IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 7 November 2023, accepted 6 December 2023, date of publication 7 December 2023,
date of current version 13 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3340680

RESEARCH ARTICLE

MicroCFI: Microarchitecture-Level Control-Flow Restrictions for Spectre Mitigation

HYEREAN JANG AND YOUNGJOO SHIN
School of Cybersecurity, Korea University, Seoul 02841, South Korea

Corresponding author: Youngjoo Shin (syoungjoo@korea.ac.kr)

This research was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIT)(No.2023R1A2C2006862). This work was supported by an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2023-00227165).

ABSTRACT Spectre attack exploits the vulnerability in speculative execution, an optimization technique for modern superscalar processors. Among the attack variants, Spectre-BTB and Spectre-RSB are the most threatening because they allow adversaries to execute arbitrary code in the transient execution context. However, there are few mitigation techniques for these Spectre variants due to the high degree of implementation difficulty. In this paper, we propose MicroCFI, a hardware/software co-design approach to mitigate Spectre-BTB and Spectre-RSB. The main idea of MicroCFI is to enforce control-flow integrity (CFI) in microarchitectural level of a program's execution. Specifically, MicroCFI strictly limits possible forward and backward indirect branch targets predicted by BTB and RSB by imposing CFI properties on all potential targets. As indirect branches only have destinations to valid targets that satisfy these properties, MicroCFI significantly reduces the chance of arbitrary code execution in Spectre attacks. We implemented a prototype of MicroCFI using an LLVM compiler and performed an evaluation on MARSSx86, a simulator for x86 microarchitectures. The security evaluation shows that MicroCFI reduces the number of available Spectre gadgets by more than 90%, significantly increasing the complexity of the attack. The performance evaluation using the SPEC CPU 2017 benchmarks shows that MicroCFI introduces negligible performance overhead.

INDEX TERMS Spectre, control-flow integrity, microarchitectural attack.

I. INTRODUCTION

Modern processors employ various optimization techniques such as out-of-order and speculative execution to maximize instruction-level parallelism. Unfortunately, these optimization techniques are prone to microarchitectural attacks [1], [2], [3], [4], [5], [6], [7], [8]. Spectre [1] is one of the attacks that exploit a vulnerability in speculative execution, allowing attackers to leak secret from a victim across a protection-domain boundary. The vulnerability arises from the fact that although misspeculation cause no changes in the architecture, it can leave a trace in microarchitectural components such as the cache. An attacker exploits this weakness by intentionally inducing misspeculation in a victim's domain, thus encoding confidential data to the cache. The attacker then uses a cache covert channel [9], [10], [11] to decode the data. The Spectre attack has several variants, including Spectre-PHT, Spectre-BTB, and Spectre-RSB, which differ in the branch prediction units they exploit [1], [12], [13], [14], [15], [16], [17].

Spectre-BTB and Spectre-RSB attacks differ from Spectre-PHT in their use of indirect branches instead of directional branch instructions. Since indirect branches have no target restrictions, these attacks can result in speculative arbitrary code execution, causing a more significant security risk than Spectre-PHT. Despite this higher threat level, most studies have mainly concentrated on Spectre-PHT [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], with less attention paid to mitigating Spectre-BTB and Spectre-RSB

The associate editor coordinating the review of this manuscript and approving it for publication was Amjad Mehmood.

attacks. This is primarily due to the high complexity in implementing mitigation techniques. Spectre-BTB (RSB) attack uses multiple branch targets, making it more difficult to block potential gadgets compared to Spectre-PHT attacks, which only utilize direct conditional branches with two branch targets.

Several security measures are available to prevent Spectre-BTB and Spectre-RSB attacks. Retpoline [28] and microcode patches by CPU vendors [29], [30] essentially avoid or disable the use of vulnerable branch prediction units. However, these methods significantly reduces processor performance due to the disabled speculation. Even worse, a vulnerability has been recently discovered in Retpoline [31], which can be exploited by Spectre-BTB-like attacks. Other security solutions [32], [33] aim to mitigate attacks by creating or modifying the hardware structure, introducing a high degree of hardware implementation complexity. In conclusion, current solutions suffer from significant performance issues and the prohibitive cost of hardware modifications.

In this paper, we address the issues of previous solutions and propose MicroCFI, a novel hardware/software co-design approach to mitigate Spectre-BTB and Spectre-RSB. Our method employs a CFI protection mechanism within a microarchitectural context to prevent control-flow hijacking during speculative execution. Specifically, MicroCFI prevents attackers from finding available Spectre gadgets by restricting a program's control flow in the microarchitectural execution context.

The concept of MicroCFI is based on the observation of a normal program's behavior. In the control flow of a normal program, the target address of forward indirect control transfer (e.g., `jmp` and `call`) will always be the entry point of a basic block or function. Likewise, the target of backward indirect control transfer (e.g., `ret`) is the next instruction of a `call` instruction. We refer to the normal forward/backward indirect control transfer target as a *valid target* (VT). Our observation is that any *valid* speculative execution at indirect branches transfers the program's control to VTs. In other words, transfer to non-VTs (i.e., targets other than VTs) is not caused by valid speculation, but only by *malicious* speculations.

In this light, MicroCFI constrains a program's execution by limiting branch targets that are predicted by BTB and RSB to only VTs. Specifically, MicroCFI implements the restriction by making all the VTs to be 2^n-bytes aligned in the program's memory. Such VT's property is ensured by applying n-bit masking to the branch target before branching speculatively, which requires the slight modification of branch prediction process in the pipeline.

Since any code chunks with base addresses that are not VTs cannot be utilized as valid Spectre gadgets, MicroCFI significantly reduces the likelihood of successful Spectre-BTB and Spectre-RSB attacks. Nevertheless, the constraint on VTs does not imply the removal of all possible Spectre gadgets. To ensure that no successful attack can occur, we insert `fence` instructions in all the remaining exploitable gadgets, which comes at the expense of introducing some additional overhead.

The amount of Spectre gadgets eliminated by MicroCFI is determined by the alignment size (n), which is a configurable parameter of MicroCFI. Our evaluation shows that 2^4-byte alignment (i.e., $n = 4$) can reduce the available gadgets by an order of magnitude, effectively mitigating the attack with a reasonable performance overhead (discussed in detail in Section V-B).

In order to implement MicroCFI, the processor's instruction pipeline process needs to undergo a minor modification that involves performing bit-masking operations on predicted addresses in the instruction-fetching unit. This makes it relatively easy to integrate MicroCFI with existing processor architectures, with lower complexity compared to previous hardware-based solutions [28], [29], [30].

We developed a prototype of MicroCFI using MARSSx86, a simulator for x86 microarchitecture, and an LLVM compiler. To evaluate the performance of MicroCFI, we measured its memory and execution overhead on the SPEC CPU 2017 benchmark suite [34]. We conducted experiments using different alignment sizes, including 2^3, 2^4, and 2^5 bytes, to analyze the performance according to the alignment size. The results of these experiments showed that MicroCFI introduces a reasonable performance overhead.

We assessed the security of our approach by measuring the reduction in the number of Spectre gadgets in various applications, including OpenSSL, Nginx, and Apache HTTP servers. Our experimental results show the reduction of more than 90% in the number of Spectre gadgets in most of these applications. We also evaluated the runtime overhead associated with incorporating the `fence` instruction to completely disable the remaining gadgets. Our experiments demonstrate that the `fence` instruction introduces only negligible overhead across all alignment sizes. This outcome is attributed to MicroCFI's significant reduction of gadgets, thereby minimizing the necessity for inserting `fence` instructions.

