Boston University

OpenBU http://open.bu.edu

Boston University Theses & Dissertations Boston University Theses & Dissertations

2021

Enabling software security mechanisms through architectural support

https://hdl.handle.net/2144/42594

Downloaded from DSpace Repository, DSpace Institution's institutional repository

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

ENABLING SOFTWARE SECURITY MECHANISMS THROUGH ARCHITECTURAL SUPPORT

by

LEILA DELSHADTEHRANI

B.S., Sharif University of Technology, Iran, 2011
M.S., Sharif University of Technology, Iran, 2013

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2021

© 2021 by
LEILA DELSHADTEHRANI
All rights reserved

Approved by

First Reader

Ajay J. Joshi, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

Manuel Egele, PhD
Assistant Professor of Electrical and Computer Engineering

Third Reader

Gianluca Stringhini, PhD
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Tali Moreshet, PhD
Senior Lecturer & Research Assistant Professor of Electrical and Computer Engineering

It is not our job to identify the secret of the red rose
Maybe our job is to dive into the magic of the red rose
And camp behind the wisdom
Wash our hand in the glory of a green leaf and go our way
We shall be born in the mornings when the sun rises
Shall let the excitement fly
Shall sprinkle moisture on understanding, space, color, voice, window and flower
Letting the sky sit between the pronunciation of being
Letting our lungs fill up and empty from eternity
Letting the load of knowledge down from the shoulders of the blue jay
Taking the name back from the fog, evergreen, mosquito and summer
Let us rise up to the height of kindness on the wet legs of rain
To open the doors for human, light, insect and tree
Maybe this is our job, to be after the song of the truth between the century and the morning glory

– Sohrab Sepehri (1964)

Acknowledgments

First, I would like to express my appreciation and gratitude to my advisor, Prof. Ajay Joshi, for his continuous support and guidance throughout my PhD. I am also deeply thankful to my co-advisor, Prof. Manuel Egele, for his invaluable advise and guidance. His guidance made me a more resilient researcher. I would like to thank Prof. Jonathan Appavoo and Prof. Michel Kinsy for their guidance during the earlier stages of my PhD research. Additionally, I like to extend my gratitude to the rest of my thesis committee members, Prof. Gianluca Stringhini and Prof. Tali Moreshet for their precious time, generous support, and insightful feedback.

I am very grateful to all my collaborators and co-authors specially Sadullah Canakci, Dr. Boyou Zhou, and Dr. Schuyler Eldrigde. I specially thank Dr. Schuyler Eldrigde for being a mentor for me during my first year of PhD and shaping my view of PhD. I would like to thank all of my friends in ICSG research group, Peac Lab research group, CAAD research group, and SeclaBU research group. I truly appreciate the long-lasting friendships with all my friends at Boston University specially with Rushi, Qingqing, Onur, Marcia, Prachi, Aselya, Zahra, and Pouya.

Finally, I want to thank my parents for their unconditional love and support and for being the source of inspiration and encouragement my whole life. Although I have not been able to visit my parents during my PhD, they never stopped supporting me and they continued being my rock through my PhD journey.

ENABLING SOFTWARE SECURITY MECHANISMS THROUGH ARCHITECTURAL SUPPORT

LEILA DELSHADTEHRANI

Boston University, College of Engineering, 2021

Major Professors: Ajay J. Joshi, PhD
Associate Professor of Electrical and Computer Engineering

Manuel Egele, PhD
Assistant Professor of Electrical and Computer Engineering

ABSTRACT

Over the past decades, there has been a growing number of attacks compromising the security of computing systems. In the first half of 2020, data breaches caused by security attacks led to the exposure of 36 billion records containing private information, where the average cost of a data breach was $3.86 million. Over the years, researchers have developed a variety of software solutions that can actively protect computing systems against different classes of security attacks. However, such software solutions are rarely deployed in practice, largely due to their significant performance overhead, ranging from ∼15% to multiple orders of magnitude. A hardware-assisted security extension can reduce the performance overhead of software-level implementations and provide a practical security solution. Hence, in recent years, there has been a growing trend in the industry to enforce security policies in hardware. Unfortunately, the current trend only implements dedicated hardware extensions for enforcing fixed security policies in hardware. As these policies are built in silicon, they cannot be updated at

the pace at which security threats evolve.

In this thesis, we propose a hybrid approach by developing and deploying both dedicated and flexible hardware-assisted security extensions. We incorporate an array of hardware engines as a security layer on top of an existing processor design. These engines are in the form of Programmable Engines (PEs) and Specialized Engines (SEs). A PE is a minimally invasive and flexible design, capable of enforcing a variety of security policies as security threats evolve. In contrast, an SE, which requires targeted modifications to an existing processor design, is a dedicated hardware security extension. An SE is less flexible than a PE, but has lower overheads.

We first propose a PE called PHMon, which can enforce a variety of security policies. PHMon can also assist with detecting software bugs and security vulnerabilities. We demonstrate the versatility of PHMon through five representative use cases, (1) a shadow stack, (2) a hardware-accelerated fuzzing engine, (3) information leak prevention, (4) hardware-accelerated debugging, and (5) a code coverage engine.

We also propose two SEs as dedicated hardware extensions. Our first SE, called SealPK, provides an efficient and secure protection key-based intra-process memory isolation mechanism for the RISC-V ISA. SealPK provides higher security guarantees than the existing hardware extension in Intel processors, through three novel sealing features. These features prevent an attacker from modifying sealed domains, sealed pages, and sealed permissions. Our second SE, called FlexFilt, provides an efficient capability to guarantee the integrity of isolation-based mechanisms by preventing the execution of various instructions in untrusted parts of the code at runtime.

We demonstrate the feasibility of our PE and SEs by providing a practical prototype of our hardware engines interfaced with a RISC-V processor on an FPGA and by providing the full Linux software stack for our design. Our FPGA-based evaluation demonstrates that PHMon improves the performance of fuzzing by 16× over the

state-of-the-art software-based implementation while a PHMon-based shadow stack has less than 1% performance overhead. An isolated shadow stack implemented by leveraging SealPK is 80× faster than an isolated implementation using mprotect, and FlexFilt incurs negligible performance overhead for filtering instructions.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Flexible Hardware Support for Security . . . 3

1.1.2 Dedicated Hardware Support for Security . . . 5

1.2 Thesis Contribution . . . 8

1.3 Organization . . . 11

2 Background and Related Work **12**

2.1 Background . . . 12

2.1.1 RISC-V Instruction Set Architecture . . . 12

2.1.2 Privilege Mode . . . 13

2.1.3 RISC-V Virtual-Memory Systems . . . 15

2.1.4 Physical Memory Protection (PMP) . . . 15

2.2 Related Work . . . 16

2.2.1 Hardware-assisted Features for Security . . . 16

2.2.2 Intra-Process Memory Isolation . . . 23

2.2.3 Runtime Instruction Filtering . . . 26

3 A Programmable Hardware Monitor for Security **30**

3.1 Threat Model . . . 30

3.2 PHMon: Design . . . 32

3.2.1 Architecture . . . 32

3.2.2 PHMon: Software Interface . . . 39

3.2.3 PHMon: OS Support 42
3.3 PHMon: Use Cases 43
3.3.1 Shadow Stack 44
3.3.2 Hardware-Accelerated Fuzzing 45
3.3.3 Preventing Information Leakage 48
3.3.4 Watchpoints and Accelerated Debugger 49
3.3.5 Code Coverage Engine 50
3.4 Evaluation Framework 51
3.4.1 Experimental Setup 51
3.4.2 Experimental Results 52
3.5 Summary 62
4 Intra-Process Memory Isolation **63**
4.1 Threat Model 64
4.2 SealPK: Design 64
4.2.1 Hardware Design 64
4.2.2 OS Support 66
4.2.3 Sealing Features 68
4.3 SealPK: Case Study 73
4.4 Evaluation 74
4.4.1 Experimental Setup 74
4.4.2 Experimental Results 75
4.5 Summary 77
5 Runtime Instruction Filtering **79**
5.1 Threat Model 79
5.2 FlexFilt: Design 80
5.2.1 FlexFilt: Hardware Design 81

5.2.2 FlexFilt: OS Support 87
5.2.3 FlexFilt: Software Interface 88
5.3 FlexFilt: Case Study 90
5.3.1 JIT compilation 90
5.3.2 V8 JIT Compilation Experiment 90
5.4 Evaluation 92
5.4.1 Experimental Setup 93
5.4.2 Experimental Results 93
5.5 Discussion 98
5.6 Summary 99

6 Conclusion and Future Work **100**
6.1 Summary of Major Contributions 100
6.2 Limitations 102
6.3 Future Research Directions 102
6.3.1 Extending Our Array of Security Engines to Other Computing Systems 103
6.3.2 An Unlimited Number of Memory Protection Domains 106
6.3.3 Fine-grained Protection Domains 108
6.4 Final Remarks 109

References **110**

Curriculum Vitae **126**

List of Tables

2.1 Comparison of previous hardware monitoring techniques. 20

2.2 Comparison of previous works that prevent the execution of target instructions at runtime. 28

3.1 PHMon's RISC-V extension instructions transmitted over RoCC. . . 40

3.2 PHMon's Application Programming Interface (API). 41

3.3 Parameters of Rocket core and PHMon. 52

3.4 List of the benchmark applications used to evaluate AFL. 52

3.5 Performance overhead of PHMon-based shadow stack compared to that of HDFI-based and LLVM-based shadow stacks. 55

3.6 Performance overhead of previous software and hardware implementations of shadow stack compared with PHMon. 55

3.7 The performance overhead of PHMon as a code coverage engine. . . . 60

3.8 The power and area of PHMon's AU and RISC-V Rocket core determined using 45nm NanGate. 62

4.1 The FPGA utilization of SealPK compared to the baseline Rocket core. 77

5.1 FlexFilt's Application Programming Interface (API). 89

5.2 The measured size of executable bytes generated for browsing the Alexa top-10 websites, on average. 92

5.3 Cycle counts for FlexFilt configuration. 95

5.4 Performance overhead of FlexFilt due to maintaining FlexFilt's information during context switches. 96
5.5 The FPGA utilization of the Rocket core enhanced with FlexFilt compared to the baseline Rocket core. 96
5.6 Opcode-based grouping of RV64I instructions. 97

List of Figures

2·1 The RISC-V Rocket Coprocessor (RoCC) interface. 14

2·2 Page Table Entry (PTE) of Sv39 (a) and Sv48 (b). 16

2·3 Simplified overview of how Intel MPK checks the permission bits of a memory access. 25

3·1 An overview of the *event-action* model provided in PHMon. 33

3·2 The RoCC interface extended with *commit log* execution trace. 35

3·3 PHMon's microarchitecture. 35

3·4 The RISC-V *custom* instruction format. 40

3·5 Integration of PHMon with AFL. 47

3·6 The performance overhead of PHMon as a shadow stack. 54

3·7 Performance improvement of PHMon over the baseline AFL compared to the fork server AFL. 57

3·8 The performance overhead of PHMon compared to GDB for a loop conditional breakpoint. 58

3·9 The power and area overheads of PHMon components compared to the baseline Rocket processor. 61

4·1 Modified MMU of the RISC-V Rocket core for SealPK support. . . . 65

4·2 Example scenario for SealPK's sealable features. 69

4·3 High-level view of SealPK's hardware support to seal pkey permissions. 72

4·4 Performance overhead of various LLVM-based shadow stack implementations for SPECint2000, SPECint2006, and MiBench benchmarks. . 76

5·1 The `Flexible Filter` design, applied to a subset of RISC-V branch instructions. 85
5·2 Simplified overview of the modifications to the RISC-V Rocket core to support FlexFilt. 86

6·1 An example of integrating various PEs and SEs to a RISC-V processing tile consisting of an out-of-order RISC-V BOOM processor and an accelerator. 105
6·2 An example of a heterogeneous SoC generated by RocketChip generator and integrated with an array of SEs and PEs. 107

List of Abbreviations

Abbreviation	Meaning
AFL	American Fuzzy Lop
ALU	Arithmetic and Logical Unit
API	Application Programming Interface
ASLR	Address-Space-Layout-Randomization
AU	Action Unit
AXI	**A**dvanced e**X**tensible **I**nterface
BC	Bounds Checking
CAM	Content-Addressable Memory
CET	Control-Flow Enforcement Technology
CFG	Control Flow Graph
CFI	Control Flow Integrity
CFU	Config Unit
CPI	Code Pointer Integrity
CSR	Control Status Register
CU	Control Unit
DBI	Dynamic Binary Instrumentation
DEP	Data Execution Prevention
DIFT	Dynamic Information Flow Tracking
DTLB	Data Translation Lookaside Buffer
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HPC	Hardware Performance Counter
IFC	Information Flow Control
IoT	Internet of Things
IPR	Instruction Protection Register
IR	Intermediate Representations
ISA	Instruction Set Architecture
ITLB	Instruction Translation Lookaside Buffer
JIT	Just-In-Time
LBA	Log-Based Architecture
LBR	Last Branch Record
MMU	Memory Management Unit
MPK	Memory Protection Key
MU	Match Unit
OS	Operating System

PC	Program Counter
PE	Programmable Engine
PHMon	**P**rogrammable **H**ardware **Mon**itor
PMO	Persistent Memory Object
PMP	Physical Memory Protection
PT	Processor Trace
PTE	Page Table Entry
PTW	Page Table Walker
RD	Read Disable
RoCC	Rocket Custom Coprocessor
ROP	Return-Oriented Programming
SE	Specialized Engine
SFI	Software Fault Isolation
SGX	Software Guard Extensions
SoC	System on Chip
SVM	Secure Virtual Machine
TLB	Translation Lookaside Buffer
TU	Trace Unit
TXT	Trusted Execution Technology
VM	Virtual Machine
WD	Write Disable

Chapter 1

Introduction

1.1 Motivation

Over the last decade, the security of an increasing number of computing systems has been compromised. In 2019, there have been 1,743 reported data breaches in the United States, leading to the exposure of 164.68 million sensitive records (Johnson, 2021). Due to the large number of security attacks and data breaches, we will only name a few of the most well-known attacks and breaches in recent years to demonstrate the prevalence of security attacks. In 2014, the Heartbleed vulnerability (Heartbleed, 2020) in the popular OpenSSL cryptography library impacted 24–55% of HTTPS servers in the Alexa Top 1 Million websites (Durumeric et al., 2014). The Heartbleed vulnerability allows attackers to read sensitive information, such as cryptographic keys and login credential, from vulnerable servers. By 2017, nearly 180,000 internet-connected devices were still vulnerable to Heartbleed bug (Wikipedia, 2021). In 2017, a data breach at Equifax leaked the social security numbers of approximately 150 million Americans (Wikipedia, 2017). In 2018, Marriott International announced that between 2014 and 2018 attackers had stolen data on about 500 million customers (Fruhlinger, 2018). The stolen data included a combination of contact information, passport number, travel information, and other personal information. The Spectre (Kocher et al., 2019) and Meltdown (Lipp et al., 2018) attacks in 2019 exploited critical architectural vulnerabilities in modern processors, including both Intel and AMD processors, to leak confidential information such as passwords and

personal data. In 2020, the Zoom application faced various security and privacy issues such as Zoom-bombing, questionable routing, and inadequate encryption (Feldman, 2020; Price, 2020). In April 2020, the founder and CEO of Zoom (Eric S. Yuan) publicly apologized for "falling short of the community's privacy and security expectations" (Yuan, 2020).

Ever since we have been building computing systems, security has been an afterthought. Every year there is a growing number of software patches to protect against reported exploits. The current pattern of enforcing software patches on an exploit-by-exploit basis results in windows of vulnerability, where attackers can compromise the security of unpatched systems. For example, in recent years software patches have been released to fix the Heartbleed bug in the popular OpenSSL cryptography library (Heartbleed, 2020), to protect against Meltdown attacks (Owaida, 2018), to harden software against Spectre attacks (Owaida, 2018), to fix zero-day vulnerabilities in Microsoft Windows (Owaida, 2020; Warren, 2020) (e.g., in Kernel Cryptography Driver and file-sharing protocol), etc.

Over the past three decades, there have been various software-only proposals for enforcing highly desirable security properties, such as Information Flow Control (IFC) (Myers, 1999; Efstathopoulos et al., 2005; Krohn et al., 2007; Zeldovich et al., 2008; Giffin et al., 2012; Xu et al., 2014) and Control-Flow Integrity (CFI) (Abadi et al., 2009; Davi et al., 2012; Kuznetzov et al., 2018), that could protect against potential security attacks. Unfortunately, many of these software-only implementations remain virtually unused in practice due to significant performance overheads, onerous requirements in terms of developer policy specification, programming language requirements, or substantial Operating System (OS) kernel modifications. Compared to a software-only implementation, a hardware-assisted implementation can reduce the performance overhead of enforcing security policies, can be independent of the

higher-level programming languages, and can reduce the burden of policy specifications by largely being transparent to developers. Hence, in order to protect against the growing number of security attacks in an efficient and timely manner, there is a pressing need to build systems where hardware and software work together for security enforcement. The existing hardware-assisted security solutions such as NX bit have helped protect against various security attacks. However, the modern exploitation techniques such as Just In Time (JIT)-based exploitation techniques (Blazakis, 2010; Snow et al., 2013) and code reuse attacks (Shacham, 2007; Buchanan et al., 2008; Checkoway et al., 2010; Bletsch et al., 2011) require stronger and more advanced hardware-assisted solutions than simple solutions like NX bit.

1.1.1 Flexible Hardware Support for Security

In recent years, there has been a growing trend in the industry to provide dedicated hardware-assisted features for security, e.g., Intel Trusted Execution Technology (TXT) (Intel Corporation, 2006), Intel Software Guard Extensions (SGX) (Anati et al., 2013), Intel Memory Protection Extensions (MPX) (Intel Corporation, 2013), Intel Control-Flow Enforcement Technology (CET) (Intel Corporation, 2017), ARM TrustZone (ARM Corporation, 2009), AMD Secure Virtual Machine (SVM) (AMD Corporation, 2006), and many more. This approach of providing dedicated hardware-assisted security extensions, however, suffers from several drawbacks. Implementing new security extensions in a new generation of processors is a time consuming and costly process. It can take up to several years to add and deploy a single hardware-assisted security extension. Given that these dedicated security extensions are built in silicon, any problems in the design or implementation of these extensions requires a fix in the next generation of the processors. For example, Intel introduced MPX (Intel Corporation, 2013) as a hardware-assisted extension for spatial memory safety. As an alternative to software-based approaches, Intel MPX provides new instructions

and registers to enforce memory safety through bounds checking. Software-based techniques, such as Safe-C (1994) (Austin et al., 1994) and SoftBound (2009) (Nagarakatte et al., 2009), existed several years before Intel MPX was announced in 2013 and introduced commercially in late 2015. Unexpectedly, Intel MPX incurred a considerable performance overhead (up to 4× slow down in the worst case) and its supporting infrastructure could not compile/run 3-10% of legacy programs. Due to various Intel MPX problems, GCC, LLVM, and Linux discontinued their support for MPX (Larabel, 2018b; Larabel, 2018a). Subsequently, by 2019 Intel deprecated MPX. The above Intel MPX example shows the lengthy and imperfect process of implementing dedicated hardware-assisted security extensions. Moreover, these extensions cannot evolve with the same pace as security threats.

Unlike dedicated hardware-assisted extensions for security, a **flexible** hardware implementation can enforce and enhance a variety of security policies as security threats evolve. Such a flexible hardware implementation can provide 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. This realistic evaluation environment increases the chance of success for the dedicated hardware features. As shown by prior work (Chen et al., 2006b; Chen et al., 2008; Deng et al., 2010; Deng and Suh, 2012; Dhawan et al., 2015), a flexible hardware to enforce security policies can be designed in the form of a hardware runtime monitor. These flexible hardware monitors can enforce a variety of security policies such as CFI, Bounds Checking (BC), data-race detection, and Dynamic Information Flow Tracking (DIFT) at runtime. However, the existing flexible hardware monitors suffer from various limitations. Most existing flexible hardware monitors (Deng et al., 2010; Deng and Suh, 2012; Dhawan et al., 2015) are only designed for tag-based memory corruption prevention. Some of the flexible hardware monitors (Chen et al., 2006b;

Chen et al., 2008; Lo et al., 2015) have a broader applicability scope but they rely on a separate general-purpose core to enforce the security policies. These techniques incur large overheads (in terms of performance, power, and area) despite leveraging filtering and hardware-acceleration strategies. Additionally, a variety of flexible hardware monitors require invasive modifications to the processor design (Corliss et al., 2003; Dhawan et al., 2015; Song et al., 2016). Such invasive modifications may limit the feasibility of hardware monitor adoption in commercial processors as well as the composition of flexible hardware monitors. These limitations indicate the need for a minimally-invasive and low-overhead implementation of a flexible hardware monitor that can enforce and enhance a variety of security policies.

1.1.2 Dedicated Hardware Support for Security

As discussed in Section 1.1.1, a flexible hardware monitor can efficiently enforce a variety of security policies. One could argue that by incorporating flexible hardware monitors with modern processors, dedicated hardware-assisted security extensions are no longer required. Unfortunately, such an argument does not consider the limitations of flexible hardware monitors. Efficient enforcement of a variety of fine-grained security policies requires dedicated hardware support through targeted modifications to various parts of a processor. A dedicated hardware extension can be more efficient than a generalized flexible hardware monitor.

Intra-process memory isolation is one of the security policies that benefits from dedicated hardware support. With the continuous increase in the number of software-based attacks, there has been a growing effort towards isolating sensitive data and trusted software components from untrusted third-party components. A hardware-assisted intra-process isolation mechanism enables software developers to partition a process into isolated components (domains), and in turn secure sensitive data from untrusted components. However, most of the existing hardware-assisted intra-process

isolation mechanisms in modern processors, such as ARM (ARM Corporation, 2018) and IBM Power (IBM Corporation, 2017), rely on costly kernel-level transactions for switching between trusted and untrusted domains. More recently, Intel proposed a hardware-assisted feature, called Intel Memory Protection Keys (MPK) (Intel Corporation, 2019), to efficiently support intra-process memory isolation by leveraging an unprivileged instruction (`WRPKRU`) for updating the associated permission of a domain. Intel MPK allows the user to create a protection domain by assigning a protection key (pkey) to a group of memory pages. Creating memory protection domains requires modifications to the Memory Management Unit (MMU) as well as the Translation Lookaside Buffer (TLB). The non-privileged `WRPKRU` instruction, which updates the pkey permissions, is fast, does not require a context switch, and does not lead to a TLB flush. However, Intel MPK suffers from security (Vahldiek-Oberwagner et al., 2019; Hedayati et al., 2019; Schrammel et al., 2020) and scalability (Park et al., 2019; Xu et al., 2020) drawbacks. Intel MPK does not prevent a compromised or malicious component from updating permissions using `WRPKRU` instruction. For example, an attacker can exploit an implicit occurrence of `WRPKRU` instruction using control-flow hijacking attacks and in turn elevate the privilege of a protection domain. Additionally, Intel MPK is vulnerable to protection-key use-after-free issue and only provides 16 protection keys. Intel MPK's drawbacks indicate the need for an efficient and secure dedicated hardware support for intra-process memory isolation.

