ an `lfence` making sure to stop the speculation before the actual target of `ret` resolved and ❸ a `jmp` to transfer the control to the target. Conceptually, this solution is similar to the Retpoline defense [69] which essentially replaces speculation on indirect branches with an empty stall gadget. Different from Retpoline, we also insert the fences for returns (Retpoline does not protect returns, and leaves the code vulnerable to Spectre-RSB attacks).
  This software approach has the advantage of not modifying the underlying hardware but imposes a noticeable overhead in the number of instructions and code size.
- *All Target Fencing*: In this approach, we show one implementation with an `lfence`, inserted in hardware, at target of each indirect branch and return (the all target fencing) since such a defense is possible without CFI. This is done by detecting every indirect call, jump, or return in the decode stage of the pipeline and inserting an `lfence` at target of them to make sure that the branch is resolved before issuing further instructions.

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

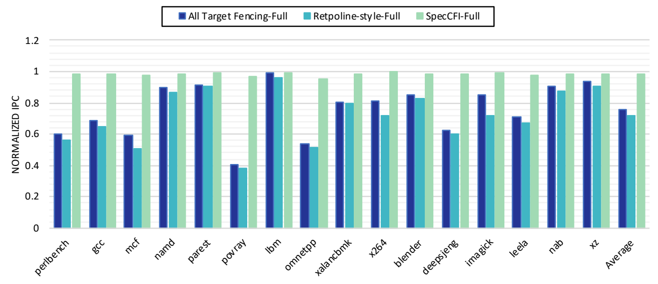


Fig. 6: Performance Impact

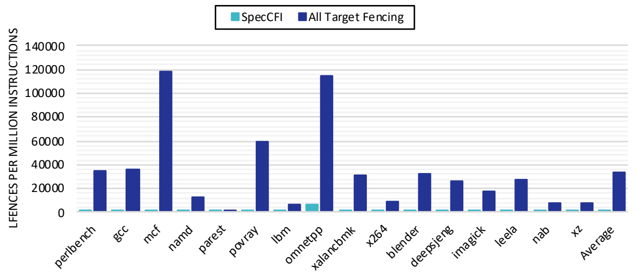


Fig. 7: Number of lfences inserted by different defenses

in our experiments:

- *Strict* `lfences`, are highly restrictive and prevent any instruction to pass through them until the fence retires [67]. This type of fences impose high overhead to the system. All the x86 serialization instructions including the `lfence` we use in our experiment, categorize as strict fences.
- *Relaxed* `lfences`, only stop certain types of instructions until the fence gets retired [67], while letting the others through. For example, LSQ-LFENCE [67], prevents any subsequent load instruction from being issued speculatively out of the load/store queue but allows any other instruction to pass it. LSQ-LFENCEs are secure against Spectre because they prevent the speculative loads, and have the advantage of letting speculation on other types of instructions proceed, substantially reducing the performance impact.

Figure 6 shows the performance overhead of SpecCFI-full (securing both forward and backward edges) in comparison to the *All Target Fencing* and *Retpoline-style software fencing* approaches. We note that in general, inserting serializing instructions (e.g, `lfence`) in the target of every indirect branch is expensive, imposing performance overhead of 39% and 48% on average for *All Target Fencing* and *Retpoline style* respectively. Using SpecCFI, by inserting `lfence` only when the CFI check fails, the number of inserted `lfence` drops significantly thus reducing the performance overhead to less than *1.9%* on average.

To illustrate the reason behind the performance reduction in the different approaches, we study the number of `lfence` instructions inserted in each approach in Figure 7. Note that benchmarks such as `mcf` and `omnet`, are C++ benchmarks which use a large number of indirect branches due to the

