Efficient Context-Sensitive CFI Enforcement through a Hardware Monitor

Sadullah Canakci, Leila Delshadtehrani, Boyou Zhou,
Ajay Joshi, and Manuel Egele

Boston University, Boston, MA, 02215, USA
{scanakci, delshad, bobzhou, joshi, megele}@bu.edu

Abstract. Recent works on Control-Flow Integrity (CFI) have mainly focused on Context-Sensitive CFI policies to provide higher security guarantees. They utilize a debugging hardware feature in modern Intel CPUs, *Processor Trace* (PT), to efficiently collect runtime contextual information. These PT-based CFI mechanisms offload the processing of the collected PT trace and CFI enforcement onto idle cores. However, a processor does not always have idle cores due to the commonly-used multi-threaded applications such as web browsers. In fact, dedicating one or more cores for CFI enforcement reduces the number of available cores for running user programs. Our evaluation with a state-of-the-art CFI mechanism (μCFI) shows that the performance overhead of a CFI mechanism can substantially increase (up to 652% on a single-core processor) when there is no idle core for CFI enforcement. To improve the performance of μCFI, we propose to leverage a hardware monitor that unlike PT does not incur trace processing overhead. We show that the hardware monitor can be used to efficiently collect program traces (<1% overhead) in their original forms and apply μCFI. We prototype the hardware-monitor based μCFI on a single-core RISC-V processor. Our analysis show that hardware-monitor based μCFI incurs, on average, 43% (up to 277%) performance overhead.

Keywords: CFI, Hardware Monitor, Processor Trace.

1 Introduction and Motivation

With the introduction of Data Execution Prevention (DEP) [8], attackers changed their focus from code injection to code-reuse attacks such as Return-Oriented Programming (ROP) [35] and Jump-Oriented Programming (JOP) [12]. Control-Flow Integrity (CFI) [7] is a security defense that aims to prevent these attacks by drastically reducing the allowed code targets for each *Indirect Control-Flow Transfer*s (ICTs). Most CFI mechanisms consist of two phases [10]: an analysis phase and an enforcement phase. The analysis phase generates a statically constructed Control-Flow Graph (CFG), which approximates the allowed code targets for each control-flow transfer. The enforcement phase ensures that all the executed control-flow transfers follow valid paths in the CFG.

The success of a CFI mechanism mainly relies on two metrics: performance and security. Recent works focus on *context-sensitive CFI* [20, 24, 27, 28] to provide stronger security guarantees than traditional context-insensitive CFI [42, 43]. Context-sensitive CFI mechanisms refine the CFG with additional contextual information. Unfortunately, introducing contextual information requires additional processing time during the enforcement phase; thus, it increases the overall performance overhead of the CFI mechanisms [20, 21].

For efficient context-sensitive CFI enforcement, researchers have repurposed an already deployed hardware feature in modern Intel CPUs, *Processor Trace* (PT) [34]. PT has been designed for **offline** debugging and failure diagnosis by capturing runtime target and timing information of ICT instructions (`ret` and indirect `jmp/call`) [32]. Although several works [15, 26, 38] used PT in its intended direction, recent works leveraged PT to efficiently collect contextual information for **online** CFI enforcement [20, 21, 24, 29].

Using PT for CFI enforcement is practical since it already exists in the commodity hardware. However, PT is not an optimal hardware feature for CFI enforcement. Although PT efficiently collects traces ($< 3\%$ overhead [29]) in the form of encoded *packets* at the hardware level, the decoding of these packets (*trace processing*) performed with a software-level decoder is significantly slower than the trace collection [21, 24, 29]. Unfortunately, any PT-based CFI mechanism requires this inefficient trace processing prior to validating ICT targets at enforcement phase. To avoid the additional performance overhead, existing PT-based CFI mechanisms [20, 21, 24, 29] offload the trace processing and ICT validation onto idle cores[1]. However, commonly used applications (such as web browsers/servers and games) are multi-threaded; thus, the processor will not always have idle cores available for CFI enforcement. In fact, dedicating one or more cores for CFI enforcement reduces the number of available cores for running user applications.

In Figure 1, we show the performance impact of the number of cores on CFI enforcement by using a state-of-the-art PT-based approach (μCFI [24]) for SPEC2006 benchmark suite [23]. The details of the experimental setup are provided in Section 4. We evaluate PT-based μCFI using four configurations: single-core (1-Core), two-core (2-Core), three-core (3-Core), and eight-core (8-Core). 8-Core and 3-Core configurations incur similar performance overheads since μCFI enforcement uses two additional idle cores (one for ICT validation and one for trace processing). 2-Core and 1-Core results clearly show that PT-based μCFI significantly impacts the performance of benchmarks if the processor does not have any idle cores. On average, the performance overhead of μCFI increases from 23% to 42% from 3-Core to 1-Core. Note that the benchmarks generating more packets show higher degradation in performance since trace processing requires more CPU time for these benchmarks. In the worst case, μCFI incurs 652% overhead on 1-Core for `h264ref`, which is significantly higher than 3-Core overhead (372%).

[1] Throughout the paper, a "core" refers to a logical core.

Based on the insights gained from our measurements, efficient context-sensitive CFI policies should be enforced through a hardware feature which efficiently collects contextual information without requiring trace processing. We show that a programmable hardware monitor (PHMon [19]) with fine-grained configuration capabilities can be used to efficiently implement a state-of-the-art context-sensitive CFI mechanism (μCFI [24]) to defend against forward-edge attacks. PHMon incurs only 1% trace collection overhead when collecting the contextual information. Moreover, PHMon does not require trace processing during CFI enforcement as opposed to PT. In addition, we integrate a hardware-based shadow stack [19] into PHMon-based μCFI to protect backward-edges as well.

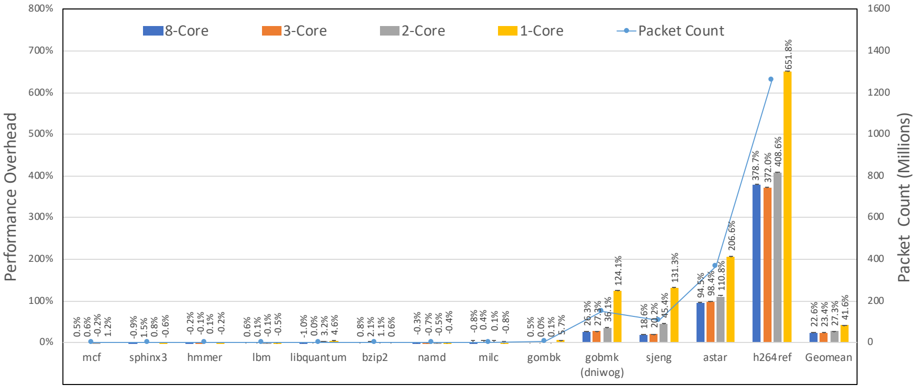

Fig. 1. Performance overhead (left y-axis) of PT-based μCFI for varying number of cores and the packet count (right y-axis) for various SPEC2006 benchmarks.[2]

.

