PHMon: A Programmable Hardware Monitor and Its Security Use Cases

ARTIFACT EVALUATED usenix ASSOCIATION PASSED

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

Department of Electrical and Computer Engineering, Boston University

{delshad, scanakci, bobzhou, schuye, joshi, megele}@bu.edu

Abstract

There has been a resurgent trend in the industry to enforce a variety of security policies in hardware. The current trend for developing dedicated hardware security extensions is an imperfect, lengthy, and costly process. In contrast to this trend, a flexible hardware monitor can efficiently enforce and enhance a variety of security policies as security threats evolve. Existing hardware monitors typically suffer from one (or more) of the following drawbacks: a restricted set of monitoring actions, considerable performance and power overheads, or an invasive design. In this paper, we propose a minimally-invasive and efficient implementation of a Programmable Hardware Monitor (PHMon) with expressive monitoring rules and flexible fine-grained actions. PHMon can enforce a variety of security policies and can also assist with detecting software bugs and security vulnerabilities.

Our prototype of PHMon on an FPGA includes the hardware monitor and its interface with a RISC-V Rocket processor as well as a complete Linux software stack. We demonstrate the versatility of PHMon and its ease of adoption through *four* different use cases: a shadow stack, a hardware-accelerated fuzzing engine, an information leak prevention mechanism, and a hardware-accelerated debugger. Our prototype implementation of PHMon incurs 0.9% performance overhead on average, while the hardware-accelerated fuzzing engine improves fuzzing performance on average by 16× over the state-of-the art software-based implementation. Our ASIC implementation of PHMon only incurs a 5% power overhead and a 13.5% area overhead.

1 Introduction

In recent years, there has been a growing demand to enforce security policies in hardware with the goal of reducing the performance overhead of their software-level counterparts. As a response to this growing demand, leading processor companies have introduced several security extensions. A successful hardware-based enforcement of security policies, such as the NX (non-executable) bit, provides an efficient permanent security solution. The processor companies have also established secure and isolated execution environments such as Intel Trusted Execution Technology (TXT) [64], Intel Software Guard Extensions (SGX) [3], ARM TrustZone [62], and AMD Secure Virtual Machine (SVM) [61]. Additionally, Intel has introduced Memory Protection Extensions (MPX) [65] and Control-Flow Enforcement Technology (CET) [67] to enforce security policies.

Unfortunately, the current trend to develop dedicated hardware security extensions suffers from several drawbacks. Implementing new security extensions in a new generation of processors is a lengthy and costly process (which can take up to several years and millions of dollars). Additionally, the implemented extensions apply **fixed** security policies. Since these fixed security policies are built in silicon, any problems in the design or implementation of these policies requires a fix in the next generation of the processors. For example, Intel introduced MPX as a hardware-assisted extension to provide spatial memory safety by adding new instructions and registers to assist with software-based bounds checking. Software-based techniques, such as Safe-C (1994) [6] and SoftBound (2009) [53], existed several years before Intel MPX was announced in 2013 and introduced commercially in late 2015. Unexpectedly, Intel MPX incurs a considerable performance overhead (up to 4× slow down in the worst case [55]) and its supporting infrastructure cannot compile/run 3-10% of legacy programs [55]. Due to various Intel MPX problems, GCC, LLVM, and Linux discontinued their support for MPX [42,43]. Additionally, MPX does not protect the stack against Return-Oriented Programming (ROP) attacks. Hence, in 2016, Intel announced a new security technology specification, called Control-Flow Enforcement Technology (CET), for full stack protection.

The above Intel MPX example shows the lengthy and imperfect process of implementing fixed hardware security extensions. As a result, these extensions cannot evolve with the same pace as security threats. In contrast to the current trend in the industry to develop rigid hardware security extensions, a **flexible** hardware implementation can enforce and enhance a variety of security policies as security threats evolve. Such a flexible hardware implementation provides a *realistic* environment (a hardware prototype with full software stack) to evaluate the security policies before a manufacturer enforces a policy as a dedicated feature in hardware.

A flexible hardware to enforce security policies can be designed in the form of a hardware-assisted runtime monitor. To characterize a general runtime monitor, we present an *event-action* model. In this model, we define the runtime monitoring by a set of *events*, where each event is defined by a finite set of monitoring rules, followed by a finite sequence of *actions*. This definition does not restrict events/actions to high-level (e.g., accessing a file) or low-level (e.g., execution of an instruction) events/actions. Accordingly, runtime monitoring consists of three main steps: 1) collecting runtime execution information, 2) evaluating the finite set of monitoring rules on the collected information to detect events, and 3) performing a finite sequence of follow-up actions. Intuitively, a monitoring system that allows the user to define generic rules, events, and actions is more widely applicable than a system that restricts the expressiveness of these aspects. Such a monitoring system can be used in a wide range of applications, including, but not limited to, enforcing security policies, debugging, and runtime optimization.

A reference monitor [4, 70] is a well-known concept, which defines the requirements for enforcing security policies. A reference monitor observes the execution of a process and halts or confines the process execution when the process is about to violate a specified security policy. The reference monitor observation can happen at different abstraction levels, e.g., OS kernel, hardware, or inline. We can describe a reference monitor using our *event-action* monitoring model, where the events are specified by security policies and the sequence of actions is limited to halting/confining the process execution. An *event-action* monitoring model has a broader scope and is not restricted to specifying reference monitors for enforcing security policies.

Software-only runtime monitoring techniques can enforce the *event–action* monitoring model with virtually no restriction. However, these software techniques are not suited for *always on* monitoring and prevention mechanisms due to their considerable performance overhead (2.5× to 10× [47, 60] caused by the dynamic translation process of Dynamic Binary Instrumentation (DBI) tools). Hardware-assisted monitoring techniques reduce this significant overhead [26, 28, 89]. Nonetheless, they commonly restrict the expressiveness of the *event–action* monitoring model. Some of the hardware-assisted monitoring techniques are designed for a specific monitoring use case, e.g., Bounds Checking (BC) [15, 27, 32, 51, 52], data-race detection [89], and Dynamic Information Flow Tracking (DIFT) [18, 19, 78, 81]. Other techniques provide some flexibility [10, 11, 25, 26, 28] and can be applied to a range of use cases including BC, DIFT, and Control Flow Integrity (CFI). We refer to these flexible techniques as Flexible Hardware Monitors (FHMons). However, the existing FHMons suffer from three common limitations:

1. Most existing FHMon techniques (e.g., [25, 26, 28]) extend each memory address and register with a tag. These techniques provide a set of actions only for *tag propagation* and raising an *exception* (handled by software), which restricts the expressiveness of their actions. Overall, this limits their deployment beyond tag-based memory corruption prevention. In principle, we can consider the tag-based FHMons as hardware reference monitors to enforce memory protection policies.

2. Some FHMon techniques [11, 12, 46] rely on a separate general-purpose core to perform generic monitoring actions. These techniques incur large overheads (in terms of performance, power, and area) despite leveraging filtering and hardware-acceleration strategies.

3. Some FHMons require invasive modifications to the processor design (e.g., [16, 28, 76]). This limits the feasibility of FHMon adoption in commercial processors as well as the composition of FHMon.

Overall, the existing hardware-assisted monitoring techniques only implement a restricted subset of an ideal *event–action* monitoring model. Hence, they suffer from limited applicability. To address the aforementioned limitations and expand the set of monitoring rules and follow-up actions, we propose a minimally-invasive and low-overhead implementation of a Programmable Hardware Monitor (PHMon).

Our PHMon can enforce a variety of security policies and it can also assist with detecting software bugs and security vulnerabilities. We interface PHMon with a RISC-V [83] Rocket [5] processor and we minimally modify the core to expose an instruction execution trace to PHMon. This execution trace captures the whole *architectural* state of the core. Each event is identified based on programmable monitoring rules applied to the instruction execution trace. Once PHMon detects an event, it performs follow-up actions in the form of hardware operations including ALU operations and memory accesses or an interrupt (handled by software). We modify the Linux Operating System (OS) to support PHMon at process level. Hence, unlike most existing FHMons and tag-based memory corruption prevention techniques, PHMon offers the option of enforcing different security policies for different processes. Additionally, we provide a software API consisting of a set of C functions to program PHMon. A user can simply use this API to specify the monitoring rules and program PHMon to monitor separate events, count the number of event occurrences, and take a series of follow-up actions. We demonstrate the versatility of PHMon and its ease of adoption through *four* representative use cases: a shadow stack, a hardware-accelerated fuzzing engine, information leak prevention, and hardware-accelerated debugging.

To evaluate PHMon in a realistic scenario, we implement a prototype of PHMon interfaced with a RISC-V Rocket core [5] using Xilinx Zedboard FPGA [63]. Our FPGA-based evaluation shows that PHMon improves the performance of fuzzing by 16× over the state-of-the art software-based implementation while our programmed shadow stack (for call stack integrity protection) has 0.9% performance overhead, on average. When implemented as an ASIC, PHMon incurs less than 5% power and 13.5% area overhead compared to an unmodified RISC-V core.

In summary, we make the following contributions:

- **Design**: We propose a minimally-invasive and efficient programmable hardware monitor to enforce an *event–action* monitoring model with programmable monitoring rules and flexible hardware-level follow-up actions. Additionally, we provide the OS and software support for our hardware monitor.
- **Application**: We demonstrate the flexibility and ease of adoption of our hardware monitor to enforce different security policies and to assist with detecting software bugs and security vulnerabilities via *four* use cases.
- **Implementation**: We implement a practical prototype, consisting of a Linux kernel and user-space running on a RISC-V processor interfaced with our PHMon, on an FPGA. Our evaluation indicates that PHMon incurs low performance, power, and area overheads. In the spirit of open science and to facilitate reproducibility of our experiments, we will open-source the hardware implementation of our PHMon, our patches to the Linux kernel, and our software API: https://github.com/bu-icsg/PHMon.

2 Related work

In this section, we discuss existing hardware features in processors and hardware-assisted monitors, which are applied in security use cases, and compare them with PHMon. We classify the hardware-assisted runtime monitors into two categories: "trace-based" and "tag-based". Trace-based monitors apply the monitoring rules and actions on the whole execution trace, while the tag-based monitors restrict the monitoring rules and/or actions to tag propagation. Table 1 compares different features of our trace-based PHMon with other tag-based and trace-based monitors. We can consider the tag-based monitors as reference monitors that can enforce one or more security policies for memory corruption prevention. In general, trace-based monitors are applied to a wider range of applications than merely memory protection. For example, as listed in Table 1, data race detection is one of the use cases of the Log-Based Architectures (LBA) [10, 11].