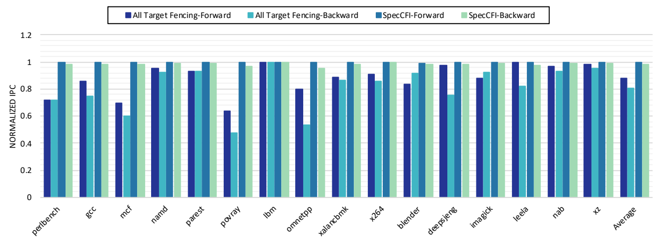


Fig. 8: Overhead breakdown for forward and backward edge

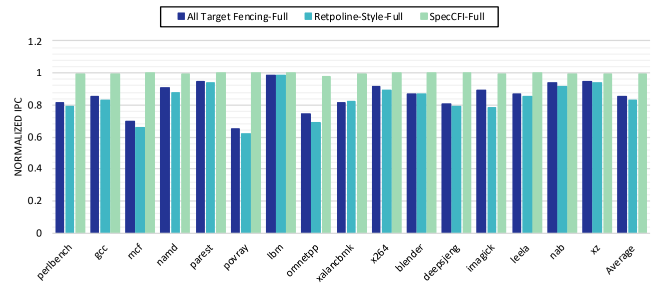


Fig. 9: Performance using relaxed fences

common use of virtual function calls and function pointers. As a result, this leads to a large number of `lfence` being inserted into the pipeline, and to a substantial performance impact compared to the baseline implementation. The only exception to this trend is `Provay` which suffers the highest overhead for all the defenses but does not have huge number of `lfence` compared to the other benchmarks. Looking more closely at this benchmark, we found out that it is a memory intensive benchmark with the highest number of load and store micro-ops among all the benchmarks. Intel manuals [37] indicate that an `lfence` is committed only when there is no preceding outstanding store. Thus, for this benchmark, each `lfence` instruction remains active for a longer period of time until it gets committed which explains the high performance impact. It is also worth mentioning that unlike the *All target fencing* and *Retpoline-style* which insert `lfence` for each indirect branch, the `lfence` instructions for SPECCFI occur due to mis-prediction detected as a label mismatch causing the insertion of the `lfence`. This means that the higher the rate of mis-prediction, the more `lfence` instructions are inserted.

In Figure 8, we study the effect of securing the forward and backward edges separately since they use separate mechanisms for protection. Note that in *Retpoline-style*, all return instructions are converted to a sequence of instructions terminating with a `jmp`, meaning that there is no remaining `ret` instruction (i.e. backward-edge) in the code compiled in this setting. Therefore, the overhead measured as the overhead of *Retpoline-style*-full is equivalent to only *Retpoline-style*-forward overhead and the overhead on the backward-edge is zero. The results from the breakdown show that as expected, the overhead in general increases with the number of indirect branches in *All Target Fencing*. As for SPECCFI, the overhead caused from forward edge defense is typically low: the overhead is incurred only on CFI mismatches which indicate misprediction of the branches. Therefore, the major part of the SPECCFI overhead is the overhead of SPECCFI-full on the backward-edge which is associated with maintaining the RSB/SCS hardware structure. It is important to consider that this maintenance effort also includes procedures to make sure the committed path is secure and therefore only a portion of this overhead is associated with defense against Spectre attacks.

Since strict `lfence` imposes a higher overhead on the system and relaxed `lfence` provides the same security guarantee with lower overhead, we implemented all discussed defenses with relaxed `lfence` as well to study the differences in overhead. Figure 9 examines the effect of relaxed `lfence`. The results show that the overhead caused by strict `lfence` is much higher than that of relaxed `lfence`. Also as expected, using strict instead of relaxed causes far more performance degradation when the benchmark is memory intensive (i.e., has a lot of stores in this case). Our results show that just by changing the type of the `lfence` from strict to relaxed, the average overhead drops down from 48.9% to 22.6% for *Retpoline-style* and from 39.9% to 18.82% for *All Target Fencing*. However, these overheads are still substantially higher than those of SPECCFI.

B. Hardware Implementation Overhead

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

Additionally, SPECCFI needs a `LCP` register to point to the last entry of the RSB/SCS from a committed call, used to distinguish between entries from speculative and committed instructions. Since RSB/SCS has 16 in-processor cache entries, the `LCP` size is 4-bit. Moreover, at two stages of the pipeline, new entries can be added to the RSB/SCS: (1) while executing call instruction and (2) load the preserved RSB/SCS entries from memory in case of underflow. Therefore, we had to update the number of write ports from 1 to 2. The same thing applies to the number of read ports, as we may use RSB/SCS to fetch next instruction while spilling over to memory in case of RSB/SCS overflow. In addition, to preserve the correct behaviour of RSB/SCS, we provided two `LCP` update mechanisms: (1) -/+1: for regular push/pop operations and (2) -/+4: for handling overflow and underflow of the structure. The cost of the RSB/SCS itself did not lead to a noticeable increase in complexity or area.