To evaluate our work, we implement a prototype of PHMon-based μCFI interfaced with the RISC-V Rocket core [9] on an FPGA. We choose the RISC-V Rocket core since its open-source nature allows us to evaluate our mechanism using an actual implementation rather than merely a simulation. In summary, we make the following contributions:

- We show that the performance impact of the trace processing on CFI becomes substantial if a processor does not have idle cores dedicated to software-level decoding. According to our measurements, a state-of-the-art CFI mechanism (μCFI) incurs up to 652% overhead on a single-core processor.
- Based on the insights gained from our measurements, we propose to implement μCFI through a hardware monitor (PHMon [19]), which unlike PT does not incur trace processing overhead.
- We evaluate PHMon-based μCFI on a single-core RISC-V processor. We demonstrate that PHMon can efficiently collect traces in their original forms

[2] The depicted performance overhead of some of the benchmarks differs from those reported in the original work [24]. Section 4.1 provides an explanation for this performance variation.

with only 1% trace collection overhead on average. PHMon-based μCFI incurs 43% performance overhead, on average, to secure forward-edges.
- We show that PHMon-based μCFI is compatible with backward-edge CFI solutions by integrating a shadow stack based on a prior work [19]. Integrating shadow stack minimally affects the performance overhead (<1% additional overhead) and allows us to secure both forward and backward edges.

The rest of the paper is organized as follows. Section 2 provides background. Section 3 describes our design and implementation. Section 4 provides our evaluation. We discuss our implementation choices in Section 5 and present related work in Section 6. Finally, Section 7 concludes our work.

2 Background

In this section, we provide the background on Intel PT [32], PHMon [19], and μCFI [24].

2.1 Intel PT

PT is a debugging hardware feature in modern Intel CPUs [34]. PT collects Change of Flow Instructions (CoFIs) that cannot be derived statically. Specifically, PT generates three types of packets while encoding the CoFIs: (1) TNT packets to record 1-bit taken or non-taken information for each conditional branch (i.e., `jcc`), (2) TIP packets to record the target addresses of indirect branches (i.e., indirect `jmp/call` and `ret`), and (3) FUP packets for the source addresses of signals and interrupts. PT uses an efficient encoding mechanism while collecting the traces of a program. A software decoder can reconstruct the control-flow of the program using the program binary and the PT packets recorded during the execution. To reduce the number of generated packets, PT can be configured to specify the address range, privilege level, and CR3 (page table pointer) value to be monitored.

2.2 PHMon

PHMon [19] is a parallel decoupled monitor interfaced with the RISC-V Rocket processor [9] via Rocket Custom Coprocessor (RoCC). A user can configure PHMon through its software API and monitor the execution of processes. In Figure 2, we present a simplified overview of PHMon. As the processor executes instructions, PHMon receives the architectural state of the processor for the monitored program in the form of a commit log from the writeback stage of the pipeline. The commit log includes the instruction (`inst`), the PC (`pc_src`), the next PC (`pc_dst`), memory/register address used in the instruction (`addr`), and the data accessed by the instruction (`data`). Note that unlike PT, PHMon collects these fields in their original forms and does not require software-decoding.

The incoming commit log is then provided to the Matching Units (MUs). Each MU applies a set of distinct monitoring rules defined via the software interface of PHMon. The MU checks the commit log to detect the matches based on these rules. For instance, a user can set an MU to detect specific instructions (e.g., `ret`, `jalr`) or instructions at specific PC values. Upon detecting a match, an MU sends a matching packet to the Action Unit (AU). The AU consists of a queue, Config Units (CFUs), an Arithmetic and Logical Unit (ALU), a local Register File, and a Control Unit. The AU stores the incoming matching packets in the queue. Each MU is associated with a CFU, where the user-defined instructions are stored for the corresponding match. The AU executes these user-defined instructions through either hardware operations (i.e., ALU or memory read/write operations) or an interrupt handled by the Operating System (OS) running on the RISC-V core.

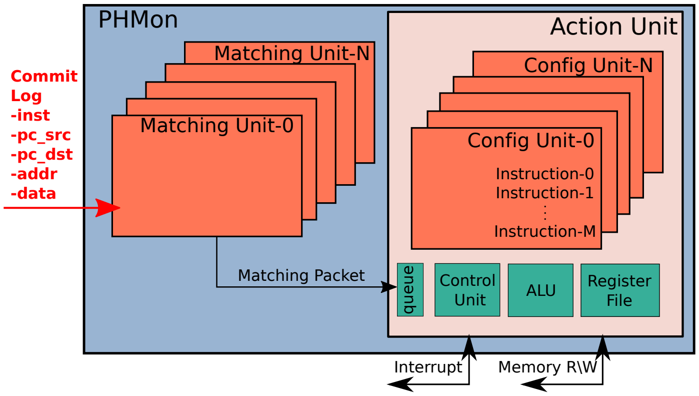

Fig. 2. A simplified overview of the PHMon [19] architecture: PHMon receives a commit log from a processor. It processes the commit log based on the user-defined rules and performs the follow-up operations such as an interrupt.

2.3 μCFI

Although context-sensitive CFI policies significantly reduce the allowed code targets for each ICT, most of them [20, 27, 40] are unable to provide a unique valid target for each ICT. As an example, we provide a code snippet (inspired by the original work [24]) in Listing 1.1. In this example, the value of the function pointer (`func_ptr`) is specified by a variable `uid` that indexes into the array `func_ptr_arr`. Since the index value (`uid`) is non-constant and resolves at run-time, most context-sensitive CFI policies identify all array elements (`A`, `B`, and `C`) as valid targets. On the contrary, μCFI [24] (a state-of-the-art context-sensitive CFI) ensures that each ICT instruction has one Unique Code Target (UCT) at each step of the program execution. μCFI achieves the UCT property by identifying *constraining data* (c-data) from the program source code and using c-data as context when enforcing CFI. c-data refers to any non-constant operand (`uid` in the example) of a *sensitive instruction*, where an instruction is considered sensitive if it is involved in a function pointer calculation (line 3-4 in the example).

For efficient CFI enforcement, μCFI uses PT when collecting c-data and ICT targets.

```
void A(); void B(); void C();
void handleReq(int uid) {
  void (*func_ptr_arr[3])() = {&A, &B, &C};
  void (*func_ptr)() = func_ptr_arr[uid];
  (*func_ptr)();
}
```

Listing 1.1. Code snippet for describing μCFI.