Runtime filtering of security-sensitive instructions also benefits from dedicated hardware support and targeted hardware modifications to an existing processor. To limit the effects of bugs and security vulnerabilities, a variety of security solutions partition the sensitive data and code into isolated components. Researchers have leveraged various techniques including OS-based techniques (Chen et al., 2016; Litton et al., 2016), virtualization-based techniques (Belay et al., 2012; Liu et al., 2015;

Koning et al., 2017), techniques built upon hardware-based trusted execution environments (Frassetto et al., 2017; Azab et al., 2014), and memory protection domains (Vahldiek-Oberwagner et al., 2019; Hedayati et al., 2019; Schrammel et al., 2020), to enforce isolation. To guarantee the integrity of the isolation, the above-mentioned security solutions have to prevent an untrusted component from accessing or modifying the isolated components. For example, the isolation-based mechanisms leveraging Intel MPK (Hedayati et al., 2019; Vahldiek-Oberwagner et al., 2019) propose solutions to prevent an attacker from exploiting the unprivileged `WRPKRU` instruction in untrusted components and modifying a domain's permissions. Similarly, a variety of prior work (Frassetto et al., 2017; Frassetto et al., 2018; Wu et al., 2018; Gu et al., 2020; Zhou et al., 2020; Azab et al., 2014; Azab et al., 2016; Park et al., 2019; Xu et al., 2020) addressed a common challenge, i.e., *preventing the execution of various unsafe instructions in untrusted parts of the code*. Such unsafe instructions could compromise the integrity of the isolation in-place by modifying access permissions, disabling protections, gaining higher privilege, etc.

To prevent the execution of unsafe instructions, previous works have leveraged various approaches such as CFI (Frassetto et al., 2018; Park et al., 2019) and binary scanning and binary rewriting (Hedayati et al., 2019; Vahldiek-Oberwagner et al., 2019; Park et al., 2019; Gu et al., 2020; Wu et al., 2018; Zhou et al., 2020; Azab et al., 2014; Azab et al., 2016). As currently existing CFI solutions (Gu et al., 2017; Ding et al., 2017; Hu et al., 2018; Liu et al., 2017; Ge et al., 2017) have non-trivial performance overhead ($> 10\%$), leveraging CFI to prevent the execution of unsafe instructions is expensive. Additionally, an efficient implementation of binary scanning and binary rewriting for dynamically generated code is challenging. A dedicated hardware feature can efficiently prevent the execution of unsafe instructions at hardware level without the need for binary scanning/binary rewriting.

1.2 Thesis Contribution

Rather than only relying on dedicated hardware-assisted features or flexible hardware support for security, in this thesis we propose a hybrid approach by leveraging both dedicated and flexible hardware-assisted security features. At a broader level, this thesis proposes an approach where security is an added concept layered on top of a hardware design. In other words, we envision security as a hardware library. Towards this end, we propose a methodology that incorporates an array of hardware engines as a security layer on top of existing processor designs. The hardware engines could be in the form of Programmable Engines (PEs) or Specialized Engines (SEs). We envision a PE as a minimally invasive design, capable of enforcing and enhancing a variety of security policies as security threats evolve. An SE provides the hardware support to efficiently enforce security policies for protecting against a specific class of security attacks. Compared to a PE, an SE is less flexible and requires more invasive and targeted modifications to an existing processor design. We present the thesis statement as following:

A hybrid array of programmable and specialized hardware security engines can efficiently enforce a variety of security policies to protect against known threats and also provides the flexibility to enforce new security policies as security threats evolve.

Therefore, we propose to build an array of hardware security engines consisting of PEs and SEs. The main contributions of this PhD research are as follows:

- **A Programmable Hardware Monitor for Runtime Enforcement of Security:** We propose a PE in form of a minimally-invasive and efficient programmable hardware monitor, called PHMon. We identify that a PE can be designed as a hardware runtime monitor. PHMon enforces an event–action monitoring model with programmable monitoring rules and flexible hardware-level

follow-up actions. PHMon can be used in a wide range of applications, including, but not limited to, enforcing a variety of security policies and assisting with detecting software bugs and security vulnerabilities. We interface PHMon with a RISC-V (Waterman et al., 2011) Rocket (Asanović et al., 2016) processor, and we minimally modify the core to expose an instruction execution trace to the hardware monitor. 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 the hardware monitor 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 OS kernel to support PHMon at the process level; hence, PHMon offers the option of enforcing different security policies for different processes. We demonstrate the versatility of PHMon and its ease of adoption through *five* representative use cases: a shadow stack, a hardware-accelerated fuzzing engine, information leak prevention, hardware-accelerated debugging, and a code coverage engine. To evaluate our design in a realistic scenario, we implement a prototype of our design interfaced with a RISC-V Rocket core (Asanović et al., 2016) on an FPGA and provide a full Linux-based software stack for our design. Similarly, we provide an FPGA-based evaluation environment for our proposed SEs. 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.

- **Intra Process Memory Isolation through Sealable Memory Protection Keys:** We propose an SE, called SealPK, for efficient intra-process memory isolation in RISC-V ISA. Similar to Intel MPK, SealPK provides a per-page protection key; however, SealPK addresses the scalability and security limitations of Intel MPK. SealPK supports up to 1024 domains (64× more than Intel MPK) by leveraging the 10 unused bits available in the Page Table Entry (PTE) of each virtual page (RISC-V Sv-39). While Intel MPK does not provide a solution to maintain the integrity of protection domains and their permissions, we propose three novel sealing features to prevent an attacker from modifying sealed domains, their corresponding sealed pages, and their permissions. In particular, our hardware-assisted permission sealing feature enables the software developer to restrict access to the `WRPKRU` instruction within a contiguous range of memory, e.g., a trusted component. Any attempt to execute a `WRPKRU` instruction outside of the specified range would lead to a hardware exception. Hence, this sealing feature efficiently prevents the manipulation of a domain's permission by an attacker. We demonstrate the efficiency of SealPK by leveraging it to implement an isolated shadow stack on our FPGA prototype. Our isolated shadow stack prototype is, on average, 80× faster than an isolated implementation using mprotect across SPECint2000 (Henning, 2000), SPECint2006 (Henning, 2006), and MiBench (Guthaus et al., 2001) benchmarks.

- **Runtime Instruction Filtering for Security:** We observe that a variety of isolation-based security solutions, on various processor architectures, have to prevent the execution of various unsafe instructions in untrusted parts of the code. The previous works are tailored to filter the execution of certain unsafe instructions. This limits the scope of the prior work. We propose an SE, called FlexFilt, for runtime filtering of unsafe instructions at page granularity.

FlexFilt creates instruction protection domains by assigning the same protection key to a group of executable pages. At hardware level, FlexFilt provides configurable filters to prevent the execution of various instructions. Each instruction protection domain can be configured to apply a combination of the configured filters to its corresponding pages, i.e., prevent the execution of various user-defined instructions at page granularity. FlexFilt is an efficient hardware-assisted feature and incurs negligible performance overhead for filtering target instructions at runtime. In addition to filtering user-space instructions, FlexFilt is capable of filtering privileged instructions (i.e., supervisor mode and hypervisor mode). To illustrate the effectiveness of FlexFilt compared to binary scanning approaches, we measure the overhead of scanning JIT compiled bytes generated by V8 JavaScript engine while browsing various webpages.

1.3 Organization

The remainder of this thesis is organized as follows. We review the background and state of the art on RISC-V, flexible hardware monitors, intra-process memory isolation, and runtime instruction filtering in Chapter 2. Chapter 3 presents our work on the design and implementation of a programmable hardware monitor for security. In Chapter 4, we introduce our hardware-assisted feature for enforcing an efficient and secure intra-process memory isolation approach. Chapter 5 presents our work on a hardware engine to efficiently filter user-defined unsafe instructions at runtime. In Chapter 6, we discuss the future directions and conclude this thesis.

Chapter 2

Background and Related Work

In this chapter, we provide a background on RISC-V and review the related work on flexible hardware monitors, intra-process memory isolation, and runtime instruction filtering.

2.1 Background

In this thesis, we leverage the RISC-V open Instruction Set Architecture (ISA) to implement and evaluate our hardware security engines. We choose the RISC-V ecosystem, which provides open-source software and hardware tools. Hence, the RISC-V ecosystem enables us to prototype a processor with the full Linux software stack. In this section, we provide the background information on the RISC-V ISA.

2.1.1 RISC-V Instruction Set Architecture

RISC-V (Waterman et al., 2011) is a free and open ISA, which has been widely used in recent years by both academia and industry for building processors and the surrounding software environment. RISC-V supports 32-bit, 64-bit, and 128-bit address spaces, namely RV32, RV64, RV128, respectively. In this thesis, our focus is on commonly used RV64 processors. RISC-V specifies a base integer instruction set (e.g., RV64I) as well as various standard extensions. RV64G is the standard general-purpose RISC-V ISA for 64-bit address space, where “G” represents the base integer prefix as well as the names of the rest of the general extensions, i.e., RV64IMAFD. Here,

"M" stands for standard extension for integer multiplication and division. "A" represents the standard extension for atomic instructions. "F" and "D" are the standard extensions for single-precision and double-precision floating-point, respectively. The RISC-V ISA also provides a standard extension for compressed instructions ("C"). Additionally, the RISC-V ISA (RV32G and RV64G) dedicates four encodings for custom instruction-set extensions. These instructions, called `custom` instructions (`custom0-custom3`) are reserved for customization and will not be used by future standard extensions. We leverage these RISC-V `custom` instructions to provide a software Application Programming Interface (API), which enables a software developer to configure our hardware security engines.

We interface our hardware engines with the open-source RISC-V Rocket core (Asanović et al., 2016). The Rocket core supports RV64-GC extensions and is capable of booting up Linux kernel. Additionally, Rocket core implements the support for `custom` instructions through the Rocket Custom Coprocessor (RoCC) interface. Figure 2·1 shows the RoCC interface, consisting of various wires and bundles. The RoCC interface (Asanović et al., 2016; Lee, 2015) provides a `command/respond` channel to transfer data to the core's register using `custom` instructions. Additionally, the coprocessor/accelerator can send read/write memory requests to the L1 data cache through the RoCC interface.

2.1.2 Privilege Mode

The RISC-V ISA provides specifications for the unprivileged ISA (Waterman et al., 2019a) as well as privileged ISA (Waterman et al., 2019b). Currently, the RISC-V ISA provides three privilege levels, i.e., user/application, supervisor, and machine modes. The highest level of privilege belongs to the machine mode, which is a mandatory privilege level for any RISC-V core. User mode and supervisor mode separate the execution of application and operating system codes. The RISC-V Rocket core sup-

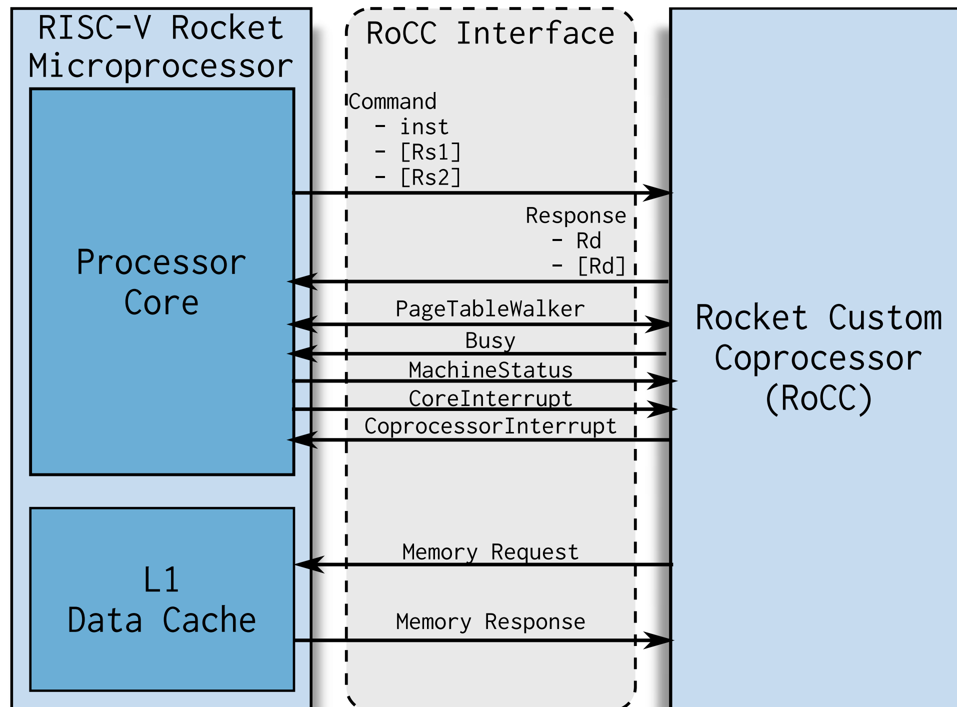


Figure 2·1: The RISC-V Rocket Coprocessor (RoCC) interface (Asanović et al., 2016; Lee, 2015). `command/respond` channel allows data to be transferred from/to the core's register. `command` carries the entire instruction as well as two source registers. `Respond` transfers the value from the coprocessor/accelerator to be written into the destination register. Additionally, the RoCC interface provides a channel to transfer memory requests to the L1 cache and memory responses to the coprocessor/accelerator.

ports all the three privilege modes. At any point, a RISC-V hardware thread runs in one of the privilege levels. This privilege level is encode in one or more Control Status Registers (CSRs).

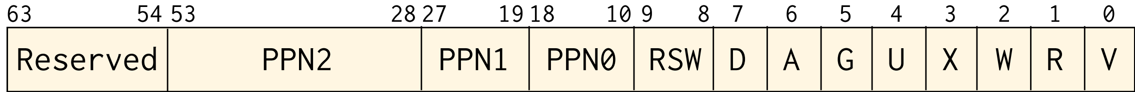
2.1.3 RISC-V Virtual-Memory Systems

For RV64 systems, RISC-V specifies two page-based virtual memory systems, i.e., Sv39 and Sv48. Sv39 and Sv48 provide a 39-bit and a 48-bit virtual address space, respectively, where in both cases the address space is divided into 4KB pages. In addition to 4KB pages, both Sv39 and Sv48 support 2MB *megapages* and 1GB *gigapages.*

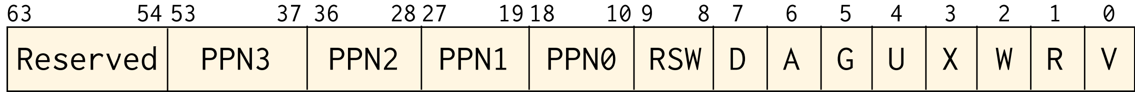
The privilege specification of RISC-V ISA specifies the virtual address translation process. Figure 2·2 shows the Page Table Entry (PTE) format of Sv39 and Sv48. Each PTE holds the mapping between a virtual address of a page and its corresponding address of a physical frame. In Figure 2·2, bits 3-1 are the page permission bits, where `R`, `W`, and `X` bits indicate whether a page is readable, writable, and executable, respectively. As shown in Figure 2·2, the top 10 bits of an Sv39 and Sv48 PTE (bits 63-54) are reserved for future use, e.g., to facilitate research experiments (Waterman et al., 2019b).

2.1.4 Physical Memory Protection (PMP)

In addition to page protection through access permissions stored in PTE, RISC-V ISA specifies the Physical Memory Protection (PMP) capability. PMP provides a per thread view (for each hart) that enables the programmable machine mode to limit the physical addresses that are accessible by software. PMP divides the physical memory address into up to 16 configurable regions, where each region can be configured with specific access permissions. At hardware level, a PMP unit utilizes machine-mode CSRs that allows the memory access permission (read, write, and execute) of each



(a)



(b)

Figure 2·2: Page Table Entry (PTE) of Sv39 (a) and Sv48 (b) (Waterman et al., 2019b). The `V` bit specifies if the PTE is valid. `R`, `W`, and `X` bits indicate whether a page is readable, writable, and executable, respectively. The `U` bit shows whether this page is accessible in user mode. The `G` bit indicates a global mapping, which exists in all address spaces. For leaf PTEs, the `A` bit specifies whether the virtual page has been read, written, or fetched from since the last time this bit was cleared. Similarly, the `D` bit shows whether the virtual page has been written since the last time the `D` bit was cleared.

region to be specified. At runtime, PMP checks are applied to all the accesses in user and supervisor modes. Various previous works (Lee et al., 2020; Kim et al., 2020; Lindemer et al., 2020; Hex Five Corporation, 2020; Karlsson, 2020) leverage PMPs for providing an additional security layer.

2.2 Related Work

2.2.1 Hardware-assisted Features for Security

The first contribution of this thesis is on providing a PE for enforcing a variety of security policies. In this subsection, we discuss the background and related work on flexible hardware features and extensions for security enforcement.

Modern processors provide hardware features and extensions to collect runtime hardware usage information. Hardware Performance Counters (HPCs) are hardware units to count the occurrence of microarchitectural events, such as cache hits and misses, at runtime. A number of previous works use the microarchitectural information collected by HPCs for malware detection (Demme et al., 2013; Tang et al., 2014;

Khasawneh et al., 2015; Ozsoy et al., 2015; Kazdagli et al., 2016; Wang et al., 2016; Singh et al., 2017). However, recent studies (Das et al., 2018; Zhou et al., 2018) shed light on the pitfalls and challenges of using HPCs for security. Moreover, HPCs are limited to a predefined pool of *microarchitectural* events, which prevents the software developer from defining new events.

Last Branch Record (LBR) (Intel Corporation, 2016) is a hardware feature available in the recent Intel processors, which records a history of the most recent N executed branches. Depending on the processor model, N can be 4, 8, 16, or 32. Several works (Cheng et al., 2014; Pappas et al., 2013; Yuan et al., 2015) rely on LBR, as a pseudo shadow stack, to mitigate Return-Oriented Programming (ROP) attacks. However, history-flushing attacks (Carlini and Wagner, 2014; Schuster et al., 2014) can evade such LBR-based detection techniques. LBR is not designed for security purposes; hence, it cannot provide a principled security solution.

Modern processors also provide architectural extensions, like Intel Processor Trace (PT) (Intel Corporation, 2016) and ARM CoreSight (Mijat, 2010), to capture debugging information. Both Intel PT and ARM CoreSight provide enormous debugging capabilities. These technologies are primarily designed to provide debugging traces for *post-processing* while online processing capabilities are essential for the timely detection of security threats. Intel PT efficiently collects the change of flow instructions that cannot be derived statically. Accordingly, PT collects three types of packets in encoded format: 1) TNT packets that record taken or non-taken information (1 bit) for conditional jumps, (2) TIP packets that record the target of indirect control transfers, and (3) FUP packets that record the control flow transfers due to signals and interrupts. Intel PT directly logs the encoded traces into physical memory. To reconstruct the control-flow of the program, we need a software decoder to decode the collected encoded PT packets. Although Intel PT is designed for offline debug-

ging and failure diagnosis, recent techniques (Ding et al., 2017; Ge et al., 2017; Hu et al., 2018) utilize this hardware extension to enforce Control Flow Integrity (CFI) at runtime. Similarly, recent works (Schumilo et al., 2017; Chen et al., 2019) leverage Intel PT to obtain code coverage information for fuzzing. To employ Intel PT for online enforcement of security policies, prior works overcame various challenges. As an example, the prior work had to design custom PT decoders to reduce the significant overhead caused during decoding of PT encoded packets (Schumilo et al., 2017; Hu et al., 2018). In contrast to debugging features, a PE designed for security enables us to efficiently collect the required execution trace information in its original format and enforce the security policies at runtime without the need for decoding the collected traces.

Given the various limitations of the debugging features, rather than leveraging the existing debugging features for enforcing security policies, a variety of previous works provide customized security extensions in hardware. These extensions can be in form of dedicated or flexible hardware monitors. A dedicated hardware monitor is designed for enforcing a specific security policy while a flexible hardware monitor is capable of enforcing various policies. 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 hardware monitor is a subset of a reference monitor. A reference monitor (Anderson, 1972; Schneider, 2000) 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. In this work, we describe a reference monitor using the *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 security policies based on 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× (Luk et al., 2005; Reddi et al., 2004) caused by the dynamic translation process of Dynamic Binary Instrumentation (DBI) tools). Hardware-assisted monitoring techniques reduce this significant overhead (Deng and Suh, 2012; Dhawan et al., 2015; Zhou et al., 2007).

We classify the hardware security 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 2.1 compares different features of the prior 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

Table 2.1: Comparison of previous hardware monitoring techniques.

Mechanism	Monitoring Mechanism	Use Cases	Source Code Requirement	Hardware Modification	Evaluation Methodology	Avg. Performance Overhead	Power/Area Overhead
Hardbound (Devietti et al., 2008)	Tag-based	BC	Yes	Inv	Sim	5%-9%	# N/A
SafeProc (Ghose et al., 2009)	Tag-based	BC	Yes	Inv	Sim	5%	# N/A
Watchdog (Nagarakatte et al., 2012)	Tag-based	BC	Yes	Inv	Sim	15%-25%	# N/A
LIFT (Qin et al., 2006)	SW (DBI)	DIFT	No	SW	SW	∼200%-300%	# N/A
TaintCheck (Newsome and Song, 2005)	SW (Tag-based)	DIFT	No	SW	SW	Avg: # N/A	# N/A
Multi-Core DIFT (Nagarajan et al., 2008)	SW (Threads)	DIFT	No	SW	Sim	48%	# N/A
DIFT (Suh et al., 2004)	Tag-based	DIFT	No	Min-inv	Sim & Emul	1.1%	# N/A
Raksha (Dalton et al., 2007)	Tag-based	DIFT	No	Inv	FPGA	48%	# N/A
FlexiTaint (Venkataramani et al., 2008)	Tag-based	DIFT	Yes	Min-inv	Sim	1%-3.7%	# N/A
MemTracker (Venkataramani et al., 2007)	Tag-based	MC	Yes	Inv	Sim	2.7%	# N/A
DataSafe (Chen et al., 2012)	Tag-based	DIFT	No	Inv	Sim	Avg: # N/A	# N/A
DISE (Corliss et al., 2003)	Binary Rewriting	FI, (De)compress	No	Inv	Sim	Avg: # N/A	# N/A
LBA (Chen et al., 2006b)	Trace-based	MC, DIFT, LOCKSET	No	Min-inv	Sim	390%-700%	# N/A
Optimized LBA (Chen et al., 2008)	Trace-based	MC, DIFT, LOCKSET	No	Min-inv	Sim	2%-327%	# N/A
FADE (Fytraki et al., 2014)	Trace-based	Memory & Propagation Tracking	No	Min-inv	Sim	20%-80%	Raw numbers
Partial Monitoring (Lo et al., 2015)	Trace-based	MC, RC, DIFT, BC	No	Min-inv	Sim	50%	(4%-11%) / (7%)
PUMP (Dhawan et al., 2015)	Tag-based	NXD+NWC, DIFT, CFI, MC	Yes	Inv	Sim	∼8%	(47%) / (55%)
Harmoni (Deng and Suh, 2012)	Tag-based	MC, RC, DIFT, BC	Yes	Min-inv	RTL Sim	∼1%-8%	(10%) / (110%)
FlexCore (Deng et al., 2010)	Tag-based	MC, DIFT, BC, SEC	Yes	Min-inv	RTL Sim	5%-44%	(14.6%) / (32.5%)
HDFI (Song et al., 2016)	Tag-based	SL Enhancement, Code Ptr Sep, Info Leak Kernel, Stack, and VTable Ptr Prot	Yes	Inv	FPGA	0.94%	# N/A
REST (Sinha and Sethumadhavan, 2018)	Tag-based	Stack & Heap Prot	No	Inv	Sim	2%-25%	# N/A
PHMon	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

prevention. In contrast, trace-based monitors have wider applicability than merely memory protection. For example, as listed in Table 2.1, the Log-Based Architecture (LBA) (Chen et al., 2006a; Chen et al., 2006b), which is a trace-based monitor, can assist with data race detection.