TABLE IV: SpecCFI hardware implementation overhead after adding it to the AO486 open-core

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

To measure the impact of SpecCFI implementation on power, area, and cycle time, we modified the open source processor (AO486) [8] to include SpecCFI design using Verilog. To synthesize the implementation of integrating SpecCFI to the processor on a DE2-115 FPGA board [1] we used Quartus 2 17.1 software. The results shown in Table IV prove that SpecCFI indeed has low implementation complexity. In terms of power, there is a 0.4% increase in core dynamic and static power. Although it is difficult to measure power accurately, we applied the power analysis tool provided by Quartus to measure power after synthesis to get more accurate results. In terms of area, there is a 0.1% increase in total logic elements. Moreover, since SpecCFI design is simple, it fits within the optimized frequency of the core. Thus, it has no effect on cycle time. The AO486 processor is an implementation of the 80486 ISA using a 32-bit in-order pipeline. Thus, these results are relative to the small pipelined core; the overheads will be much smaller if compared to a modern out-of-order superscalar core.

C. Empirical Security Evaluation

1) Against real exploits: To verify our analysis, we evaluated the effectiveness of SpecCFI against real-world exploits. We ran previously disclosed Spectre-BTB [43], Spectre-RSB [45], and SMoTHerSpecter [13] PoC inside the emulator. Table V summarizes the results, using the same classification scheme proposed in [15]. The experiment results show that SpecCFI was able to prevent all information leaks.

TABLE V: Empirical security evaluation of SpecCFI.

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

2) Impact of CFG precision: To study the difference between coarse-grained CFI (e.g., Inte CET [38]) and fine-grained CFI (e.g., SpecCFI) against BTB injection attacks, we used the SMoTherSpectre [13] for a demonstration. In this scenario, the attacker has to find a BTI gadget in the victim process which loads a secret in a register and terminates by an indirect branch to be able to perform BTB injection. By poisoning the BTB, the attacker transfers control to a SMoTHer Gadget to leak the secret. The SMoTHer Gadget starts with a comparison based on the target register followed by a conditional jump which enables SMoTherSpectre to leak the secret through a port contention side-channel. Figure 10 compares the required SMoTHer gadgets and feasibility of the attack under coarse-grained and fine-grained CFI.

Table VI shows the number of available SMoTher Gadgets from several standard libraries. Using the constraints for the

TABLE VI: Available SMother Gadgets in Standard Libraries

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

SMother Gadget identified by Bhattacharyya et al. [13], we scanned for valid SMoTHer gadgets in the first 70 instructions after label instructions (`endbr64` and `cfi_lbl`). For SpecCFI, we used a function signature based approach for generating labels [53], [54]. As we can see, although fine-grained CFI still permits some gadgets, the number is much smaller than that available under coarse-grained CFI.

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

VIII. Related Work

Since the initial announcement of Spectre and Meltdown in January of 2018, several Spectre variants have appeared [27], [30], [42], [43], [45], [47]. Spectre attacks are characterized by manipulating the prediction mechanisms to trigger speculation to an attacker chosen gadget. They differ in what they exploit to trigger speculation: branch direction predictor (variant 1, variant 1.1) [27], [42], [43], branch target predictor (or branch target buffer) for variant 2 [43], return stack buffer for Spectre-RSB (also called variant 5) [45], [47], or load-store aliasing predictor for variant 4 [30]. To mitigate these attacks, several software and hardware defenses ranging from programming guidelines for cryptographic software developers [18] to architectural changes [40], [77] have been proposed. In this section, we will overview these defenses categorized into the Spectre attack variants that they defend against. Table VII shows the Spectre attacks defenses and which attacks they mitigate and Table VIII shows the Spectre attacks defenses and their impact on hardware complexity, software modifications, and performance. SpecCFI is the only defense that provides complete protection against all Spectre attacks with little impact on performance and implementation overhead. Note that we are not considering Meltdown style attacks [46], [50], [61], [62], [70], [73] since they rely on speculation within a single instruction and therefore do not rely on manipulating the branch prediction structures.