2.1 Custom Hardware for Monitoring

Dedicated hardware monitors have been used for a variety of debugging and security applications including hardware-assisted watchpoints for software debugging [35, 88] and hardware-assisted Bounds Checking (BC) [27, 32, 51]. Similar to [35, 88], PHMon can be integrated with an interactive debugger, such as GDB, and provide watchpoints by effectively filtering and monitoring different ranges of memory addresses. PHMon can also evaluate conditional break points and we illustrate this capability in Section 5.4.

Dynamic Information Flow Tracking (DIFT) is a technique for tracking information during the program's execution by adding tags to data and tracking the tag propagation. Software-only implementations of DIFT [50, 54, 59] have large performance overheads. To reduce the performance overhead, hardware implementations for DIFT have been proposed [13, 19, 78, 81]. These techniques provide different levels of flexibility for DIFT, from 1-bit tags [59] and multi-bit tags [19] to more flexible designs [13, 81]. Instead of comparing PHMon with custom hardware for BC and DIFT, Section 2.2 provides a comparison with FHMons that are capable of performing both BC and DIFT.

2.2 Flexible Hardware Monitors (FHMons)

FHMons provide flexible monitoring capabilities and can be applied to a range of applications. MemTracker [82] implements tag-based hardware support to detect memory bugs. Several existing works [25, 26, 28] extend DIFT tag-based monitoring into more flexible frameworks capable of supporting different security use cases. PUMP [28] provides programmable software policies for tag-based monitoring with invasive changes to the processor pipeline. FlexCore [25] is a re-configurable architecture decoupled from the processor, which provides a range of runtime monitoring techniques. The programmable FPGA fabric of FlexCore restricts its integration with a high-performance core. Harmoni [26] is a coprocessor designed to apply different runtime tag-based monitoring techniques, where the tagging capability is not as flexible as FlexCore or PUMP. HDFI [76] and REST [74] provide memory safety through data-flow isolation by adding a 1-bit tag to the L1 data cache.

Among the tag-based FHMons, HDFI [76] is the closest work to PHMon in terms of providing a realistic evaluation environment. Both HDFI and PHMon implement a hardware prototype, rather than relying on simulations, and evaluate a full Linux-based software stack on an FPGA. Contrary to PHMon, HDFI applies invasive modifications to the processor pipeline (adds a 1 bit tag to L1 data cache and modifies the decode and execute stages of the pipeline). HDFI is restricted to enforcing data-flow isolation policies to prevent memory corruption. Although PHMon can be used for sensitive data protection (e.g., preventing Heartbleed), compared to HDFI, PHMon has limited capabilities to protect against memory corruption. However, unlike HDFI, PHMon can be applied in security use cases beyond memory corruption prevention, such as accelerating the detection of security vulnerabilities

Table 1: Comparison of previous hardware monitoring techniques with PHMon

Mechanism	Monitoring Mechanism	Use Cases	Source Code Requirement	Hardware Modification	Evaluation Methodology	Avg. Performance Overhead	Power/Area Overhead
Hardbound [27]	Tag-based	BC	Yes	Inv	Sim	5%-9%	# N/A
SafeProc [32]	Tag-based	BC	Yes	Inv	Sim	5%	# N/A
Watchdog [51]	Tag-based	BC	Yes	Inv	Sim	15%-25%	# N/A
LIFT [59]	SW (DBI)	DIFT	No	SW	SW	∼200%-300%	# N/A
TaintCheck [54]	SW (Tag-based)	DIFT	No	SW	SW	Avg: # N/A	# N/A
Multi-Core DIFT [50]	SW (Threads)	DIFT	No	SW	Sim	48%	# N/A
DIFT [78]	Tag-based	DIFT	No	Min-inv	Sim & Emul	1.1%	# N/A
Raksha [19]	Tag-based	DIFT	No	Inv	FPGA	48%	# N/A
FlexiTaint [81]	Tag-based	DIFT	Yes	Min-inv	Sim	1%-3.7%	# N/A
MemTracker [82]	Tag-based	MC	Yes	Inv	Sim	2.7%	# N/A
DataSafe [13]	Tag-based	DIFT	No	Inv	Sim	Avg: # N/A	# N/A
DISE [16]	Binary Rewriting	FI, (De)compress	No	Inv	Sim	Avg: # N/A	# N/A
LBA [11]	Trace-based	MC, DIFT, LOCKSET	No	Min-inv	Sim	390%-700%	# N/A
Optimized LBA [12]	Trace-based	MC, DIFT, LOCKSET	No	Min-inv	Sim	2%-327%	# N/A
FADE [30]	Trace-based	Memory & Propagation Tracking	No	Min-inv	Sim	20%-80%	Raw numbers
Partial Monitoring [46]	Trace-based	MC, RC, DIFT, BC	No	Min-inv	Sim	50%	(4%-11%) / (7%)
PUMP [28]	Tag-based	NXD+NWC, DIFT, CFI, MC	Yes	Inv	Sim	∼8%	(47%) / (55%)
Harmoni [26]	Tag-based	MC, RC, DIFT, BC	Yes	Min-inv	RTL Sim	∼1%-8%	(10%) / (110%)
FlexCore [25]	Tag-based	MC, DIFT, BC, SEC	Yes	Min-inv	RTL Sim	5%-44%	(14.6%) / (32.5%)
HDFI [76]	Tag-based	SL Enhancement, Code Ptr Sep, Info Leak Kernel, Stack, and VTable Ptr Prot	Yes	Inv	FPGA	0.94%	# N/A
Nile [23]	Trace-based	Shadow Stack	No	Min-inv	FPGA	0.78%	(26%) / (15%)
REST [74]	Tag-based	Stack & Heap Prot	No	Inv	Sim	2%-25%	# N/A
PHMon (This Work)	Trace-based	Shadow Stack, Fuzzing Info Leak, Debugging	No	Min-inv	FPGA	0.94%	(5%) / (13.5%)

"Inv" = Invasive; "Min-inv" = Minimally-invasive; "# N/A" = Numbers not available; Sim = "Simulation"; Emul = "Emulation"; "MC" = Memory Checking; "RC" = Reference Counting
"BC" = Bounds Checking; "FI" = Fault Isolation; "SEC" = Soft Error Checking; "SEP" = Seperation; "SL" = Standard Library; "Ptr" = Pointer; "Prot" = Protection; "Info" = Information; "Leak" = Leakage

(we demonstrate this capability in Section 5).

Overall, to the best of our knowledge, the existing flexible tag-based monitoring techniques are a subset of an *event-action* monitoring model, where the actions are restricted to tag-propagation and raising an exception (handled by software). In this regard, these tag-based FHMons are reference monitors that enforce memory protection policies. PHMon provides a more comprehensive language for actions. Hence, we can leverage PHMon in a wider range of security applications, not limited as a reference monitor to enforce memory protection policies. An efficient implementation of a tag-based FHMon, such as HDFI, is complementary to PHMon.

In a multi-core system, Log-Based Architectures (LBA) [10, 11] implement trace-based monitors that capture an execution log from a monitored program on one core and transfer the collected log to another general-purpose core, where a dynamic tool (lifeguard) executes and enforces the security policies. The optimized LBA [12] considerably reduces the performance overhead of LBA [11] (from $3\times$-$5\times$ to $\sim$50%) at the cost of higher power and area overheads. From the perspective of the *event-action* monitoring model, LBA's expressiveness in terms of monitoring rules and actions is close to software-based techniques. However, the LBA trace-based monitor suffers from considerable performance, power, and area overheads. Similar to optimized LBA, FADE [30], DISE [16], and partial monitoring [46] apply filtering, pattern matching, and dropping decisions to the execution trace, respectively. Rather than utilizing an additional general-purpose core, PHMon provides a programmable hardware capable of performing a smaller range of monitoring techniques, but does so efficiently and with significantly lower power and area overheads. Among the trace-based FHMons, Nile [23] is the closest work to PHMon. Compared to LBA architectures and PHMon, Nile provides a restricted set of possible actions; however, Nile's actions are not limited to tag propagation. Nile only supports comparison operations (no other arithmetic or logical operations), which restricts its applicability for different use cases.

2.3 Generic Monitoring Hardware Extensions

Modern processors provide hardware features and extensions to collect runtime hardware usage information. Hardware Performance Counters (HPCs) are hardware units for counting the occurrence of microarchitectural events, such as cache hits and misses, at runtime. A number of previous works use HPCs for malware detection [24, 40, 57, 73]. However, recent studies [21, 87] shed light on the pitfalls and challenges of using HPCs for security. Moreover, HPCs are limited to a predefined pool of *microarchitectural* events, while PHMon and FHMons provide a set of monitoring rules to specify cus-

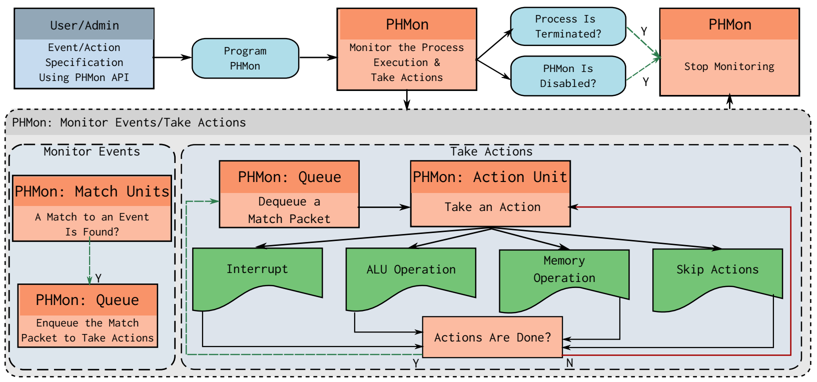

Figure 1: An overview of the *event-action* model provided in PHMon.

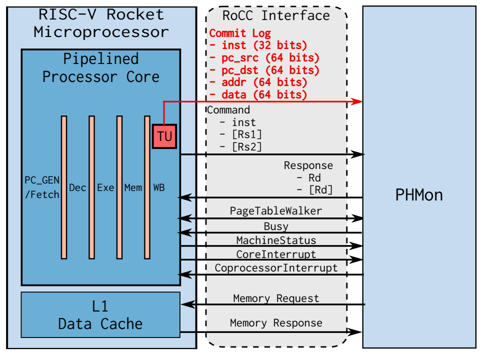

Figure 2: The RoCC interface extended with *commit log* execution trace.

tom events. Additionally, PHMon and FHMons are capable of performing follow-up actions, while HPCs are restricted to interrupts.