Dedicated hardware monitors have been used for a variety of debugging and security applications including hardware-assisted watchpoints for software debugging (Greathouse et al., 2012; Zhou et al., 2004) and hardware-assisted Bounds Checking (BC) (Devietti et al., 2008; Ghose et al., 2009; Nagarakatte et al., 2012). 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 (Nagarajan et al., 2008; Newsome and Song, 2005; Qin et al., 2006) have large performance overheads. To reduce the performance overhead, hardware implementations for DIFT have been proposed (Chen et al., 2012; Dalton et al., 2007; Suh et al., 2004; Venkataramani et al., 2008). These dedicated hardware implementations for DIFT provide different levels of flexibility, from 1-bit tags (Qin et al., 2006) and multi-bit tags (Dalton et al., 2007) to more flexible designs (Chen et al., 2012; Venkataramani et al., 2008).

Flexible hardware monitors provide flexible monitoring capabilities and can be applied to a range of applications. Over the years, a variety of flexible hardware monitors have been proposed. MemTracker (Venkataramani et al., 2007) implements tag-based hardware support to detect memory bugs. Several existing works (Deng et al., 2010; Deng and Suh, 2012; Dhawan et al., 2015) extend DIFT tag-based monitoring into more flexible frameworks capable of supporting different security use cases. PUMP (Dhawan et al., 2015) provides programmable software policies for tag-based monitoring with invasive changes to the processor pipeline. FlexCore (Deng et al., 2010) 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 (Deng and Suh, 2012) 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 (Song et al., 2016) and REST (Sinha and Sethumadhavan, 2018) provide memory safety through data-flow isolation by adding a 1-bit tag to the L1 data cache.

Among the tag-based flexible hardware monitors, HDFI (Song et al., 2016) is the closest work to our envisioned PE (called PHMon) in terms of providing a realistic evaluation environment. HDFI implements a hardware prototype, rather than relying on simulations, and evaluates a full Linux-based software stack on an FPGA. However, 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. 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 flexible hardware monitors are reference monitors that enforce memory protection policies. We envision a more comprehensive language for actions of our PE. Hence, we could leverage PHMon in a wider range of security applications, not limited as a reference monitor to enforce memory protection policies.

In a multi-core system, Log-Based Architectures (LBA) (Chen et al., 2006a; Chen et al., 2006b) 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 (Chen et al., 2008) considerably reduces the performance overhead of LBA (Chen et al., 2006b) (from 3×-5× to ∼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 (Fytraki et al., 2014), DISE (Corliss et al., 2003), and partial monitoring (Lo et al., 2015) apply filtering, pattern matching, and dropping decisions to the execution trace, respectively.

2.2.2 Intra-Process Memory Isolation

The second contribution of this thesis is an efficient and secure hardware-assisted support for intra-process memory isolation. In this subsection, we discuss the related work on intra-process memory isolation. A variety of prior works rely on software-based approaches. A Software Fault Isolation (SFI) technique (Wahbe et al., 1993) instruments each memory access by address masking instructions to prevent unintended memory accesses; however, it suffers from large performance overhead. As the original implementation of SFI does not protect against control-flow hijacking attacks (which might bypass SFI's inserted memory access checks), various SFI-based techniques leverage CFI and binary rewriting to achieve stronger security (McCamant and Morrisett, 2006; Ford and Cox, 2008; Yee et al., 2009; Sehr et al., 2010; Zhao et al., 2011; Deng et al., 2015). However, existing CFI solutions incur non-trivial performance overheads ($> 10\%$).

Hardware approaches can reduce the overhead of isolating the code running within the same virtual address space. CODOMs (Vilanova et al., 2014) and CHERI (Watson et al., 2015) propose efficient capability-based systems, which require significant and invasive hardware modifications. Compared to capability-based architectures, our proposed SE requires less invasive modifications. IMIX (Frassetto et al., 2018) and MicroStach (Mogosanu et al., 2018) enable secure data encapsulation by mini-

mally extending the x86 ISA with secure load and store instructions. However, these mechanisms require CFI to protect against control-flow hijacking attacks.

In recent years, a growing number of modern processors, including ARM (ARM Corporation, 2018), IBM Power (IBM Corporation, 2017) and Intel (Intel Corporation, 2019), have provided a per-page protection key capability, where a group of virtual memory pages form a domain and all pages in the domain are assigned the same protection key (pkey). Intel MPK utilizes 4 previously unused bits of the PTE to specify the pkey of each page and to divide the address space into up to 16 different protection domains. Intel MPK stores the permission bits of all the pkeys in a single 32-bit register (per logical core), called protection key rights register (`PKRU`). The access permission of each pkey is specified using a 2-bit value in the `PKRU`. Accordingly, each pkey specifies a domain as `readable/writable`, `read-only` or `non-accessible`. As shown in Figure 2·3, Intel MPK modifies the Memory Management Unit (MMU) in a way that for each memory access, in addition to checking the page-table permission bits, it checks the corresponding `PKRU` bits of the page's pkey. The intersection of these two checks determines the validity of the memory access. A `PKRU` check is only applicable to memory accesses and not an instruction fetch.

Intel MPK provides two new unprivileged instructions, i.e., `WRPKRU` and `RDPKRU`, to write/read into/from `PKRU`. A user can leverage the `WRPKRU` instruction to update the permission bits of all domains without the need for a context switch. Hence, updating the permission bits of a domain is fast (11-260 cycles (Vahldiek-Oberwagner et al., 2019)); however, `PKRU` is not protected from manipulation by control-flow hijacking attacks (Vahldiek-Oberwagner et al., 2019; Hedayati et al., 2019; Schrammel et al., 2020).

The Linux kernel provides the support for Intel MPK (since v4.6) through three new system calls, i.e., `pkey_alloc`, `pkey_free`, and `pkey_mprotect`. The kernel main-

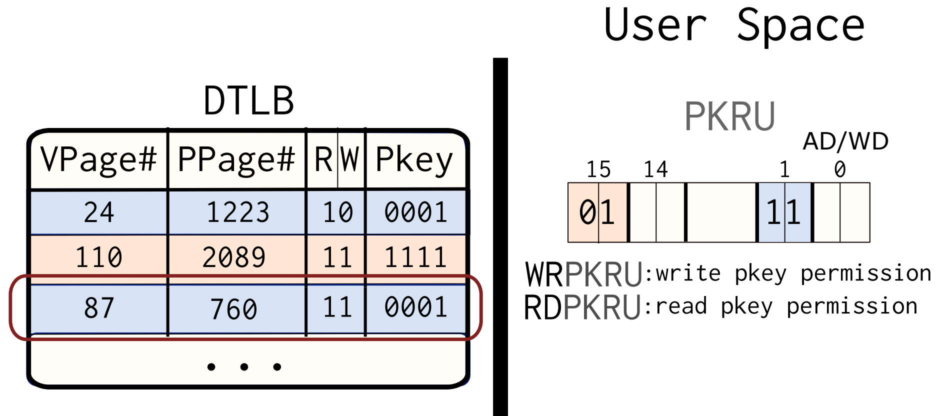


Figure 2·3: Simplified overview of how Intel MPK checks the permission bits of a memory access. The intersection of the page permissions stored in the DTLB and the corresponding permission bits of the protection domain (stored in `PKRU`) specifies the effective access permission.

tains a 16-bit allocation bitmap to keep track of the allocated keys. A user-space thread has to allocate a new pkey using the `pkey_alloc` system call prior to assigning the pkey to a page (group) by invoking the `pkey_mprotect` system call. Using the `pkey_free` system call, the user frees an allocated pkey; however, the kernel only updates the allocation map to indicate that the corresponding pkey is free without erasing the pkey from the PTE of all the corresponding memory pages. The same pkey might be assigned to another domain on future `pkey_alloc` invocations; hence, unintentionally the previous domain would share the same pkey as the new domain, giving rise to the pkey use-after-free problem.

To address Intel MPK's limitations, ERIM (Vahldiek-Oberwagner et al., 2019) and Hodor (Hedayati et al., 2019) combine Intel MPK with binary inspection to prevent reusing of `WRPKRU` instruction by an attacker. The sealing permission feature of SealPK, our SE for intra-process memory isolation, provides a similar capability by restricting valid `WRPKRU` instructions to a contiguous range of memory addresses for each pkey. Although our sealing feature is limited to one valid memory range for each pkey, its simplicity and efficiency distinguishes our work form ERIM and Hodor. Intel

MPK only provides up to 16 protection domains; however, some real-world use cases such as Persistent Memory Object (PMO) (Xu et al., 2020) and OpenSSL (Park et al., 2019) require more than 1000 pkeys. libmpk (Park et al., 2019) and Xu et al. (Xu et al., 2020) provide a software-based and a hardware-based virtualization technique, respectively, to address the limited number of pkeys. These virtualization techniques are complementary to our SE. We can leverage such virtualization techniques to support more than 1024 domains for SealPK. Donky (Schrammel et al., 2020) provides a secure user-space software framework to protect the domain permissions against control-flow hijacking attacks without relying on binary inspection or CFI. Donky proposes a pkey extension for RISC-V ISA, and implements it on the Ariane core (Ariane, 2018). Donky uses the 10 unused bits of Sv39 PTEs to store the pkeys; however, Donky relies on a 64-bit CSR (managed by a software library) to store the permission bits of only 4 pkeys at a time. If the pkey of the accessed memory address is not loaded into that CSR, Donky requires extra cycles for the software library (which stores all the pkey information) to load the missing pkey and its permission into the register. In our SE, we leverage an on-chip memory (rather than a CSR) and access it in the same cycle as page-table permission checks.

2.2.3 Runtime Instruction Filtering

The third contribution of this thesis is on providing an SE for runtime filtering of unsafe instructions. A variety of previous works tackled the challenge of preventing unsafe instructions in untrusted parts of the code. Table 2.2 lists the prior works, their target instructions, and the approaches they used for filtering the instructions. Here, "target" instructions refer to the unsafe instructions that should be filtered.

A recent example of the need for runtime instruction filtering is Intel MPK. To ensure the security of Intel MPK for intra-process memory isolation, it is necessary to prevent an untrusted component from executing `WRPKRU` or `XRSTOR` instructions, which

might elevate the privilege of a protection domain.[1] As shown in Table 2.2, several recent works leveraged different approaches to prevent the unsafe execution of `WRPKRU` (and `XRSTOR`) instructions. As mentioned before, ERIM (Vahldiek-Oberwagner et al., 2019) relies on binary inspection and binary rewriting techniques to prevent an unsafe execution of a `WRPKRU` or an `XRSTOR` instructions. Hodor (Hedayati et al., 2019) leverages binary scanning and hardware watchpoints to prevent the execution of unsafe `WRPKRU` instructions. The virtualization techniques for Intel MPK (Park et al., 2019; Xu et al., 2020) rely on CFI or previous approaches such as ERIM and Hodor to filter unsafe `WRPKRU` instructions. Donky (Schrammel et al., 2020) uses a hardware-assisted call-gate mechanism to secure the domain transitions of MPK, without the need for binary scanning or CFI. Prior to Intel MPK, IMIX (Frassetto et al., 2018) proposed a secure data encapsulation approach by extending the x86 ISA with `SMOV` instruction for secure load and store. IMIX assumes the mitigation approaches such as CFI and Code-Pointer Integrity (CPI) (Kuznetzov et al., 2018) to prevent an attacker from reusing the trusted code containing `SMOV`.

The need to filter target instructions in x86 architecture is not limited to user-space instructions protecting MPK. Fidelius (Wu et al., 2018) and Underbridge (Gu et al., 2020) provide security isolation in hypervisor and kernel space, respectively. Hence, as shown in Table 2.2, Fidelius and Underbridge needed to restrict the execution of privileged target instructions as part of their protection mechanisms. Fidelius proposes a software-based extension to protect the Virtual Machine (VM) against an untrusted hypervisor. Fidelius utilizes binary scanning to restrict the execution of instructions that might hijack the control flow (e.g., `VMRUN`) or switch the address space (e.g., `MOV CR3`). Underbridge retrofits Intel MPK for kernel-space isolation. To

[1] `XRSTOR` restores the full or partial state of a processor's state during a context switch. The `XRSTOR` instruction can modify the contents of the `PKRU` register (which stores the permission bits of all the domains) by setting a specific bit in the `eax` register before executing the instruction (Vahldiek-Oberwagner et al., 2019).

Table 2.2: Comparison of previous works that prevent the execution of target instructions at runtime.

Architecture	Mechanism	Target Instructions	Privilege Level	Filtering Approach
x86	(Vahldiek-Oberwagner et al., 2019)	WRPKRU, XRSTOR	User	Binary inspection and rewriting
	(Hedayati et al., 2019)	WRPKRU	User	Binary scanning and hardware watchpoints
	(Park et al., 2019)	WRPKRU	User	CFI or relying on an approach like ERIM
	(Xu et al., 2020)	WRPKRU	User	Relying on approach like Hodor or ERIM
	(Schrammel et al., 2020)	WRPKRU	User	Hardware-assisted call-gates
	(Frassetto et al., 2018)	Extended instruction (SMOV)	User	CFI
	(Wu et al., 2018)	MOV CR0, MOV CR4, WRMSR, VMRUN, MOV CR3	Supervisor	Binary scanning
	(Gu et al., 2020)	MOV CR3	Supervisor	Binary scanning and rewriting
ARM	(Zhou et al., 2020)	MSR	User	Binary scanning
	(Azab et al., 2014)	LDC, MCR	Supervisor	Binary scanning
	(Azab et al., 2016)	N/A	Supervisor	Binary scanning
RISC-V	(Schrammel et al., 2020)	N/A	User	Hardware-assisted call-gates
	SealPK	Extended instruction (WRPKR)	User	Hardware-assisted instruction filtering
	FlexFilt	Various instructions	User & Supervisor	Hardware-assisted flexible filters

prevent the bypassing of the isolation enforced by MPK, Underbridge leverages binary scanning and rewriting. Subsequently, Underbridge ensures that system servers do not contain any explicit or implicit `CR3` instructions that modify the page table base register.

Researchers have faced the instruction filtering requirement on ARM processors too. Silhouette (Zhou et al., 2020) provides a protected implementation of the shadow stack on embedded ARM processors. Silhouette scans the code to ensure that it does not contain an instruction, such as Move to Special register from Register (`MSR`), that can be used to modify the program state without the need for a store instruction. TZ-PKR (Azab et al., 2014) provides a real-time protection of the OS kernel by leveraging ARM TrustZone (ARM Corporation, 2009). SKEE (Azab et al., 2016) implements

a light-weight framework for a secure kernel-level execution environment on ARM architectures, without relying on a higher privilege layer. To prevent the kernel from executing target privileged instructions, both TZ-PKR and SKEE scan the kernel executables looking for certain control instructions, such as Move to Coprocessor from ARM Register (`MCR`) and Transfer Data from memory to Coprocessor (`LDC`). These instructions are replaced with hooks that jump to a switch gate.

Although RISC-V is a relatively new ISA, Donky (Schrammel et al., 2020) provides the memory protection key capability for RISC-V and leverages hardware-assisted call-gates to secure its implementation. SealPK (our SE for intra-process memory isolation in RISC-V) also implements the memory protection keys for RISC-V. SealPK provides a hardware-assisted feature allowing the software developer to restrict the execution of the `WRPKR` instruction to a contiguous range of memory addresses (e.g., one trusted function). Unlike the flexible design of our proposed SE for instruction filtering (FlexFilt), SealPK's implementation is limited to allowing the execution of a fixed instruction in only one trusted function.

Chapter 3

A Programmable Hardware Monitor for Security

A successful hardware extension to enforce security policies provides an efficient permanent security solution against a specific class of security attacks. However, fixed security policies built in dedicated hardware extensions cannot get updated at the same pace as security threats evolve. In this chapter, we discuss our design of a PE in form of a programmable hardware monitor that can enforce and enhance a variety of security policies as security threats evolve. Such a flexible hardware implementation can also provide 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. Our programmable hardware monitor, called PHMon, can enforce a variety of security policies and it can also assist with detecting software bugs and security vulnerabilities.

3.1 Threat Model

In this chapter, 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. In principle, PHMon can be extended to protect (part of) the OS kernel. 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 CET (Intel Corporation, 2017). 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 we will discuss in Section 3.2.2) 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.

Also, we assume all hardware components are trusted and bug free. Hence, hardware-based attacks such as row hammer (Kim et al., 2014) and cache-based side-channel attacks are out-of-scope of this work. For security enforcement use cases, we can consider PHMon as a reference monitor (Anderson, 1972; Schneider, 2000). 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. Ad-

ditionally, PHMon monitors the execution of the forked processes of a parent process. Regarding tamperproofness, 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) through simulation and/or formal methods.

3.2 PHMon: Design

We propose a minimally-invasive programmable hardware monitor (for a general-purpose processor) to enforce an *event-action* monitoring model. Figure 3·1 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.

3.2.1 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 ac-

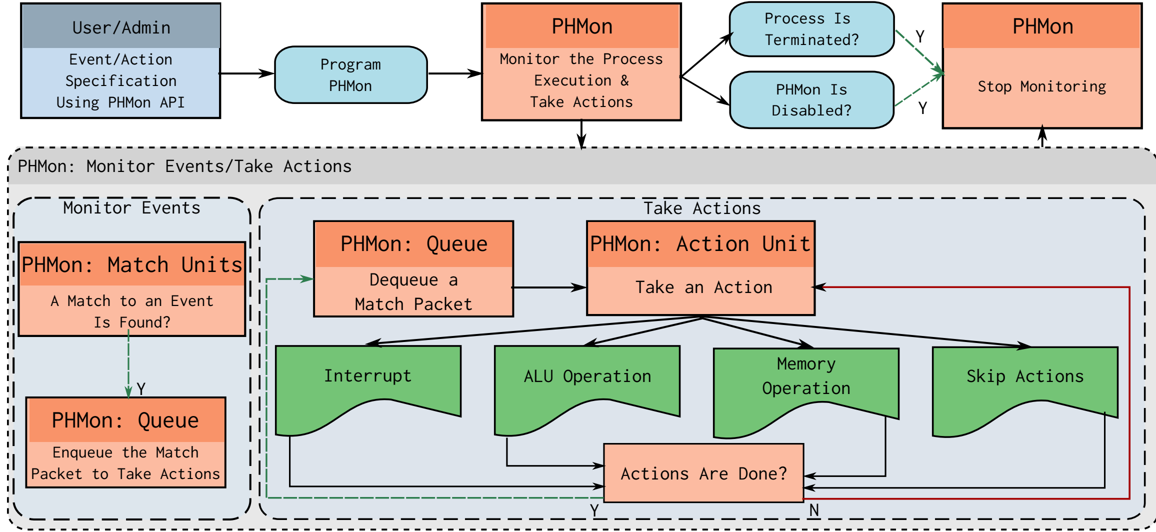


Figure 3·1: An overview of the *event-action* model provided in PHMon.

tions. To perform these tasks, PHMon consists of three main architectural units: a Trace Unit (TU), Match Units (MUs), and an Action Unit (AU).

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 the current implementation of PHMon, 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 (Li et al., 2016) 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 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 via its RoCC interface. 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 2·1 depicts the extended RoCC interface used in our design to communicate with the Rocket processor. We have extended the RoCC interface to carry the commit log trace (shown in red in Figure 3·2). 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 3·3, 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 PHMon is decoupled from the processor and processes the incoming commit logs one by one, 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

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

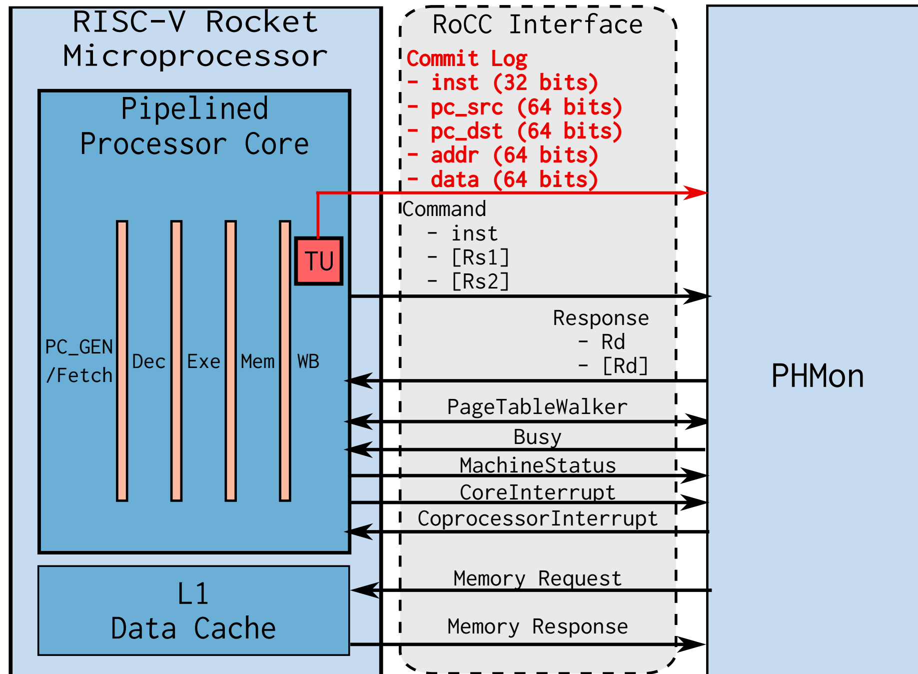


Figure 3·2: The RoCC interface extended with *commit log* execution trace.