In Figure 3, we present the overview of μCFI enforcement using PT (PT-based μCFI). μCFI consists of a compiler (μCFI-compiler) and a dynamic monitor (μCFI-monitor). The μCFI-compiler instruments the program source to identify c-data and generates an instrumented binary. During the execution of the instrumented binary, PT writes the encoded traces into its trace buffer. When the trace buffer reaches capacity, the kernel driver (PT-Driver) copies the trace buffer into a kernel buffer. μCFI-monitor obtains the encoded PT trace of the instrumented program from the kernel buffer by signaling PT-Driver, decodes the PT trace in its trace decoder unit, and validates the ICT targets in the points-to analyzer unit.

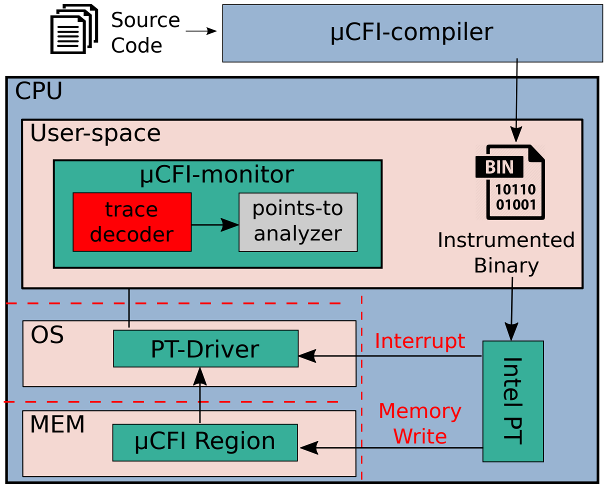

Fig. 3. μCFI [24] design overview: μCFI consists of two components: a compiler (μCFI-compiler) that instruments the program to identify c-data, and a runtime monitor (μCFI-monitor) to validate ICTs.

To guarantee the protection of forward-edges, μCFI-monitor requires collecting c-data, the target of indirect `calls`[3], and the target of sensitive `rets` from

[3] Since the current μCFI implementation does not protect indirect `jmps`, μCFI-compiler converts each indirect `jmp` to a conditional branch.

the instrumented program. Note that μCFI-monitor requires the target value of some of the returns (only sensitive ones) for forward-edge protection since they are involved in the function pointer calculation. More specifically, a return is "sensitive" if its corresponding function contains at least one sensitive instruction. For backward-edge protection, the μCFI-compiler instruments the program to implement a software-based shadow stack based on a prior work [16].

3 PHMon-Based μCFI

PHMon-based μCFI is a hardware-assisted context-sensitive CFI enforcement. There are two main advantages of leveraging PHMon when enforcing μCFI. First, PHMon collects the program traces in their original forms. Therefore, it does not introduce trace processing overhead when enforcing CFI. Second, PHMon offers a variety of configuration capabilities. This feature allows PHMon to easily collect both contextual data and ICT targets.

3.1 Design

In Figure 4, we show the overview of μCFI enforcement using PHMon (PHMon-based μCFI). First, we explain how we leverage PHMon to protect forward-edges. As a first step, we compile a program with a modified version of the μCFI-compiler (detailed in Section 3.2) and generate the instrumented binary. Prior to the execution of this binary, PHMon is programmed to collect the required information (i.e., c-data, the target of indirect calls, and the target of sensitive returns) for μCFI enforcement. While the processor executes the binary, PHMon collects the commit log through RoCC. Then, PHMon applies the user-defined monitoring rules to the commit log to determine if the commit log includes any information for μCFI enforcement. PHMon writes the collected information from the binary into a trace buffer depicted as μCFI Region in Figure 4. Whenever the trace buffer becomes full, PHMon raises an interrupt. Our kernel module (PHMon-Driver) copies the collected trace buffer to a kernel buffer and informs the OS such that the OS can resume the execution of the instrumented binary. The PHMon-Driver is also in charge of providing the collected traces to the μCFI-monitor as the μCFI-monitor performs the enforcement of the ICT instructions.

We protect backward-edges by implementing a shadow stack. Delshadtehrani et al. [19] already showed that PHMon can be used to implement a shadow stack (PHMon-based shadow stack). Instead of implementing a software-only shadow stack like PT-based μCFI, we implement PHMon-based shadow stack in our prototype for completeness. We program PHMon to validate the return targets by monitoring `call`/`ret` instructions (details provided in Section 3.2). In case of a `call` instruction, PHMon stores the new return address in a shared memory space (Shadow Stack in Figure 4). Whenever the function returns, PHMon compares the return address with the one stored in the Shadow Stack to validate the return target.

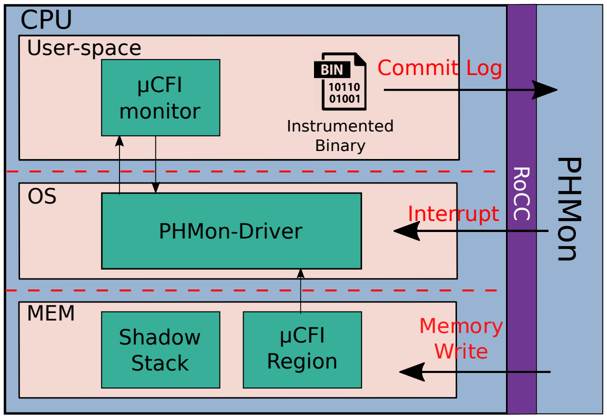

Fig. 4. Design Overview of PHMon-based μCFI: PHMon writes the collected program traces into μCFI Region and Shadow Stack in memory for forward-edge protection and backward-edge protection, respectively. μCFI-monitor fetches the traces via PHMon-Driver and enforces CFI.

One of the main differences between protecting forward and backward edges is the enforcement mechanism. For forward-edge protection, PT-based μCFI stores the collected packets in trace buffers and provides the buffer content to the software monitor by raising an interrupt. We keep our PHMon-based μCFI implementation similar to PT-based μCFI to fairly represent the architectural benefit of using PHMon. Specifically, PHMon stores the necessary information in a memory buffer and provides this information to the software monitor by triggering an interrupt. Here, we use PHMon as a trace collection mechanism (similar to PT) by leaving the ICT validation to the software monitor. For a shadow stack, PHMon validates the ICT targets at the hardware level by using a sequence of ALU instructions. Therefore, PHMon validates the ICT targets without requiring a software monitor. Note that we implement the shadow stack to show that our PHMon-based protection mechanism for forward edges is compatible with a backward-edge protection mechanism. The performance benefit of this work arises from the forward-edge protection mechanism.