A. Spectre-PHT Defenses

Spectre-PHT exploits the directional predictor (also called the Pattern History Table or PHT) to perform the attack. To defend against this attack, Intel, AMD, and ARM proposed to use instructions that serialize the execution (e.g. `lfence`) to stop speculation around conditional branches [6], [9], [34].

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

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

Attacker **Victim**

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

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

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

Attacker **Victim**

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

Fig. 10: Speculative control-flow bending attack example.

TABLE VII: Spectre defenses and the attacks they mitigate. Symbols show if an attack is mitigated (●), not mitigated (○), or partially mitigated (◐).

Attacks	Side-channel prevention		Speculation prevention							
	DAWG [41]	SafeSpec [40], InvisiSpec [77]	LFENCE [6], [9], [34]	IRBS, IBPB, STIBP [6], [36]	(SLH) [19], (YSNB) [55]	Retpoline [69]	RSB Stuffing [35]	CSF [67]	ConTExT [63]	SPECCFI
Spectre-PHT	◐	◐	●	○	●	○	○	●	◐	●[a]
Spectre-BTB	◐	◐	○	●	○	●	○	◐	◐	●
Spectre-RSB	◐	◐	○	○	○	○	●	◐	◐	●
SmotherSpectre	○	○	○	○	○	◐	◐	◐	◐	●

[a]Combined with any Spectre-PHT defense

TABLE VIII: Spectre defenses and their overhead in terms of hardware complexity, software modification, and performance. Symbols show if overhead is high (↑), low (↓), or no overhead (-). The performance overhead results are based on what was reported in the studies; Please note that these values are not based on experiments on identical benchmarks and are only reported to provide a general sense of performance.

Overhead	Side-channel prevention		Speculation prevention							
	DAWG [41]	SafeSpec [40], InvisiSpec [77]	LFENCE [6], [9], [34]	IRBS, IBPB, STIBP [6], [36]	(SLH) [19], (YSNB) [55]	Retpoline [69]	RSB Stuffing [35]	CSF [67]	ConTExT [63]	SPECCFI
Hardware	↑	↑	–	–	–	–	–	↓	↓	↓
Software modification	–	–	↑	–	↑	↓	↓	↓	↓	↓
Performance	1 - 5 %	SafeSpec: -3% InvisiSpec: 22%	62 - 74.8 %	20 - 50 %	SLH: 29 - 36.4 % YSNB: 60 %	5 - 10 %	↓	2.7 - 15.2 %	1 - 71.14 %	1.9 %

Although aggressive serialization (e.g., at every branch instruction) can mitigate Spectre-PHT, it hurts performance substantially [34]: serializing all branch instructions will eliminate the performance benefit of the branch predictor (e.g., up to 10x slowdown [55]). Therefore, multiple proposals tried to reduce the number of serialization points introduced using static analysis to identify and serialize exploitable gadgets only [33], [34], [49], [74]. However, these approaches miss some of the gadgets that can be exploited [44]. Another weakness of these defenses is that even though they stop speculative execution around exploitable gadgets, they do not stop speculative code fetches and other micro-architectural behaviors before execution (e.g., instruction cache and iTLB fills) which can also leak data [64].

Speculative Load Hardening (SLH) [19] and You Shall Not Bypass (YSNB) [55] try to reduce the high overhead by identifying Spectre gadgets, then injecting artificial dependencies between branches and these identified gadgets. Although this results in performance advantages over liberal fencing, they still have 36%-60% performance overhead [67]. Context-Sensitive Fencing (CSF) [67] is a micro-code mitigation technique where serialization instructions are added dynamically based on run-time conditions that identify potential exploit execution. Although CSF focuses primarily on Spectre-PHT, the authors propose to defend against Spectre-BTB and Spectre-RSB using a special fence that would flush the BTB/RSB when transferring control to higher domains. However, flushing BTB and RSB hurts performance since it results in mis-predictions. In addition, in a simultaneous multithreading (SMT) processor, flushing the BTB/RSB after control transfer is not enough to protect against Spectre-BTB and Spectre-RSB since structures can be polluted after a control transfer using other threads.