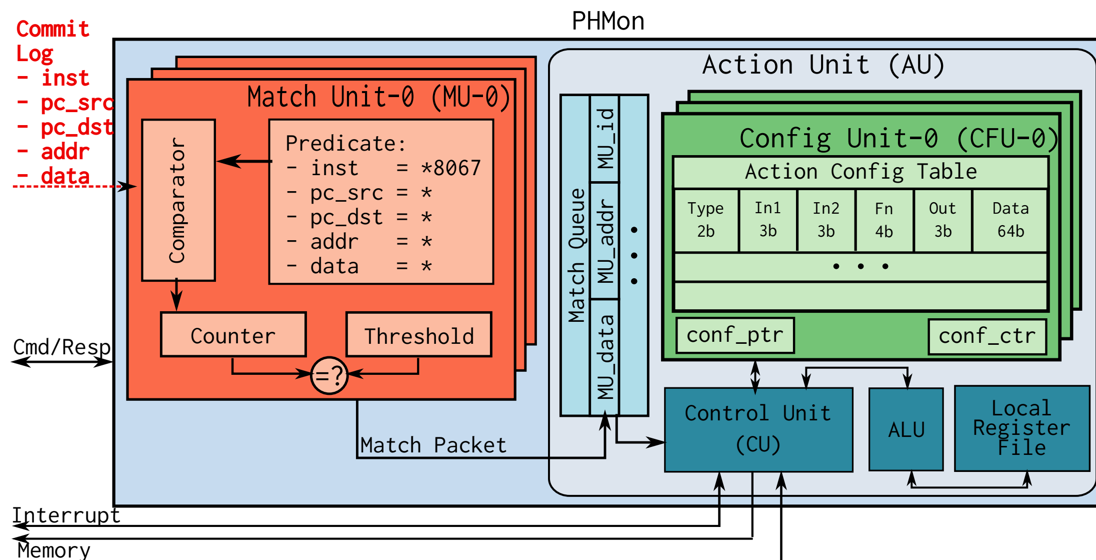


Figure 3·3: PHMon's microarchitecture.

matched events in a queue prior to taking actions.

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 (Waterman et al., 2011). 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 3.2.2, 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. 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.

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 3·3).

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) and leverages the registers in the Local Register File (6 registers in total) to perform these operations. 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.

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 3·3). `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. 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.

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.

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.

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.

3.2.2 PHMon: Software Interface

We use RISC-V's custom instructions to configure PHMon's MUs and CFUs, as well as to communicate with PHMon. Figure 3·4 shows the instruction format of the custom instructions. According to this format, the type of an instruction is determined based on the combined values of the opcode and funct fields. The xs1, xs2, and xd fields are corresponding booleans of rs1, rs2, and rd registers, respectively, indicating whether these registers are being used in the instruction. In our design, we

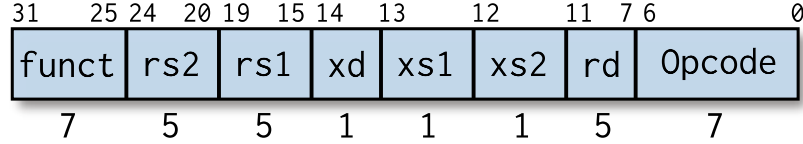


Figure 3·4: The RISC-V *custom* instruction format (Waterman et al., 2011).

Table 3.1: PHMon's RISC-V extension instructions transmitted over RoCC.

Instruction Category	funct	xd	xs1	xs2
Configure matching patterns	0	0	1	1
Configure actions	1	0	1	1
Control	2	0	1	1
Read status	3	1	1	0
Write status	4	0	1	1

only use the `custom1` opcode; accordingly, the value of the `funct` field distinguishes between our different instruction types. As listed in Table 3.1, we use custom instructions to augment the RISC-V ISA with five instruction categories that send/receive data to/from PHMon by `rs1`, `rs2`, and `rd` registers. Each of our extended instruction categories consists of one or more instructions. As an example, the `read status` instruction category contains instructions for reading the values of local registers as well as the counters of each MU. PHMon distinguishes between the instructions in the same category by decoding the values stored in the `rs1` register.

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. Table 3.2 shows the list of functions that one can use to communicate with PHMon, and it also specifies whether the function is accessible through a user-space process, a supervisor, or both. 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 (i.e.,

Table 3.2: PHMon's Application Programming Interface (API).

Function	Instruction Category	Accessibility	Description
set_pattern(MU_id, *mask, pid)	Configure matching patterns	User/Supervisor	Set the MU's matching pattern
set_thresh(MU_id, count)	Configure matching patterns	User/Supervisor	Set the matching threshold
set_action(CFU_id, *action)	Configure actions	User/Supervisor	Program the CFU's list of actions
conf_matchpacket(MU_id, element)	Configure actions	User/Supervisor	Select MU_data of match packet
reset(MU_id)	Control	Supervisor	Reset an MU and its paired CFU
enable(MU_id)	Control	Supervisor	Enable monitoring for an MU
disable(MU_id)	Control	Supervisor	Disable monitoring for an MU
count = rd_count(MU_id)	Read status	User/Supervisor	Read the value of an MU's counter
reg = rd_register(reg_id)	Read status	User/Supervisor	Read the value of a local register
wr_count(MU_id, count)	Write status	User/Supervisor	Write the value of an MU's counter
wr_register(count, reg_id)	Write status	User/Supervisor	Write the value of a local register

to seal PHMon's 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.

The **set_pattern** function in Table 3.2 configures the matching patterns of a specific MU using the `MU_id` and a matching input for a specific process or all processes. The matching input defines the matching conditions for each commit log entry. As discussed before, a match condition consists of matching and masking bits. As an example, according to the RISC-V ISA, a `ret` instruction is a pseudo-instruction defined by `JALR` when `rd=x0` and `rs1=x1`. We can monitor a `ret` (“`jalr x0,x1,0`”) using the following matching condition:

```
ret: inst = 0x00008067; mask bit = 0x00000000
```

Accordingly, the matching condition for `inst` evaluates to true when the current instruction is an exact match with the value of a `ret` instruction. For all the other entries of the commit log (`pc_src`, `pc_dst`, `addr`, and `data`), we set the masking bits to `0xffffffffffffffff`, indicating all these fields are `don't cares`. **set_thresh** function programs the number of matches that a specific MU needs to monitor prior to triggering an action. Once an MU triggers an action, it sends the commit log element specified by **conf_matchpacket** function to the `Match Queue` as part of the `match packet`. The actions are programmed using **set_actions** function for a specific CFU based on the value of an `action` struct. This struct has the same elements as

an `action description` and the user can program each element. Our API provides two separate functions for reading an MU's counter value and reading the values of registers stored in the Local Register File, namely **rd_count** and **rd_register**, respectively. Similarly, **wr_count** and **wr_register** are available for writing into an MU's counter and local registers.

3.2.3 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.

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. 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.

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. For debugging use cases, upon an interrupt, we can trap into GDB for further debugging.

3.3 PHMon: Use Cases

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

3.3.1 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 `call`s to the `ret`s in the chain). A shadow stack can therefore detect ROP attacks.

Rather than providing a dedicated hardware solution (e.g., Intel CET (Intel Corporation, 2017)), we leverage PHMon's flexibility to implement a hardware-assisted 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 the integrity of the shadow stack against unauthorized accesses by monitoring `load` and `store` accesses to this range of addresses leveraging a third MU. 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. 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. Analogous to (Broadwell et al., 2003), 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. In addition to implementing a shadow stack (backward-edge CFI enforcement), as shown in (Canakci et al., 2020), PHMon can assist with forward-edge CFI enforcement.

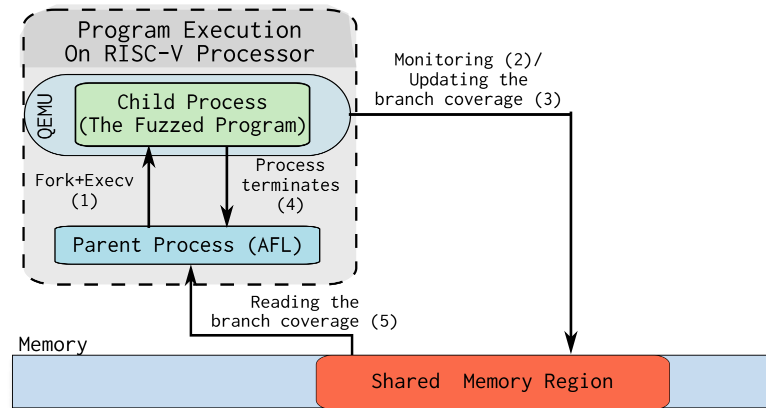
3.3.2 Hardware-Accelerated Fuzzing

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 (Aizatsky et al., 2016) and Microsoft (Microsoft Corporation, 2017) use fuzzing extensively and *continuously.* For instance, Google's OSS-Fuzz platform found over 1,000 bugs in 5 months (Google Corporation, 2017). Similarly, American Fuzzy Lop (AFL) (Zalewski, 2017) is one of the state-of-the-art

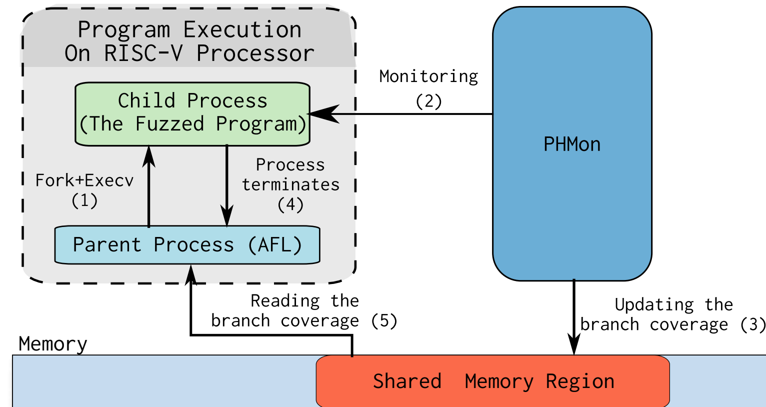
fuzzers that successfully identified zero-day vulnerabilities in popular programs, such as PHP and OpenSSH.

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× to 5× (Reddi et al., 2004)). 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 3·5): (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



(a)



(b)

Figure 3·5: Integration of PHMon with AFL. (a) The overview of QEMU-based AFL. (b) The overview of PHMon-based AFL.

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`.

3.3.3 Preventing Information Leakage

PHMon can also be used to prevent the leakage of sensitive information, such as cryptographic keys. A concrete example is Heartbleed (Graham-Cumming, 2015), 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 (Graham-Cumming, 2015). 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

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

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.

3.3.4 Watchpoints and Accelerated Debugger

As another 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 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 3.3.3.

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 100th 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. For the de-

bugging 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.

3.3.5 Code Coverage Engine

As software complexity increases, the importance of software testing, i.e., the process of evaluating a software product to detect possible bugs and errors, has gained significant traction in academia and industry (Schumilo et al., 2017; Aizatsky et al., 2016; Microsoft Corporation, 2017). Since the number of possible input test cases for most programs is infinite, a smart method for selecting rational test cases is vital. One of the widely-used methods to evaluate the quality of software testing is code coverage; a measurement of how many lines, branches, or functions of the program execute due to a particular set of test inputs. Code coverage can be collected by applying dynamic instrumentation using DBI tools such as Pin and DynamoRIO or static instrumentation by annotating the source of the program (e.g., gcov [68]). The former method adds software probes into the executable during runtime using dynamic binary translation; therefore, this technique introduces significant performance overhead. The latter requires access to source code which prevents the analysis of closed-source software.

Using PHMon, we collect the branch coverage and function coverage metrics for both open-source and closed-source propriety software with low performance overhead. To this end, we use two MUs, where MU0 monitors call instructions and MU1 monitors branch instructions with `threshold = 1`. Subsequently, MU0 and MU1 are responsible for collecting the call and branch statistics to measure the function and branch coverage metrics, respectively. When either of the MUs detects a match, it writes the `pc_src`, `pc_dst`, and its `id` into a shared memory and appropriately increases the address pointer. Measuring coverage does not require an online response.

Hence, analogous to gcov, we post-process the collected information to present the coverage results in a human-understandable way (function names, control flow graphs, etc.).

3.4 Evaluation Framework

In this section, we discuss our evaluation framework for PHMon as well as our performance, power, and area evaluation results.

3.4.1 Experimental Setup

We implemented PHMon as a RoCC (using Chisel HDL (Bachrach et al., 2012)) and interfaced it with the RISC-V Rocket processor (Asanović et al., 2016) that we prototyped on a Xilinx Zynq Zedboard evaluation platform (Digilent, 2017). 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 3.3 lists the microarchitectural parameters of Rocket core and PHMon. Note that similar to HDFI (Song et al., 2016), we do not include an L2 data cache in our experiments running on Rocket core. 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). 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 runtime overhead of 14 applications from MiBench (Guthaus et al., 2001), 9 applications (out of 12) from SPECint2000 (Henning, 2000), and 8 applications (out of 12) from SPECint2006 (Henning, 2006) benchmark suites. To measure the performance improvement of our hardware-accelerated AFL, we evaluated 6 vulnerable applications (Zalewski, 2017)

Table 3.3: 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	32 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

Table 3.4: List of the benchmark applications used to evaluate AFL.

Application	Description	Version
indent	Indentation for C code writing	2.2.1
zstd	A compression application	N/A
PCRE	A regular expression library	8.38
sleuthkit	A library to view file systems	4.1.3
nasm	An assembler tool	2.11.07
unace	Managing ACE archives	1.2b

listed in Table 3.4. To assess power and area, we used Cadence ASIC toolflow for 45nm NanGate process (Nangate Corporation, 2008) 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 (Thoziyoor et al., 2008) to estimate their power and area.

3.4.2 Experimental Results

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 that 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, 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 3·6 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 3.5 (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

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

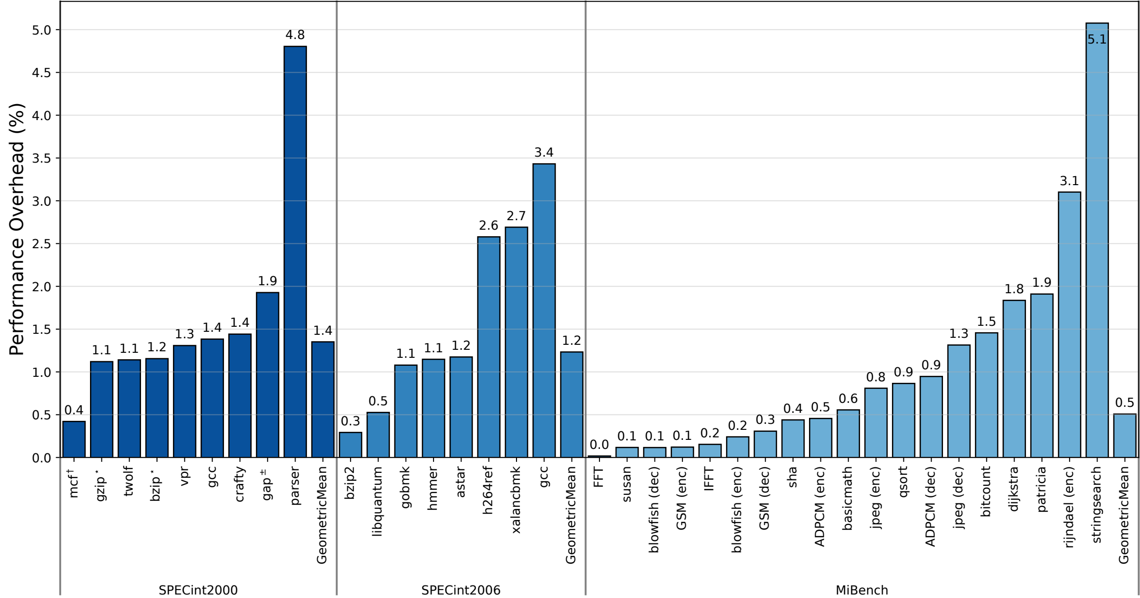


Figure 3·6: The performance overhead of PHMon as a shadow stack.
† We were not able to run mcf benchmark with `reference` input on our evaluation board; as a result, we used the `test` input for this benchmark.
⋆ Due to the memory limitations of our evaluation board, we had to reduce the buffer size 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.

benchmarks (Song et al., 2016). 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 3.5, 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. 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

Table 3.5: Performance overhead of PHMon-based shadow stack compared to that of HDFI-based (as reported in (Song et al., 2016)) 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%$^\pm$	3.34%	12.30%$^\pm$
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.
$^\pm$ 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.

Table 3.6: Performance overhead of previous software and hardware implementations of shadow stack compared with PHMon.

Mechanism	Methodology	Performance Overhead
(Szekeres et al., 2013)	Software (LLVM plugin)	5% on SPEC2006
(Abadi et al., 2009)	Software (binary rewriting)	21% on SPEC2000 (CFI + ID check)
(Corliss et al., 2005)	Software (binary rewriting)	20.53% on SPEC2000 (encoding) 53.60% on SPEC2000 (memory isolation)
(Davi et al., 2011)	Software (Pin tool)	2.17× on SPEC2006
(Sinnadurai et al., 2008)	Software (DynamoRIO)	18.21% on SPEC2000
(Zhang et al., 2014)	Software (static binary instrumentation)	18% on SPEC2006
(Dang et al., 2015)	Software	3.5% on SPEC2006
(Ozdoganoglu et al., 2006)	Hardware	∼0.5%-∼2.4% on SPEC2000
(Moon, 2017)	Hardware	0.24% on SPEC2006
(Song et al., 2016)	Hardware	2.1% on SPEC2000
PHMon	Hardware	1.4% on SPEC2000, 1.2% on SPEC2006

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 3.6 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 (Szekeres et al., 2013), which PHMon's shadow stack implementation 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 (Schumilo et al., 2017; Stephens et al., 2016), 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 3·7 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

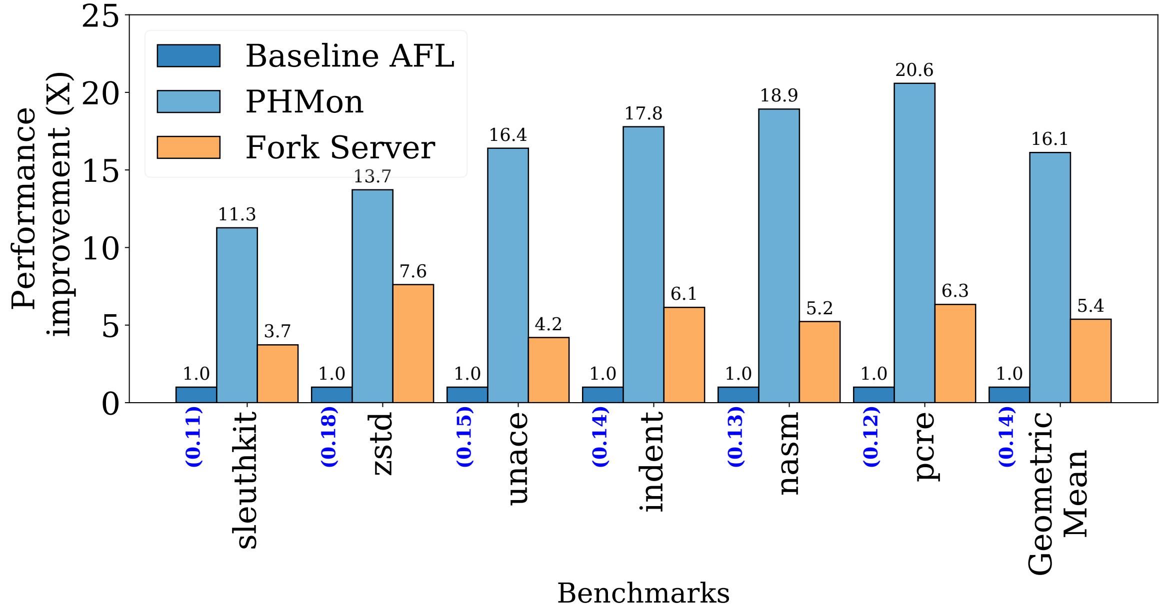


Figure 3·7: Performance improvement of PHMon over the baseline AFL compared to the fork server AFL. The numbers below the "Baseline AFL" bars show the number of executions per second for the baseline AFL.

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

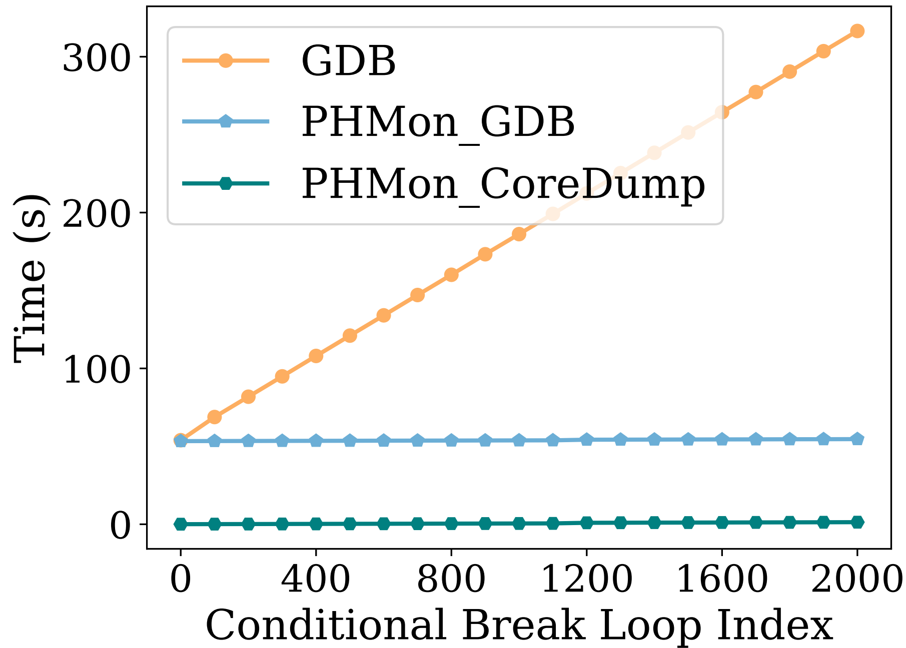


Figure 3·8: The performance overhead of PHMon compared to GDB for a loop conditional breakpoint.

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 3·8 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.

Code Coverage Engine: To validate the correct functionality of our code coverage engine, we collected the function call and branch coverage information from three different applications. As a first application, we wrote a basic program that includes several branches and function calls. Additionally, we used two more applications, i.e., PCRE and zstd. For these three applications, we examined several different inputs that produce different coverage results and compared our results with those reported by gcov (Gcov, 2021) results for the same set of inputs. These results are based on two different methods; 1) running the applications through the user-mode RISC-V QEMU on x86-64 and 2) running the applications on RISC-V Linux. We compiled the test applications with RISC-V GCC (-fprofile-arcs -ftest-coverage), and visualized the branch and function call coverage using gcov and lcov (LCOV, 2021), the graphical front-end of gcov. The branch coverage results of gcov for both of our methods match with that of PHMon results. For the function coverage, PHMon detects all the function calls successfully. To evaluate the performance overhead of our code coverage engine over the unmodified baseline RISC-V processor, we used 6 vulnerable applications tested with 5 different inputs for each application. As listed in Table 3.7, on average, PHMon incurs 2.15% performance overhead (3.55% in worst case).