3.2 Implementation

To enforce μCFI using PHMon, we applied changes to both the μCFI-compiler and μCFI-monitor. We used the same front-end IR-level instrumentation (LLVM 3.6) used by the original μCFI-compiler [2]. This front-end instrumentation is in charge of identifying c-data. We used our RISC-V back-end instrumentation to collect c-data, indirect call targets, and sensitive return targets (detailed later in this section) using PHMon. At the release of LLVM 3.6, RISC-V was not a supported architecture. Therefore, we used a newer version (LLVM 7.0) for the back-end instrumentation. We removed the code that implements the trace decoder unit (color-coded with red in Figure 3) from the μCFI-monitor since PHMon does not perform any encoding while collecting the traces from the binary. We applied the necessary changes to allow the μCFI monitor to communicate with the PHMon-Driver. We used LLVM 7.0 to cross-compile the μCFI-monitor [3] for RISC-V. Overall, we aim to minimize the software-level implementation differences between PT-based μCFI and PHMon-based μCFI, so that we can fairly represent the architectural benefit of using PHMon.

We slightly modified the Linux kernel to support PHMon-based μCFI. Since the frequent suspension of the protected program increases the performance overhead, μCFI suspends the protected program only at security-sensitive system calls to validate the target of the collected ICTs. Similar to many prior works [13, 20, 24, 40], we modify our kernel to suspend the execution of the protected program at the following security-sensitive system calls: `mmap`, `mremap`, `remap_file_pages`, `mprotect`, `execve`, `execveat`, `sendmsg`, `sendmmsg`, `sendto`, and `write`.

As we explained in Section 2.2, PHMon maintains the incoming match packets in a queue prior to executing the user-defined instructions stored in the CFU. When the queue gets full, an obvious option for PHMon [19] is to stall the fetch stage of the Rocket core's pipeline until PHMon processes all the packets waiting in the queue. However, this is not the proper way of handling the queue problem for PHMon-based μCFI since match packets frequently require an action that should be processed by the processor, i.e., interrupt. Instead of stalling the processor, we modified PHMon to raise an interrupt handled by the OS whenever the queue becomes full. We then perform busy-waiting in the interrupt handler until all the match packets in the queue are processed. To provide full protection against control-flow attacks, in addition to leveraging PHMon for forward-edge protection (PHMon-based μCFI), we also use it for backward-edge protection (PHMon-based shadow stack). In total, we program 5 MUs to simultaneously implement PHMon-based μCFI and PHMon-based shadow stack. In the rest of this section, we explain about programming PHMon for forward-edge as well as backward-edge protection.

Programming PHMon for Forward-edge Protection For μCFI forward-edge protection, we use three MUs: one MU for indirect calls, one MU for sensitive returns, and one MU for c-data collection. We use two registers from the Register File of PHMon to store the base address and the current pointer of μCFI Region.

Indirect calls: To collect indirect call targets with PHMon, we replace each indirect call in the program with an indirect jump to a special function (`ICF` as shown in Listing 1.2). The `ICF` loads the indirect call target into a temporary register (`t1`) from a fixed memory address, and jumps to the target address stored in `t1`. To obtain the target address during the instrumented binary execution, we use one MU which compares the `pc_src` of the collected commit log with the PC value of the load instruction (0x104b4 in Listing 1.2) in the `ICF`. This PC value can be statically obtained by disassembling the binary. Whenever PHMon detects a match, it writes the content of `data` field of the commit log (this field holds the `t1` value) to the trace buffer allocated for μCFI.

Sensitive returns: As explained in Section 2, the target address of each sensitive return is required by the μCFI-monitor for forward-edge protection. To obtain the target values for sensitive returns, we insert a `mv t1,ra` instruction before each sensitive return instruction in the application. This instruction copies the return address value to the temporary register `t1`. PHMon can then simply collect the value of `t1`. To do this, we use one MU to detect the execution of the `mv t1,ra` instruction. Whenever the `inst` value in the incoming commit log matches with the machine code of the `mv t1,ra`, PHMon writes the content of `data` field of the commit log (`t1` in this case) into the trace buffer allocated for μCFI.

c-data Collection: To collect c-data, we instrument the program to call a special `write_cdata` function as shown in Listing 1.3. The program calls the `write_cdata` function to send c-data to the μCFI-monitor. The `write_cdata` loads the value of c-data into a temporary register (`t1`) from a fixed memory address, and immediately returns. We program one MU to monitor the `ld` instruction in the `write_cdata`. The MU compares the `pc_src` value of the incoming commit logs with the PC value of the `ld` instruction (0x104bc in Listing 1.3). Whenever PHMon detects a match, PHMon writes the content of `data` field (`t1` in this case) into the trace buffer allocated for μCFI.

```
#load indirect call target to t1
<ICF>:
  104b4:  ld  t1,-728(gp)
  104b8:  jr  t1
```

Listing 1.2. RISC-V assembly code of the function `ICF`

```
#load c-data value into t1
<write_cdata>:
  104bc:  ld  t1,-720(gp)
  104c0:  ret
```

Listing 1.3. RISC-V assembly code of the function `write_cdata`

Programming PHMon for Backward-edge Protection We implement a shadow stack (similar to original work [19]) using PHMon to demonstrate the compatibility of PHMon-based μCFI with backward-edge CFI mechanisms. We use one MU to monitor calls and one MU to monitor returns. We use two registers from the Register File of PHMon to store the base address of the Shadow Stack and the Shadow Stack pointer. We program PHMon to write the original return addresses into a trace buffer (Shadow Stack in Figure 4) when a call instruction is executed. Upon a return instruction, PHMon pops a value from the shadow stack and compares it with the current return value. PHMon performs the comparison directly on hardware using its ALU unit. In case of a mismatch, PHMon raises an interrupt and the OS terminates the process.

4 Evaluation

We evaluated our PHMon-based μCFI system to answer the following questions: **(1)** What is the execution time overhead of our source code instrumentation to leverage PHMon?

Table 1. Microarchitectural details of Intel processor, Rocket core and PHMon.

Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz	
Pipeline	out-of-order
L1 instruction cache	32 KB, 8-way set-associative
L1 data cache	32 KB, 8-way set-associative
L2 cache	256 KB, 4-way set-associative
L3 cache	12 MB, 16-way set-associative
Rocket Core @ 25MHz	
Pipeline	6-stage, in-order
L1 I Cache	16 KB, 4-way set-associative
L1 D cache	16 KB, 4-way set-associative
Register file	31 entries, 64-bit
PHMon	
MUs	5
Local Register File	6 entries, 64-bit
Match Queue	1,024 entries, 129-bit
Action Config Table	16 entries

(2) How much overhead does PHMon incur to collect the program traces?
(3) How much overhead does PHMon-based μCFI incur when protecting forward edges only?
(4) What is the performance degradation of integrating a backward-edge CFI mechanism into PHMon-based μCFI?