B. Spectre-BTB and Spectre-RSB Defenses

Spectre-BTB exploits the BTB and Spectre-RSB exploit the RSB to perform the attack. Google proposed Return Trampoline (retpoline) [69] as a software mitigation technique that defends against Spectre-BTB by replacing indirect branches with push+return instruction sequences that prevent BTB poisoning. However, this solution has high performance overhead since it stops speculation (similar to serialization). In addition, it can be bypassed using *ret* instructions since they cause mis-speculation through the BTB in some processors (e.g., Intel's core i7 processors starting from Skylake). In particular, those processors predict the address of a *ret* instruction from the BTB when the RSB is empty (which can be forced by executing unmatched returns). To solve this exploit, RSB stuffing [35] was proposed to intentionally fill the RSB with the address of a benign delay gadget to avoid misspeculation on context switches. Although this technique can partially mitigate Spectre-BTB (when using *ret* to trigger speculation through BTB), it can also defend against SpectreRSB cross-domains attack. However, since we are filling the RSB on context switch, stored entries for the currently running process will be lost when execution is switched back to the current process (i.e. performance loss due to losing speculation information). In contrast, SPECCFI saves committed RSB entries per process in case of a context switch out of the process and restores them when execution returns to the process, which results in improving the prediction performance of *ret* instructions.

Intel and AMD added new instructions to their instruction set architecture (ISA) that can control indirect branches to defend against Spectre-BTB [6], [36]. The addition consists of three controls:

- Indirect Branch Restricted Speculation (IBRS): allows processors to enter IBRS mode (privileged mode) and execute indirect branches that are not influenced by less privileged mode.
- Single Thread Indirect Branch Prediction (STIBP): will not allow a hyperthread running on a core to use branch predictor entries inserted by the other thread running on the same core.
- The Indirect Branch Predictor Barrier (IBPB): allows processors to flush BTB and clear their state. This way, the code executed before the barrier cannot impact branch prediction of the code executed after this instruction.

These new ISA instructions defend only against Spectre-BTB. In addition, they have a high performance overhead; up to 24% on Skylake and up to 53% on Haswell [23].

C. Spectre All Variants Defenses

Several mitigations were proposed to defend against all variants of Spectre. Dynamically Allocated Way Guard (DAWG) [25], [41] was proposed to provide isolation between protection domains by partitioning the cache at the cache way granularity. Although this method can prevent leakage of the data through a cache side-channel, it requires domains enforcement management in the software, defending cache as leakage source only, and it can not protect against attacks that are performed within the same address space or isolation domain. In addition, since it is a cache specific defense, other micro-architectural structures can be used for communication (e.g. branch predictor).

SafeSpec [40] and InvisiSpec [77] are hardware mitigation techniques that are similar to DAWG in the way that they are both trying to prevent side-channel communication from speculative instructions. Therefore, they propose to mitigate the side-effect of speculative execution on the micro-architectural state; shadow micro-architectural structures for caches and Translation Lookaside Buffers (TLBs) were added to store transient effect of speculative instructions. These effects will be committed to caches and TLBs only when speculation is deemed correct and flush the changes from the shadow structures otherwise. Although these solutions outperform software solutions, they require making disruptive changes to the processor/memory architecture and consistency models.

ConTExT [63] introduced protection for secret data from speculative execution. Specifically, the technique introduces a new memory mapping (called non-transient mapping) which tracks data that must not be accessed by speculative instructions. Nevertheless, this solution requires changes to the architecture and the operating system, and developer involvement to annotate the secret data. It also incurs high performance overhead for security-critical applications.

IX. CONCLUDING REMARKS

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

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

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

ACKNOWLEDGEMENTS

This paper was partially supported by NPRP grant 8-1474-2-626 from the Qatar National Research Fund (a member of Qatar Foundation). The statements in the paper are solely the responsibility of the authors.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

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

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

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

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

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

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

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

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

[34] 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.

[35] Intel. Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, 2018.

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

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

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

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

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

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

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

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

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

[45] E. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[46] 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 *USENIX Security Symposium (Security)*, 2018.

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

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

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

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

[51] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[52] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

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

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

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

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

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

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

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

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

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

[62] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726*, 2019.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[81] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dong Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[82] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.