In summary, our main contributions are as follows:

- We propose MicroCFI, a novel Spectre-BTB and Spectre-RSB mitigation technique. It follows a CFI-based approach and significantly increases the attack complexity by reducing the number of available Spectre gadgets.
- We implement and evaluate a prototype of MicroCFI using the SPEC CPU benchmark suite. Our results demonstrate that MicroCFI can be implemented with only a minor hardware modification and introduces a reasonable performance overhead.
- We perform a security analysis by measuring the number of available gadgets with different alignment sizes. Our results demonstrate that MicroCFI provides the best tradeoff between security and performance when using a 2^5-byte alignment.

TABLE 1. Summary of defense techniques. For Spectre attack, ● indicates that the technique mitigates the attack. For Functionality of speculation, ● indicates that the technique preserves the functionality of speculative execution.

Technique	Methodology	Spectre attack		Functionality of speculation	Hardware modification
		Spectre-BTB	Spectre-RSB		
Retpoline [28]	Limit the use of BTB	●	○	○	-
IRBS, IBPB, SITBP [29], [30]	Limit the use of BTB	●	●	○	-
Venkman [35]	Prevent BTB and RSB poisoning	●	○	●	-
SpecCFI [32]	Prevent malicious speculative execution	○	●	●	BTB entry extension & register addition
ConTExT [33]	Limit specific speculative execution	●	●	●	1-bit extension for all registers
MicroCFI **(This paper)**	Constrain speculative control-flow transfer	●	●	●	Add a bit-masking operation in the fetch stage

The remainder of this paper is organized as follows. In Section II, we discuss existing defense techniques for Spectre-BTB and Spectre-RSB in comparison with MicroCFI. In Section III, we provide background knowledge regarding Spectre attacks and the proposed mitigation technique. Section IV presents the details of the main approaches to MicroCFI. Section V presents the implementation of MicroCFI and its evaluation results, in terms of performance and security. Finally, we conclude the study in Section VI.

II. RELATED WORK

The defense techniques for Spectre-RSB and Spectre-BTB can be broadly classified into hardware- [32], [33] and software-based techniques [28], [29], [30], [35]. In this section, we describe the existing defense techniques based on this classification. Table 1 provides a summary of the defense techniques, including their basic methodology, the attacks they attempt to mitigate, and the requirements for hardware modification.

A. SOFTWARE-BASED PROTECTION

Google proposed Retpoline [28], a protection technique for Spectre-BTB. The Retpoline transforms a `jmp` instruction into a sequence of instructions that ends with a `ret` instruction. In this manner, the Retpoline avoids using BTB for branch prediction, and the attacker cannot produce malicious speculative execution with BTB poisoning. Unlike MicroCFI, Retpoline does not mitigate Spectre-RSB. Moreover, it causes significant performance degradation because it fundamentally disables speculative execution using the BTB.

CPU vendors extended the x86-instruction set [29], [30] by adding new instructions that control branch prediction to prevent Spectre-BTB. The first is an indirect branch restricted specification (IBRS), which puts the processor in a special mode called IBRS. It prevents privileged branch instructions from being affected by less-privileged ones. The second is single-thread indirect branch prediction (STIBP), which prohibits the sharing of branch predictors between different threads on the same core. Finally, the indirect branch predictor barrier (IBPB) flushes the BTB state so that any code executed prior to the IBPB cannot affect the branch predictions after the IBPB. These x86 extensions can be activated via microcode updates and require support from an operating system and BIOS. Like Retpolines, they provide only defense against Spectre-BTB. Because they avoid or restrict speculative execution, they introduce more than four times the performance overhead [1].

Zhuojia et al. proposed Venkman [35], a software-based Spectre-mitigation technique. The goal of Venkman is to constrain all speculative executions at target addresses to prevent attackers from poisoning BTB and RSB. To this end, Venkman aligns the target addresses of all the `jmp`, `call`, and `ret` instructions to ensure that only the aligned addresses are stored in the BTB and RSB. Then, Venkman forces indirect branch instructions to jump only to the start address of the aligned code chunks, which is achieved by instrumenting bit-masking operations prior to the branch instructions.

MicroCFI is similar to Venkman in its basic idea, but differs in the approach that implements the mitigation. Basically, Venkman is a software-based approach that prohibits the execution of unaligned binary to prevent BTB and RSB poisoning. That is, there are no micro-architectural restrictions imposed on the addresses stored in BTB, RSB, or predicted branch target addresses. Assuming the presence of a skilled attacker who can bypass the alignment check or compromise the operating system, they could potentially run a malicious program that does not comply with the aligned binary format, thereby corrupting BTB and RSB. Nonetheless, MicroCFI can thwart such advanced attackers by placing stringent restrictions on the projected addresses within the hardware.

B. HARDWARE-BASED PROTECTION

SpecCFI [32] attempts to mitigate Spectre-BTB and Spectre-RSB using a CFI mechanism. SpecCFI applies an ID-based CFI [36], [37], [38] to prevent Spectre-BTB. This technique

assigns the same ID to both an indirect branch and its valid targets in the CFG. This guarantees the integrity of the control flow by checking whether the indirect branch has the same ID before execution. SpecCFI requires hardware modification; it adds additional storage to each BTB entry to maintain the ID of the target address. Furthermore, it requires additional support from hardware-assisted protection techniques such as Intel CET or ARM BTI to embed IDs in indirect branch instructions. To mitigate Spectre-RSB, SpecCFI also applies a shadow call stack (SCS) that can detect tampering with the return address. Therefore, a special register that stores a return address is required through hardware modification.

ConTExT [33] proposed a new type of memory mapping called non-transient mapping. Memory regions containing confidential data were annotated with C/C++ directives by a program developer. The memory region is then assigned to a non-transient mapping. ConTExT prevents speculative execution in the memory region with non-transient mapping because the non-transient memory regions are not accessible during transient execution. For ConTExT to be effective, non-transient mapping must be guaranteed not only for the confidential data itself, but also for any data propagated from the secret. Therefore, to protect the secret propagated to the register, ConTExT requires hardware modification; it requires an additional bit to identify whether all registers should be operated with non-transient mapping.

These hardware-based protection techniques provide effective and efficient protection against Spectre-RSB and Spectre-BTB. However, they require additional internal storage equipment, such as registers and tags, in the microarchitectural hardware components. Employing such internal storage is very expensive and causes huge hardware complexity in microprocessors. On the other hand, MicroCFI requires no storage but only a bit-masking unit in the instruction fetching hardware, which causes negligible complexity compared to previous solutions.

III. BACKGROUND

A. SPECULATIVE EXECUTION

Modern processors utilize a speculative execution technique to avoid pipeline stalls. This involves predicting and fetching a branch target in advance, so that instructions at the target address can be speculatively executed. Successful prediction provides a considerable performance benefit. However, if the prediction fails, the pipeline should be flushed, and all architectural states affected by the speculative execution must be reverted.

To improve prediction accuracy, processors typically use a specialized hardware unit called a branch predictor. This unit keeps a history of successfully executed branches and, based on this history, predicts the next instruction address when the fetch process begins.