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 of a context switch by 4.01%. The required operation to maintain PHMon's configuration during a context switch is constant. Hence, we

Table 3.7: The performance overhead of PHMon as a code coverage engine.

Application	Performance Overhead (%)
indent	2.39
zstd	3.11
PCRE	1.16
sleuthkit	0.11
nasm	3.55
unace	2.58
Geometric Mean	1.43

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.

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`

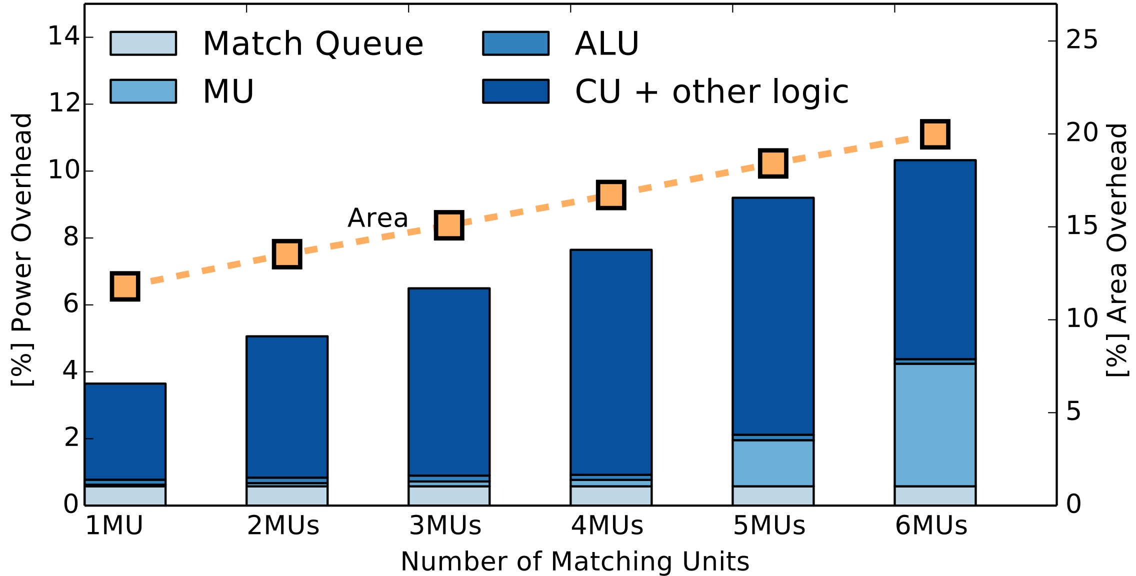


Figure 3·9: The power and area overheads of PHMon components compared to the baseline Rocket processor.

`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 umber 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 3·9 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 work, we used a design with only 2 MUs. This design has a 5% power

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

Table 3.8: 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

overhead and it incurs a 13.5% area overhead. Table 3.8 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%.

3.5 Summary

In this chapter, we have presented PHMon, a PE in form of a programmable hardware monitor. We have provided a minimally invasive and efficient implementation for PHMon with expressive monitoring rules and flexible fine-grained actions. PHMon is capable of enforcing a variety of security policies as well as assisting with detecting software bugs and security vulnerabilities. We have implemented a practical prototype, consisting of a Linux kernel and user-space running on a RISC-V processor interfaced with PHMon, on an FPGA. We have demonstrate the flexibility and ease of adoption of PHMon via five representative use cases. Our evaluations indicate that PHMon incurs low performance, power, and area overheads. In the spirit of open science and to facilitate reproducibility of our experiments, we have open-sourced the hardware implementation of PHMon, our patches to the Linux kernel, and our software API: https://github.com/buicsg/PHMon.

[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 (Lee et al., 2014). Here, we use a non-optimized but publicly available process (45nm NanGate) for power and area measurements.

Chapter 4

Intra-Process Memory Isolation

Intel MPK provides an efficient protection key-based approach for intra-process memory isolation. However, the current hardware implementation of Intel MPK and its software support suffers from security and scalability issues, i.e., allowing a compromised or malicious component to update permissions using a user-space instruction, being vulnerable to protection-key use-after-free issue, and providing only 16 protection keys. While the flexible design of PHMon enables us to enforce a variety of security policies, due to the limited number of MUs, PHMon is not suited for assisting with intra-process memory isolation at page granularity. In this chapter, we propose an SE, called SealPK, for intra-process memory isolation in RISC-V ISA. Similar to Intel MPK, SealPK provides a per-page protection key (pkey); however, SealPK supports up to 1024 domains (64× more than Intel MPK) by leveraging the 10 unused bits available in the PTE of each virtual page (RISC-V Sv-39). We mitigate the pkey use-after-free problem at OS level by keeping track of the number of pages belonging to the same domain and a lazy de-allocation approach. While Intel MPK does not provide a solution to maintain the integrity of protection domains and their permissions, we propose three novel sealing features to prevent an attacker from modifying sealed domains, their corresponding sealed pages, and their permissions.

4.1 Threat Model

We assume that the software may contain one or more memory corruption vulnerabilities that an adversary can leverage to perform an attack. SealPK's goal is to prevent the adversary from reading or writing from/into sensitive memory pages. We assume that the OS and all hardware components are trusted and bug free. Side-channel and rowhammer attacks as well as microachitectural leaks are out-of-scope of this work.

4.2 SealPK: Design

In this section, we discuss the design of SealPK at hardware level, OS level, and software level.

4.2.1 Hardware Design

We leverage the 10 unused bits of the RISC-V Sv39 PTE to store the pkey. The Sv48 PTE also has 10 unused bits while a 32-bit RISC-V processor uses Sv32, where there are no unused bits in PTE. In this case, we can store the pkey information in a separate OS-managed data structure and use a TLB to cache the information at hardware level. Figure 4·1 demonstrates our hardware modifications to support SealPK. We add a new entry to each line of the DTLB to store the corresponding 10-bit pkey of each virtual page.[1] Hence, our SealPK design supports up to 1024 domains, which is 64× more than the 16 domains supported by Intel MPK. We can use a virtualization-based mechanism, like libmpk (Park et al., 2019), to support more than 1024 domains. Although with a virtualization technique we can create more than 1024 domains, in reality we are still limited to 1024 concurrent physical pkeys. We store the permission bits of the pkeys separately. In our design, we use 2 bits, i.e., `(Read Disable (RD), Write Disable (WD))`, to specify the access permission of

[1] As that pkey checks are only applicable to data memory accesses and not an instruction fetch, we do not modify the ITLB.

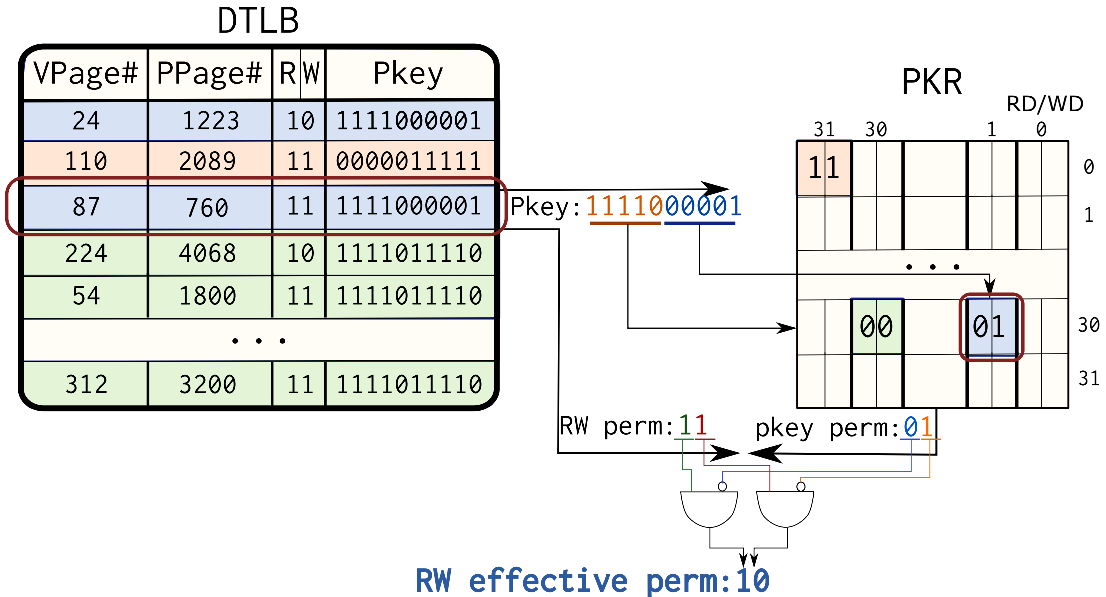


Figure 4·1: Modified MMU of the RISC-V Rocket core for SealPK support. Here, we color-code the TLB entries of each domain, consisting of various pages sharing the same pkey. For each data memory access, the effective permission bits are determined by the intersection of the PTE permissions and pkey permissions stored in `PKR`.

each protection key. Following the principle of the least privilege, unlike Intel MPK and previous works, our design enables a `write-only` page, which can in turn reduce the attack surface. Such a `write-only` page is specifically useful for log entries, where one thread is responsible for writing the log and another thread processes the written log. Note that the RISC-V ISA does not support `write-only` pages,[2] and our design provides this feature by leveraging pkeys regardless of the support in PTE permissions.

We support 1024 pkeys in our design; hence, unlike Intel MPK, we cannot simply use a single register to store all the pkey permission bits. To provide fast access to these bits, we use a 2Kb on-chip SRAM-based memory to store the permission bits. This memory, called `PKR` (shown in Figure 4·1), consists of 32 rows, where each row stores the permission bits of 32 pkeys (64 bits total). We utilize the custom

[2]The PTE permissions for a `write-only` page is a feature reserved for the future use.

instruction extension of the RISC-V ISA to define two new instructions, `RDPKR` and `WRPKR`, to read from and write to `PKR`.[3] The `RDPKR` instruction uses two registers, i.e., `rs1` and `rd`, for its operation. The input register (`rs1`) contains the pkey. At hardware level, the upper 5-bits of the pkey are used to index into `PKR` and read the corresponding 64-bit row of permissions. This 64-bit value is returned as the output and stored in `rd`. The `WRPKR` instruction uses two input registers, i.e., `rs1` and `rs2`, for its operation. The first input register (`rs1`) contains the pkey, which is used to index into `PKR`. The second input register (`rs2`) contains the new value of 64-bit permissions of the corresponding row. The hardware uses this new 64-bit value to overwrite the permission bits of the row indexed by pkey.

In our hardware design, we provide a control logic to determine the effective permission bits of each data memory access. Consider the example shown in Figure 4·1, where there is an incoming write request to the virtual page #87. In addition to reading the page's read/write permission bits stored in DTLB (`11`), the control logic reads the corresponding 2-bit permission bits of the pkey (`1111000001`) stored in `PKR`. The control logic uses the upper 5 bits of the pkey to index into a specific 64-bit row of `PKR` and the lower 5 bits to select the 2 permission bits (`01`). The effective permission is the intersection of the DTLB's and pkey's permission bits. In this example, the effective permission is `10`; hence, the write access is not allowed. If a data access is not allowed according to the effective permission, it leads to a load/store page fault; the processor triggers an exception, and the OS handles the page fault.

4.2.2 OS Support

At the OS level, we add the support to store each page's pkey in the 10 unused bits of the PTE. Our RISC-V kernel support is built upon the existing Linux kernel support

[3] To simplify the implementation of the custom instructions, we leverage the RoCC extension of the Rocket core, which adds the support for decoding and executing custom instructions to the Rocket core's pipeline.

for MPK, i.e., through `pkey_alloc`, `pkey_free`, and `pkey_mprotect` system calls.

Lazy de-allocation

To keep track of the allocated pkeys, we implement a 1024-bit allocation bitmap. To efficiently address the pkey use-after-free problem of Intel MPK, we leverage a lazy de-allocation approach. We implement a 1024-bit dirty map to indicate whether each pkey has been lazily de-allocated. We also keep track of the number of pages currently associated with each pkey using a counter map. If a pkey's corresponding counter is not zero, `pkey_free` updates the permission bits of the pkey in `PKR` to `(0,0)`; hence, the page-table permissions determine the effective permission of the corresponding pages. Rather than clearing the corresponding bit of the pkey in the allocation map, `pkey_free` sets the dirty bit and `pkey_alloc` would not allocate a dirty pkey. Whenever a memory page with a dirty pkey gets freed, we update the number of pages associated with the dirty pkey in the counter map, accordingly. Once the counter becomes zero, we erase the dirty bit of the corresponding pkey; hence, it can safely be allocated afterwards. If `pkey_alloc` cannot find a free non-dirty pkey, it returns an allocation error to indicate no free pkey is available.

Per thread OS support

We modify the `task_struct` in the Linux kernel to maintain the contents of `PKR` for each thread during the context switches. According to our evaluations, maintaining `PKR` information during context switches incurs less than 2% performance overhead. Furthermore, we modify the RISC-V page fault handler in the Linux kernel to identify a page fault caused by a pkey permission violation. We augment the segmentation fault with the pkey information to accurately reflect the cause of the fault to the developer and assist with debugging.

4.2.3 Sealing Features

In this subsection, we describe three novel sealing features to protect allocated domains, their associated pages, and their permission bits from tampering by an attacker. To clarify the defensive capabilities of these features, consider the example shown in Figure 4·2. In this example, a software developer writes a program that handles sensitive financial records. The `Main` function (written in-house) initially allocates the memory pages for the financial record (`log`) as `readable-writable` and assigns a protection key to these pages. Following the principle of the least privilege, the initial value of the pkey restricts the permission to `read-only` pages. In this example, `Func-A` updates the contents of the `log`. We assume that this function is developed in-house and has access to the pkey. Prior to writing the sensitive financial information into the `log`, `Func-A` modifies the domain permission of the `log` to `write-only`. For performance reasons, the software developer leverages third-party untrusted libraries in the implementation of `Func-B`, `Func-C`, and `Func-D`. `Funct-B` reads the `log` and returns a sorted copy of the `log`. `Func-C` does not have access to the `log`, instead it receives a list of prices and converts them to a different currency. `Funct-D` reads the `log` and prints all the transactions of a specific account. Hence, `Func-B` and `Func-D`, can only access the `log` as **read-only** memory. For security reasons, the untrusted functions are not aware of the pkey value. In the rest of this section, we explain how each of our sealing features protects the `log` against potential attacks originating from the untrusted components.

Sealing the domain: In this scenario, `Func-B` is a malicious third-party component, which receives the `log` as a `read-only` input. `Funct-B` is supposed to read the `log` and return a sorted copy of it. However, as shown in Figure 4·2, this untrusted component allocates a new `readable-writable` pkey, invokes the `mprotect` system call and assigns the new pkey to the `log`. In this way, `Func-B` can falsify the finan-

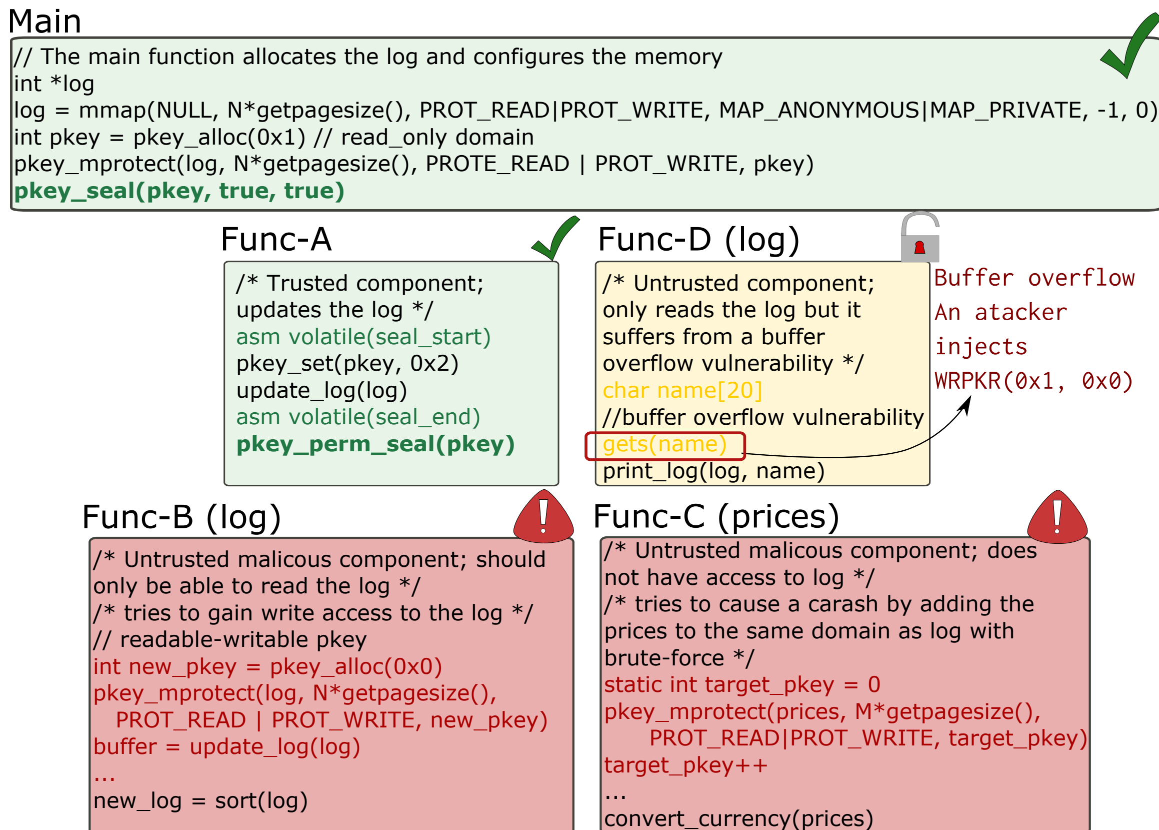


Figure 4·2: Example scenario for SealPK's sealable features. The red texts in `Func-B` and `Func-C` show an effort to attack the pkeys. The yellow texts in `Func-D` show a vulnerability that can be leveraged by an attacker to compromise the pkey permissions. The green texts in the `Main` and `Func-A` functions show our sealing features to protect the domain, its associated pages, and its permissions from unauthorized modifications.

cial records stored in the `log`. Unfortunately, the developer who uses this untrusted function does not have access to its source-code and is unaware of its maliciousness. In this scenario, Intel MPK is not capable of preventing this malicious modification to the `log` within the same thread. To prevent such unauthorized modifications, we provide a domain sealing option by adding a `sealed_domain` map to the kernel. We modify the `pkey_mprotect` system call to check the `sealed_domain` map prior to modifying a domain's pkey. Once a domain is sealed, `pkey_mprotect` prevents any further modifications to PTE permissions as well as the pkey value, efficiently throwing such attacks.

Sealing pages: We assume that after the initialization step in the `Main` function, no more pages will be added to the protection domain. Consider a scenario where `Func-C`, a malicious third-party component, aims to crash this financial application. Crashing the application at runtime could lead to denial-of-service and financial losses. `Func-C` does not have access to the `log`; it only receives a list of prices and converts them from one currency to another one. This price list does not include any sensitive information; hence, `Func-A` does not assign a protection domain to it. In this example, in each call, the malicious `Func-C` adds the pages associated with the price list to a different domain, hoping that the new domain would restrict the `read` permission. As a result, after the price list is assigned with the same pkey as the `log`, once `Func-A` tries to read the price list the program crashes with a segmentation fault. Intel MPK cannot prevent this issue within the same thread; similarly, our domain sealing feature is not sufficient in this scenario. To ensure that no more pages can be added to a domain (either by mistake or by a malicious component), we provide a page sealing option by adding a `sealed_page` map to the kernel, indicating whether the pages associated with each pkey are sealed. We modify the `pkey_mprotect` system call to check the `sealed_page` map and only allow adding new pages to a pkey

domain if the associated pages of that domain are not sealed. As shown in Figure 4·2, we add a new system call, `pkey_seal(int pkey, bool seal_domain, bool seal_page)`, which allows the programmer to seal a domain and/or its associated pages. Note that once a domain or its associated pages are sealed, the seal cannot be broken unless the corresponding pkey and all its associated pages are freed.

Sealing permissions: In this scenario, we assume `Func-D` is a third-party component, suffering from a buffer overflow vulnerability. As shown in Figure 4·2, an attacker can leverage this vulnerability to inject a `WRPKR` instruction at runtime and modify the permission bits of the `log` to `readable-writable`. Subsequently, the attacker can falsify the sensitive contents of the financial record. Intel MPK does not protect pkey permissions against control-flow hijacking attacks that leverage the `WRPKRU` instruction. To prevent such a tampering, we provide a permission sealing feature, which allows the developer to restrict the execution of the `WRPKR` instruction to a specified range of memory addresses. In this example, we aim to restrict the occurrence of the `WRPKR` instruction to the address range of `Func-A`.

At hardware level, as shown in Figure 4·3, we keep track of sealed pkey permissions using a local memory, called `SealReg`. We modify the Rocket core's pipeline to consult `SealReg` prior to executing a `WRPKR` instruction. If the permission bits of the pkey are sealed, the `WRPKR` instruction is only allowed in the permissible range, specified by the developer. We leverage a Content-Addressable Memory (CAM) like structure, named `PK-CAM`, to cache the permissible range of each pkey. If the pkey information is available in `PK-CAM` but the current address of the `WRPKR` instruction is not in the permissible range, then SealPK prevents the execution of `WRPKR` and causes an exception. If `PK-CAM` does not include the pkey information, we will refill `PK-CAM`. To do this, we trigger an interrupt and insert the pkey and its permissible range to `PK-CAM` in the OS interrupt handler. As part of our future work, we plan to delegate

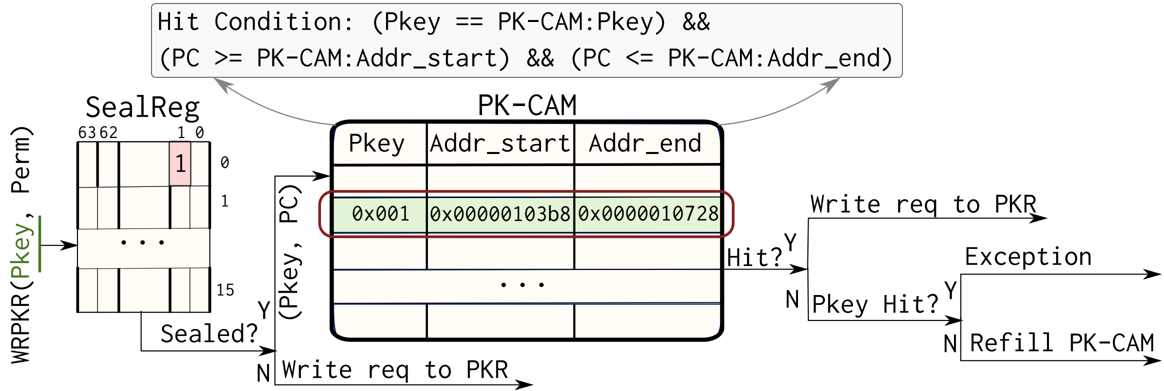


Figure 4·3: High-level view of SealPK's hardware support to seal pkey permissions.

this interrupt to user level and provide a secure software library to update `PK-CAM`.