Last Branch Record (LBR) is a hardware feature available in the recent Intel processors, which records a history of the 16 most recent indirect `jumps`. Several works [14, 58, 84] rely on LBR, as a pseudo shadow stack, to mitigate Return-Oriented Programming (ROP) attacks. However, history-flushing attacks [9, 72] can evade such LBR-based detection techniques. LBR is not designed for security purposes; hence, it cannot provide a principled security solution. Unlike LBR, PHMon's implemented shadow stack is not limited to maintaining only the last 16 branch records (the limit for PHMon is the allocated memory size); hence, PHMon is not vulnerable to history flushing attacks.

Modern processors also provide architectural extensions, like Intel Processor Trace (PT) [66] and ARM CoreSight [48], to capture debugging information. Both Intel PT and ARM CoreSight provide enormous debugging capabilities; however, these technologies are primarily designed to provide debugging traces for *post-processing*. Online processing capabilities, however, are essential for the timely detection of security threats. FHMons and PHMon expand the online monitoring with efficient online processing and prevention capabilities. Although Intel PT is designed for offline debugging and failure diagnosis, recent techniques [29, 31, 39] utilize this hardware extension to enforce Control Flow Integrity (CFI) at runtime. Similarly, kAFL [71] is a kernel fuzzing engine that uses Intel PT to obtain code coverage information.

3 Threat Model and Assumptions

In this work, we focus on detecting software security vulnerabilities and preventing attackers from leveraging these vulnerabilities. We follow the common threat model among the related works. We assume software may include one or more security bugs and vulnerabilities that attackers can leverage to perform an attack. We do not assume any restrictions about what an attacker would do after a successful attack. Specifically for our use cases, we assume an application may suffer from a security vulnerability such as buffer overflow and an attack can leverage that to gain the control of program's stack. Also, motivated by our information leakage prevention use case, we assume that sensitive memory contents can be leaked to unauthorized entities.

Since PHMon relies on OS support, we assume that the OS kernel is trusted. However, in principle, PHMon can be extended to protect (part of) the OS kernel. Section 7.2 provides a more detailed discussion about PHMon's capabilities and limitations in protecting the OS kernel. Also, we assume all hardware components are trusted and bug free. Hence, hardware-based attacks such as row hammer [41] and cache-based side-channel attacks are out-of-scope of this work.

As mentioned before, for security enforcement use cases, we can consider PHMon as a reference monitor [4, 70]. A reference monitor should satisfy three principles: complete mediation, tamperproofness, and verifiability. PHMon satisfies the complete mediation principle. Whenever a context switch into a monitored process occurs, PHMon continues monitoring. Additionally, PHMon monitors the execution of the forked processes of a parent process. Regarding tamperproofness, as we will discuss in Section 4.2, PHMon provides the option of "sealing" configurations to prevent further modifications. With respect to verifiability, PHMon is *small enough* to be subject to verification (13.5% area overhead compared to an in-order processor).

4 PHMon

We propose a minimally-invasive programmable hardware monitor (for a general-purpose processor) to enforce an *event-action* monitoring model. Figure 1 presents a high-level overview of PHMon that implements such an *event-action* monitoring model. To enable per process monitoring, software API (to configure/program the hardware monitor) and OS support are mandatory. A user/admin can configure the hardware to monitor the execution of one or more processes. Then, the hardware monitor collects the runtime execution information of the processor, checks for the specified events, and performs follow-up actions. Once the process terminates or the user/admin disables the monitoring, the hardware monitor stops monitoring. In the rest of this section, we discuss the challenges associated with designing PHMon and our design decisions to address these challenges. In the next three subsections, we explain the hardware design for PHMon, its software interface, and the OS support for PHMon.

4.1 PHMon: Architecture

In this subsection, we present the hardware design of PHMon. Our main design goal for our hardware monitor is to provide an **efficient** and **minimally invasive** design. According to the *event-action* monitoring model, our hardware monitor should perform three main tasks: collect the instruction execution trace of a processor, examine the execution trace to find matches with programmed events, and take follow-up actions. To perform these tasks, PHMon consists of three main architectural units: a Trace Unit (TU), Match Units (MUs), and an Action Unit (AU).

4.1.1 Trace Unit (TU)

The TU is responsible for performing the first task, i.e., collecting the instruction execution trace. To design our TU, we need to answer the following questions: **what** information should the TU collect, from **where** should it collect this information, and **how** to transfer the collected information to the hardware monitor?

In this work, we only collect information about the architectural state of the processor (not the micro-architectural state). To this end, the TU collects the entire architectural state of the processor using five separate entries, i.e., the undecoded instruction (`inst`), the current Program Counter (PC) (`pc_src`), the next PC (`pc_dst`), the memory/register address used in the current instruction (`addr`), and the data accessed by the current instruction (`data`). The `inst` entry contains the `opcode` as well as the input and output `operand identifiers`. In principle, we can collect this information from different stages of a processor's pipeline (i.e., decode, execute, memory, and write-back stages). We can take advantage of the FIRRTL [45] compiler[1] (via annotations) to extract specific signals with low effort and transfer them to PHMon. To ensure that we monitor the instructions that are actually executed and in the order they are committed, we collect the above-mentioned information from the commit stage of the pipeline. Hence, we call the collected information a *commit log*.

During each execution cycle, the TU collects a commit log and transfers it to our hardware monitor. To prevent stalling the processor's pipeline while PHMon processes each commit log, we design PHMon as a parallel decoupled monitor. Such

[1]FIRRTL is an Intermediate Representation (IR) for digital circuits. The FIRRTL compiler is analogous to the LLVM compiler.

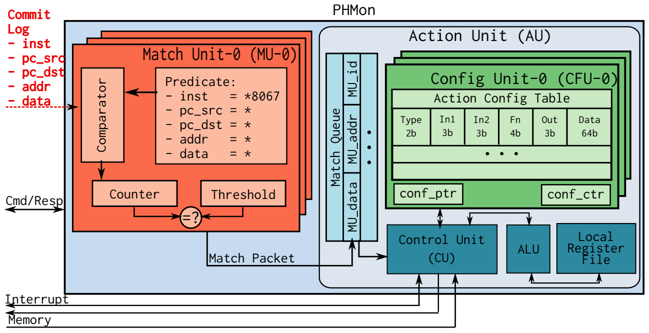

Figure 3: PHMon's microarchitecture.

a decoupled monitor requires an interface to receive the commit log from the processor. In this work, we design PHMon as an extension to the open-source RISC-V Rocket processor [5] via its Rocket Custom Coprocessor (RoCC) interface. RISC-V [83] is an open standard Instruction Set Architecture (ISA). We choose the Rocket processor due to the availability of its RISC-V open ISA and the capability of running the Linux OS on the processor. However, our PHMon design is **independent** of the transport interface and ISA.

Figure 2 depicts the extended RoCC interface used in our design to communicate with the Rocket processor. The RoCC interface provides transmitting/receiving register data for communication, status/exception bits, and direct communication with the memory hierarchy (L1 data cache in our design). We have extended the RoCC interface to carry the commit log trace (shown in red in Figure 2). Since Rocket is an in-order processor, we **minimally** modify the **write-back** stage of the Rocket processor's pipeline to collect the commit log trace.

PHMon receives the commit log, collected by the TU, from the RoCC interface. Then, as shown in Figure 3, PHMon applies the configured monitoring rules to the commit log to detect events (handled by MUs) and performs follow-up actions (managed by the AU). As mentioned before, PHMon is decoupled from the processor and it processes the incoming commit logs one by one. Hence, we need a queuing mechanism to record incoming commit log traces. Rather than placing a queue between the RoCC interface and PHMon, we filter the incoming packets using MUs and only record the matched events in a queue prior to taking actions.

4.1.2 Match Units (MUs)

MUs are responsible for monitoring an incoming commit log and finding matches with programmed events. Each MU is in charge of detecting a distinct event using a set of monitoring rules. An event is specified at **bit-granularity** by a `match entry` and its corresponding `care`/`don't care` mask entry, which are applied on each commit log entry. An MU matches the `care` bits of each `match entry` with the corresponding bits in the commit log entry. As an example, consider a scenario where a user wants to monitor any of the four branch instructions including `BLT`, `BGE`, `BLTU`, and `BGEU`. The user can configure an MU to monitor these four instructions using the following matching condition:

```
BLT, BGE, BLTU, BGEU: inst = 0x00004063; mask bit = 0xffffbf80
```

The matching condition for `inst` evaluates to true when the current instruction is a match with one of the `BLT`, `BGE`, `BLTU`, or `BGEU` instructions. Note that each of these instructions is identified based on the `opcode` and `func3` bits (refer to [83]). For each of the remaining entries of the commit log (i.e., `pc_src`, `pc_dst`, `addr`, and `data`), we set the masking bits to `0xffffffffffffffff`, indicating these fields are `don't cares`. In Section 4.2, we will present our software interface for programming MUs to monitor the target events. Whenever the `predicate` (the logical conjunction of the matches on all the commit log entries) evaluates to true, a counter in the corresponding MU increases. Once the counter reaches a programmed threshold value, the MU triggers an activation signal and sends a `match packet` to the AU. The AU queues the incoming `match packets`, while it performs actions for the packets arrived earlier. To reduce the queuing traffic, an MU filters commit log traces based on the monitoring rules before queuing them.

An MU may be programmed by a user process to monitor only its own execution or by an admin to monitor processes with lower permissions. In both cases, MU configuration becomes part of a process' context and is preserved across context switches by the OS. In Section 6.2, we evaluate the performance overhead caused by preserving PHMon's configuration across context switches.

Although each MU monitors a separate event, PHMon is capable of monitoring a **sequence** of events using multiple MUs communicating through a shared memory space set up by either the OS or the monitored process itself. For example, multiple MUs may all write to or read from the shared memory.

4.1.3 Action Unit (AU)

The AU is responsible for performing the follow-up actions. Our main goal in designing the AU is to provide a **minimal** design that supports a variety of actions including arithmetic and logical operations, memory operations, and interrupts. To this end, we effectively design our AU as a small *microcontroller* with restricted I/O consisting of four microarchitectural components: Config Units (CFUs), an Arithmetic and Logical Unit (ALU), a Local Register File, and a Control Unit (CU). In addition to these four components, the `Match Queue` that records the `match packets` (generated by MUs) is placed in the AU (see Figure 3).