4.1 Evaluation Framework

To evaluate the performance of PT-based μCFI for systems with varying core numbers, we used an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz machine running Ubuntu 16.04. The microarchitectural details are provided in Table 1. We run PT-based μCFI on four different configurations; 1-Core, 2-Core, 3-Core, and 8-Core. As discussed in Section 1, in Figure 1, we reported the performance overhead of μCFI for these four configurations. In addition, we reported the packet count collected by PT for enforcing μCFI. We used the open-source μCFI-monitor [3], μCFI-compiler [2], and μCFI-kernel [4][4] repositories. By using these three repositories with no modifications, we successfully reproduced the results (within 1% standard deviation) reported in the original work [24] on an 8-Core processor. However, we observed that μCFI-kernel does not suspend the execution of the protected program for three of the security-sensitive system calls (`mremap`, `remap_file_pages`, and `write`) reported in the paper [24]. Hence, we modified μCFI-kernel to include these missing system calls. This modification lead to higher performance overhead of some benchmarks (i.e., +9% for sjeng, +84% for astar, and +234% for h264ref) compared to the overheads reported

[4] μCFI-kernel is the modified Linux kernel which supports μCFI.

in the original work [24]. In our analysis, we ran each benchmark three times and calculated the average (geometric mean) overhead. The standard deviation in our measurements is less than 1%. We also provide the error bars for each benchmark in Figure 1.

To measure the PHMon-based μCFI performance overhead, we compared PHMon-based μCFI with the baseline implementation of the Rocket processor. The microarchitectural parameters of Rocket core and PHMon are listed in Table 1. For both experiments, Rocket core includes a 16K L1 instruction cache and a 16K L1 data cache without an L2 or an L3 cache[5]. Due to the limitation of our FPGA-base evaluation platform, we could run Rocket core with maximum frequency of 25 MHz for both experiments. We modified the open-source PHMon architecture [1] interfaced with the 6-stage in-order RISC-V Rocket processor [9] via RoCC interface. PHMon-based μCFI is prototyped on a Xilinx Zynq Zedboard [33], running a modified version of RISC-V Linux (v4.20) kernel. For both experiments, i.e., the baseline and PHMon-based μCFI, we setup the Rocket processor with the same configurations including a 16K L1 instruction and data cache. We performed each experiment three times and calculated the average value. All standard deviations were below 1%. To show the stability of our measurements, we include the error bars as well (both in Figure 5 and 6).

During development, we observed that the μCFI-monitor validates the traces much slower than PHMon's trace collection speed. Hence, the collected traces accumulated in the kernel. Due to the limited available memory in our evaluation framework, the accumulated traces eventually resulted in an out of memory situation for some of the benchmarks. To circumvent this issue when we reach the memory limit, PHMon-Driver suspends the protected program until all the collected traces are processed by the μCFI-monitor. Note that this increases the duration that we suspend the process and potentially increases the performance overhead of PHMon-based μCFI. In an evaluation framework with more available memory, PHMon-based μCFI could outperform our current prototype.

4.2 Evaluation Benchmarks

We calculated the runtime overhead of PT-based μCFI (Figure 1) for C/C++ applications using the 'test' workload of SPEC2006 benchmark suite [23]. We could not obtain results (similar to original work [24]) for `gcc`, `dealII`, `povray`, `omnetpp` and `xalancbmk` since PT loses packets at the hardware level, which manifests as a segmentation fault in the μCFI-monitor. Additionally, in our evaluation framework, `soplex` caused a segmentation fault in the μCFI-monitor. We did not include `perlbench` in our evaluation since this benchmark frequently uses the `fork` system call. The heavy usage of `fork` puts more pressure on a single-core compared to an 8-Core; hence, using this benchmark in our evaluation framework misrepresents the performance impact of the trace processing for μCFI enforcement. Note that we represent the `dniwog` input as a separate data

[5] At time of our evaluation, Rocket core was not supporting L2 and L3 cache.

point for gobmk since its packet number is drastically higher than the remaining data points.

To evaluate PHMon-based μCFI on a Rocket processor, we used 8 out of 12 benchmarks which successfully run with PT-based μCFI. Unfortunately, we could not run lbm, namd, and gobmk with a PHMon-based μCFI due to RISC-V cross-compilation errors using LLVM 7.0. We could not run mcf and perlbench due to the limited memory on our FPGA. Similarly, sjeng was also too large for our FPGA. However, by reducing the value of TTSize (which controls the size of one of the hashtables in sjeng) to 3000 in sjeng's source code, we were able to run it with PHMon-based μCFI. For a fair evaluation, we also report the overhead of PT-based μCFI for sjeng using TTSize=3000 in Figure 6.

4.3 Evaluation Results

Figure 5 depicts the performance impact of the source code instrumentation to collect program traces (instrumentation overhead) using PHMon. We measured the instrumentation overhead of the benchmarks by comparing the execution time of a baseline program with the instrumented program for μCFI enforcement. Our results demonstrate that the code instrumentation (Instru in Figure 5), including our RISC-V back-end passes to transfer c-data and ICT targets to the μCFI-monitor, incurs very low performance overhead (1.7% on average). Note that the instrumentation overhead is higher (peak 5%) for the benchmarks generating more packets such as sjeng, astar, and h264ref.

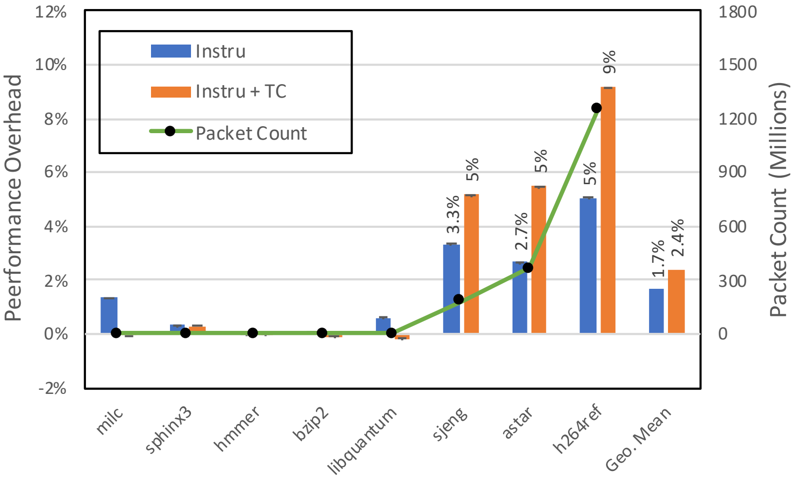

Fig. 5. Performance overhead (left y-axis) of the instrumented binary (Instru) and trace collection (TC) overhead of PHMon. We use the right y-axis to show the packet count for each benchmark.