Branch predictors can be classified according to the type of branch instructions, as shown in Figure 1. Pattern-history table (PHT) is used for conditional direct branch instructions,

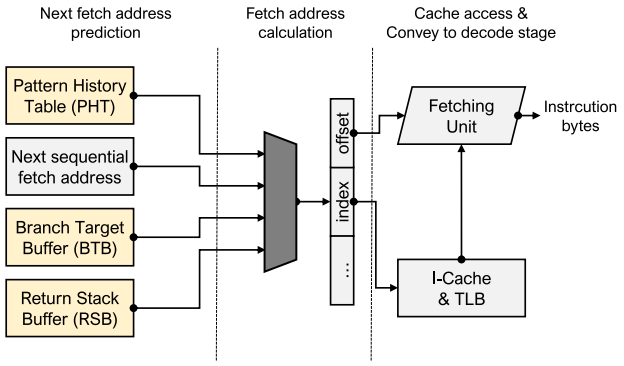


FIGURE 1. Detail of fetch process including branch prediction.

which keeps track of the previous prediction outcomes of directional branch instructions. Forward indirect branch instructions like `call` and `jmp` use a branch target buffer (BTB) for prediction, which stores the addresses of the most recently executed branch instructions along with their corresponding branch target addresses. For backward indirect branch instructions like `ret`, a return stack buffer (RSB) is used, which is a stack structure where the return address is pushed when a `call` instruction is executed.

B. SPECTRE ATTACKS

Speculative execution has a fundamental design flaw that allows for microarchitectural side-channel attacks. This vulnerability originates from the fact that although mispredicted speculative execution reverts the affected architectural states (e.g., registers and memory), microarchitectural states such as cache remain unchanged. Hence, any data derived from speculative execution remain in the cache, even if the execution has been rolled back.

Spectre [1] is a microarchitectural attack that exploits these vulnerabilities in speculative execution. This attack exploits the observation that speculative execution may cause the processor to execute unnecessary instructions. The control over such speculatively executed instructions is achieved through the deliberate manipulation of the branch predictor. The attack proceeds in three steps. In the first step, the attacker locates gadgets to use for the attack within the victim program. The attacker can compose the payload by chaining several gadgets like ROP attack. In the second step, the attacker poisons the branch predictor, causing the victim process to execute attacker-chosen gadgets mis-speculatively. Finally, when these gadgets are speculatively executed and load the victim's confidential data into the cache, the attacker leaks that data through cache covert-channel techniques such as Flush+Reload.

As aforementioned, to make the victim execute the attacker-chosen gadgets speculatively, the attacker must poison a branch predictor, that is, train the branch predictor to perform misspeculation.

Spectre has multiple variants and are classified according to the branch predictors they attempt to poison. Table 2

TABLE 2. Classification of spectre attacks.

Branch type	Branch predictor	Attack
Conditional branch	Pattern History Table (PHT)	Spectre-PHT (Variant 1)
Forward indirect branch	Branch Target Buffer (BTB)	Spectre-BTB (Variant 2)
Backward indirect branch	Return Stack Buffer (RSB)	Spectre-RSB

presents the Spectre variants and the corresponding branch predictors.

Spectre-PHT [1] targets the PHT to cause mispredictions in conditional branch instructions, while Spectre-BTB [1] targets the BTB to poison the predicted branch target of the forward indirect branches. In addition, Spectre-RSB [12] attempts to poison the target address of the return instruction in the RSB.

C. CONTROL-FLOW INTEGRITY

In software attacks, attackers attempt to hijack the control flow of a program by manipulating the memory or register where the branch target addresses are stored. These attacks cause abnormal control flow, which does not occur during normal program execution.

Control-flow integrity (CFI)-based protection technique [36], [39] was designed to mitigate control-flow hijacking attacks in software. It establishes a policy for a valid indirect control flow and forces any control flow transfers caused by indirect branches to follow the policy. Simple but strong CFI techniques use a control-flow graph (CFG). In general, control-flow hijacking attacks cause an invalid control flow over CFG. Therefore, these CFI techniques [36], [40], [41], [42], [43], [44], [45], [46] can detect violations by checking whether an indirect branch moves to a valid target on the CFG for every execution.

However, applying CFG-based CFI techniques to software is not trivial, as it may increase the performance overhead and complexity of software development owing to CFG calculations. To overcome these limitations, lightweight CFI-based [47], [48], [49], [50] techniques have been proposed. Without the use of the CFG, the lightweight techniques perform verification based on the characteristics of normal indirect branches. As the lightweight CFI technique introduces low overhead, it can be easily applied regardless of the program complexity and has the advantage of being able to supplement it by combining it with other protection techniques.

IV. MicroCFI

A. THREAT MODEL

In this study, we consider attackers as follows. Firstly, we assume that the attacker is able to locate their process on the same physical core as the victim. The victim could be running in a different protection domain than the attacker, such as a different process or a privileged kernel. Secondly, the attacker can poison the shared BTB or RSB using their knowledge of the source code or binary of the victim's program. This leads the victim to speculatively execute the attacker's chosen gadgets.

We do not consider microarchitectural side-channel attacks [1], [2], [3], [4] other than Spectre-BTB and Spectre-RSB. That is, our attack model does not include attackers who deliver Meltdown-type or Spectre-PHT attacks, as several countermeasures have already been proposed [18], [19], [20], [21], [22], [23], [24], [25], [51]. It is noteworthy that MicroCFI is orthogonal to the existing countermeasures and can be easily integrated with them.

B. MOTIVATION AND OVERVIEW

1) MOTIVATION

Our approach is mainly motivated by that Spectre attacks lead to abnormal control flow in the speculative execution of a victim's program, which would never occur in normal program execution. This is similar to software-level control flow hijacking attacks, which can be effectively mitigated by CFI technique. However, the existing CFI technique cannot directly mitigate Spectre due to differences in the level at which control flow violations occur.

We aim to achieve control-flow integrity not only in software but also in microarchitecture. In particular, we focus on lightweight coarse-grained CFI techniques to achieve the goal. In general, existing coarse-grained CFI techniques [47], [48], [49] are built with the insight that a normal program execution has the following immutable properties of indirect branches:

- Indirect call and jump instructions always branch to an entry of a basic block or a function. In fact, function entry can be considered as an entry of basic blocks because functions are composed of basic blocks.
- Return instructions always jump to the instruction right after a `call` instruction.

Based on the properties above, coarse-grained CFI techniques impose restrictions on control-flow transfer driven by indirect branches. In order to mitigate Spectre-BTB and Spectre-RSB, these immutable properties must also be satisfied with speculative executed indirect branches.

2) OVERVIEW

The basic idea of MicroCFI is to ensure the aforementioned immutable properties for microarchitectural control-flow transfers predicted using the BTB or RSB. In this paper, we refer these branch targets that meet the immutable properties to as *valid targets* (VT). MicroCFI employs two methods to implement its control flow integrity mechanism: (1) code alignment and (2) bit masking. The first method, code alignment, re-arranges all VTs in the program so that they are equipped with address-side characteristics, i.e., the VTs have 2^n-byte-aligned addresses in the memory. By doing so, any speculative control flows must be directed to aligned

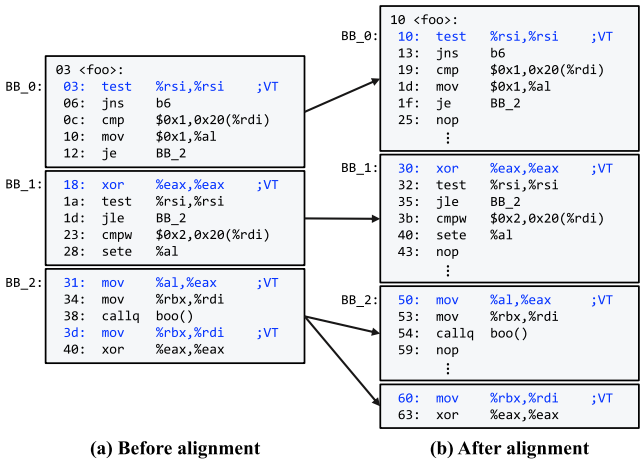


FIGURE 2. Example of 2^4-byte code alignment.

VTs. The second method, bit masking, involves adding a bit masking unit to a fetching hardware in the instruction pipeline. This unit efficiently restricts any speculative indirect branches to only target the aligned VT by performing n-bit masking operations on the predicted branch target address.