We also provide the software support for sealing the permissions. We provide two new custom instructions, i.e., `seal_start` and `seal_end`, to specify the contiguous permissible range of each pkey. Although these instructions can be added to the source code (Figure 4·2), the more efficient way of using them is by a compiler pass or through runtime mechanisms such as ld-preload. After specifying the start and end addresses of a permissible range for `WRPKR`, the developer has to invoke a newly added system call (`pkey_perm_seal`) to seal the permissions. This system call leverages a custom instruction, which is only accessible to the supervisor mode, to seal the permission bits by updating the `SealReg` and `PK-CAM`. We modify the Linux kernel to maintain the `SealReg` information as well as permissible range of each pkey during context switches for each process. Note that `SealReg` and the permissible range of a pkey are implemented similar to a one-time fuse, i.e., they can only be written once for each process. Hence, after configuration, the permission sealing feature cannot be modified. The simplicity and efficiency of our permission sealing feature distinguishes our work from existing works focused on preventing the manipulation of a domain's permissions by an attacker, e.g., (Hedayati et al., 2019) and (Vahldiek-Oberwagner et al., 2019). By leveraging SealPK's sealing features, the software developer can implement a

tamper-proof `log` of financial records in the face of buggy and malicious third-party components.

4.3 SealPK: Case Study

To demonstrate the effectiveness of SealPK, as a case study, we use SealPK to protect an isolated shadow stack that prevents ROP attacks. It is imperative to guarantee the integrity of the shadow stack (Burow et al., 2019), i.e., the shadow stack area should be an **isolated** area within the process' address space to prevent attackers from modifying it. We isolate the shadow stack memory in a protection domain. Once the shadow stack memory is allocated and assigned to a domain, no more pages will be added and the protection domain stays the same during the process execution. We leverage the domain and page sealing features to protect the allocated domain and pages of the shadow stack from further modifications (similar to scenarios described in Section 4.2.3) after the initial configuration.

For the shadow stack implementation, we first implement a baseline front-end pass LLVM plugin (the same plugin that we used in PHMon's evaluation). This front-end pass allocates a memory area for the shadow stack and instruments the prologue and epilogue of each function to push the original return address into the shadow stack memory and pop the shadow return address from that memory, respectively. To isolate the shadow stack, we modify the front-end pass to allocate a pkey and to assign it to the shadow stack memory pages. To protect the shadow stack from modifications, we initialize the pkey as `read-only`. We implement a RISC-V back-end pass to temporarily update the pkey permission to `readable-writable` in the prologue, where we push the return address into the shadow stack. Right after pushing the return address, the back-end pass disables the pkey write permission. Our back-end pass inserts the required `RDPKR` and `WRPKR` instructions to update the pkey's

permission bits. We can leverage our permission sealing feature to restrict the `WRPKR` occurrences to the memory range of the back-end pass.

4.4 Evaluation

In this section, we evaluate the performance overhead of SealPK for the isolated shadow stack. Additionally, to demonstrate the hardware overhead of SealPK, we report its FPGA resource utilization.

4.4.1 Experimental Setup

We use the Chisel HDL to implement SealPK on the RISC-V Rocket core, with the same configurations that we described in Section 3.4.1. We add the OS support for SealPK to the Linux kernel v4.15. As a case study, we implement an isolated shadow stack using LLVM front-end and back-end passes. We use Clang v.7 and v.8 for our front-end and back-end passes, respectively. We prototype our hardware design with the full software stack on a Xilinx Zedboard FPGA.

For performance evaluation, we use RISC-V LLVM to cross-compile 6 applications (out of 12) from SPECint2000 (Henning, 2000), 4 applications (out of 12) from SPECint2006 (Henning, 2006), and 7 applications from MiBench (Guthaus et al., 2001) benchmark suites. Due to compilation issues and memory limitations of our FPGA, we were not able to successfully cross-compile and run all the applications from these benchmark suites. In particular, for SPECint2000, we got a segmentation fault for the baseline execution of `vortex` and `gcc`, and faced LLVM cross-compilation issues for the remaining 4 applications. For SPECint2006, with the baseline code, we got an out of memory error for `mcf` and a segmentation fault for `gcc`. We faced various LLVM cross-compilation issues for the remaining 6 applications. Note that RISC-V LLVM is still not as mature as GCC support for RISC-V. In our evaluations, we use the `large` inputs for MiBench and evaluate SPECint2000 and SPECint2006

applications using `test` inputs. We use the `test` inputs for SPEC evaluations due to the memory limitation of our FPGA board (256MB) as well as the long execution time of the benchmarks for the `mprotect` comparison point (multiple days). In our evaluations, we measured the total execution time of an application as our performance metric. For the baseline, we compiled the benchmarks using Clang v.8 without applying any passes and ran the benchmarks on an unmodified core and Linux kernel. We ran each application three times and report the geometric mean of the execution times.

4.4.2 Experimental Results

Performance Results

Figure 4·4 shows the performance overhead of various shadow stack implementations compared to the baseline. The `Inline` implementation is a non-isolated shadow stack as a front-end LLVM pass, where the shadow stack capability is inserted as an inline code. The `Inline` implementation has a 7.12%, a 4.44%, and a 2.18% performance overhead, on average, for SPEC2000, SPEC2006, and MiBench benchmarks, respectively. The performance overhead difference between the `Inline` and `Func` implementations (6.73% for SPEC2000, 4.94% for SPEC2006, and 1.47% for MiBench) shows the effect of using a function call in the front-end pass rather than an inline code. Note that `Inline` and `Func` implementations of the shadow stack cannot guarantee its integrity; hence, the shadow stack memory remains unprotected. `SealPK-WR` is an isolated implementation of the shadow stack, which leverages SealPK to temporarily allow `write` permission into the shadow stack memory. We implement `SealPK-WR` as a back-end pass that uses the function name in the `Func` implementation to simply identify the exact location to insert the necessary `WRPKR` instructions. For SPEC2000 and SPEC2006, respectively, `SealPK-WR` has an average of 19.59% and 13.95% performance overhead while for less-intensive MiBench applica-

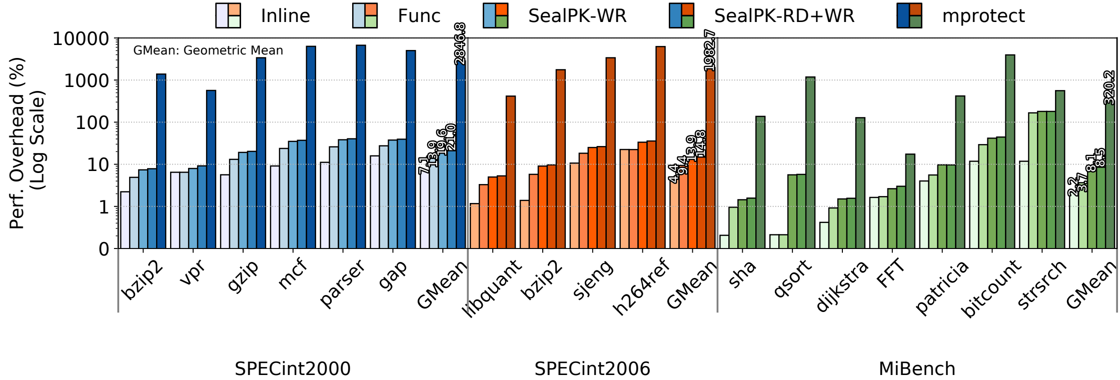


Figure 4·4: Performance overhead of various LLVM-based shadow stack implementations for SPECint2000, SPECint2006, and MiBench benchmarks. The `Inline` implementation is a front-end LLVM pass, where the shadow stack capability is inserted as an inline code. The `Func` implementation uses a function call in the front-end pass rather than an inline code. `SealPK-WR` is implemented as a back-end pass built upon `Func`, where it writes the new value of pkey permission bits without maintaining the rest of the permission bits. `SealPK-RD+RW` adds the support to read the corresponding row of the pkey before updating it. `mprotect` is implemented as an inline front-end pass by invoking `mprotect` system call before and after writing the return address into the shadow stack.

tions `SealPK-WR` only incurs 8.12% performance overhead. `SealPK-WR` assumes that during the program execution, only one pkey is allocated; hence, it writes the new value of pkey permission bits without reading and maintaining the rest of the permission bits. Adding the support for reading the corresponding row of the pkey before updating it, i.e., `SealPK-RD+RW`, increases the performance overhead by 1.41%, 0.86% and 0.40% for SPEC2000, SPEC2006, and MiBench, respectively.

As a comparison point, we implemented an isolated version of the shadow stack leveraging the `mprotect` system call. In this scenario, we modified our inline front-end pass to invoke `mprotect` before and after writing the return address into the shadow stack to allow and disallow the write permission, respectively. As expected, using `mprotect` incurs considerable performance overhead, i.e., 2875.62% for SPEC2000, 1982.70% for SPEC2006, and 320.21% for MiBench, on average, which makes it an infeasible option. `mprotect` requires a context switch into the kernel, followed by a

Table 4.1: The FPGA utilization of SealPK compared to the baseline Rocket core.

	Baseline		Rocket Core + SealPK	
	Used	Utilization	Used	Utilization
Total Slice LUTs	32030	60.21	35019	65.83
LUTs as logic	30907	58.1	33852	63.63
LUTs as Memory	1123	6.45	1167	6.71
Slice Registers as Flip Flop	16506	15.51	19392	18.23

full page table walk to change the permissions of all the specified pages, and then a TLB flush. On the contrary, leveraging SealPK to implement an isolated shadow stack uses a user-space instruction to modify the pkey permission bits. We leverage our sealing features to protect the allocated pkey, its associated pages, and their permissions from tampering by an attacker.

FPGA Resource Utilization

Table 4.1 shows the FPGA utilization of adding SealPK to the Rocket core compared to the baseline unmodified Rocket core. In our FPGA prototype, enhancing Rocket core with SealPK increases the LUT and FF utilization by 5.62% and 2.72%, respectively. The main source of area and power overhead for SealPK is `PKR`, a 2Kb local memory. Accordingly, we estimate that our power overhead is also less than 6%, even when considering a 100% access rate to `PKR`. In our FPGA evaluation, the Rocket core operated with a maximum frequency of 25 MHz (both in the baseline and the enhanced version with SealPK experiments).

4.5 Summary

In this chapter, we have presented SealPK, an SE for intra-process memory isolation in RISC-V ISA. The goal of our SE is to provide an efficient and secure per-page protection capability that can support a large number of protection domains. We have leveraged the 10 unused bits available in the Sv39 PTE of each virtual page

to support up to 1024 protection domains. We have developed three novel sealing features to prevent an attacker from modifying sealed domains, their corresponding sealed pages, and their permissions. We have demonstrated the efficiency of our design by securing a shadow stack on our FPGA prototype.

Chapter 5

Runtime Instruction Filtering

In this chapter, we discuss our design of an SE for runtime filtering of unsafe instructions. Such unsafe instructions could compromise the integrity of the isolation in-place by modifying access permissions, disabling protections, gaining higher privilege, etc. Our SE, called FlexFilt, assists with securing various isolation-based mechanisms. Our goal is to design a flexible and efficient hardware engine capable of filtering different user-defined instructions at different privilege levels in various parts of the code. Although a flexible hardware feature such as PHMon enables us to prevent the execution of various unsafe instructions in specific ranges of memory addresses, the number of memory regions for filtering instructions is limited to the number of MUs. FlexFilt enables the software developer to create up to 16 instruction domains, where each instruction domain can be configured to filter the execution of user-specified instructions at page granularity. At hardware level, FlexFilt provides configurable filters to prevent the execution of various user-defined instructions.

5.1 Threat Model

FlexFilt can be leveraged in a variety of security use cases introduced by prior work (see Table 2.2). In our work, for each use case, we follow the common threat model in the prior work. For intra-process memory isolation approaches, we assume that the untrusted parts of the code might contain vulnerabilities that an adversary can exploit to inject arbitrary instructions including the target instructions (e.g., `WRPKRU`).

We do not assume any restrictions about what an attacker would do after a successful attack.

As the OS is responsible for allocating the instruction domains and maintaining FlexFilt's information, we assume the OS kernel is (partially) trusted. In Section 5.5, we discuss about FlexFilt's capabilities and limitations in filtering kernel-level instructions. We assume all hardware components, including our modifications, are trusted and bug free. Hence, we consider rowhammer, side-channel, and fault attacks beyond the scope of this work.

5.2 FlexFilt: Design

In this section, we discuss FlexFilt's design goals, the challenges involved in implementing FlexFilt, and our solutions to address those challenges. Unlike prior works that provide a solution capable of filtering a small number of specific target instructions, we strive to provide a generalized solution for filtering user-defined instructions at runtime. Such a generalized solution should be flexible, efficient, and fine-grained. To be compatible with existing OS-supported memory protections, we implement FlexFilt at page granularity, i.e., each instruction page can apply a combination of the configured instruction filters. This design choice allows us to leverage the already existing OS-managed structures such as PTE as well as hardware structures such as TLB in our implementation. Providing a finer granularity for instruction filtering requires substantial modifications at both OS-level and hardware-level. We need to provide the OS support as well as a software API to enable a software developer to utilize FlexFilt. In the rest of this section, we will first discuss our hardware design choices, followed by the OS support for FlexFilt, and then the software support to configure our flexible instruction filters.

5.2.1 FlexFilt: Hardware Design

In this subsection, we discuss the hardware design of FlexFilt.

Instruction Protection Domains

To leverage the existing OS-level and hardware-level structures for memory protection, we implement FlexFilt at page granularity. Inspired by the design of memory protection keys, we devise *instruction protection keys*, which enables us to simply divide the software code into trusted and untrusted executable partitions. The software developer can assign the same instruction protection key to a group of executable pages, which subsequently creates instruction protection domains. The existing memory protection keys such as Intel MPK are only applicable to data memory accesses, not instruction addresses. In this work, our focus is on associating fetched instructions to protection domains according to their corresponding addresses.

The prior work such as Donky (Schrammel et al., 2020) and SealPK leverage the 10 unused bits of Sv39/Sv48 PTE to store the memory protection key information. Similarly, we can utilize these 10 unused bits to store the instruction protection keys, which would provide up to 1024 instruction protection domains. Supporting a large number of data *memory protection domains* is a necessity in various use cases, such as PMO (Xu et al., 2020) and OpenSSL (Park et al., 2019). However, supporting a large number of domains is not required for instruction protection domains. According to our literature review, the previous works with the instruction filtering requirement only needed two instruction domains, i.e., a trusted and an untrusted domain. However, providing only two instruction protection domains could be restrictive for some use cases (e.g., the combination of various protection mechanisms). But, we do not need 1024 instruction domains, even if we apply all the protection mechanisms proposed by a variety of the previous works into a single system. As a trade-off for the

number of instruction domains, we utilize the 4 lower bits of the 10 unused bits in the PTE to store the instruction protection keys (ipkey). Accordingly, FlexFilt supports up to 16 instruction protection domains, where each domain filters target instructions in the domain's corresponding pages.

Flexible Filters

Each instruction protection domain prevents the execution of target instructions in its corresponding pages. Ideally, we are interested in a flexible feature capable of filtering any number of target instructions in each domain. In reality, providing such a capability is not practical due to resource limitations and substantial area and power overheads. With a limited number of instruction filters for each domain, we consider two design options. First, each instruction domain has a fixed number of dedicated instruction filters. Second, there is a fixed number of shared instruction filters, and each instruction domain can apply a combination of these shared filters to its corresponding pages. Although the first option provides more flexibility in terms of filtering capabilities, it requires more hardware resources. Additionally, the instruction filter information for all the domains should be maintained at OS level during context switches. Considering the overheads involved with the first design option, we choose the second option in our design. We leave further investigations into the overheads involved in implementing the first design option as part of our future work.

By choosing the second design option, i.e., a fixed number of shared configurable instruction filters, the next design question we have to answer is the exact number of shared instruction filters. To choose the number of instruction filters, we examine the number of required filters in the previous works (listed in Table 2.2). Most of the previous works required to filter only one target instruction. In the worst case scenario, Fidelius (Wu et al., 2018) needed to filter the execution of three unique instruction

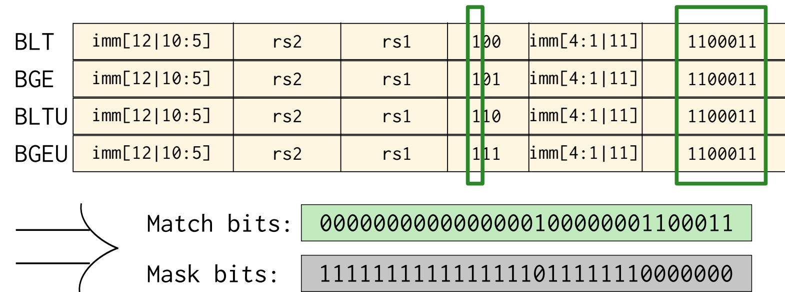
types (five instructions in total). Considering the possibility of enforcing a combination of protection mechanisms, we choose to implement four shared instruction filters in our design. With four instruction filters we can filter all the target instructions in Fidelius (Wu et al., 2018) based on their instruction type. Once the software developer configures the shared instruction filters, each instruction domain applies a combination of the shared filters to its corresponding pages, i.e., an instruction domain can apply or not apply each of the filters. Four configurable filters results in 16 possible combinations of applicable filters. As FlexFilt supports up to 16 instruction domains, each domain can apply one of the 16 possible combinations of configured filters.

One of the main design goals of FlexFilt is providing **flexible** instruction filters. To achieve this goal, we enable a software developer to configure the filters (Section 5.2.3). Additionally, we design each filter in an inherently flexible way. To do this, as a first step, we examine the RISC-V instruction formats (Waterman et al., 2019a). At hardware level, we can design a simple filter by maintaining the `opcode` of the target instruction in a register. Then, we can apply this filter to the instruction trace at runtime by comparing the `opcode` of each instruction at the execution stage with the filter. However, various RISC-V instructions are identified by a combination of `funct3`/`funct7` and the `opcode`. As an example, consider various branch instructions, including `BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, and `BGEU`, in the RISC-V ISA. These branch instructions share the same `opcode` value (`1100011`) and they are distinguished based on the value of `funct3` bits. In this scenario, a flexible instruction filter offers the user the option of filtering a specific branch instruction, a subset of the branch instructions, or all of the above-mentioned branch instructions. To provide such flexible filtering options, we leverage a bit-granular match/mask mechanism, similar to the matching mechanism that we used for PHMon.

Consider a scenario where the user is interested in filtering four of the previously mentioned branch instructions, i.e., `BLT`, `BGE`, `BLTU`, `BGEU`. Figure 5·1 shows the format of these instructions. The common bits of these instructions, specified with a green box, can be used for identifying them. In this example, the uncommon bits are `don't cares`, i.e., they serve no purpose in identifying the target branch instructions. Accordingly, the software developer can simply describe the four branch instructions using `Match` and `Mask` bits. The `Match` bits specify the 32-bit value of an instruction identified as one of the four branch instructions and the `Mask` bits specify the `don't care` bits. At hardware level, our `Flexible Filters` enforce such a matching/masking approach (the bottom part of Figure 5·1). The control logic of the `Mask` acts like a filter that blocks the masked parts of an instruction (specified by `1` bits in the `Mask` and shown by dark gray color in Figure 5·1) and passes through the rest of the instruction bits (shown by transparent gray color in Figure 5·1). The output of this control logic, which contains the `don't care` bits, is passed into a comparator module to be compared with the `Match`. If these two values match, then the `Flexible Filter` activates an output signal, indicating that current instruction should be filtered.

Microarchitecture Support

FlexFilt supports up to four instruction domains and provides four `Flexible Filters`, where a combination of these filters is applicable to each instruction domain. For each page, we specify the instruction domain by storing the ipkey in the 4 previously unused bits of PTE. For each instruction execution, in addition to checking the PTE permission bits (e.g., the `X` bit, which indicates that the page is executable), we need to determine if the instruction should be filtered. To this end, we first have to identify the corresponding domain of the instruction. At hardware level, the Instruction TLB (ITLB) maintains the virtual to physical address translation of the instructions as well as their corresponding permission bits. We augment the ITLB with a new field



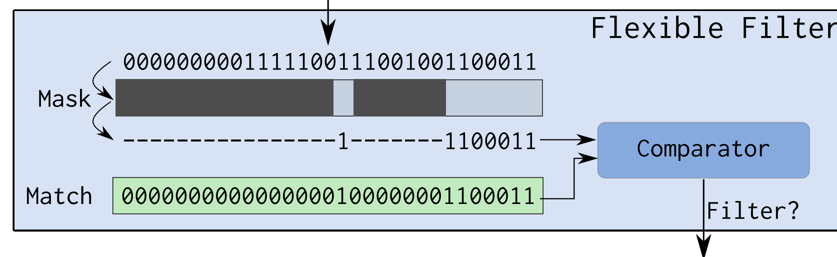


Figure 5·1: The `Flexible Filter` design, applied to a subset of RISC-V branch instructions.

to store the associated ipkey of each virtual address. Whenever there is an ITLB miss, the hardware Page Table Walker (PTW) walks the page table and fills the ITLB with the missing information including the ipkey. As each instruction domain applies a combination of the `Flexible Filters`, we need to maintain the configured combination of each domain. To this end, for each domain, we associate a valid bit to each of the `Flexible Filters`. We store all the valid bits in a separate 64-bit register, called Instruction Protection Register (`IPR`).