To demonstrate that PHMon can efficiently collect contextual information, we measured the trace collection overhead of PHMon (TC in Figure 5). To do this, we ran the instrumented benchmarks under the monitoring of PHMon without μCFI enforcement. Whenever the trace buffer of PHMon became full, PHMon triggered an interrupt and returned from the interrupt handler without processing the trace buffers. Note that `Instru+TC` overhead also includes the instrumentation overhead. Our results show that PHMon can efficiently collect program traces (<1% overhead on average and 4% peak). Note that PHMon's trace collection overhead is maximum of 4% even for benchmarks generating more packets such as `sjeng`, `astar`, and `h264ref`.

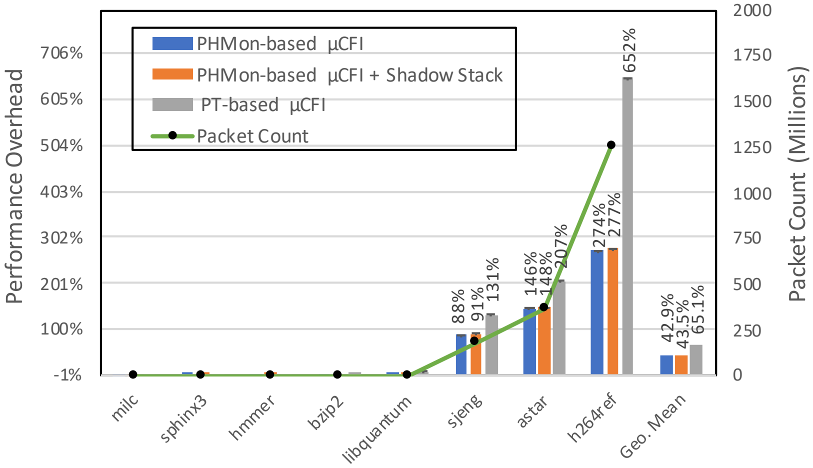

Fig. 6. Performance overhead of PHMon-based μCFI, PHMon-based μCFI + Shadow Stack, and PT-based μCFI. We use the right y-axis for providing the packet count for each benchmark.

Using Figure 6, we first depict the performance overhead of PHMon-based μCFI and PT-based μCFI when protecting only the forward-edges. Both PHMon-based and PT-based μCFI perform efficiently for benchmarks such as `milc`, `sphinx3`, `hmmer`, `bzip2`, and `libquantum` that generate fewer packets. PHMon-based μCFI introduces 88%, 146%, and 274% overhead for packet-intensive `sjeng`, `astar`, and `h264ref` benchmarks, respectively. For these benchmarks, PT-based μCFI results in 131%, 207%, and 652% performance overhead, respectively. Since PHMon-based μCFI does not incur trace processing overhead, its performance bottleneck mainly arises from the ICT validation performed by the μCFI-monitor. For PT-based μCFI, there is an additional trace processing overhead prior to ICT validation.

In Figure 6, we also show the full PHMon-based CFI protection overhead. The full protection secures forward-edges and backward-edges with PHMon-based μCFI and PHMon-based shadow stack, respectively. Adding the shadow stack increases the performance overhead of PHMon-based μCFI by less than 1% on average (peak 3%) and allows us to fully protect the programs against control-flow hijacking attacks.

The original work [19] reports the power and area overhead of PHMon with varying number of MUs. Based on those results, PHMon-based μCFI using three MUs incurs 6.5% power and 15.1% area overhead. The full protection requires five MUs, which results in 9.2% power and 18.4% area overhead.

5 Discussion

In this section, we discuss some of our design choices when implementing μCFI using PHMon. We specifically discuss aspects of our source code instrumentation (Section 3.2) to protect forward-edges.

When enforcing μCFI using PHMon, we aim to minimize the software-level implementation differences with the original PT-based μCFI work so that we can fairly represent the architectural benefit of using PHMon. Therefore, similar to PT-based μCFI, we collected indirect call targets by redirecting the control-flow to the special function (ICF as shown in Listing 1.2). We initially aimed to replace indirect calls with direct calls to ICF similar to PT-based μCFI targeting x86_64. Unfortunately, direct calls in RISC-V can target a limited range (± 1 MiB) since the offset is encoded into the operand of the the instruction and that operand is only 20 bits. Therefore, we could not replace each indirect call with a direct call in RISC-V, especially for benchmarks with bigger code size, and had to use indirect jumps instead. Unfortunately, we do not provide additional checks to ensure that these indirect jumps are not subverted by an attacker at run-time. However, we could easily avoid these indirect jumps in the binary by instrumenting the code with “custom” instructions. The RISC-V ISA allows adding a custom ISA extension. We could insert a custom instruction that stores the target address of an indirect call in a register before each indirect call instruction. This way, we could obtain indirect call targets without redirecting indirect calls to the ICF using indirect jumps. We redirect indirect calls to ICF using indirect jumps instead of inserting custom instructions to have a similar implementation with PT-based μCFI.

We collect c-data by monitoring the `ld` instruction at a fixed address (see Listing 1.3). We obtain the PC value of `ld` instruction using a static analysis. Our design choice aligns with PT-based μCFI which generates packets only for fixed addresses. To minimize the software-level implementation differences, we implement PHMon-based μCFI in a similar way. Unfortunately, this design choice can result in some portability issues. For instance, it can cause problems with randomization mechanisms like ASLR. In fact, this issue could also easily be addressed by inserting a custom instruction that will help us store the constraining data in a register. This way, we could program PHMon to monitor the custom instruction rather than a specific program counter. Since the instruction machine code is the same regardless of the program layout, μCFI enforcement would be more portable than our current implementation.

We insert `mv t1,ra` instruction before each sensitive return to collect the return value using PHMon. We choose the `mv t1,ra` instruction since none of the SPEC2006 benchmarks contains it when compiled without our back-end pass. We acknowledge that the proper way to collect sensitive return values would be to insert a custom instruction before each sensitive return. For instance, the custom instruction could store the return value in a register. We could program PHMon to monitor this custom instruction and write the content of the register value into memory. This way, we could ensure that the original program does not contain the inserted instruction unless it enforces μCFI.

6 Related Work

Our PHMon-based μCFI approach is closely related to works which use hardware support for CFI enforcement. We divide these hardware mechanisms into two categories: the ones already deployed in modern processors, and the new hardware designs proposed for future deployment.