The combination of these two techniques creates a robust control flow integrity mechanism at the microarchitecture level. Specifically, an attacker is restricted to choose *valid gadgets*, i.e., gadgets of which the base addresses are aligned VTs, which are a small fraction of all possible gadgets found in the program. As a result, it becomes more challenging for attackers to find available valid gadgets and greatly increases attack complexity, effectively mitigating Spectre-BTB and Spectre-RSB.

C. CODE ALIGNMENT

MicroCFI transforms a program into one in which all VTs in the program code are aligned by 2^n bytes. Figure 2 shows an example of a program transformation with 2^4-byte code alignment. Figure 2(a) shows a program to be transformed; it has four VTs at addresses 0×03, 0×18, 0×31, and 0×3d. Figure 2(b) shows the code of the transformed program consisting of four code chunks at addresses 0×10, 0×30, 0×50, and 0×60, all of which are aligned with 2^4 bytes in the memory.

The code alignment inevitably creates empty spaces between code chunks in the transformed program. In most cases, we fill these gaps by just inserting `NOP` instructions right after the last instruction of the code chunk. However, in the case of the code chunk that ends with a `call` instruction, we have to take further consideration. That is, inserting `NOP`s right after `call` will result in persistent misspeculations in the RSB, negatively impacting the performance.

This is because the RSB behaves like a software stack. Suppose that `NOP`s are appended immediately after the `call` instruction, as shown in Figure 3(a). The execution of `callq` will push the return address, i.e., the address of the next instruction (0×59 in Figure 3(a)) to both the stack and RSB. When the subsequent return instruction is fetched, the processor will consult the RSB to predict the return address. However, as MicroCFI applies bit-masking to the predicted address (the detail is described in the next section), the RSB will consequently present the aligned address (i.e., 0×50), resulting in inconsistency with the return address stored in the stack. This will lead to misspeculation. Therefore, we have to address the code chunk ending with a `call` instruction in a different manner.

To prevent misspeculation caused by the inconsistency between the stack and RSB, we insert `NOP`s prior to the `call` instead of padding right after the instruction, as shown in Figure 3(b). This method moves `callq` to a location adjacent to the desired next instruction located at the aligned address. In this case, `callq` pushes the (aligned) return address to both the stack and the RSB. Therefore, a return address in the RSB will remain consistent with the stack even after bit masking is applied, and misspeculation will never occur in this case.

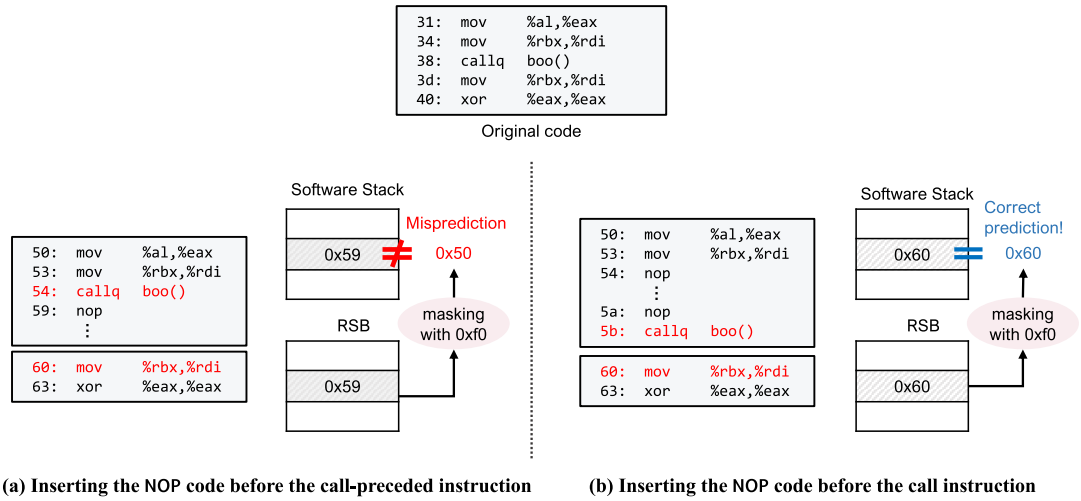
MicroCFI ensures that all VTs in the transformed program are aligned with 2^n bytes in memory. However, it should be noted that the alignment of memory addresses to 2^n bytes does not guarantee that all 2^n-byte-aligned addresses are VTs. In fact, there may exist code chunks whose base addresses are aligned but are not VTs, which creates a potential avenue for attackers to construct valid gadgets using these chunks. Even more concerning, there is a risk that valid gadgets can be constructed using code chunks whose base addresses are VTs. Therefore, the number of valid gadgets that can be constructed in a transformed program is a crucial factor in assessing the security of MicroCFI. Our experiments with practical applications demonstrate that MicroCFI significantly reduces the number of valid gadgets through code alignment, and the specific details of this analysis are presented in Section V-C.

D. BIT-MASKING ON PREDICTED ADDRESS

Code alignment is achieved by software-based program transformation. However, relying solely on this approach is inadequate for enforcing CFI and mitigating Spectre attacks, as these attacks occur during speculative execution. To restrict the branch targets of speculative execution to only aligned VTs, the microarchitecture must include specialized hardware functionality to guarantee that the BTB and RSB provide aligned predicted branch target addresses.

To enforce MicroCFI, a simple hardware mechanism is introduced at the instruction fetch stage. That is, we add a bit-masking unit that performs n-bit masking on the predicted branch target addresses provided by the BTB and RSB. Figure 4 shows the modified fetch stage. When n-bit masking is applied, only addresses aligned to 2^n-byte boundaries are fetched and utilized for speculative execution.

The bit-masking unit is an effective means of limiting the attacker's abilities without any negative impact on the execution of a program. When a program is transformed



(a) Inserting the NOP code before the call-preceded instruction (b) Inserting the NOP code before the call instruction

FIGURE 3. Examples of two alignment schemes for call-preceded instructions.

with code-alignment, only aligned addresses are kept in the BTB and RSB, as all indirect branch targets in the program are aligned in memory. Consequently, bit masking on the predicted address does not affect the branch prediction performance in the transformed program. In contrast, bit masking impedes the attacker's abilities. To carry out Spectre-BTB (or Spectre-RSB), the attacker must corrupt the BTB (RSB) with their chosen target addresses. If the attacker selects a non-aligned address as a target, the bit masking will result in an aligned but incorrectly predicted address. Therefore, the attack will fail, and the victim will potentially execute arbitrary code instead of the desired gadget selected by the attacker. Additionally, the executed code is treated as a misprediction and is rolled back, so it has no effect on program execution.

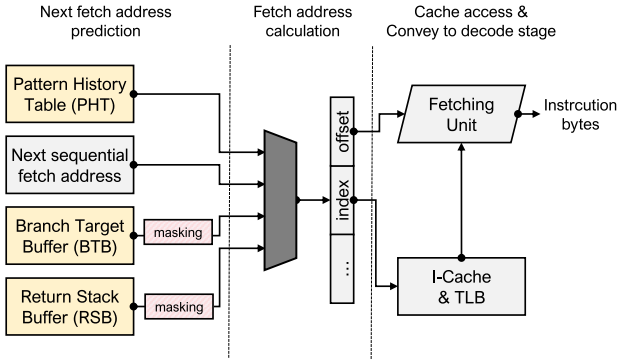


FIGURE 4. A pipeline that performs bit-masking on the predicted branch target address.

V. IMPLEMENTATION AND EVALUATION

We implement a prototype of MicroCFI to evaluate its effectiveness in terms of the performance and security. In this section, we present our implementation of MicroCFI in detail and the evaluation results.

