A Programmable Hardware Monitor for Security of RISC-V Processors

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 is a growing trend in the industry to implement security policies in hardware; however, the current approach to do this is a lengthy and costly process. Additionally, the dedicated hardware security extensions enforce fixed security policies, which cannot evolve as security threats evolve. In contrast to this trend, we propose a minimally-invasive and efficient implementation of a Programmable Hardware Monitor (PHMon) that can enhance and enforce a variety of security policies. We interface our hardware monitor with an open-source RISC-V processor and leverage our design for four different security use cases. We have prototyped our RISC-V processor interfaced with PHMon design on an FPGA, and we have open-sourced our design to the community.[1]**

I. Introduction

In recent years, there has been a growing trend in the industry to implement security policies in hardware. A successful hardware implementation provides a permanent solution against a specific class of security attacks without the need for software patches. Additionally, such hardware solutions typically have considerably lower performance overhead compared to their software counterparts. However, implementing security policies in a new generation of processors is a lengthy and costly process. Moreover, these solutions are typically customized for a specific class of attacks and cannot evolve as security threats evolve.

In contrast to the current industrial trend to implement security policies in customized hardware, a **flexible** hardware monitor can enforce a variety of security policies and can be enhanced as security threats evolve. However, the existing flexible hardware monitors typically suffer from one (or more) of three common drawbacks:

1) A broad class of hardware monitors are tag-based monitors, where each memory address and register is extended with a tag. These tag-based monitors are limited only to enforcing memory protection policies.
2) Some hardware monitors require an idle core on a general-purpose processor to enforce the security policies, which results in high power and area consumption.
3) The implementation of some hardware monitors results in invasive modifications in the processor design, which prohibits their adoption in commercial processors.

To address the aforementioned drawbacks, we propose a minimally-invasive and low-overhead implementation of a Programmable Hardware Monitor (PHMon) [2] interfaced with an open-source RISC-V [4] Rocket processor [1]. PHMon is a trace-based monitor applicable to a wide range of security use cases (not limited to memory protection policies). To evaluate PHMon for real security use cases, we provide a software API and Operating System (OS) support for our monitor and prototype our design (including a full software stack) on an FPGA board. We demonstrate the versatility of PHMon and its ease of adoption by leveraging it for four different security-based use cases: a shadow stack, a hardware-accelerated fuzzing engine, an information leak prevention mechanism, and a hardware-accelerated debugger.

Our FPGA evaluation shows that a PHMon-based shadow stack incurs only 0.9% performance overhead, on average, while our hardware-accelerated fuzzing engine improves the performance by 16× over the state-of-the-art software-based approach. According to our ASIC evaluation, PHMon incurs 5% and 13.5% power and area overhead, respectively.

II. PHMon

PHMon is a minimally-invasive hardware monitor capable of monitoring one or more processes running on a RISC-V processor. To enable per process monitoring, we provide a software API as well as OS support for PHMon. At the hardware level, PHMon monitors the execution of the program, collects the runtime execution trace, checks for specified monitoring patterns, and performs a series of follow-up actions. In this section, we briefly discuss PHMon's architecture and its software support.

A. PHMon: Architecture

PHMon is interfaced with the open-source RISC-V Rocket processor [1] through the RoCC interface. At the hardware level, we minimally modify the write-back stage of the pipeline to collect the runtime execution trace information, called *commit log*. The commit log consists of 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). We transfer the commit log to PHMon through a modified RoCC interface. Figure 1 shows a simplified overview of PHMon's hardware connected to the RISC-V Rocket core through the modified RoCC interface.

In PHMon's architecture, Match Units (MUs) are responsible to monitor each incoming commit log and finding matches with programmed patterns. The user can specify the matching

[1] https://github.com/bu-icsg/PHMon

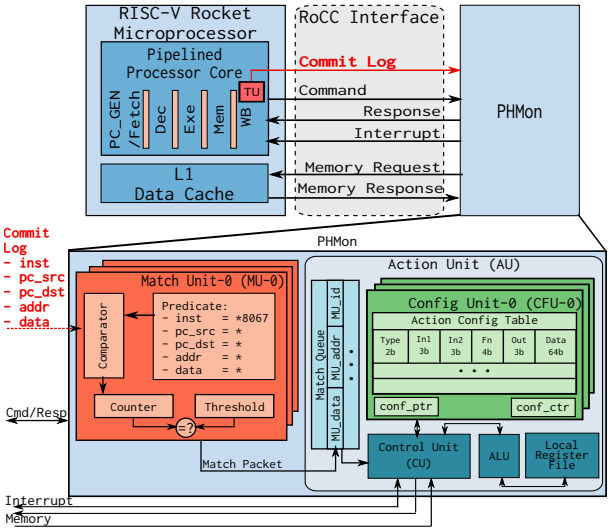

Fig. 1. A simplified overview of PHMon's microarchitecture connected to the RISC-V Rocket core through the modified RoCC interface.

patterns at bit-granularity using PHMon's software API. Once an MU finds a match, the Action Unit (AU) performs a series of programmed actions in the form of arithmetic and logical operations, memory operations, or triggering an interrupt.