Each MU is paired with a CFU, where the CFU stores the sequence of actions to be executed once the MU detects a match. These programmable actions are in fact the instructions of a small *program* that executes in the AU. The CU performs the sequence of actions via hardware operations (i.e., ALU operations and memory requests) or an interrupt (handled by software). The CU uses the registers in the Local Register File (6 registers in total) to perform the hardware operations. Our AU implementation enforces the *atomic* execution of actions. To this end, the CU executes all of the follow-up actions of one `match packet` before switching to the actions of the next `match packet`.

As part of the actions, the AU can access memory by sending requests to the L1 data cache, a virtually-indexed physically-tagged cache, through the RoCC interface. Hence, all memory accesses are to virtual addresses. The L1 data cache of Rocket processor has an arbiter to handle incoming requests from several agents including the Rocket core and the RoCC interface. Note that the memory hierarchy of Rocket core manages the memory consistency.

In Appendix A, we provide a detailed description about each of the AU's microarchitectural components.

4.2 PHMon: Software Interface

We use RISC-V's standard ISA extensions [83], called `custom` RISC-V instructions, to configure PHMon's MUs and CFUs, as well as to communicate with PHMon. We provide a list of functions that one can use to communicate with PHMon, where each function is accessible by a user-space process, a supervisor, or both. Note that when a user process programs PHMon, then PHMon only monitors that process' execution. When an admin programs PHMon, it can be configured to monitor a specific user process or monitor all user processes. To prevent an unauthorized process from reconfiguring PHMon (after an MU and its paired CFU are configured), we provide an optional feature to stop any further configuration. To this end, we leverage the Rocket's privilege level (*MStatus.priv*) provided to PHMon through the RoCC interface. According to the privilege level, PHMon permits or blocks incoming configuration requests.

4.3 PHMon: OS Support

In this section, we discuss the necessary modifications to the Linux OS kernel to support PHMon. We categorize our modifications into two classes: per process modifications and interrupt handling modifications.

4.3.1 Per Process OS Support

We extend Linux to support PHMon and provide a complete computing stack including the hardware, the OS, and software applications. We provide the OS support for PHMon at the process level. To this end, we alter the `task_struct` in the Linux Kernel to maintain PHMon's state for each process. We store the MUs' counters, MUs' thresholds, the value of local registers, and CFUs' configurations as part of the `task_struct` (using the `custom` instructions for reading PHMon register values).

We modify the Linux kernel to initialize the PHMon information before the process starts its execution. Once PHMon is configured to monitor a process, we enable a flag (part of the `task_struct`) for that process. Our modified OS allocates a shared memory space for communication between MUs. After allocation, the OS maintains the base address and the size of the shared memory as part of the PHMon information for the process in the `task_struct`. Additionally, the OS sends the base and size values to PHMon. PHMon can simply protect the shared memory from unauthorized accesses, where only the AU and the OS are authorized to access the shared memory. To provide this protection, one of the MUs can monitor any user-space `load` or `store` accesses to this range of memory and trigger an interrupt in case of memory access violation.

During a context switch, the OS reads the MU information (counter and threshold values) as well as the Local Register File information from PHMon and stores them as the PHMon information of the `previous` process in the `task_struct`. Before the OS context switches to a monitored process, it reads the MU information of the `next` process and writes it to PHMon registers using the functions provided in the PHMon API. Note that to retain the atomicity of the programmed actions, our modifications to the OS delay a context switch until the execution of the current set of actions and the corresponding actions of all the `match packets` stored in the `Match Queue` are completed. It is worth mentioning that our current implementation of PHMon is not designed for real-time systems. Hence, we currently do not provide any guarantees for meeting stringent real-time deadlines.

4.3.2 Interrupt Handling OS Support

The OS is responsible for handling an incoming interrupt triggered by the CU. We configure our RISC-V processor to delegate the interrupt to the OS. Additionally, we modify the Linux kernel to handle the incoming interrupts from the RoCC interface. In our security-oriented use case, the OS terminates the process that caused the interrupt based on the assumption that an anomaly or violation has triggered the interrupt. Note that the OS can handle the interrupt in various ways according to the user's requirements (e.g., trapping into GDB for the debugging use case in Section 5.4).

5 Use Cases

PHMon distinguishes itself from related work by its flexibility, versatile application domains, and its ease of adoption. To demonstrate the versatility of PHMon, we present *four* use cases: a shadow stack, a hardware-accelerated fuzzing engine, an information leakage prevention mechanism, and hardware-accelerated debugging.

5.1 Shadow Stack

Our first use case is a shadow stack, a security mechanism that detects and prevents stack-based buffer overflows as well as Return-Oriented-Programming (ROP) attacks. As data on the stack is interleaved with control information such as function return addresses, an overflow of a buffer can violate the integrity of such control information and in consequence compromise system security. A shadow stack is a secondary stack that keeps track of function return addresses to protect them from being tampered with by an attacker. A stack buffer overflow attack occurs when a program writes data into a stack-allocated buffer, such that the data is larger than the buffer itself. ROP is a contemporary code-reuse attack that combines a sequence of so-called gadgets into a ROP-chain. Gadgets typically consist of a small number of instructions ending in a `ret` instruction. However, executing a ROP-chain violates function call semantics (i.e., there are no corresponding `calls` to the `rets` in the chain). A shadow stack can therefore detect ROP attacks.

Rather than providing a dedicated hardware solution (e.g., Intel's proposed shadow stack [67]), we leverage PHMon's flexibility to implement a hardware shadow stack. A shadow stack can easily be realized in PHMon with two MUs. We program one MU (MU0) to monitor `call` instructions and another MU (MU1) to monitor `ret` instructions. Also, we configure each of the MUs to trigger an action for every monitored instance of `call` and `ret` (`threshold = 1`).

The OS allocates a shared memory space, i.e., space for the shadow stack, for each process that is being monitored. Both MUs have access to this shared memory space. We can simply protect this shared memory space against unauthorized accesses by monitoring `load` and `store` accesses to this range of addresses leveraging a third MU (as described in Section 4.3). Any user-space access to this memory space results in an interrupt and termination of the violating process. Once the OS allocates this memory space (during the initialization of a new process), it stores the base address and the size of the allocated memory in the first two general-purpose registers of the Local Register File in PHMon (refer to Appendix A for more information about the Local Register File). We configure the CFUs to use the base address register as the shadow stack pointer. The AU accesses the shadow stack by sending memory requests to the L1 cache using the RoCC interface.

The summary of our *event-action* scenario for implementing a shadow stack is as follows: the first MU (MU0) monitors `calls` and pushes the corresponding `pc_src` value to the shadow stack. The second MU (MU1) monitors `rets` and compares the `pc_dst` value with the value stored on the top of the shadow stack. If there is a mismatch between `calls` and `rets` (e.g., an illegal `ret` address or a ROP attack), PHMon triggers an interrupt and the OS handles the interrupt. In our current implementation, the OS simply terminates the process that caused the interrupt. Note that analogous to [8], we can address `call-ret` matching violations caused by `setjmp/longjmp` by augmenting the `jmp_buf` struct with one more field to store the shadow stack pointer.

5.2 Hardware-Accelerated Fuzzing

Fuzzing is the process of providing a program under test with random inputs with the goal of eliciting a crash due to a software bug. It is commonly used by software developers and security experts to discover bugs and security vulnerabilities during the development of a software product and *mostly* for the deployed software. Big software companies such as Google [2] and Microsoft [68] use fuzzing extensively and *continuously*. For instance, Google's OSS-Fuzz platform found over 1,000 bugs in 5 months [33]. Similarly, American Fuzzy Lop (AFL) [85] is one of the state-of-the-art fuzzers that successfully identified zero-day vulnerabilities in popular programs, such as PHP and OpenSSH.

AFL aims to explore new execution paths in the code to discover potential vulnerabilities. AFL consists of two main units: the fuzzing logic and the instrumentation suite. The fuzzing logic controls the mutation and scheduling of the inputs, and also decides if the current input is interesting enough for further fuzzing. During fuzzing, the instrumentation suite collects branch coverage information of the program for the current input. In the current version of AFL (2.52b), the instrumentation can be applied either at compile time with a modified gcc compiler (afl-gcc) if source is available or at runtime by adding instructions to the native binary through user-mode QEMU for closed-source programs. As QEMU uses DBI, it can instrument each control-flow instruction with the necessary book-keeping logic. While this capability is flexible, DBI comes at a significant performance overhead (2.5$\times$ to 5$\times$ [60]). PHMon can easily monitor the control-flow instructions and apply the necessary book-keeping logic without incurring the DBI overhead. In this study, we do not modify the fuzzing logic of AFL. However, we program PHMon to implement the instrumentation suite.

AFL uses a shared memory region, called `bitmap`, to store the encountered basic block transitions (a basic block is an instruction sequence with only one entry and one exit point) for the program executed with the most recent input. Each basic block has an id, calculated by performing logical and bitwise operations using the current basic block address. The address that points to the transition information in the `bitmap` is calculated based on the current and the previous block id.

We use PHMon as part of AFL as follows (see Figure 4): (1) AFL starts executing the target program on the RISC-V processor. (2) PHMon monitors the control-flow instructions of the target binary. (3) Whenever PHMon detects a control-flow instruction, it updates the `bitmap`. (4) The child process (fuzzed program) terminates. (5) The fuzzing unit compares the output `bitmap` with the global `bitmap` (the collection of the previously observed basic block transitions) and determines whether the current input is interesting enough for further fuzzing.

PHMon conducts step (2) and step (3) of the above-described AFL process. To this end, we program two MUs to monitor the control-flow instructions (`branches` and `jumps`) with `threshold = 1`. Both of these MUs have access to the `bitmap` allocated by AFL. We program each MU with 12 actions to update the `bitmap`.

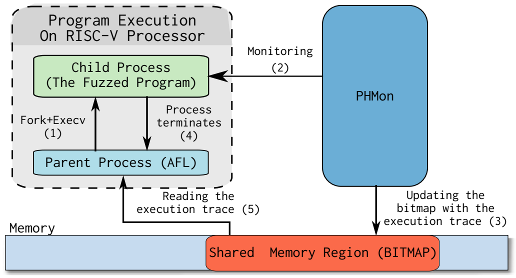
Program Execution On RISC-V Processor
Child Process (The Fuzzed Program)
Parent Process (AFL)
Fork+Execv (1)
Process terminates (4)
Monitoring (2)
PHMon
Reading the execution trace (5)
Updating the bitmap with the execution trace (3)
Memory
Shared Memory Region (BITMAP)

Figure 4: Integration of PHMon with AFL.

5.3 Preventing Information Leakage

PHMon can also be used to prevent the leakage of sensitive information, such as cryptographic keys. A concrete example is Heartbleed [34], a buffer over-read vulnerability in the popular OpenSSL library that allowed attackers to leak the private key[2] of any web-server relying on that library [34].