A. IMPLEMENTATION

MicroCFI has two requirements of modification in both software and hardware. First, all VTs in a program must be aligned to 2^n bytes in memory. The code alignment can be implemented with the aid of the LLVM compiler [52], a reusable and modular compilation framework. In particular, we extend the compiler to transform a program such that all the VTs in the program are aligned. For this purpose, we use alignment directives for assembly files generated in a compilation process. The extended LLVM compiler automatically inserts `NOP` bytes in the empty spaces introduced by the code alignment. For the specific case of a `call` instruction, as described in Section IV-C, the compiler inserts `NOP`s in front of the `call` instruction to avoid persistent misspeculation.

The second requirement of MicroCFI is that branch target addresses predicted via BTB or RSB has to be bit-masked prior to speculative execution. Therefore, bit-masking must be included in the branch prediction process, which occurs in the fetch stage of an instruction pipeline. Unfortunately, CPU vendors neither allow access to the internals of CPU microarchitectures nor disclose any implementation details. Therefore, we decide to implement MicroCFI based on a processor simulator, instead of using real microprocessors. In particular, we use the MARSSx86 simulator [53], which supports cycle-accurate full-system simulation of an x86-64 architecture. MARSSx86 provides a simulation environment with detailed pipeline models including branch prediction based on PTLsim [54]. We implement MicroCFI by modifying the branch prediction process implemented in MARSSx86. The modified MARSSx86 performs a bit-masking operation on the predicted target address by referring to BTB and RSB.

There is a limitation to the current implementation of our MicroCFI prototype. Because MicroCFI modifies CPU internals to perform bit-masking on predicted addresses, all

TABLE 3. CPU configuration for the MARSSx86 simulator.

Parameter	Configuration
CPU	SkyLake
L1I & L1D	32KB, 8-way, 64B line, 4 cycles, private per core
L2	256KB, 8-way, 64B line, 12 cycles, private per core
L3	8M, 16-way, 64B line, 42 cycles, shared by all cores
ROB	224-entry Reorder Buffer
iTLB	4-entry instruction Translation Lookaside Buffer
dTLB	64-entry data Translation Lookaside Buffer
RSB	16-entry Return Stack Buffer
FQ	48-entry Fetch Queue
IQ	64-entry Issue Queue
LDQ	72-entry Load Queue
STQ	56-entry Store Queue

programs running in the MicroCFI-protected system must have their VTs aligned with 2^n bytes to avoid misspeculations. The programs include an operating system, such as a Linux kernel, and C/C++ standard libraries (e.g., glibc and libstdc++), which have built environments dedicated to GCC toolchains [55]. As we use an LLVM compiler, the current implementation of MicroCFI does not support the transformation of these types of programs. We would like to emphasize that this limitation is not due to the inherent properties of MicroCFI, but due to implementation issues.

B. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the MicroCFI through experiments. The experiments are conducted using the MARSSx86 simulator [53] running on Ubuntu 18.04 Linux. For the evaluation, MARSSx86 is configured to simulate an Intel Skylake processor [56]. Details of the configuration are presented in Table 3.

We run the SPEC CPU 2017 benchmark on the simulator to measure the performance overhead of MicroCFI. In particular, we choose 14 benchmark programs among them, which are written in C/C++ language, and measure the runtime and memory overhead while running these benchmarks. To evaluate the performance of MicroCFI with respect to the alignment size, experiments are conducted by varying the alignment from 2^4 to 2^6 bytes.

1) RUNTIME OVERHEAD

We perform an experiment to evaluate the runtime overhead. The overall runtime overhead, RO_T, is represented by the following equation:

$$RO_T = RO_M + RO_A + \alpha. \tag{1}$$

RO_M represents the runtime overhead directly caused by a bit-masking operation, which is the essential operation of MicroCFI to enforce indirect branches to jump to aligned VTs. To evaluate the RO_M, we measure the average number of executed instructions per clock cycle for each benchmark following a common approach to simulation-based performance evaluation [32], [57]. MARSSx86 provides an instruction per cycle (IPC) metric for this purpose.

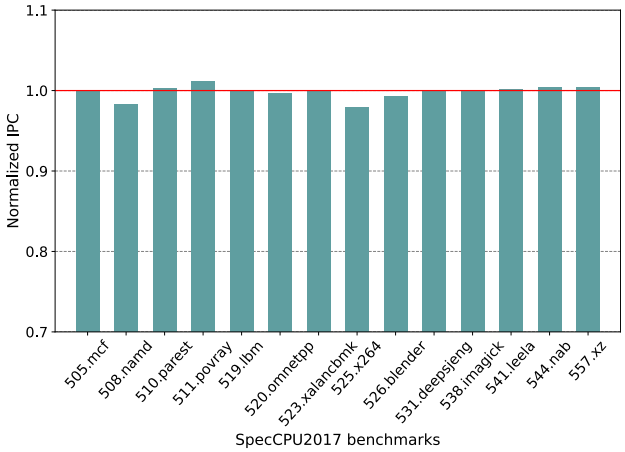


FIGURE 5. Normalized IPC of SPEC benchmarks.

RO_A in Equation (1) indicates the runtime overhead directly induced by the code alignment. As the code alignment introduces dummy `NOP` instructions between aligned code chunks, additional execution overhead is inevitable, which may affect IPC because of more misses in an L1-I cache and increased instruction stream. Hence, to evaluate RO_A, we use a metric that measures the increase in execution time of the benchmark. We perform this experiment in a real host environment due to the extremely low execution speed of the simulator.

In addition, there may be unknown but negligible factors that also affect the runtime overhead (α), which we did not consider in our evaluation. Therefore, we focus on RO_A and RO_M to estimate the overall runtime overhead of MicroCFI.

a: OVERHEAD DUE TO BIT-MASKING

First, we measure the runtime overhead RO_M induced by the bit-masking operation. For accurate measurements, we use the IPC metric offered by the MARSSx86 simulator while running the SPEC benchmarks.

The experimental results are presented in Figure 5, which shows the measured IPC values for all benchmarks. These IPCs are obtained from the execution of the first billion user-mode instructions after launching the benchmark program. The IPC in the graph is normalized; it is the ratio of the measured IPC of the MicroCFI-enabled program to that of the same program without MicroCFI. Hence, a value greater than 1 implies better execution performance when MicroCFI is applied; otherwise, it indicates a lower execution performance. We observe that the normalized IPCs values are close to one for all MicroCFI-enabled benchmark programs. The results indicate that the bit-masking operation results in a negligible overhead during program execution.