Figure 5·2 demonstrates our modifications to the RISC-V Rocket core to implement FlexFilt. The modified ITLB stores the ipkey information (received from PTW) for each entry. On an ITLB hit, the ipkey value gets transferred to the I-Cache (alongside the physical address) and subsequently on an I-Cache hit, the instruction and its associated ipkey gets stored in the Instruction Queue. Subsequently, the ipkey value gets transferred to the decode and then execute stage without any modifica-

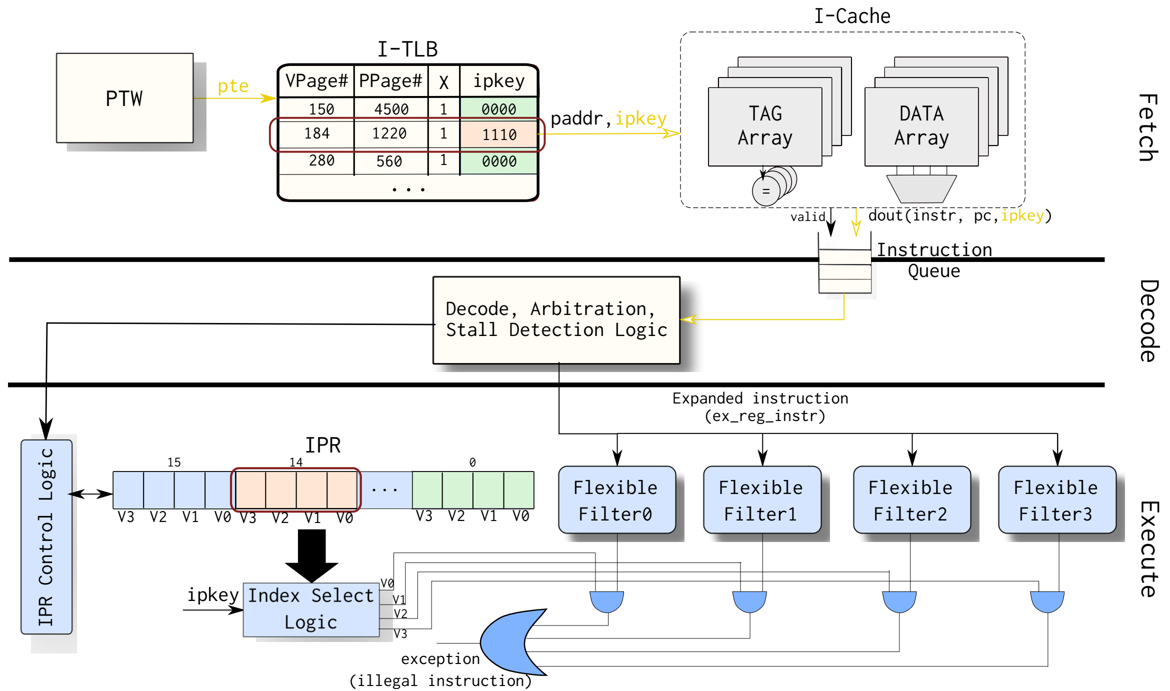


Figure 5·2: Simplified overview of the modifications to the RISC-V Rocket core to support FlexFilt. The blue components are the new components added to provide our page-granular flexible filtering capability. The components and arrows shown by yellow are the existing parts of the Rocket core, which are minimally modified to support FlexFilt. The gray components are unmodified but they communicate with modified signals. We do not show the rest of unmodified components in the Rocket core's pipeline.

tions. At the execute stage, FlexFilt uses the ipkey value to read the corresponding valid bits (4-bits) of the instruction domain from `IPR`. In the same cycle, each of the 4 `Flexible Filters` receives the 32-bit instruction in the execute stage, and performs the filtering operation based on its `Match` and `Mask` bits configuration (Figure 5·1). If the resulting filter signal of any of the `Flexible Filters` is high and at the same time its corresponding valid bit is active, then FlexFilt prevents the execution of the instruction by causing an illegal instruction exception.

While Figure 5·2 shows the main components of FlexFilt, this implementation does not take the privilege level of the instructions into account. Some of the previous works focused on preventing the execution of target instructions in the kernel

space while others focused on preventing the execution in user space. To distinguish between privilege levels at hardware level, we simply access the `priv` field provided in the `MStatus` CSR of the Rocket core. We consider two design options for incorporating the privilege level into our filtering mechanism. As the first option, we can augment each `Flexible Filter` with a configurable `privilege` field. Subsequently, each instruction domain applies a combination of `Flexible Filters` in its corresponding pre-configured privilege level. As the second option, we can allow each instruction domain to configure the privilege level of its associated valid filters. To support this option, in addition to valid bits, we need to maintain the privilege level of each `Flexible Filter` for each instruction domain. Although the second option is more flexible, it requires more hardware resources. To choose between these two design options, we examined the requirements of the previous works listed in Table 2.2. As prior work did not need to prevent the execution of a target instruction at various privilege levels in different parts of the code, we choose the first design option (described above).

5.2.2 FlexFilt: OS Support

In our design, we consider scenarios where each process can filter different target instructions. To enable a per-process instruction filtering capability, we need to provide the OS support for FlexFilt. In this subsection, we discuss the Linux kernel modifications to support FlexFilt.

Instruction Protection Keys

To provide the support for instruction protection keys in the Linux kernel, we leverage the existing support for memory protection keys. We use the existing support for `pkey_alloc` and `pkey_mprotect` in the kernel to allocate an instruction protection key and associate the specified executable pages with an ipkey, respectively. Unlike

the existing memory protection proposals on RISC-V, we only use 4 bits of the unused PTE bits to store our protection keys. Hence, we update the allocation bitmap to keep track of 16 instruction protection domains.

Per Process OS Support

To enable a per process view for instruction protection domains, we maintain the domain information during context switches. We modify the `task_struct` in the Linux kernel to keep the configuration of each `Flexible Filter`, which includes the `Match`, `Mask`, and `privilege` bits. Additionally, we maintain the `bitmap` of allocated ipkeys as well as `IPR` contents.

5.2.3 FlexFilt: Software Interface

We leverage the standard RISC-V `custom` instruction extension to define new instructions for configuring FlexFilt. Table 5.1 shows our software API and the unprivileged `custom` instructions that each API invokes to configure FlexFilt. We provide the `config_filter` function to configure each `Flexible Filter` by specifying its corresponding `Match`, `Mask`, and `privilege` bits. The software developer can leverage the `config_instr_domain` function to set the valid bit of a `Flexible Filter` for a specific instruction domain. Additionally, we provide five more `custom` instructions, which are accessible only at the supervisor level. We leverage these five instructions to maintain FlexFilt's information during context switches.

We leverage the existing `pkey_mprotect` system call to associate a group of executable pages, specified by `addr` and `len`, with an ipkey. To invoke the `pkey_mprotect` system call, we should obtain the address range of each instruction domain. To this end, the software developer can annotate the source code to specify the sections of the program belonging to an instruction domain. Then, we can modify the loader to invoke `pkey_mprotect` based on the extracted information from annotations. Rather

Table 5.1: FlexFilt's Application Programming Interface (API).

Function	Invoked Custom Instruction
config_filter(uint32_t match, uint32_t mask, uint8_t priv, uint8_t index)	SETMATCH, SETMASK, and SETPRIV
config_instr_domain(uint64_t d_index, uint64_t v_index)	WRIPR

than modifying the loader, as a proof of concept, we leveraged `LD_PRELOAD`. We leave the required modifications to the loader as part of our future work.

To clarify the use of `LD_PRELOAD` in specifying the instruction domains, consider a scenario where the software developer aims to only allow the execution of `WRPKRU` instruction (or its RISC-V equivalent, i.e., `WRPKR` instruction that we defined in SealPK) in trusted parts of the code. In this example, we assume that the software developer writes two trusted functions, namely `good_code1` and `good_code2`. She modifies the permission bits of her memory protection domains through `WRPKRU` instructions only in the two trusted functions and she wants to prevent the execution of `WRPKRU` in other parts of the code. We annotate these two functions with a function attribute to put them in a separate section. Then, we modify the linker script to ensure that the instructions in this separate section are page aligned. Additionally, we use the `config_filter` function to configure one of the `Flexible Filters` with an exact instruction match with `WRPKRU` running in user-level privilege. Then, we leverage `config_instr_domain` to set the corresponding valid bit of that `Flexible Filter` for the default instruction domain, i.e., `domain0`. To allow the execution of `WRPKRU` instruction in `good_code1` and `good_code2`, we can associate the instruction pages of these two functions to a separate instruction domain, e.g., `domain1`. This new instruction domain does not set the valid bit of the configured filter and in turn does not filter the execution of `WRPKRU`. We leverage `LD_PRELOAD` to obtain the address range of the two functions and to invoke `pkey_mprotect` for associating them with `domain1` instruction domain. We use extern function pointers for `good_code1` and

`good_code2` to facilitate leveraging `LD_PRELOAD` on them.

5.3 FlexFilt: Case Study

Filtering target instructions in dynamically generated code is more challenging than static code. In this section, we provide an experimental study to demonstrate the advantages of leveraging FlexFilt for runtime instruction filtering of dynamically generated code through Just-In-Time (JIT) compilation.

5.3.1 JIT compilation

JIT compilation dynamically compiles interpreted programming languages such as JavaScript into bytecode (an intermediate representation) or native machine code. A JavaScript engine (e.g., ChakraCore (Microsoft, 2020), SpiderMonkey (Mozilla, 2020), and V8 (Google Corporation, 2020b)) is responsible for compiling and executing the JavaScript code, memory management, and optimization. In our experiment, we rely on V8, which is Google's open-source JavaScript engine used in Chrome, Chromium, and Node.js. V8 first compiles the JavaScript code into a bytecode. Then, V8's optimizing compiler generates an optimized machine code from the bytecode.

5.3.2 V8 JIT Compilation Experiment

As a case study, we consider the scenario of leveraging Intel MPK for intra-process memory isolation of the Chromium browser. While browsing webpages, the V8 engine dynamically compiles the JavaScript code and translates it to optimized native machine code. To prevent reuse of JIT compiled code for unauthorized modification of a protection domain's permission bits, we need to ensure that the code does not contain any implicit occurrences of `WRPKRU` instruction (as an attacker can exploit an implicit occurrence of `WRPKRU` instruction using control-flow hijacking attacks and in turn elevate the privilege of a protection domain). To this end, the previous

works continuously scan the newly generated code at runtime and rewrite the code if necessary. In this section, we analyze the overhead of scanning the dynamically generated code by measuring the number of generated bytes in native machine code while browsing various webpages.

For this measurement, we built and ran the Chromium browser (Google Corporation, 2020a) on an Intel® Core™ i7-4700MQ processor @ 3.4GHz machine running Ubuntu 18.04.5 LTS. We used `v8_enable_disassembler=true` flag for building Chromium to enable disassembler support in V8. To measure the total number of generated bytes during JIT compilation, we ran Chromium with `--js-flags=''--print-bytecode''` flag and browsed the Alexa top-10 websites (Amazon Corporation, 2020). Table 5.2 shows the total size of executable bytes generated by V8 engine while browsing the Alexa top-10 websites. For each website, we report two numbers: 1) the total size of executable bytes generated when loading the frontpage of the website, and 2) the number of bytes generated per second while browsing each website for 5 minutes. For a website such as `Google.com` and `Baidu.com`, we do the browsing by searching various keywords without opening any of the search results. For each website, we repeated both the experiments, i.e., loading of the frontpage of the website and 5 minutes browsing, three times and reported the geometric mean. As shown in Table 5.2, some websites had zero executable bytes generated. When loading the frontpage of these website, they did not result in any native bytes. As a workaround for calculating the geometric mean in the presence of samples with zero values, we converted each zero value to one. On average, a binary scanning approach has to scan around 3,432 bytes and 3,258 bytes per second, respectively, for loading the Alexa top-10 pages and browsing them. In the worst case, 366,003 bytes (for `Tmall.com`) and 15,454 bytes per second (for `Taobao.com`) should be scanned. Considering 4KB pages, a binary scanning approach has to scan about

Table 5.2: The measured size of executable bytes generated for browsing the Alexa top-10 websites, on average.

Website	Executable bytes generated when loading the frontpage	Executable bytes generated per second while browsing the page
Google.com	0	3,458
Youtube.com	266,798	2,620
Tmall.com	366,003	15,323
Baidu.com	0	1,532
Qq.com	159,565	2,043
Sohu.com	34,096	2,014
Facebook.com	20,938	9,712
Taobao.com	220,299	15,454
Amazon.com	92,442	3,098
360.cn	0	400
Geometric mean	3,432	3,258

90 pages for loading `Tmall.com`, while for continuous browsing of `Taobao.com` around 4 pages should be scanned almost every 1 second. Once the binary scanning finds an implicit occurrence of a target instruction, it can rely on a binary rewriting tool to eliminate the target instruction from the code. For a continuous process such as web browsing, the binary scanning and binary rewriting should be implemented very efficiently, which is a challenging task. A dynamic binary rewriting approach incurs considerable performance overhead. In contrast, FlexFilt examines each executed instruction at hardware level with negligible performance overhead and prevents the execution of target instructions without the need for binary rewriting.

5.4 Evaluation

In this section, we discuss our experimental framework as well as FlexFilt's performance, power, and area overheads.

5.4.1 Experimental Setup

Similar to the implementation of PHMon and SealPK, we implemented FlexFilt on a RISC-V Rocket core using Chisel HDL and prototyped our full design on a Xilinx Zedboard FPGA. We modified the Linux kernel (v4.15) to add the support for FlexFilt. We leveraged SealPK's memory protection support in Linux kernel to build the instruction protection domain support for RISC-V.

To verify the correct functionality of FlexFilt, we implemented tests that created scenarios similar to the one described in Section 5.2.3. As an example, we leveraged the `WRPKR` extended instruction to implement memory protection domains. Then, we prevented the execution of the `WRPKR` instruction except in the trusted functions specified by the user. To this end, we leveraged FlexFilt's API and the `LD_PRELOAD` approach. We cross-compiled the code using RISC-V GNU toolchain and ran the program on our FPGA prototype. As expected, FlexFilt allows the execution of `WRPKR` in trusted functions and prevents its execution anywhere else in the code by causing an illegal instruction exception.

5.4.2 Experimental Results

As FlexFilt is an SE that can assist with filtering target instructions in various systems, the efficiency of our design in terms of performance, power, and area metrics are of great importance. In this section, we evaluate the efficiency of FlexFilt.

Performance Results

To demonstrate the performance overhead of integrating FlexFilt with the RISC-V Rocket processor, in this section, we evaluate FlexFilt's performance overhead using microbenchmarks and measure the context switch overhead to maintain FlexFilt's information.

Microbenchmarks

FlexFilt provides four `Flexible Filters`. During the program execution, each `Flexible Filter` receives the current instruction at the execution stage and applies its configured filter on the instruction. As the filtering operation does not need any extra cycles, we expect FlexFilt to incur negligible performance overhead. At hardware level, all the `Flexible Filters` perform the filtering operation in parallel. Hence, regardless of the number of activated configured filters, we expect FlexFilt's performance overhead to remain the same. To examine the effect of number of activated configured `Flexible Filters` on the performance overhead, we ran the `mcf` benchmark from SPEC2000 benchmark suite (Henning, 2000) for active filter count ranging from 0 to 4. We repeated each experiment 3 times and considered the geometric mean of execution times as the performance metric. As expected, this experiment showed that the execution time overhead of FlexFilt did not change with the number of activated filters. With various number of filters, the total execution time stayed the same (the geometric mean of the execution time overhead across various configurations was 0.16% with a standard deviation of less than 1%). Although we performed this experiment for only the `mcf` benchmark, we expect a similar behavior in other benchmarks.

We devised a microbenchmark to measure the overhead of configuring `Flexible Filters` as well as the overhead for applying a combination of filters to an instruction domain. Table 5.3 shows the average number of cycles to configure a `Flexible Filter` and an instruction domain. As discussed in Section 5.2.3, we provide a software API to configure FlexFilt. The software API creates a wrapper function around the `custom` instructions to facilitate their use. As expected, using the software API is more costly compared to leveraging the `custom` instructions as inline assembly. For example, leveraging `config_filter` function to configure the `Flexible Filters` takes

Table 5.3: Cycle counts for FlexFilt configuration.

Operation or Instruction	Mechanism	#Cycles
API	config_filter	46
	config_instr_domain	69
`Custom` Instruction as Inline Assembly	`SETMATCH`	4
	`SETMASK`	5
	`SETPRIV`	4
	`WRIPR`	4

46 cycles, on average, while using its corresponding `custom` instructions (`SETMATCH`, `SETMASK`, and `SETPRIV`) as inline assembly takes less than 15 cycles.

Context Switch Overhead

During the context switches, we maintain FlexFilt's information including the configuration of each `Flexible Filter` and the contents of `IPR`. As the amount of FlexFilt's information to maintain stays the same during context switches, we expect FlexFilt's context switch overhead to be the same across all applications. We measured the performance overhead of FlexFilt during context switches for 4 SPEC2000 benchmarks (Henning, 2000), i.e., bzip2, gcc, gzip, and mcf. We ran each benchmark three times and determined the geometric mean of the execution time overheads. Table 5.4 shows the performance overhead of maintaining FlexFilt's information during each context switch. On average, FlexFilt increases the execution time of each context switch by less than 2%. As expected, the context switch overhead virtually stays the same for various benchmarks. As the number of context switches varies across different benchmarks, the total performance overhead of FlexFilt is different for each benchmark. However, FlexFilt's overall performance overhead is negligible (Table 5.4). In this experiment, we used four benchmarks from SPEC2000; however, as the amount of FlexFilt's information maintained during context switches is independent of the benchmark, we expect similar context switch overheads in any application.

Table 5.4: Performance overhead of FlexFilt due to maintaining FlexFilt's information during context switches.

Benchmark	Increase in Context Switch Execution Time	Overall Execution Time Overhead
bzip2	1.77%	0.02%
gcc	1.81%	0.07%
gzip	1.75%	0.05%
mcf	1.78	0.16%

Table 5.5: The FPGA utilization of the Rocket core enhanced with FlexFilt compared to the baseline Rocket core.

	Baseline		Rocket Core + FlexFilt	
	#Used	% Utilization	#Used	% Utilization
Total Slice LUTs	32030	60.21	32584	61.25
LUTs as logic	30907	58.1	31409	59.04
LUTs as Memory	1123	6.45	1175	6.75
Slice Registers as Flip Flop	16506	15.51	17056	16.03

FPGA Resource Utilization

In our FPGA prototype, the RISC-V Rocket core enhanced with FlexFilt operated with a maximum frequency of 25MHz. The unmodified baseline RISC-V Rocket core also has the same maximum frequency, i.e., our microarchitectural modifications did not reduce the operating frequency. Table 5.5 shows the FPGA resource utilization of an enhanced Rocket core with FlexFilt compared to the baseline Rocket core. Accordingly, as FlexFilt has less than 1% area overhead, we estimate the power overhead of FlexFilt to be negligible.

Flexible Filtering Capability

Our `Flexible Filters` enable a software developer to filter instructions at bit granularity. For example, a `ret` instruction in RISC-V ISA is a pseudoinstruction defined using a `JALR` (jump and link register) instruction with `rd = x0`, `rs1 = x1`, and `imm = 0`. Rather than filtering all the `JALR` instructions, the software developer can configure a `Flexible Filter` to only filter a `ret` instruction (`MATCH = 0x00008067` and

Table 5.6: Opcode-based grouping of RV64I instructions (Waterman et al., 2019a).

Instruction group	Opcode	Number of instructions
LUI	0110111	1
AUIPC	0010111	1
JAL	1101111	1
JALR	1100111	1
BRANCH	1100011	6
LOAD	0000011	7
STORE	0100011	4
ALUI	0010011	16
ALU	0110011	15
FENCE	0001111	1
ECALL/EBREAK	1110011	2

`MASK = 0xffff0000`). In addition to filtering a specific instruction or a subset of an instruction (e.g., a `ret` instruction), `Flexible Filters` can be used to filter a group of instructions. To clarify this capability, we examined the RV64I base instruction set, which consists of 55 instructions (Waterman et al., 2019a). As shown in Table 5.6, these instructions can be divided into 11 groups, based on their `opcodes`. A software developer can configure a single `Flexible Filter` to filter any of the above-mentioned group of instructions. Two or more groups of instructions can be merged together and form a larger instruction filtering group. As an example, consider the scenario where a security developer defines secure versions of load and store instructions. Then, she specifies a trusted portion of the code, where she replaces all the load and store instructions with their secure counterparts. To ensure that the trusted code does not execute an ordinary store or load instruction at runtime, we can leverage FlexFilt. We group all the `LOAD` and `STORE` instructions together (11 instructions with the `opcode = 0-00011`) and configure one `Flexible Filter` to prevent the execution of all the instructions in this group.

5.5 Discussion

We configure FlexFilt once a process gets loaded (or during `LD_PRELOAD`). To prevent further modifications to FlexFilt's configuration, the software developer can leverage one of the `Flexible Filters` to prevent the execution of the configuration `custom` instructions. This `Flexible Filter` can be sealed at hardware level from further modifications. As a result, once the process is loaded and FlexFilt is configured, any further execution of the configuring `custom` instructions causes an exception.

FlexFilt is capable of filtering instructions both in the user space and kernel space. The software developer specifies the target privilege level of each `Flexible Filter`. At hardware level, FlexFilt examines the `priv` bit to distinguish between various privilege levels. To prevent the execution of target instructions at kernel level, we can configure `Flexible Filters` with `supervisor` privilege for each process. Alternatively, FlexFilt can be configured to prevent the execution of target instructions at kernel level for all the processes during the loading of the Linux kernel. Sealing FlexFilt's configuration raises the bar against kernel-level adversaries. However, as OS is responsible for maintaining FlexFilt's information during context switches, protecting FlexFilt completely against kernel-level adversaries requires further study. We leave this study as part of our future work.

In previous sections, we discussed FlexFilt's capability in filtering the execution of target instructions in untrusted parts of the code. FlexFilt allows the execution of target instructions in trusted parts of the code. However, an adversary might leverage the vulnerabilities in untrusted parts of the code (e.g., buffer overflow) to launch a control-flow hijacking attack and execute the target instructions. To prevent such attacks, we can protect the entrance and exit points of trusted functions using trampoline or call gates.

5.6 Summary

In this chapter, we have presented FlexFilt, an SE for runtime filtering of various user-defined instructions. The goal of this SE is to assist with securing various isolation-based mechanisms by preventing the execution of unsafe instructions in untrusted parts of the code. FlexFilt supports up to 16 unique instruction domains. To this end, FlexFilt creates instruction protection domains by assigning the same protection key to a group of executable pages. At hardware level, FlexFilt provides configurable filters and each instruction protection domain can be configured to apply a combination of the configured filters to its corresponding pages. We have demonstrated the feasibility of FlexFilt's design by implementing a practical prototype, consisting of the RISC-V Rocket core enhanced with FlexFilt and the Linux kernel support for FlexFilt, on an FPGA.

Chapter 6

Conclusion and Future Work

In this thesis, we present and evaluate an array of hardware-assisted engines for efficiently enforcing a variety of security policies at runtime. In this chapter, we summarize the contributions of this thesis, discuss the limitations of our proposed hardware engines, and outline the directions for future work.

6.1 Summary of Major Contributions

In this work, we devise a methodology that incorporates an array of hardware engines as a security layer on top of the existing RISC-V Rocket processor design. We propose a PE, PHMon, in form of a minimally-invasive and efficient programmable hardware monitor. We demonstrate that by enforcing an event-action monitoring model, PHMon can enforce a variety of security policies. Additionally, PHMon can assist with detecting software bugs and security vulnerabilities. We interface PHMon with the RISC-V Rocket processor and minimally modify the commit (write-back) stage of the pipeline to expose an instruction execution trace to PHMon. A user can monitor various events on the collected execution trace. After detecting an event, PHMon performs a series of follow-up actions to enforce the security policies specified by the user. As various processes in the system can have different security requirements, PHMon enables a per-process monitoring capability. To this end, we provide the OS support for PHMon by maintaining its information during context switches. We demonstrate the versatility of PHMon through five representative use cases, a shadow

stack, a hardware-accelerated fuzzing engine, information leak prevention, hardware-accelerated debugging, and a code coverage engine. Our evaluation of PHMon on a realistic FPGA prototype demonstrates that PHMon improves the performance of fuzzing by 16× over the state-of-the-art software-based implementation. Additionally, a PHMon-based shadow stack implementation has 0.9% performance overhead, on average.