To prevent Heartbleed, we first identified the memory addresses that contain the private key. Second, we manually white-listed all legitimate read accesses (i.e., instructions that access the key). As legitimate accesses to the key are confined to three functions that implement cryptographic primitives, this was a straightforward task. Finally, we programmed PHMon to trigger an interrupt in case any instruction but those white-listed above accesses the key. To this end, we configure an MU to monitor `load` instructions that access the key, and the CFU contains a series of actions that compare the `pc_src` of the `load` instruction against the white-list. As a proof of concept, we programmed PHMon to prevent the leakage of the prime number p and PHMon successfully prevented the disclosure. Note that the location of sensitive information and its legitimate accesses can vary in different environments. Ideally, the information about the location of an instruction that accesses sensitive data would be produced by a compiler (e.g., by annotating sensitive variables). However, we leave augmenting a compiler tool-chain to produce such meta-information which can be readily enforced by PHMon as future work.

5.4 Watchpoints and Accelerated Debugger

As the last use case, we focus on the debugging capabilities of PHMon. PHMon can provide watchpoints for an interactive debugger, such as GDB, by monitoring memory addresses (`addr` entry of the commit log) and then triggering an interrupt. Although the number of MUs dictates the maximum number of *unique* watchpoints that PHMon can monitor, our watchpoint capability is not limited by the number of MUs. Each MU can monitor a range of monitoring addresses, specified by `match` and `mask` bits. Here, the range of watchpoint addresses can be contiguous or non-contiguous. Additionally, for each range, the user can configure PHMon to monitor read

[2]More precisely, the attack leaks the private prime number p which allows the attacker to reconstruct the private key.

accesses, write accesses, or both by specifying the `inst` entry of the commit log. It is worth mentioning that most modern architectures only provide a few watchpoint registers (e.g., four in Intel x86). We have used and validated the watchpoint capability of PHMon as part of the information leak prevention use case, described in Section 5.3.

In addition to watchpoints, PHMon accelerates the debugging process. As an example, PHMon can provide an efficient conditional breakpoint and trap into GDB. Consider a debugging scenario for a conditional breakpoint in a loop as "`break foo.c:1234 if i==100`", where `i` is the loop counter. Here, we want to have a breakpoint and trap into GDB when the loop reaches its 100^{th} iteration. To this end, PHMon monitors an event where `pc_src` has the corresponding PC value of line 1234. Then, PHMon triggers an interrupt when the MU's `counter` reaches the `threshold` of 100. Subsequently, the interrupt handler traps into GDB. In Section 6.2, we measure the performance improvement of PHMon over GDB for such a conditional breakpoint.

For the debugging use cases, such as watchpoints and conditional breakpoints, the only required action in case of detecting an event is triggering an interrupt. As a result, PHMon is synchronized with the program's execution.

6 Evaluation

In this section, we discuss our approach to validate the functionality of PHMon as well as our evaluation of PHMon using performance, power, and area metrics.

6.1 Experimental Setup

We implemented PHMon as a RoCC (using Chisel HDL [7]) and interfaced it with the RISC-V Rocket processor [5] that we prototyped on a Xilinx Zynq Zedboard evaluation platform [63]. We performed all experiments with a modified RISC-V Linux (v4.15) kernel. We compared the PHMon design with a baseline implementation of the Rocket processor. For both the baseline and PHMon experiments, we used the same Rocket processor configurations featuring a 16K L1 instruction cache and a 16K L1 data cache. Table 2 lists the microarchitectural parameters of Rocket core and PHMon. Note that similar to HDFI [76], we do not include an L2 data cache in our experiments running on Rocket core. Currently, TileLink2 (the protocol that Rocket Chip uses to implement the cache coherent interconnect) does not support L2 cache while the L2 cache in older versions of TileLink is not mature enough [76]. Due to the limitations of our evaluation board, in our experiments, the Rocket Core operated with a maximum frequency of 25 MHz (both in the baseline and PHMon experiments). Note that for our ASIC evaluation, we synthesized the Rocket core with a target frequency of 1 GHz.

For our shadow stack use case, we calculated the run time overhead of 14 applications from MiBench [36], 9 applications (out of 12) from SPECint2000 [37], and 8 applications (out of 12) from SPECint2006 [38] benchmark suites. To measure the performance improvement of our hardware-

Table 2: Parameters of Rocket core and PHMon.

Rocket Core	
Pipeline	6-stage, in-order
L1 instruction cache	16 KB, 4-way set-associative
L1 data cache	16 KB, 4-way set-associative
Register file	31 entries, 64-bit
PHMon	
MUs	2
Local Register File	6 entries, 64-bit
Match Queue	2,048 entries, 129-bit
Action Config Table	16 entries

accelerated AFL, we evaluated 6 vulnerable applications [85] including indent 2.2.1, zstd, PCRE 8.38, sleuthkit 4.1.3, nasm 2.11.07, and unace 1.2b.

To assess power and area, we used Cadence ASIC toolflow for 45nm NanGate process [69] to synthesize PHMon and the Rocket processor to operate at 1 GHz. We then measured the post-extraction power consumption and the area of our system as well as our baseline system, i.e., the unmodified Rocket processor. We considered all memory blocks (both in PHMon and Rocket) as SRAM blocks and used CACTI 6.5 [80] to estimate their power and area.

6.2 Functionality Validation and Performance Results

In this subsection, we validate the functionality of our use cases and evaluate their performance overhead. Additionally, we evaluate the performance overhead PHMon imposes during context switches.

Shadow Stack. We validated the functionality of our shadow stack using benign benchmarks and programs vulnerable to buffer overflow attacks. All benchmark programs ran successfully with the shadow stack enabled resulting in no false detections from PHMon. We developed simple programs vulnerable to the buffer overflow using `strcpy` and exploited this vulnerability.[3] As designed, PHMon detected the mismatches between `calls` and `rets`, triggered an interrupt, and the Linux Kernel terminated the process.

We measured the runtime overhead of our shadow stack on different benchmark applications from MiBench, SPECint2000, and SPECint2006 benchmark suites. We ran each benchmark five times and calculated the average runtime overhead. All standard deviations were below 1.5%. Unfortunately, we were not able to successfully cross-compile and run three of the SPECint2000 benchmarks, i.e., eon, perlbmk, and vortex, for RISC-V. For the rest of the SPECint2000 benchmarks, we used `-O2` for compilation and `reference` input for evaluation (we clarify the exceptions in the results). For SPECint2006 benchmark applications, we used `-O2` for compilation. Considering the limitations of our evaluation board,

[3]We disabled Address Space Layout Randomization (ASLR) to simplify our buffer overflow attack.

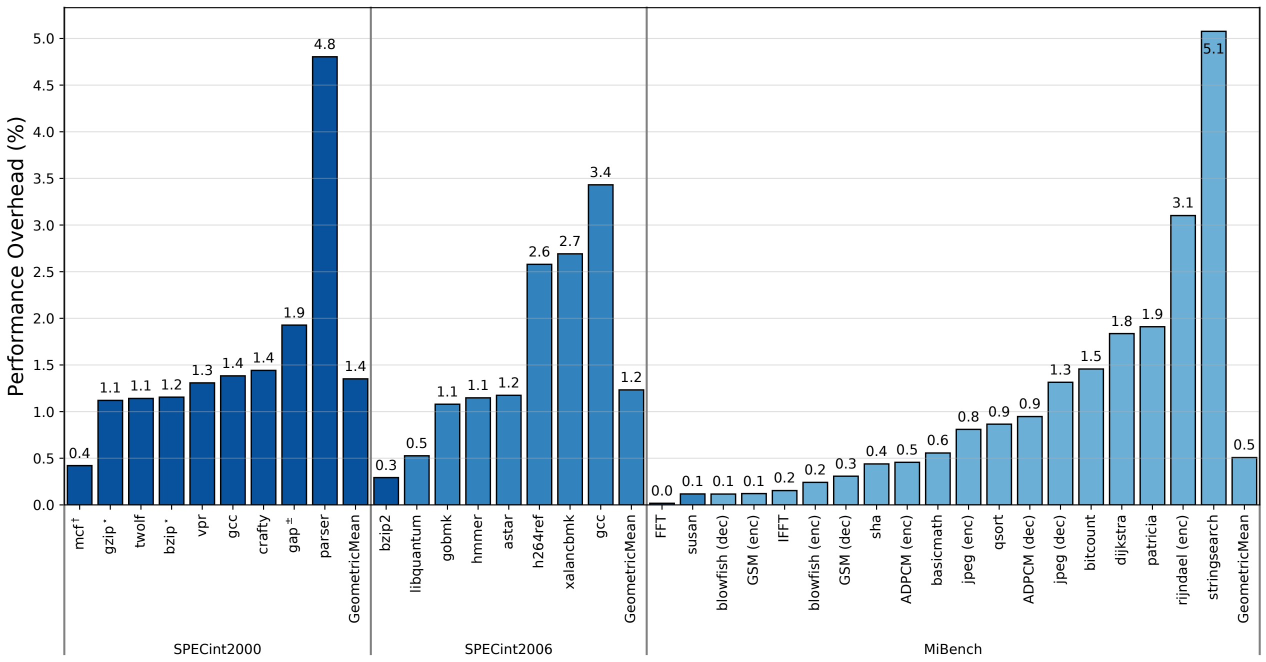

Figure 5: The performance overhead of PHMon as a shadow stack.

† We were not able to run mcf benchmark with `reference` input on our evaluation board; as a result, we used the `test` input for this benchmark.

⋆ Due to the memory limitations of our evaluation board, we had to reduce the buffer size of the `reference` input to 3 MB for gzip and bzip2 benchmarks.

± We had to use `-O0` and an input buffer size of 96 MB to successfully run gap benchmark.

we used the `test` inputs to evaluate SPECint2006. Nevertheless, we were not able to run mcf, sjeng, omnetpp, and perlbench benchmarks mainly due to memory limitations. Figure 5 shows the performance overhead of PHMon as a shadow stack over the baseline Rocket processor. On average, PHMon incurs 0.5%, 1.4%, and 1.2% performance overhead for our evaluated MiBench, SPECint2000, and SPECint2006 applications, respectively. Overall, PHMon has a 0.9% performance overhead on the evaluated benchmarks.