For more precise analysis, we count the number of bit-masking operations performed during the execution of each benchmark. Table 4 presents the average number of executed masking operations for 10,000 user mode instructions for each benchmark. Although the number varies

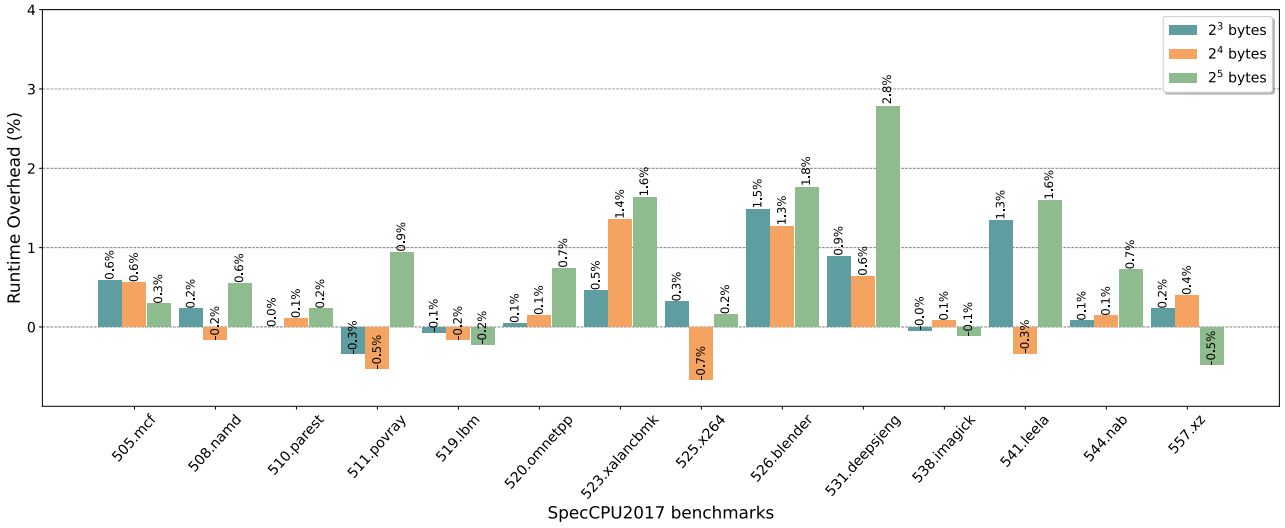


FIGURE 6. Runtime overhead with code alignment.

TABLE 4. The number of maskings per 10,000 instructions in SPEC CPU 2017 benchmarks.

Benchmark	No. maskings per 10K	Benchmark	No. maskings per 10K
505.mcf	303.5	525.x264	94.5
508.namd	1.0	526.blender	54.2
510.parest	40.9	531.deepsjeng	86.9
511.povray	362.4	538.imagick	29.1
519.lbm	0.2	541.leela	227.0
520.omnetpp	219.1	544.nab	10.8
523.xalancbmk	305.5	557.xz	0.2

significantly ranging from 0.2 to 362.4 in all benchmark programs, we observe that there is no correlation between the number of bit-masking operations and the normalized IPC. From these results, we conclude that the runtime overhead caused by the bit-masking operation is negligible.

b: OVERHEAD DUE TO CODE ALIGNMENT

To obtain an accurate measurement of the overhead RO_A, we conduct an experiment where only code alignment is employed in MicroCFI, and bit-masking operations are excluded. As these operations aim to restrict malicious transient control flows, running experiments without bit-masking has no impact on the execution behavior of benign SPEC benchmark programs.

To evaluate the runtime overhead, we calculate the geometric mean of the execution times from 150 runs for each benchmark. The experimental results are depicted in Figure 6, where the overhead RO_A is presented as the percentage ratio of the execution time of a benchmark with MicroCFI to that of the same benchmark without MicroCFI. For 2^3-byte alignment, the average overhead is 0.37%, while it is 0.20% for 2^4 bytes and 0.76% for 2^5 bytes.

The results reveal that for 2^3-byte alignment, the best case is recorded at -0.34% for the **511.povray** benchmark, while the worst case is 1.48% for **526.blender**. For 2^4 bytes, the best case is -0.66% for **525.x264**, while the worst case is 1.35% for **523.xalancebmk**. For 2^5 bytes, the best case is -0.47% for the **557.lbm**, while the worst case is 2.78% for the **531.deepsjeng**, which is the same as for 2^4 bytes.

Most of the benchmarks show negligible overhead for all alignment sizes, but interestingly, the overhead does not always increase proportionally with the alignment size. In some benchmarks, such as **508.namd**, **511.povray**, **519.lbm**, **525.x264**, **538.imagick**, **541.leela**, and **557.xz**, the overhead is negative, indicating that these benchmarks perform better when MicroCFI is applied. We postulate that code alignment leads to unexpected performance improvement. The arrangement of code inherently influences cache and memory locality. Thus, alterations in code layout induced by code alignment can increase the cache hit rate for instructions, which we estimate is the underlying reason for performance enhancements observed in some benchmark.

Typically, when the `NOP` instruction is executed, it takes a certain amount of time. As the alignment size grows, more `NOP` instructions are added, resulting in greater overhead. Surprisingly, our findings contradict this pattern. We believe that this is due to the influence of code alignment on the cache state during program execution. The aligned code in the cache may affect the cache hit/miss rate, eventually affecting the program's performance.

2) MEMORY OVERHEAD

We measure the memory overhead of MicroCFI using the SPEC benchmarks. The overhead is obviously caused by the code alignment, as it actually increases the program size. Figure 7 shows an increase in the text sections of the SPEC benchmark programs. From the experiment, we observe that

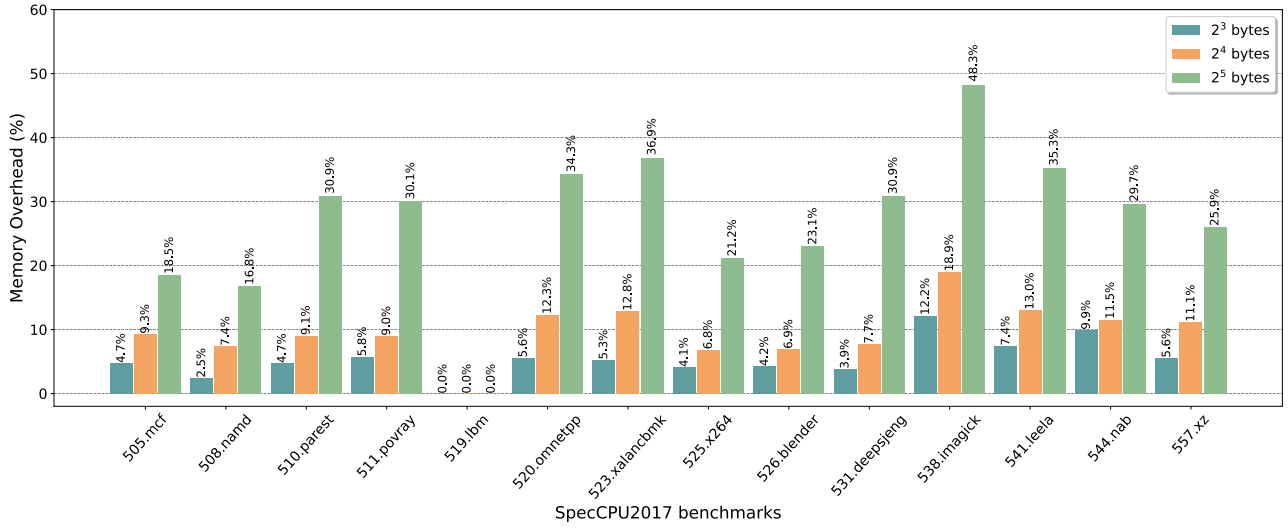


FIGURE 7. Memory overhead with code alignment.

TABLE 5. Number of alignments in SPEC CPU 2017 benchmarks.

Benchmark	No. alignment per 10K	Benchmark	No. alignment per 10K
505.mcf	13.6	525.x264	16.4
508.namd	9.1	526.blender	16.7
510.parest	23.4	531.deepsjeng	20.9
511.povray	22.4	538.imagick	27.8
519.lbm	3.9	541.leela	23.2
520.omnetpp	24.8	544.nab	19.0
523.xalancbmk	26.0	557.xz	15.8

the program size increases on average by 5.4%, 9.7%, and 27.3% for 2^3, 2^4, and 2^5-byte alignments, respectively. The memory overhead increases as the alignment size increases. This is because the larger the alignment size, the more `NOP` the instructions inserted into the program.