6.1 Reusing Deployed Hardware Features

CFI mechanisms that rely on existing hardware features are practical since they can be readily deployed on commodity hardware. Unfortunately, existing hardware features have several drawbacks since the hardware features are not designed with security in mind. Specifically, CFIMon [42] utilizes Branch Trace Store (BTS) [34] to enforce CFI. However, BTS incurs high performance overheads (20×-40×) [40]. To reduce the overhead, several works [13,31,40] use Last Branch Record (LBR) [34] for CFI enforcement. LBR can record the last N executed branches where N can be 4, 8, 16, or 32 depending on the processor model. For instance, kBouncer [31] aims to protect backward-edges from ROP attacks using LBR. kBouncer checks the control flow of the program whenever the program makes a security sensitive system call. ROPecker [13] extended kBouncer's approach by emulating the potential program execution with the help of a statically generated ROP gadget database. The key idea is to detect ROP gadgets which can possibly be stitched together towards a malicious purpose. Due to LBR's branch recording capacity, kBouncer and ROPecker are shown to be vulnerable to history flushing attacks [11]. This attack initially cleanses any evidence of the ROP attack in the short-term history and then creates a view of history that the defense will not classify as an attack. Another LBR-based CFI mechanism (PathArmor [40]) raises the bar for history flushing attacks thanks to its context-sensitive CFI policy. PathArmor uses LBR to record the last 16 indirect branches and direct calls as the context. Unfortunately, PathArmor checks less than 0.1% of total returns on NGINX [21] for backward-edge protection because of the LBR's limited trace recording capability.

CFIGuard [41] overcomes the limited size of LBR by combining it with the Performance Monitoring Unit. CFIGuard raises an interrupt whenever LBR buffer is full. However, triggering an interrupt every 16 branches can significantly increase the performance overhead, especially for CPU-intensive applications. OS-CFI [28] implements an origin sensitive context-sensitive CFI mechanism to reduce the attack surface for C-style indirect calls and C++ virtual calls. For the former, the origin is the the most recently updated code location. For the latter, the origin refers to code location where receiving object's constructor is called. OS-CFI uses Intel MPX for efficiently storing and retrieving the origin of the code pointers. OS-CFI uses inline reference monitors to collect and maintain the contextual information. Since these monitors extensively use memory to store the temporary data for searching hash table, they are vulnerable to race conditions for a short interval. To protect the integrity of inline reference monitors, OS-CFI utilizes the transactional memory (Intel TSX). LMP [25] uses MPX for protecting backward-edges by implementing a shadow stack via program source instrumentation. Unfortunately, Intel MPX is not adopted by industry widely due to the considerable performance overhead and compatibility issues [30]. MPX is not available on future Intel processors [5].

Several researchers also leverage Intel PT for CFI enforcement. PT can record higher number of indirect branches than LBR, which allows researchers to enforce more precise CFI mechanisms. For instance, PT-CFI [22] enforces backward-edge CFI by implementing a shadow stack for the COTS binaries based on the PT traces. Griffin [21] implements three different CFI policies over unmodified binaries and shows the tradeoff between precision and performance. Also, Griffin shows the performance impact of the number of kernel threads on the speed of buffer trace processing and CFI enforcement, which goes up from $\sim$%8 to $\sim$%19 on NGINX as we increase the number of threads from one to six. FlowGuard [29] attempts to minimize the performance overhead of PT with its fuzzing-assisted approach. The key idea is to collect program traces prior to the program execution by using a fuzzer and minimize the overhead of expensive software-level decoding of PT. Dynamic analysis-based approaches [20, 24] increase the precision of CFI by obtaining additional information from the program at runtime, but at the expense of introducing higher performance overhead. More specifically, PITTYPAT [20] implements a path-sensitive CFI policy, which verifies the whole executed control path of the program. μCFI uses constraining data to provide unique code target for each ICT.

PHMon-based μCFI enforces CFI without weakening any security guarantees. As opposed to PT-based μCFI, PHMon-based μCFI can collect the original form of the data and does not require software-level decoding of collected information when validating control-flows. Also, PHMon-based μCFI is not vulnerable to history flushing attacks as opposed to LBR-based CFI mechanisms.

6.2 New Hardware Designs

Several works propose new hardware designs to enforce CFI. For instance, HAFIX [17] proposes a fine-grained backward-edge CFI system which confines function returns to active call sites. It assigns unique labels to each function by instrumenting the program source with compiler support and enforces the CFI policy directly on hardware for efficiency. Unfortunately, recent work shows that HAFIX is vulnerable to Back-Call-Site attack [39] and cannot fully protect backward-edges. Also, it is vulnerable to any forward-edge attacks. HCFI [14] can fully protect backward-edges by implementing a shadow stack. Additionally, it implements the forward-edge CFI policy discussed by Abadi et al. [7]. Similar to HAFIX, HCFI also modifies the ISA and introduces new instructions to provide CFI capability to the core. Sullivan et al. [37] enhance HAFIX by supporting forward edge protection. Although both Sullivan et al. [37] and HCFI implement efficient forward-edge CFI policies directly on hardware, unlike μCFI, they are still unable to provide a unique target for each ICT and cannot fully protect against forward edge attacks. Intel announced its hardware support for CFI in the form of CET [6]. CET offers strong backward-edge protection with a shadow stack. Unfortunately, the forward-edge policy protection (i.e., Indirect branch tracking) is coarse-grained and vulnerable to advanced attacks such as JOP [12] and COOP [36]. Nile [18] and PHMon [19] offer full protection against backward-edge attacks by implementing a shadow stack with less than 2% performance overhead. However, these two works cannot protect against forward-edge attacks. This work complements PHMon by offering forward-edge protection.

7 Conclusion

In this work, we show that the hardware features originally designed for debugging on Intel processors are not efficient when used for enforcing CFI. Specifically, Intel PT-based CFI mechanisms put high pressure onto idle cores in processor since they require expensive software-level decoding prior to ICT enforcement. All of these PT-based mechanisms assume that idle cores are readily available for CFI enforcement, which is not necessarily the case considering the multi-threaded nature of common applications. We evaluate the performance impact of the trace processing on PT-based CFI enforcement and show that a state-of-the-art CFI mechanism (μCFI) incurs up to 652% overhead on a single-core compared to 372% overhead on a 3-Core processor. When enforcing CFI, we leverage a programmable hardware monitor (PHMon) which does not introduce trace processing overhead unlike PT. Our PHMon-based μCFI mechanism incurs 43% performance overhead, on average, to secure forward edges. We also integrate a hardware-based shadow stack to fully secure the program including backward-edges. Adding the shadow stack increases the performance overhead of PHMon-based μCFI by less than 1% on average.

Acknowledgements. This work was supported in part by NSF SaTC Award 1916393 and Google Faculty Research Award.