B. PHMon: Software Support

Using RISC-V's standard ISA extension, called *custom* instructions, we provide a list of functions written in C for programming PHMon and communicating with it. We also provide a feature to "seal" PHMon's configuration to prevent an unauthorized process from reconfiguring PHMon. To enable per process monitoring capabilities, the OS support for PHMon is mandatory. To provide this support, we modify the `task_struct` in the Linux kernel to maintain PHMon's state for each process. Also, we delegate the machine-level interrupts of the Rocket core to OS, where we alter the interrupt handler to manage these interrupts.

III. Use Cases

To demonstrate the versatility of PHMon and its ease of adoption, in this section, we briefly describe four use cases, i.e., a shadow stack, a hardware-accelerated fuzzing engine, preventing information leakage, and hardware-accelerated debugging, of PHMon. For a more detailed discussion and evaluation of these four use cases, refer to our USENIX Security 2020 paper [2].

A. Shadow Stack

A shadow stack is a secondary stack that can prevent Return-Oriented Programming (ROP) attacks by keeping track of function return addresses. We can simply program PHMon to act as a shadow stack using two MUs, where one MU monitors `call` instructions and the other MU monitors `ret` instructions. If there is a mismatch between the `call` and `ret` addresses, PHMon triggers an interrupt and the OS terminates the violating process. We evaluated our PHMon-based shadow stack on an FPGA prototype. On average, PHMon only has a 0.9% performance overhead on evaluated applications from MiBench, SPECint2000, and SPECint2006 benchmark suites.

B. Hardware-Accelerated Fuzzing Engine

Fuzzing is an automated software testing techniques that provides a program under test with invalid, unexpected, or random inputs with the goal of finding bugs, crashes, and security vulnerabilities. Big software companies such as Google, Microsoft, and Facebook, use fuzzing constantly and extensively. As our hardware-accelerated fuzzing engine, we integrate PHMon with American Fuzzy Lop (AFL), a state-of-the-art fuzzer. For closed-source programs, AFL uses QEMU as its instrumentation suite. QEMU, which is a software emulator, incurs considerable performance overhead for instrumentation. We simply replace QEMU with PHMon to improve the performance of AFL. Based on our evaluation, PHMon improves AFL's performance by 16× over QEMU-based implementation of AFL.

C. Preventing Information Leakage

Preventing information leakage is another use case of PHMon. As a concrete example, we leverage PHMon for preventing Heartbleed [3], a buffer over-read vulnerability in the popular OpenSSL library. The Heartbleed vulnerability allowed attackers to leak the private key of any web-server relying on the OpenSSL library. To prevent Heartbleed, we identify the memory addresses containing the private key and manually white-list all legitimate read accesses. We program PHMon to trigger an interrupt for any illegitimate accesses (any accesses other than white-listed ones). Our PHMon prototype shows that we can prevent a Heartbleed attack with negligible performance overhead.

D. Watchpoints and Accelerated Debugging

PHMon can provide watchpoints by monitoring (a range of) memory addresses and trapping into GDB by triggering an interrupt. PHMon can also accelerate the debugging process. For example, PHMon can provide an efficient conditional breakpoint and trap into GDB.

IV. Conclusion

In this paper, we discussed PHMon, a programmable Hardware monitor with a full software stack. The flexible design of PHMon enables us to use it in a wide range of security use cases. We demonstrate the flexibility of PHMon through four use cases: a shadow stack, a hardware-accelerated fuzzing engine, information leak prevention, and a hardware-accelerated debugger. PHMon has a 5% power and 13.5% area overhead while it incurs low performance overhead.

References

[1] Krste Asanović et al. The Rocket Chip generator. *Tech. Report, EECS Dep., UCB*, 2016.

[2] Leila Delshadtehrani et al. Phmon: a programmable hardware monitor and its security use cases. In *Proc. USENIX Security*, 2020.

[3] John Graham-Cumming. 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.

[4] Andrew Waterman et al. The RISC-V instruction set manual, volume i: Base user-level ISA. *Tech. Report UCB/EECS-2011-62, EECS Department, UC Berkeley*, 2011.