Table 5 presents the average number of alignments per 1 KB of the text section of each benchmark program. The table shows that the number of alignments is proportional to the memory overhead, as presented in Figure 7. For example, the 519.lbm benchmark, which has the smallest alignments, has the lowest memory overhead for all alignment sizes. Similarly, the 538.imagick benchmark that recorded the highest memory overhead for all alignment sizes also had the highest average number of alignments.

The correlation between the number of alignments and memory overhead is evident as the number of inserted `NOP` instructions increases according to the number of alignments in the program. Consequently, it can be observed that the memory overhead of MicroCFI depends on the alignment size and number of alignments.

Choosing smaller byte alignment can decrease this overhead, but it might simultaneously increase the pool of available gadgets. Alternatively, memory overhead can be reduced by selecting forward or backward-edge to protect, like most CFIs are classified into two categories: forward-edge CFI and backward-edge CFI. Protecting only one edge reduces the amount of code that requires alignment, which alleviates memory overhead. Nevertheless, this selective protection exposes a potential vulnerability at the unprotected edge, compromising overall security. Thus, MicroCFI has a trade-off between memory overhead and security.

The proposed technique imposes memory cost to enhance program security. However, the cost of memory has been declining for a long time, and this memory cost is becoming more tolerable than the runtime cost. Hence, MicroCFI has an acceptable memory overhead in practical applications.

C. SECURITY EVALUATION

The security goal of MicroCFI is to make it extremely challenging for attackers to deliver Spectre attacks by reducing the number of available Spectre gadgets. Therefore, the security of MicroCFI can be evaluated by the number of reduced Spectre gadgets as a result of applying MicroCFI to a target program. From this perspective, we measure the extent to which the number of available Spectre gadgets is reduced by MicroCFI.

To count the number of Spectre gadgets, we utilize SpecFication [58], a tool designed to identify Spectre gadgets in a program's binary. In MicroCFI-protected programs, only *valid gadgets* whose base addresses are aligned to 2^n bytes in memory can be used to construct Spectre gadgets. The number of valid gadgets in the programs varies according to the alignment size n. Hence, we measure the number of valid gadgets for the program with 2^3, 2^4, and 2^5-byte code alignments.

Spectre attacks can utilize ROP techniques that chain multiple gadgets together to create an end-to-end exploit. SpecFication supports identifying small-sized gadgets that

TABLE 6. Number of valid gadgets.

	libcrypto			
	Shifting	*Loading from*	*Loading to*	*Total*
Original	287	15052	20783	36122
2^3 **bytes**	28 (90.2%)	2246 (85.0%)	3004 (85.5%)	5278 (85.3%)
2^4 **bytes**	12 (95.8%)	1149 (92.3%)	1555 (92.5%)	2716 (92.4%)
2^5 **bytes**	2 (99.3%)	234 (98.4%)	328 (98.4%)	564 (98.4%)

	libssl			
	Shifting	*Loading from*	*Loading to*	*Total*
Original	30	3399	4369	7798
2^3 **bytes**	3 (90.0%)	516 (84.7%)	637 (85.4%)	1157 (85.1%)
2^4 **bytes**	2 (93.3%)	280 (91.7%)	337 (92.2%)	619 (92.0%)
2^5 **bytes**	2 (93.3%)	148 (95.6%)	169 (96.1%)	319 (95.9%)

	Nginx			
	Shifting	*Loading from*	*Loading to*	*Total*
Original	83	2743	3618	6444
2^3 **bytes**	9 (89.1%)	366 (86.6%)	478 (86.7%)	853 (86.7%)
2^4 **bytes**	3 (96.3%)	213 (92.2%)	285 (92.1%)	501 (92.2%)
2^5 **bytes**	2 (97.5%)	147 (94.6%)	188 (94.8%)	337 (94.7%)

	mod_ssl			
	Shifting	*Loading from*	*Loading to*	*Total*
Original	3	116	173	292
2^3 **bytes**	1 (66.6%)	22 (81.0%)	25 (85.5%)	48 (83.5%)
2^4 **bytes**	0 (100%)	18 (84.4%)	19 (89.0%)	37 (87.3%)
2^5 **bytes**	0 (100%)	13 (88.7%)	15 (91.3%)	28 (90.4%)

	mod_proxy			
	Shifting	*Loading from*	*Loading to*	*Total*
Original	5	286	359	650
2^3 **bytes**	0 (100%)	11 (96.1%)	19 (94.7%)	30 (95.3%)
2^4 **bytes**	0 (100%)	9 (96.8%)	18 (94.9%)	27 (95.8%)
2^5 **bytes**	0 (100%)	5 (98.2%)	9 (97.4%)	14 (97.8%)

can be chained together. The tool classifies gadgets based on their functions as either *Shifting*, *Loading from*, or *Loading to*. By chaining gadgets from different categories, a functional payload for the Spectre attack can be constructed.

To evaluate security of MicroCFI, we slightly modify the tool so that it looks only for valid gadgets. For our experiment, we use some popular open-source software, such as OpenSSL(v1.1.1d) library, Nginx(v1.22), and Apache HTTP server(v2.4.53), as target programs. In particular, we use two shared libraries, `libcrypto1.1.so` and `libssl.so` in OpenSSL, and two modules, `mod_ssl`, and `mod_proxy` in the Apache HTTP server to measure the number of valid gadgets. We perform the experiment on Ubuntu 18.04 Linux, which is the same as the previous experiment, and the target program is compiled with the LLVM 3.4 compiler modified for MicroCFI.

Table 6 shows the number of valid gadgets found in the MicroCFI-protected binaries and the corresponding percentage of gadgets in the original program. Based on our analysis, we find that the count of valid gadgets decreases by more than 80% in all programs with 2^3-byte alignment and by over 90% in most programs with 2^4-byte alignment. Additionally, the findings suggest that the reduction in the number of gadgets increases with the increase in the alignment size.

In the case of `mod_ssl`, the smallest decrease in the number of valid gadgets occurs with 2^3- and 2^4-byte alignments. However, we would like to emphasize that there are no longer *shifting* gadgets in the transformed `mod_ssl` binary. As the *shifting* gadget is essential for constructing Spectre gadgets [58], it is extremely challenging or infeasible for an attacker to deliver a successful Spectre attack without using the *shifting* gadget.

Although MicroCFI can significantly decrease the amount of valid gadgets in a program, it does not completely eliminate the possibility of the existence of valid gadgets. To evaluate the risk of potential vulnerabilities, we manually inspect the remaining valid gadgets, specifically focusing on *shifting* gadgets that are crucial for an attack. Our analysis reveals that, on average, 42% of the remaining *shifting* gadgets are false positives, meaning that they are not genuinely valid gadgets. In particular, when using 2^5-byte alignment, all programs except Nginx have no valid gadgets remaining, while Nginx only has one confirmed valid gadget. With 2^4-byte alignment, Nginx and `libcrypto1.1.so` has 2 and 10 valid gadgets, respectively. Finally, on the 2^3-byte alignment condition, we have confirmed 7 valid gadgets in Nginx, 10 in `libcrypto1.1.so`, and 1 in `libssl.so`. Consequently, the number of valid gadgets available to attackers is lower than what is displayed in Table 6. It's worth mentioning that the figures presented in Table 6 represent the total number of valid gadgets across all registers. Thus, if attackers have limited access to registers, the number of valid gadgets they can use is further restricted, making it more challenging to construct a Spectre gadget chain.

In the unlikely scenario that attackers can still create an attack using the remaining valid gadgets, we can thwart such attacks by incorporating the `fence` instruction into the remaining gadgets. It is unnecessary to insert the `fence` instruction into every gadget; inserting it into even one gadget category, such as *shifting*, can substantially hinder the construction of a gadget chain. This method may introduce some overhead, but it provides more comprehensive protection. Furthermore, unlike other existing techniques [28], [29], [30], this approach incurs minimal overhead since it restricts speculative execution solely for the remaining valid gadgets. We performed experiments with Nginx and its benchmarking tool [59] to evaluate the impact of the `fence` instruction. The results of the experiments revealed that the `fence` command had a negligible effect across all alignment configurations.