Table 3 (the first three columns) provides a head-to-head comparison for the performance overhead of PHMon-based and HDFI-based shadow stacks. For both PHMon and HDFI, the evaluation baseline is the RISC-V Rocket processor. Unfortunately, HDFI only provides the shadow stack overhead numbers for four SPECint2000 benchmarks [76]. These four benchmarks are cross-compiled for RISC-V using the GCC toolchain. On average, for these four benchmarks, PHMon has a 1.0% performance overhead compared to a 2.1% performance overhead of HDFI.

In the last column of Table 3, we reported the performance overhead of our front-end pass LLVM implementation of a shadow stack. Our LLVM pass instruments the prologue and epilogue of each function to push the original return address and pop the shadow return address, respectively. We used Clang to compile four SPECint2000 benchmarks and used the `reference` input for our evaluations. We only compiled the main executable of SPEC benchmarks (without libraries such as glibc) using Clang. Hence, the implemented front-end pass only protects the main executable. On average, our LLVM plugin has a 5.4% performance overhead.

Table 3: Performance overhead of PHMon-based shadow stack compared to that of HDFI-based (as reported in [76]) and LLVM-based shadow stacks.

Benchmark	PHMon	HDFI	LLVM Plugin
gzip	1.12%⋆	1.12%	2.24%⋆
mcf	0.42%†	1.76%	8.42%†
gap	1.92%±	3.34%	12.30%±
bzip2	1.15%⋆	3.05%	3.66%⋆

⋆ Similar to HDFI, due to the memory limitations of our evaluation board, we had to reduce the buffer size of the `reference` input to 3 MB for gzip and bzip2 benchmarks.

± We used `-O0` for PHMon and `-O2` for LLVM and an input buffer size of 96 MB to run gap.

† Due to memory limitation of our evaluation board, we used `test` input for mcf benchmark.

The main source of performance overhead for PHMon is an increase in the number of memory accesses. Unlike our Rocket processor configuration, in a realistic deployment, the processor would at least include an L2 data cache. Hence, we expect PHMon's performance overhead to be lower in a realistic deployment, which alleviates the significant performance overhead caused by a cache miss.

To put PHMon's performance overhead into perspective, Table 4 compares PHMon's overhead with that of other state-of-the-art software and hardware shadow stack implementations. To facilitate this comparison, we have only listed the implementations that measure their performance overhead on SPEC benchmarks. As an overall criterion, the average overhead of a technique should be less than 5% for getting adopted by industry [79], which PHMon's shadow stack im-

Table 4: Performance overhead of previous software and hardware implementations of shadow stack compared with PHMon.

Mechanism	Methodology	Performance Overhead
[79]	Software (LLVM plugin)	5% on SPEC2006
[1]	Software (binary rewriting)	21% on SPEC2000 (CFI + ID check)
[17]	Software (binary rewriting)	20.53% on SPEC2000 (encoding) 53.60% on SPEC2000 (memory isolation)
[22]	Software (Pin tool)	2.17× on SPEC2006
[75]	Software (DynamoRIO)	18.21% on SPEC2000
[86]	Software (static binary instrumentation)	18% on SPEC2006
[20]	Software	3.5% on SPEC2006
[56]	Hardware	∼0.5%-∼2.4% on SPEC2000
[49]	Hardware	0.24% on SPEC2006
[76]	Hardware	2.1% on SPEC2000
PHMon	Hardware	1.4% on SPEC2000, 1.2% on SPEC2006

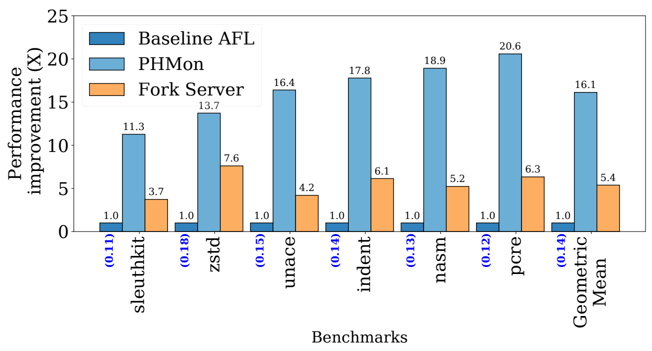

Figure 6: Performance improvement of PHMon over the baseline AFL compared to fork server AFL. The numbers below the "Baseline AFL" bars show the number of executions per second for the baseline AFL.

plementation satisfies.

Hardware-Accelerated Fuzzing. To fuzz RISC-V programs, we integrated AFL into the user-mode RISC-V QEMU version 2.7.5. We fuzzed each of the 6 vulnerable programs for 24 hours using QEMU on the Zedboard FPGA. To provide a fair comparison, for the PHMon-based AFL experiments, we fuzzed each of these programs for the same number of executions as in the QEMU experiments. Similar to other works in fuzzing [71, 77], we used the number of executions per second as our performance metric. We fuzzed each vulnerable program three times and calculated the average value of performance (all standard deviations were below 1%).

For performance evaluation, we used the user-mode QEMU-based AFL running on the FPGA as our baseline. We also ran the QEMU-based fork server version of AFL as a comparison point for PHMon. Figure 6 shows the performance improvement of the PHMon-based AFL over our baseline compared to the performance improvement of the fork server version of AFL. On average, PHMon improves AFL's performance by 16× and 3× over the baseline and fork server version, respectively. Similar to the baseline AFL, we can integrate PHMon with the fork server version of AFL. We expect this integration to further enhance PHMon's performance improvement of AFL. We validated the correct functionality of the PHMon-based AFL by examining the found crashes. On average, for the 6 evaluated vulnerable programs, PHMon-based AFL and the baseline AFL detected 12 and 11 crashes, respectively, for the same number of executions. The mismatch between the two approaches is due to the probabilistic nature of AFL-based fuzzing. Since PHMon improves the performance of AFL, it increases the probability of finding more unique crashes compared to the baseline.

Detecting Information Leakage. To validate that PHMon detects and prevents confidential information leakage, specifically private key of a server, we reproduced the Heartbleed attack on the FPGA by using OpenSSL version 1.0.1f. We initially sent non-malicious heartbeat messages to the server. As expected, none of these messages resulted in false positives. Next, we sent malicious heartbeat messages to the server to leak information. PHMon successfully detected the information leakage attempt and triggered an interrupt; and then, the OS terminated the process. For the non-malicious heartbeat messages, PHMon has virtually no performance overhead (only once a key is accessed, PHMon performs a few ALU operations).

Watchpoints and Accelerated Debugger. We have used and validated the watchpoint capability of PHMon as part of the information leak prevention use case. Also, we evaluated PHMon's capability in accelerating a conditional breakpoint in a loop. Once the program execution reaches the breakpoint, PHMon triggers an interrupt. We evaluated two scenarios for handling the interrupt, trapping into GDB (`PHMon_GDB`) and terminating the process by generating the core dump file (`PHMon_CoreDump`). Figure 7 shows the activation time of the breakpoint over the loop index value for GDB compared to two PHMon-accelerated scenarios. In case of GDB, which uses software breakpoints, each loop iteration results in two context switches to/from GDB, where GDB compares the current value of the loop index with the target value.

For the `PHMon_GDB` case, since PHMon monitors and evaluates the conditional breakpoint, GDB can omit the software breakpoints used in the previous case. Due to the initial overhead of running GDB, `PHMon_GDB` has a similar execution time as GDB for the first breakpoint index ($i = 0$). By increasing the breakpoint index, `PHMon_GDB`'s execution time virtually stays the same while GDB's execution time increases linearly. For the `PHMon_CoreDump` case, since PHMon monitors the conditional breakpoint and generates a core dump (without running GDB), the performance overhead is negligible (i.e., virtually 0). This experiment clearly indicates PHMon's advantage as an accelerated debugger.

Context Switch Performance Overhead. We measured the performance overhead of maintaining PHMon's configuration (including the configuration of MUs and CFUs, the `counter` and `threshold` of each MU, and local registers) across context switches for `mcf` benchmark with `test` input. On average, over three runs, PHMon increases the execution time

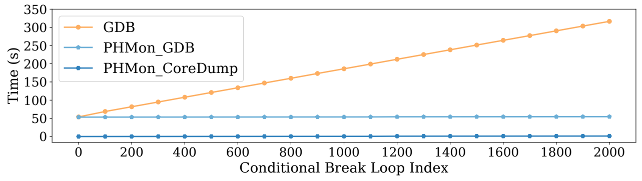

Figure 7: The performance overhead of PHMon compared to GDB for a loop conditional breakpoint.

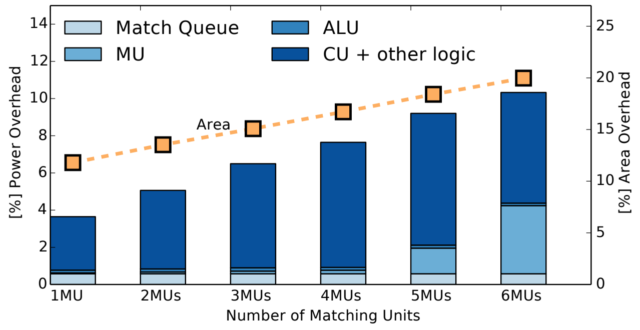

Figure 8: The power and area overheads of PHMon components compared to the baseline Rocket processor.

of a context switch by 4.01%. In total, for mcf benchmark, maintaining PHMon's configuration during context switches takes 0.14 ns, while overall context switches on the baseline processor take 23.80 ns (the total execution time of the process is 5.93 s, where on average 175 context switches happen). The required operation to maintain PHMon's configuration during a context switch is constant. Hence, we expect the performance overhead of PHMon during context switches to be the same for other benchmarks. According to our evaluations for the shadow stack use case, the activation queue is empty before each context switch and there is no need to delay a context switch to complete the remaining actions. However, for different use cases depending on the actions, we might need to delay a context switch to perform the remaining actions.

6.3 Power and Area Results

We measured the post-extraction power and area consumption of PHMon and the Rocket processor using the Cadence Genus and Innovus tools (at 1 GHz clock frequency). In this measurement, we used black box SRAMs for all of the memory components; then, we used CACTI 6.5 to estimate the leakage power and energy/access of memory components. Rocket contains an L1 data cache and L1 instruction cache while PHMon includes a `Match Queue` and `Action Config Table` as the main memory components. In our implementation, the `Match Queue` and each `Action Config Table` consist of 2,048 and 16 elements, respectively. Each `Match Queue` element is 129-bit wide (for a configuration with 2 MUs), while each `Action Config Table` is 79-bit wide. Due to the small size of the `Action Config Table`, its power and area overheads are negligible.