References

1. bu-icsg. `https://github.com/bu-icsg/PHMon`, accessed on 10-Feb-2020
2. ucfi-gatech. `github.com/uCFI-GATech/ucfi-compiler/commit/6502e1c`, accessed on 10-Feb-2020
3. ucfi-gatech. `github.com/uCFI-GATech/ucfi-monitor/commit/8787121`, accessed on 10-Feb-2020
4. ucfi-gatech. `github.com/uCFI-GATech/ucfi-kernel/commit/08a15f7`, accessed on 10-Feb-2020
5. Intel memory protection extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions` (2013)
6. Intel control-flow enforcement technology. `software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf` (2019), accessed on 10-Feb-2020
7. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC) **13**(1), 1–40 (2009)

8. Andersen, S.: Changes to functionality in microsoft windows xp servicepack 2 part 3: memory protection technologies (2004)
9. Asanović, K., et al.: The Rocket Chip generator. Tech. Report, EECS Department, UC Berkeley (2016)
10. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Computing Surveys (CSUR) **50**(1), 1–33 (2017)
11. Carlini, N., Wagner, D.: ROP is still dangerous: Breaking modern defenses. In: Proceedings of the USENIX Security Symposium (2014)
12. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the Conference on Computer and Communications Security. CCS (2010)
13. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng, H.R.: Ropecker: A generic and practical approach for defending against rop attacks. In: Symposium on Network and Distributed System Security. NDSS (2014)
14. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: Hcfi: Hardware-enforced control-flow integrity. In: Proceedings of the Sixth Conference on Data and Application Security and Privacy. CODASPY (2016)
15. Cui, W., Ge, X., Kasikci, B., Niu, B., Sharma, U., Wang, R., Yun, I.: REPT: Reverse debugging of failures in deployed software. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. OSDI (2018)
16. Dang, T.H., Maniatis, P., Wagner, D.: The performance cost of shadow stacks and stack canaries. In: Proceedings of the ACM Symposium on Information, Computer and Communications Security. ASIA CCS (2015)
17. Davi, L., Hanreich, M., Paul, D., Sadeghi, A., Koeberl, P., Sullivan, D., Arias, O., Jin, Y.: Hafix: Hardware-assisted flow integrity extension. In: Proceedings of the Design Automation Conference. DAC (2015)
18. Delshadtehrani, L., Eldridge, S., Canakci, S., Egele, M., Joshi, A.: Nile: A programmable monitoring coprocessor. IEEE Computer Architecture Letters **17**(1), 92–95 (2018)
19. Delshadtehrani, L., Canakci, S., Zhou, B., Eldridge, S., Joshi, A., Egele, M.: Phmon: A programmable hardware monitor and its security use cases. In: Proceedings of the USENIX Security Symposium (2020)
20. Ding, R., Qian, C., Song, C., Harris, B., Kim, T., Lee, W.: Efficient protection of path-sensitive control security. In: Proceedings of the USENIX Security Symposium (2017)
21. Ge, X., Cui, W., Jaeger, T.: Griffin: Guarding control flows using intel processor trace. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS (2017)
22. Gu, Y., Zhao, Q., Zhang, Y., Lin, Z.: Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In: Proceedings of the Conference on Data and Application Security and Privacy. CODASPY (2017)
23. Henning, J.L.: Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture News **34**(4), 1–17 (2006)
24. Hu, H., Qian, C., Yagemann, C., Chung, S.P.H., Harris, W.R., Kim, T., Lee, W.: Enforcing unique code target property for control-flow integrity. In: Proceedings of the Conference on Computer and Communications Security. CCS (2018)
25. Huang, W., Huang, Z., Miyani, D., Lie, D.: Lmp: Light-weighted memory protection with hardware assistance. In: Proceedings of the Annual Conference on Computer Security Applications. ACSAC (2016)

26. Kasikci, B., Schubert, B., Pereira, C., Pokam, G., Candea, G.: Failure sketching: A technique for automated root cause diagnosis of in-production failures. In: Proceedings of the Symposium on Operating Systems Principles. SOSP (2015)
27. Khandaker, M., Naser, A., Liu, W., Wang, Z., Zhou, Y., Cheng, Y.: Adaptive callsite sensitive control flow integrity. In: Proceedings of the European Symposium on Security and Privacy. EuroS&P (2019)
28. Khandaker, M.R., Liu, W., Naser, A., Wang, Z., Yang, J.: Origin-sensitive control flow integrity. In: Proceedings of the USENIX Security Symposium (2019)
29. Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., Guan, H.: Transparent and efficient cfi enforcement with intel processor trace. In: Proceedings of the International Symposium on High Performance Computer Architecture (HPCA). HPCA (2017)
30. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., Fetzer, C.: Intel mpx explained: A cross-layer analysis of the intel mpx system stack. Proceedings of the ACM on Measurement and Analysis of Computing Systems **2**(2), 1–30 (2018)
31. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Proceedings of the USENIX Security Symposium (2013)
32. Reinders, J.: Processor tracing (Jun 2017), `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`
33. Digilent's ZedBoard Zynq FPGA: `http://www.digilentinc.com/Products/Detail.cfm?Prod=ZEDBOARD/` (2017), accessed on 10-Feb-2020
34. Intel Corporation: Intel 64 and IA-32 architectures software developer's manual. System Programming Guide, volume 3 (3A, 3B, 3C & 3D) (2016)
35. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC) **15**(1), 1–34 (2012)
36. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: Proceedings of the Symposium on Security and Privacy. S&P (2015)
37. Sullivan, D., Arias, O., Davi, L., Larsen, P., Sadeghi, A., Jin, Y.: Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In: Proceedings of the Design Automation Conference. DAC (2016)
38. Thalheim, J., Bhatotia, P., Fetzer, C.: Inspector: data provenance using intel processor trace (pt). In: International Conference on Distributed Computing Systems (ICDCS) (2016)
39. Theodorides, M., Wagner, D.: Breaking active-set backward-edge cfi. In: International Symposium on Hardware Oriented Security and Trust. HOST (2017)
40. van der Veen, V., Andriesse, D., Goktas, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical context-sensitive cfi. In: Proceedings of the Conference on Computer and Communications Security. CCS (2015)
41. Yuan, P., Zeng, Q., Ding, X.: Hardware-assisted fine-grained code-reuse attack detection. In: Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses. RAID (2015)
42. Yubin Xia, Yutao Liu, Chen, H., Zang, B.: Cfimon: Detecting violation of control flow integrity using performance counters. In: IEEE/IFIP International Conference on Dependable Systems and Networks. DSN (2012)
43. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: Proceedings of the USENIX Security Symposium (2013)