VI. CONCLUSION

In this paper, we proposed MicroCFI, a novel hardware/software co-design solution to counter Spectre-BTB and Spectre-RSB attacks. The proposed technique enforces

strict control on the speculative control flow by implementing a CFI mechanism within the microarchitectural context. This is achieved by utilizing a code alignment and bit-masking technique, where forward/backward indirect branch targets are transformed into VT that are aligned to memory addresses of 2^n bytes. In an instruction fetch stage, a bit-masking unit applies bit-masking operations to predicted target addresses from BTB and RSB, ensuring they are always aligned to VTs. This restriction significantly reduces the number of available Spectre gadgets for attackers, making successful attacks much more difficult.

The security evaluation conducted on real-world applications like OpenSSL, Nginx, and Apache HTTP server revealed that MicroCFI with 2^4-byte alignment reduces the number of available gadgets by over 90%, making it a highly effective solution to mitigate Spectre attacks. The experiments with SPEC CPU 2017 benchmarks showed that MicroCFI provides reasonable performance overhead with minimal hardware modification while offering protection against Spectre-BTB and Spectre-RSB attacks.

In future work, we will improve MicroCFI compatibility. MicroCFI forces all indirect branches to speculatively execute aligned addresses at the hardware level, requiring the rebuilding of all programs run on the processor—an aspect that may be perceived as a limitation. This constraint is also intrinsic to the CFI mechanism. Consequently, our focus will involve studying ways to improve MicroCFI compatibility to address these limitations.

REFERENCES

[1] 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 *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[2] 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 *Proc. 27th USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 973–990.

[3] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 753–768.

[4] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 88–105.

[5] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Proc. Eur. Symp. Res. Comput. Secur. (ESORICS)*. Cham, Switzerland: Springer, 2019, pp. 279–299.

[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 785–800.

[7] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," in *Proc. 30th USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2021, pp. 1415–1432.

[8] H. Ponce-de-Leon and J. Kinder, "Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 235–248.

[9] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham, Switzerland: Springer, 2016, pp. 279–299.

[10] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, Jan. 2010.

[11] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732.

[12] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *Proc. 12th USENIX Workshop Offensive Technol. (WOOT)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 1–12.

[13] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O'Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, "Spectre declassified: Reading from the right place at the wrong time," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 1753–1770.

[14] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM pointer authentication with speculative execution," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 685–698.

[15] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks," in *Proc. 31st USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2022, pp. 971–988.

[16] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for new speculative attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 681–698.

[17] A.-T. Le, T.-T. Hoang, B.-A. Dao, A. Tsukamoto, K. Suzaki, and C.-K. Pham, "A cross-process Spectre attack via cache on RISC-V processor with trusted execution environment," *Comput. Electr. Eng.*, vol. 105, Jan. 2023, Art. no. 108546.

[18] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against Spectre attacks via program analysis," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2504–2519, Nov. 2021.

[19] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "SPECUSYM: Speculative symbolic execution for cache timing leak detection," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 1235–1247.

[20] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1–19.

[21] 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," 2019, *arXiv:1902.05178*.

[22] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2020, pp. 1481–1498.

[23] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 3, pp. 1–31, Jul. 2020.

[24] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely, "The code that never ran: Modeling attacks on speculative evaluation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1238–1255.

[25] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 802–815.

[26] B. A. Shivakumar, G. Barthe, B. Gregoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean, "Typing high-speed cryptography against Spectre v1," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 1592–1609.

[27] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for software Spectre defenses," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 666–680.

[28] Intel. (2018). *Retpoline: A Branch Target Injection Mitigation*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf

[29] AMD. (2022). *Software Techniques for Managing Speculation on AMD Processors*. [Online]. Available: https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf

[30] Intel. (2018). *Speculative Execution Side Channel Mitigations*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf

[31] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *Proc. 31st USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2022, pp. 3825–3842.

[32] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SpecCFI: Mitigating Spectre attacks using CFI informed speculation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 39–53.

[33] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "ConTExT: Leakage-free transient execution," 2019, *arXiv:1905.09100*.

[34] SPEC. (2017). *SPEC CPU 2017*. [Online]. Available: https://www.spec.org/cpu2017

[35] Z. Shen, J. Zhou, D. Ojha, and J. Criswell, "Restricting control flow during speculative execution with venkman," 2019, *arXiv:1903.10651*.

[36] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 340–353.

[37] M. T. Yourst, "PTLsim: A cycle accurate full system x86–64 microarchitectural simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2007, pp. 23–34.

[38] B. Niu and G. Tan, "Modular control-flow integrity," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, Jun. 2014, pp. 577–587.

[39] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009.

[40] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Mar. 2016, pp. 179–194.

[41] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, 2015, pp. 927–940.

[42] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 927–940.

[43] H. Jang, M. C. Park, and D. H. Lee, "IBV-CFI: Efficient fine-grained control-flow integrity preserving CFG precision," *Comput. Secur.*, vol. 94, Jul. 2020, Art. no. 101828.

[44] D. Bounov, R. G. Kıcı, and S. Lerner, "Protecting C++ dynamic dispatch through VTable interleaving," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[45] C. She, L. Chen, and G. Shi, "TFCFI: Transparent forward fine-grained control-flow integrity protection," in *Proc. IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2022, pp. 407–414.

[46] M. C. Park and D. H. Lee, "BGCFI: Efficient verification in fine-grained control-flow integrity based on bipartite graph," *IEEE Access*, vol. 11, pp. 4291–4305, 2023.

[47] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 292–307.

[48] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 199–210.

[49] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 1470–1486.

[50] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Eenforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 941–955.

[51] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *Engineering Secure Software and Systems*. Cham, Switzerland: Springer, 2017, pp. 161–176.

[52] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2004, pp. 75–86.

[53] A. Patela, F. Afram, and K. Ghose, "Marss-x86: A QEMU-based microarchitectural and systems simulator for x86 multicore processors," in *Proc. 1st Int. QEMU Users' Forum*, 2011, pp. 29–30.

[54] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 559–573.

[55] R. Love, *Linux Kernel Development*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2003.

[56] Intel. (2012). *Intel64 and IA-32 Architectures Optimization Reference Manual*. [Online]. Available: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[57] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for NVM," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 178–190.

[58] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *Proc. 23rd Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*. Berkeley, CA, USA: USENIX Association, 2020, pp. 1–16.

[59] Apache. (2012). *Apache HTTP Server Benchmarking Tool*. [Online]. Available: https://httpd.apache.org/docs/2.4/en/programs/ab.html

HYEREAN JANG received the B.S. degree in mathematics from Chungnam National University, Daejeon, South Korea, in 2017, and the M.S. degree in cybersecurity from Korea University, Seoul, in 2020, where she is currently pursuing the Ph.D. degree in cybersecurity. Her research interests include software security, system security, and CPU micro-architectural security.

YOUNGJOO SHIN received the B.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 2006, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2008 and 2014, respectively. From 2008 to 2017, he was with the National Security Research Institute (NSR), Daejeon, as a Senior Researcher. From 2017 to 2020, he was with Kwangwoon University, Seoul, as an Assistant Professor. From 2020 to 2021, he was with Korea University as an Assistant Professor, where he is currently an Associate Professor with the School of Cybersecurity. His research interests include system and network security, CPU micro-architectural security, cloud computing security, and vulnerability analysis on embedded systems.