To estimate the dynamic power of the Rocket's L1 caches and PHMon's `Match Queue`, we determined the average memory access rate of these components using PHMon and CSR `cycle` address. We estimated the access rate of the `Match Queue` for two of our use cases,[4] i.e., the shadow stack and the hardware-accelerated AFL, by leveraging PHMon (2 MUs with `threshold=max`) to count the number of `calls` and `rets`, `jumps` and `branches`, and `call` and `branches`, respectively. We averaged the access rates of our two use cases and determined the average dynamic power consumption based on this metric. Figure 8 depicts the total area overhead as well as the power overhead of the main components of PHMon compared to the baseline Rocket processor. There is a trade-off between the number of MUs and the power and area overheads of PHMon. For the number of MUs ranging from 1 to 6, PHMon incurs a power overhead ranging from 3.6% to 10.4%. Similarly, area overhead ranges from 11% to 19.9% as we increase the MU count from 1 to 6. For all of our use cases in this paper, we used a design with only 2 MUs. This design has a 5% power overhead and it incurs a 13.5% area overhead. Table 5 lists the absolute power and area consumed by PHMon's AU and the Rocket core.[5] Our FPGA evaluation shows that a PHMon configuration with 2 MUs increases the number of logic Slice LUTs by 16%.

Table 5: The power and area of PHMon's AU and RISC-V Rocket core determined using 45nm NanGate.

Description	Power ($\mu W/MHz$) @1 GHz	Power ($\mu W/MHz$) @180 MHz	Area (mm^2)
Rocket core	534.3	556.7	0.359
PHMon's AU	43.8	25.0	0.048

7 Discussion and Future Work

In this Section, we address some of undiscussed aspects of PHMon and present our future work.

7.1 Architecture Aspect

As discussed in Section 4, PHMon maintains the incoming match packets in a queue prior to performing follow-up actions. The size of this queue is a design decision, which affects the number of match packets that PHMon can have in flight. We envision that when the queue is full, PHMon can take one of the following actions: 1) PHMon may opt to drop the incoming match packets; 2) PHMon could stall the instruction fetch stage of Rocket's pipeline; 3) PHMon could raise an interrupt, then the OS stays in a sleep state, until a certain number of empty slots are available. In our current prototype, PHMon stalls the pipeline once the queue gets full. For all our experiments, a size of 2KB entries for the queue was sufficient to avoid any stalling.

PHMon performs actions in a blocking manner, i.e., it only performs one action at a time. Although the L1 data cache

[4]The access rate for the other two use cases is negligible.

[5]Note that in 40GPLUS TSMC process, Rocket processor has 0.034 mW/MHz dynamic power consumption and its area is 0.39 mm^2 [44]. Here, we use a non-optimized but publicly available process (45nm NanGate) for power and area measurements.

in Rocket is non-blocking, PHMon blocks the rest of the actions while waiting to receive a memory response. This can increase the run time for performing actions. The evaluation results presented in the paper include the effect of blocking actions. Potentially, we can modify PHMon such that it can perform non-blocking actions. Although such a design will improve the performance, it will increase the complexity and power/area overheads of PHMon.

In this paper, we interface our PHMon with an in-order RISC-V processor. We implement the AU of PHMon as a microcontroller with restricted I/O, which implements a limited hand-crafted 16-bit ISA and provides a safe and restricted domain to take actions. Our developed ISA does not include branches/jumps, i.e., our AU is not Turing complete. This limited processing implementation is useful for preventing security threats. However, if a user requires actions that cannot be implemented by our restricted ISA, the option of triggering an interrupt provides the user with flexibility of executing actions in form of arbitrary programs. Then, PHMon can enforce the programmed security policies on these arbitrary action programs.

In the current implementation, we monitor the committed instruction stream. However, PHMon can apply the same monitoring model using other data streams, e.g., execution information from different stages of the pipeline or cache access information. Applying PHMon to other data streams will require minimal modifications to the processor for collecting the data streams and transmitting them to PHMon.

The number of MUs is another design decision when designing PHMon. The number of MUs directly affects power and area overheads. A user can monitor more events than the available number of MUs by time-multiplexing the MUs (similar to HPCs). Note that several MUs may trigger actions simultaneously; in this case, several `match packets` enter the `Match Queue`, where the MU with the lowest `MU_id` gets the highest priority to enter the queue. The user has an option to set a priority order for MUs. Currently, PHMon does not include a dedicated local memory shared between MUs. For future work, we will include a scratchpad memory or a Content Addressable Memory (CAM) in PHMon to reduce the number of outgoing accesses to the L1 data cache and in turn further reduce the performance overhead.

7.2 Security Aspect

Regarding the security capabilities, in principle, we can extend PHMon to protect (parts of) the OS kernel as well. However, to achieve this protection from an attacker who has compromised the kernel, PHMon must be able to guarantee that an attacker cannot reprogram or disable engaged protections. As PHMon is configured from the kernel, providing such a guarantee is challenging against an adversary who holds the same privilege as the defense mechanism. The same is true for most architecturally supported security features, such as page permissions or Intel's proposed CET. While PHMon can easily be configured to ensure the integrity of configuration information and control instructions, integrity is merely a necessary condition to protect against a kernel-level adversary, it is not sufficient. For example, with integrity intact, attackers can launch mimicry or confused deputy attacks to reprogram PHMon. "Sealing" configurations (as mentioned in Section 4.2) and protecting integrity will raise the bar against kernel-level adversaries, but a complete solution that protects an OS kernel with a kernel-controlled defense mechanism requires further study.

7.3 Application Aspect

The user can leverage multiple MUs to apply several monitoring policies simultaneously. For example, one can use 6 MUs to simultaneously apply all four use cases of PHMon presented in this paper. PHMon enables per process monitoring capabilities; hence, we can reuse an MU to apply different policies based on the requirements of the running process. For example, an MU that is used for debugging of a specific process can be reconfigured to prevent Heartbleed in any other process that is using openssl.

8 Conclusion

We presented the design, implementation, and evaluation of PHMon, a minimally-invasive programmable hardware monitor. PHMon is capable of enforcing a variety of security policies at runtime and also assisting with detecting software bugs and security vulnerabilities. Our PHMon prototype includes a full FPGA implementation that interfaces the monitor with a RISC-V processor, along with the necessary OS and software support. We demonstrated the versatility and ease of adoption of PHMon through *four* use cases; a shadow stack, a hardware-accelerated fuzzing engine, information leak prevention, and a hardware-accelerated debugger. On average, our shadow stack incurs 0.9% performance overhead while our hardware-assisted AFL improves the performance by up to 16×. An ASIC implementation of PHMon with 2 MUs has less than 5% and 13.5% power and area overheads, respectively.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1916393 and CCF-1533663 and a Google Faculty Research award.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC) 13*, 1 (2009).

[2] AIZATSKY, M., SEREBRYANY, K., CHANG, O., ARYA, A., AND WHITTAKER, M. Announcing OSS-Fuzz: continuous fuzzing for open source software. *Google Testing Blog* (2016).

[3] Anati, I., Gueron, S., Johnson, S. P., and Scarlata, V. R. Innovative technology for CPU based attestation and sealing. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[4] Anderson, J. P. Computer security technology planning study. Tech. Report ESD-TR-73-51, *The Mitre Corporation, Air Force Systems Division, Hanscom AFB, Badford*, 1972.

[5] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. The Rocket Chip generator. *Tech. Report, EECS Department, UC Berkeley* (2016).

[6] Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (1994).

[7] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the Design Automation Conference (DAC)* (2012).

[8] Broadwell, P., Harren, M., and Sastry, N. Scrash: a system for generating secure crash information. In *Proceedings of the USENIX Security Symposium* (2003).

[9] Carlini, N., and Wagner, D. ROP is still dangerous: breaking modern defenses. In *Proceedings of the USENIX Security Symposium* (2014).

[10] Chen, S., Falsafi, B., Gibbons, P., Kozuch, M., Mowry, T., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G., and Schlosser, S. Logs and lifeguards: accelerating dynamic program monitoring. *Tech. Report IRP-TR-06-05, Intel Research* (2006).

[11] Chen, S., Falsafi, B., Gibbons, P. B., Kozuch, M., Mowry, T. C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G. R., Lin, B., and Schlosser, S. W. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)* (2006).

[12] Chen, S., Kozuch, M., Strigkos, T., Falsafi, B., Gibbons, P. B., Mowry, T. C., Ramachandran, V., Ruwase, O., Ryan, M., and Vlachos, E. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2008).

[13] Chen, Y.-Y., Jamkhedkar, P. A., and Lee, R. B. A software-hardware architecture for self-protecting data. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2012).

[14] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Robert H., D. ROPecker: A generic and practical approach for defending against ROP attack. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).

[15] Clause, J., Doudalis, I., Orso, A., and Prvulovic, M. Effective memory protection using dynamic tainting. In *Proceedings of the International Conference on Automated Software Engineering (ASE)* (2007).

[16] Corliss, M. L., Lewis, E. C., and Roth, A. DISE: a programmable macro engine for customizing applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2003).

[17] Corliss, M. L., Lewis, E. C., and Roth, A. Using DISE to protect return addresses from attack. *ACM SIGARCH Computer Architecture News 33*, 1 (2005).

[18] Crandall, J. R., and Chong, F. T. Minos: control data attack prevention orthogonal to memory model. In *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2004).

[19] Dalton, M., Kannan, H., and Kozyrakis, C. Raksha: a flexible information flow architecture for software security. *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2007).

[20] Dang, T. H., Maniatis, P., and Wagner, D. The performance cost of shadow stacks and stack canaries. In *Proceedings of the Symposium on Information, Computer and Communications Security (ASIACCS)* (2015).

[21] Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. SoK: the challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of the Symposium on Security and Privacy (S&P)* (2018).

[22] Davi, L., Sadeghi, A.-R., and Winandy, M. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the Symposium on Information, Computer and Communications Security (ASIACCS)* (2011).

[23] Delshadtehrani, L., Eldridge, S., Canakci, S., Egele, M., and Joshi, A. Nile: a programmable monitoring coprocessor. *Computer Architecture Letters (CAL) 17*, 1 (2018).

[24] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. On the feasibility of online malware detection with performance counters. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2013).

[25] Deng, D. Y., Lo, D., Malysa, G., Schneider, S., and Suh, G. E. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2010).

[26] Deng, D. Y., and Suh, G. E. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (2012).

[27] Devietti, J., Blundell, C., Martin, M. M., and Zdancewic, S. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS) (2008).

[28] Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Knight Jr, T. F., Pierce, B. C., and DeHon, A. Architectural support for software-defined metadata processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[29] Ding, R., Qian, C., Song, C., Harris, B., Kim, T., and Lee, W. Efficient protection of path-sensitive control security. In *Proceedings of the USENIX Security Symposium* (2017).

[30] Fytraki, S., Vlachos, E., Kocberber, O., Falsafi, B., and Grot, B. FADE: a programmable filtering accelerator for instruction-grain monitoring. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2014).

[31] Ge, X., Cui, W., and 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).

[32] Ghose, S., Gilgeous, L., Dudnik, P., Aggarwal, A., and Waxman, C. Architectural support for low overhead detection of memory violations. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)* (2009).

[33] Google. OSS-Fuzz: five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html/, 2017.

[34] Graham-Cumming, J. Searching for the prime suspect: how heartbleed leaked private keys. https://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-/leaked-private-keys/, 2015.

[35] Greathouse, J. L., Xin, H., Luo, Y., and Austin, T. A case for unlimited watchpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[36] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC)* (2001).

[37] Henning, J. L. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer 33*, 7 (2000).

[38] Henning, J. L. SPEC CPU2006 benchmark descriptions. *Special Interest Group on Computer Architecture News (SIGARCH) 34*, 4 (2006).

[39] Hu, H., Qian, C., Yagemann, C., Chung, S. P. H., Harris, W. R., Kim, T., and Lee, W. Enforcing unique code target property for control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)* (2018).

[40] Khasawneh, K. N., Ozsoy, M., Donovick, C., Abu-Ghazaleh, N., and Ponomarev, D. Ensemble learning for low-level hardware-supported malware detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2015).

[41] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2014).

[42] Larabel, M. Intel MPX support will be removed from Linux. https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Kernel-Removal-Patch/, 2018.

[43] Larabel, M. Intel MPX support removed from GCC 9. https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9/, 2018.

[44] Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanović, V., and Asanović, K. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *Proceedings of the European Solid State Circuits Conference (ESSCIRC)* (2014).

[45] Li, P. S., Izraelevitz, A. M., and Bachrach, J. Specification for the FIRRTL language. *Tech. Report UCB/EECS-2016-9, EECS Department, UC Berkeley* (2016).

[46] Lo, D., Chen, T., Ismail, M., and Suh, G. E. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2015).

[47] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2005).

[48] Mijat, R. Better trace for better software: introducing the new ARM CoreSight system trace macrocell and trace memory controller. *ARM, White Paper* (2010).

[49] Moon, H. *Hardware techniques against memory corruption attacks*. PhD thesis, Seoul National University, 2017.

[50] Nagarajan, V., Kim, H.-S., Wu, Y., and Gupta, R. Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)* (2008).

[51] Nagarakatte, S., Martin, M. M., and Zdancewic, S. Watchdog: hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2012).

[52] Nagarakatte, S., Martin, M. M., and Zdancewic, S. Watchdoglite: hardware-accelerated compiler-based pointer checking. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2014).

[53] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: highly compatible and complete spatial memory safety for C. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2009).

[54] Newsome, J., and Song, D. Dynamic taint analysis: automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2005).

[55] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. Intel MPX explained: a cross-layer analysis of the Intel MPX system stack. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)* (2018).

[56] Ozdoganoglu, H., Vijaykumar, T., Brodley, C. E., Kuperman, B. A., and Jalote, A. SmashGuard: a hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers (TC) 55*, 10 (2006).

[57] Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. Malware-aware processors: a framework for efficient online malware detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2015).

[58] Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the USENIX Security Symposium* (2013).

[59] Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. Lift: a low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2006).

[60] Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education (WCAE)* (2004).

[61] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: system programming. https://support.amd.com/techdocs/24593.pdf, 2006.

[62] ARM. ARM security technology, building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[63] Digilent's ZedBoard Zynq FPGA. Development board documentation. http://www.digilentinc.com/Products/Detail.cfm?Prod=ZEDBOARD/, 2017.

[64] Intel Corporation. Intel trusted execution technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf, 2006.

[65] Intel Corporation. Introduction to Intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions/, 2013.

[66] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. *System Programming Guide, Part 3C* (2016).

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

[68] Microsoft Corporation. Microsoft security development lifecycle. https://www.microsoft.com/en-us/sdl/process/verification.aspx/, 2017.

[69] Nangate, Sunnyvale, California. 45nm open cell library.

[70] Schneider, F. B. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC) 3*, 1 (2000).

[71] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium* (2017).

[72] Schuster, F., Tendyck, T., Pewny, J., Maass, A., Steegmanns, M., Contag, M., and Holz, T. Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2014).

[73] Singh, B., Evtyushkin, D., Elwell, J., Riley, R., and Cervesato, I. On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)* (2017).

[74] Sinha, K., and Sethumadhavan, S. Practical memory safety with REST. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2018).

[75] Sinnadurai, S., Zhao, Q., and fai Wong, W. Transparent runtime shadow stack: protection against malicious return address modifications. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf, 2008.

[76] Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., and Paek, Y. HDFI: hardware-assisted data-flow isolation. In *Proceedings of the Symposium on Security and Privacy (S&P)* (2016).

[77] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2016).

[78] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004).

[79] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal war in memory. In *Proceedings of the Symposium on Security and Privacy (S&P)* (2013).

[80] Thoziyoor, S., Muralimanohar, N., Ahn, J. H., and Jouppi, N. P. CACTI 5.1. Tech. rep., HPL-2008-20, HP Labs, 2008.

[81] VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2008).

[82] VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y., AND PRVULOVIC, M. Memtracker: efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[83] WATERMAN, A., LEE, Y., PATTERSON, D. A., AND ASANOVIĆ, K. The RISC-V instruction set manual, volume i: Base user-level ISA. *Tech. Report UCB/EECS-2011-62, EECS Department, UC Berkeley* (2011).

[84] YUAN, P., ZENG, Q., AND 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).

[85] ZALEWSKI, M. American fuzzy lop (AFL) fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2017.

[86] ZHANG, M., QIAO, R., HASABNIS, N., AND SEKAR, R. A platform for secure static binary instrumentation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)* (2014).

[87] ZHOU, B., GUPTA, A., JAHANSHAHI, R., EGELE, M., AND JOSHI, A. Hardware performance counters can detect malware: myth or fact? In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)* (2018).

[88] ZHOU, P., QIN, F., LIU, W., ZHOU, Y., AND TORRELLAS, J. iWatcher: efficient architectural support for software debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2004).

[89] ZHOU, P., TEODORESCU, R., AND ZHOU, Y. HARD: hardware-assisted lockset-based race detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)* (2007).

A Appendix

In this appendix, we present the microarchitectural details of PHMon's Action Unit (AU) design. As discussed in Section 4.1, PHMon receives the commit log from the RoCC interface and then PHMon applies the configured monitoring rules to the commit log to detect events and perform follow-up actions. Once an MU finds a match, the MU sends an activation signal alongside a `match packet` to the AU. The match packet consists of an address (`MU_addr`), data (`MU_data`), and an MU identification number (`MU_id`). The `MU_addr` contains the address of the instruction in the commit log (i.e., `pc_src` element), while `MU_data` is programmable and can contain the contents of any one of the commit log entries. The `MU_id` specifies the index of the MU that triggered the activation signal. The AU enqueues an incoming match packet from the MU into the `Match Queue` while it performs actions for the packets arrived earlier. To perform actions, as shown in Figure 3, the AU consists of four distinct microarchitectural components: Config Units (CFUs), Local Register File, Arithmetic and Logic Unit (ALU), and Control Unit (CU). In the next subsections, we explain each of AU's microarchitectural components in detail.

A.1 Config Units (CFUs)

In the PHMon design, each MU is paired with a CFU. Each CFU consists of three main components: an `Action Config Table`, a `conf_ctr`, and a `conf_ptr`. The `Action Config Table` contains the list of actions (programmed by the user) that PHMon should perform after the MU finds a match and triggers the activation signal. The `conf_ctr` and `conf_ptr` preserve the index of the total number of actions and the current action, respectively. Each entry in the Action Config Table, called `action description`, consists of `Type`, `In1`, `In2`, `Fn`, `Out`, and `Data` elements (see Figure 3).

`Type` specifies one of the following four types: *ALU operation*, *memory operation*, *interrupt*, and *skip actions*. In case of an ALU operation, `In1` and `In2` act as programmable input arguments of the ALU whereas for memory operations, `In1` and `In2` are interpreted as `data` and `address` of the memory request. In both cases, `In1` and `In2` can be programmed to hold the local register values (maintained in Local Register File) or an immediate value. The `Out` element specifies where the output of the ALU/memory operation is stored. The `Fn` element determines the functionality of an ALU operation or the type of the memory request. The `Data` element only applies to an ALU operation as immediate data. Note that in case of a memory operation, PHMon sends a memory request through the L1 data cache using the RoCC interface. The *interrupt* action triggers an interrupt, which will be handled by the OS. The *skip actions* provide the option of early action termination. In this case, when the result of an ALU operation is equal to zero, the AU will skip the remaining actions of the current event.

A.2 Local Register File

The Local Register File consists of three dedicated registers for memory requests and their responses: `Mem_addr`, `Mem_data`, and `Mem_resp`, and three general-purpose registers: `Local_1`, `Local_2`, and `Local_3`. Memory operations occur using `Mem_addr` and `Mem_data` registers as the `addr` and `data` of the request while the result gets stored in the `Mem_resp` register. The user can use `Local_1`, `Local_2`, and `Local_3` registers for ALU operations.

A.3 Arithmetic and Logic Unit (ALU)

We include a small ALU in PHMon to support a variety of actions. The ALU operations are restricted inside PHMon; however, these operations can be combined with other PHMon's actions (i.e., memory operations and interrupts) to provide the user with the capability to influence the process' execution. The input and output arguments of our ALU (including `In1`, `In2`, `Fn`, and `Out`) are programmable. The `Fn` argument determines the ALU function out of the following 10 different operations: `Addition`, `Subtraction`, `Logical Shift Left`, `Logical Shift Right`, `Set Less Than`, `Set Equal`, `AND`, `OR`, `XOR`, and `NOP`.

A.4 Control Unit (CU)

The CU handles all the tasks related to performing actions. Our CU consists of a small FSM with three states: `ready`, `wait`, and `busy`. Depending on the current state of the CU, it performs one or more of the following tasks: dequeue a match packet from the `Match Queue`, update the Local Register File, receive the next *action description*, and perform an action. Once all of the listed actions are performed, the CFU notifies the CU. In this case, the CU enters the `ready` state, repeating all of the described tasks for the next element stored in the `Match Queue`.