We propose an SE, SealPK, to enforce an efficient and secure protection key-based intra-process memory isolation mechanism for the RISC-V ISA. SealPK provides a per-page protection key and supports up to 1024 domains (64× more than Intel MPK) by leveraging the 10 unused bits available in the RISC-V Sv39 PTE. To provide higher security guarantees than Intel MPK, we propose three novel sealing features to prevent an attacker from modifying sealed domains, sealed pages, and sealed permissions. Additionally, we propose an OS-level lazy deallocation approach to mitigate the pkey use-after-free problem of Intel MPK. We demonstrate the efficiency of SealPK by leveraging it to implement an isolated shadow stack on an FPGA prototype. The SealPK-based isolated shadow stack prototype is, on average, 80× faster than an isolated implementation using mprotect.

To guarantee the integrity of various isolation-based security mechanisms, we propose FlexFilt, a flexible implementation of an SE. FlexFilt efficiently prevents the execution of unsafe instructions that could compromise the integrity of the isolation in-place. While previous works are tailored to filter the execution of certain target instructions, FlexFilt provides configurable filters to prevent the execution of various instructions at page granularity. In addition to filtering user-space instructions, FlexFilt is capable of filtering privileged instructions (i.e., supervisor mode and hypervisor mode). We demonstrate the feasibility of FlexFilt's design by implementing it on the RISC-V Rocket processor, providing the OS support for it, and prototyping the full

design on an FPGA. Our FPGA evaluation shows the FlexFilt design has negligible area and power overheads.

6.2 Limitations

Despite the capabilities of our hardware engines in enforcing a variety of security policies, they have some limitations. In the current implementation, we interface PHMon with an in-order processor. In principle, we can interface PHMon with an out-of-order processor and simply collect the execution trace (commit log) from the commit stage. Collecting the execution trace from different stages of an out-of-order processor requires further investigations. Additionally, in this work, PHMon is limited to monitoring the execution of programs on a single-core processor. Leveraging PHMon for enforcing security policies in a multi-core processor requires further studies and experiments.

SealPK's sealing features improve the security guarantees of memory protection-key based approaches. However, SealPK's capability in sealing permissions is limited to restricting `WRPKRU` instructions in only one trusted function (contiguous memory addresses). FlexFilt's capability in preventing the execution of unsafe instructions at page granularity addresses the above-mentioned limitation of SealPK. Nevertheless, both SealPK and FlexFilt are limited to creating protection domains at page granularity. Supporting finer-grained protection domains requires further study. Additionally, SealPK provides up to 1024 memory protection domains, which might not be sufficient for some use cases.

6.3 Future Research Directions

In this subsection, we outline the potential directions for future work.

6.3.1 Extending Our Array of Security Engines to Other Computing Systems

In this work, we interfaced our hardware security engines with an in-order RISC-V Rocket processor. A potential future direction is extending our methodology, i.e., providing security through an array of hardware engines, to other computing systems from small-scale systems such as Internet of Things (IoT) devices to large-scale systems such as heterogeneous multi-core processors.

Extending Our Security Engines to IoT Devices

A potential future direction is adapting the design of our hardware engines for IoT devices. While IoT devices can be used in a wide range of applications, our focus here is on deeply embedded devices relying on smaller microcontrollers such as ARM Cortex-M family. The RISC-V Rocket processor is comparable to the higher-end ARM Cortex-A5 core (Lee et al., 2014); hence, our current design for hardware engines are not directly applicable to our target IoT devices.

During the past decade, the number of IoT devices has increased drastically. The number of IoT devices in 2020 was more than 11.7 billion and by 2025 it is projected to reach 30.9 billion devices around the world (Holst, 2021). In recent years, there has been a growing concern regarding the security attacks on IoT devices. According to HP analysis (HP Corporation, 2014), 70% of the most commonly used IoT devices suffer from security vulnerabilities and on average each IoT device has 25 vulnerabilities. As IoT devices have limited computational processing, low power budget, and limited memory, it is a challenging task to provide efficient security solutions for them. As a potential future direction, we can develop light-weight and low-power SEs for securing deeply embedded IoT devices. Such IoT devices typically consist of pre-defined software tasks. Hence, we can leverage our instruction filtering engine to limit the execution of valid instructions to a pre-defined set and in turn prevent the

execution of any instruction not employed in the existing software tasks. Additionally, developing light-weight PEs for IoT devices enables us to update the security policies at hardware level through software configuration and without the need for updating the IoT device hardware. As PHMon is designed for higher-end microcontrollers such as RISC-V Rocket core, it cannot directly be deployed for deeply embedded IoT devices. As part of our future work, we can focus on designing a low power PE suited for integration with smaller microcontrollers like ARM Cortex-M family.

Extending Our Security Engines to Out-of-Order Processors

In principle, our hardware engines are also applicable to out-of-order processors. However, interfacing our hardware engines with an out-of-order processor requires additional efforts. One potential future direction is extending our hardware engines for the out-of-order RISC-V BOOM processor (Asanovic et al., 2015; Celio et al., 2017). Figure 6·1 shows an example integration of our hardware engines with a processing tile consisting of various components including a BOOM processor. Recently, the RoCC interface has been extended for the RISC-V BOOM processor; however, the current implementation does not support interrupts, exceptions, or direct access to the L1 data cache. As communicating with the L1 data cache and the capability to trigger interrupts are necessary parts of PHMon's follow-up actions, we need to extend the RoCC interface with these capabilities for the BOOM processor.

To extend PHMon's monitoring capabilities to an out-of-order processor, we can simply collect the execution trace information from the commit stage of the pipeline. Although it is feasible to collect the trace information from different stages of an out-of-order core's pipeline, such information might be leveraged by an attacker for side-channel attacks. Evaluating the pros and cons of collecting execution trace information from various stages of the pipeline could be another future direction. As an out-of-order core has a more complex design compared to an in-order core, we

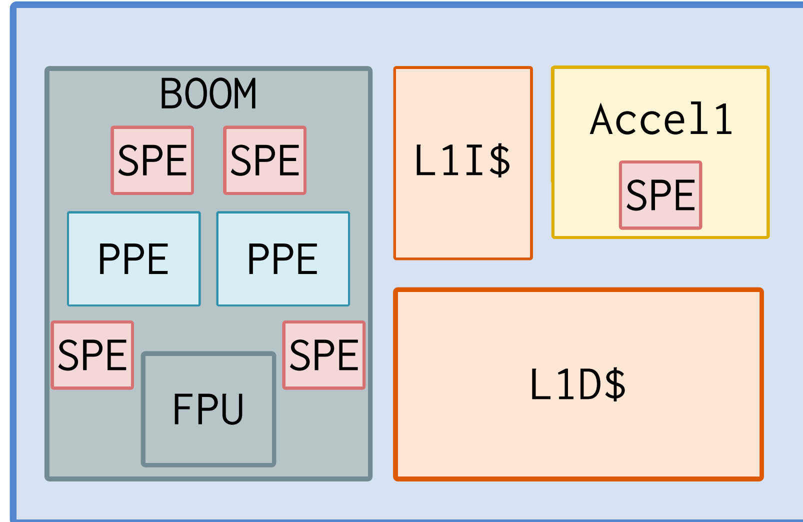


Figure 6·1: An example of integrating various PEs and SEs to a RISC-V processing tile consisting of an out-of-order RISC-V BOOM processor and an accelerator.

can potentially develop a number of SEs to enforce security policies in various parts of an out-of-order core. As an example, an SE could be dedicated for protecting the sensitive parts of the code through execution obfuscation. We can investigate the possibility of designing an SE that prevents time-based side-channel attacks by providing constant-time execution (e.g., by injecting nop instructions at runtime or by stalling the pipeline) for sensitive parts of the code.

Extending Our Security Engines to Multi-Core Processors

In a multi-core processor, each core can be integrated with a number of PEs and SEs. While in this work we only discussed the incorporation of hardware engines with a processor, a potential future direction is providing a distributed array of hardware engines across a homogeneous multi-core system. In such a system, in addition to PEs and SEs for each core, we can incorporate different hardware engines in various parts of the system, e.g., cache subsystem and I/O networks. For example, we can design a PE to monitor the memory transactions in various parts of the memory

subsystem. Such a PE can be designed as a simple monitor, which is programmed to detect anomalies in memory transactions. Once the PE detects an anomaly, it can transfer the log of the most recent memory transactions to stronger hardware engine for further investigations. Similarly, PEs and SEs can be deployed to monitor the transactions on I/O bridges, memory controllers, and high-speed I/O devices for detecting anomalies or enforcing specific transaction policies.

A distributed array of hardware engines can also be deployed for a heterogeneous multi-core System-on-Chip (SoC). As shown in Figure 6·2, in such a system, different cores and computing nodes can leverage various number of PEs and SEs. Each processing tile in such a system can consist of an in-order or out-of-order core integrated with a number of accelerators. An accelerator might be an untrusted closed-source IP integrated with other trusted components. Such a closed-source accelerator might contain malicious components threatening the security of the system. We can design an SE to monitor and confine the transactions between the accelerator and the rest of the system and/or to detect anomalies in communications.

The communication between various SEs and PEs in a large-scale SoC requires further investigation. Simpler hardware engines can transfer their signals into a central processing unit for further processing and performing follow-up actions. Depending on the size of the SoC, another design option could be providing distributed processing units. As a future direction, we can examine various design options for integrating hardware engines into the hierarchies of an SoC.

6.3.2 An Unlimited Number of Memory Protection Domains

In this thesis, we develop SealPK that creates memory protection domains for intra-process memory isolation. SealPK utilizes the unused bits of Sv39 PTEs to store the pkey, which allows us to support up to 1024 unique memory protection domains. A potential solution to support more than 1024 pkeys is virtualization. Although the

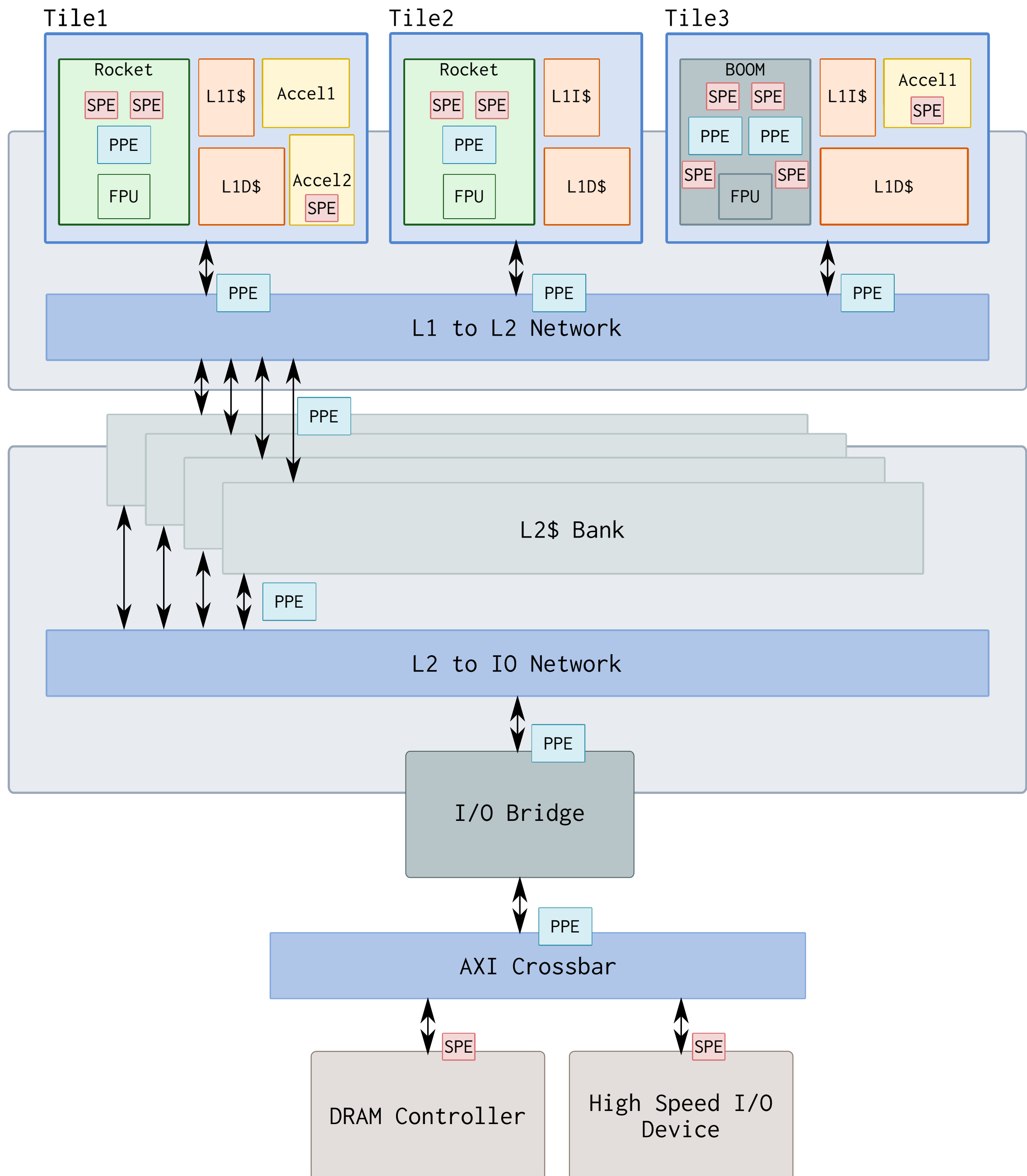


Figure 6·2: An example of a heterogeneous SoC generated by RocketChip generator and integrated with an array of SEs and PEs.

hardware-based virtualization technique by Xu et al. (Xu et al., 2020) is efficient, it is tailored for the specific use case of PMOs. As a potential future direction, we can develop a generic solution, where we integrate libmpk (a software-based virtualization technique) with SealPK. Although with a virtualization technique, e.g., libmpk and the work by Xu et al. (Xu et al., 2020), the user can create more than 1024 sequential domains, in reality we are still limited to 1024 concurrent physical pkeys. To support an `unlimited` number of domains, we can leverage a generic hardware-assisted approach. In this approach, rather than storing the pkey information as part of a PTE, we will use a hierarchical structure similar to that of page tables to store the associated pkey of each page. To avoid a pkey walk and a subsequent memory request for each DTLB access, we will add a hardware structure similar to TLB to cache the range of virtual addresses associated with a pkey as well as the pkey's permission bits. Even with this approach, the number of domains is still restricted by the size of pkey. This potential future direction could also be used for cases where the PTE does not have any unused bits. In such cases, we can store the pkey information in a separate structure, similar to the solution described for the `unlimited` pkey.

6.3.3 Fine-grained Protection Domains

In this thesis, we proposed two SEs that provide protection domains at page granularity. SealPK creates memory protection domains while FlexFilt provides instruction protection domains. Creating the protection domains at page granularity enables us to leverage the existing access permissions at PTE level and TLB level. However, depending on the size of allocated memories, creating protection domains at page granularity might result in unused chunks of memory. A potential future direction is providing a flexible fine-granular support for creating protection domains. Such an implementation requires both hardware and OS support. Recently, Intel VT-X provided the support for sub-page permissions, where it enables setting write

access permissions at 128 byte granularity (Intel Corporation, 2020). However, we need a feature that enables/disables both read and write permissions at fine granularity. Ideally, we are interested in an implementation that enables us to create flexible protection domains with varying sizes from one byte to a page. However, the performance overhead of defining protection domains at byte granularity will be prohibitive. Performing a through profiling experiment on a range of real applications such as Chromium browser will provide us with a better understanding on the distribution and size of allocated memories. Subsequently, we can decide on the smallest supported granularity for protection domains.

6.4 Final Remarks

In summary, we propose, implement, and evaluate a hybrid array of hardware engines, consisting of a PE and two SEs, as a security layer on top of the RISC-V Rocket core. Our proposed PE enables us to enforce a variety of security policies and also assists us with finding software bugs and security vulnerabilities. Our proposed SEs provide hardware assistance for intra-process memory isolation and runtime filtering of unsafe instructions. In this thesis, we have taken the necessary steps towards evaluating our proposed hardware security engines in a realistic environment including an FPGA prototype of the hardware design with the full Linux-based software stack. We strongly believe that such a realistic evaluation environment paves the path for future research in the area of hardware security.

References

Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC'09)*, 13(1):1–40.

Aizatsky, M., Serebryany, K., Chang, O., Arya, A., and Whittaker, M. (2016). Announcing OSS-Fuzz: continuous fuzzing for open source software. [online] `https://bit.ly/3voiFJA`. (Accessed on 02/24/2021).

Amazon Corporation (2020). The top 500 sites on the web. [online] `https://www.alexa.com/topsites`. (Accessed on 03/14/2021).

AMD Corporation (2006). AMD64 architecture programmer's manual volume 2: system programming. [online] `https://bit.ly/3klGmxg`. (Accessed on 02/24/2021).

Anati, I., Gueron, S., Johnson, S. P., and Scarlata, V. R. (2013). Innovative technology for CPU based attestation and sealing. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, volume 13, page 7. ACM.

Anderson, J. P. (1972). Computer security technology planning study. Technical Report ESD-TR-73-51, The Mitre Corporation, Air Force Systems Division, Hanscom AFB, Bedford, Massachusetts. [online] `https://bit.ly/3qQFx1L`. (Accessed on 02/24/2021).

Ariane (2018). Ariane RISC-V CPU. [online] `https://github.com/lowRISC/ariane`. (Accessed on 02/24/2021).

ARM Corporation (2009). ARM security technology, building a secure system using TrustZone technology. [online] `https://bit.ly/3qQaqUg`. (Accessed on 02/06/2021).

ARM Corporation (2018). Arm architecture reference manual ARMv7-A and ARMv7-R edition. [online] `https://bit.ly/38dRFmq`. (Accessed on 02/24/2021).

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., Lovẽ, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C. n., Twigg, S., Vo, H.,

and Waterman, A. (2016). The Rocket Chip generator. EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2016-17. [online] `https://bit.ly/38BHLuX`. (Accessed on 02/24/2021).

Asanovic, K., Patterson, D. A., and Celio, C. (2015). The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized risc-v processor. EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2015-167. [online] `https://bit.ly/2OmejSW`. (Accessed on 02/24/2021).

Austin, T. M., Breach, S. E., and Sohi, G. S. (1994). Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 290–301. ACM.

Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W. (2014). Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'14)*, pages 90–102. ACM.

Azab, A. M., Swidowski, K., Bhutkar, R., Ma, J., Shen, W., Wang, R., and Ning, P. (2016). SKEE: a lightweight secure kernel-level execution environment for ARM. In *Proceedings of Network & Distributed System Security Symposium (NDSS'16)*, volume 16, pages 21–24. Internet Society.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asañović, K. (2012). Chisel: constructing hardware in a scala embedded language. In *Proceedings of the Design Automation Conference (DAC'12)*, pages 1212–1221. IEEE.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. (2012). Dune: safe user-level access to privileged CPU features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 335–348. USENIX Association.

Blazakis, D. (2010). Interpreter exploitation: pointer inference and JIT spraying. BlackHat DC. [online] `https://bit.ly/3rLMEt0`. (Accessed on 02/24/2021).

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, pages 30–40. ACM.

Broadwell, P., Harren, M., and Sastry, N. (2003). Scrash: a system for generating secure crash information. In *Proceedings of the USENIX Security Symposium (Security'03)*, pages 273–284. USENIX Association.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'08)*, pages 27–38. ACM.

Burow, N., Zhang, X., and Payer, M. (2019). SoK: Shining light on shadow stacks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*, pages 985–999. IEEE.

Canakci, S., Delshadtehrani, L., Zhou, B., Joshi, A., and Egele, M. (2020). Efficient context-sensitive CFI enforcement through a hardware monitor. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'20)*, pages 259–279. Springer.

Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium (Security'14)*, pages 385–399. USENIX Association.

Celio, C., Chiu, P.-F., Nikolic, B., Patterson, D. A., and Asanovic, K. (2017). Boomv2: an open-source out-of-order risc-v core. In *Workshop on Computer Architecture Research with RISC-V (CARRV'17)*.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the ACM Conference on Computer and communications security (CCS'10)*, pages 559–572. ACM.

Chen, S., Falsafi, B., Gibbons, P., Kozuch, M., Mowry, T., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G., and S̃chlosser, S. (2006a). Logs and lifeguards: accelerating dynamic program monitoring. Intel Research, Technical Report IRP-TR-06-05. [online] `https://bit.ly/3bJg4Ct`. (Accessed on 02/24/2021).

Chen, S., Falsafi, B., Gibbons, P. B., Kozuch, M., Mowry, T. C., Teodorescu, R., Ailamaki, Anastassia an d Fix, L., Ganger, G. R., Lin, B., and Schlosser, S. W. (2006b). 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'06)*, pages 63–65. ACM.

Chen, S., Kozuch, M., Strigkos, T., Falsafi, B., Gibbons, P. B., Mowry, T. C., Ramachandran, V., Ruwase, O., Ryan, M., and Vlachos, E. (2008). Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the International Symposium on Computer Architecture (ISCA'08)*. ACM.

Chen, Y., Mu, D., Xu, J., Sun, Z., Shen, W., Xing, X., Lu, L., and Mao, B. (2019). PTrix: Efficient hardware-assisted fuzzing for COTS binary. In *Proceedings of the*

ACM Asia Conference on Computer and Communications Security (AsiaCCS'19), pages 633–645. ACM.

Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L. (2016). Shreds: fine-grained execution units with private memory. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'16)*, pages 56–71. IEEE.

Chen, Y.-Y., Jamkhedkar, P. A., and Lee, R. B. (2012). A software-hardware architecture for self-protecting data. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*, pages 14–27. ACM.

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., and Deng, R. H. (2014). ROPecker: a generic and practical approach for defending against ROP attack. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'14)*, pages 1–14. Internet Society.

Corliss, M. L., Lewis, E. C., and Roth, A. (2003). DISE: a programmable macro engine for customizing applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA'03)*, pages 362–373. ACM.

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

Dalton, M., Kannan, H., and Kozyrakis, C. (2007). Raksha: a flexible information flow architecture for software security. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*, pages 482–493. ACM.

Dang, T. H., Maniatis, P., and Wagner, D. (2015). The performance cost of shadow stacks and stack canaries. In *Proceedings of the Symposium on Information, Computer and Communications Security (ASIACCS'15)*, pages 555–566. ACM.

Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. (2018). SoK: the challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of the Symposium on Security and Privacy (S&P'18)*, pages 20–38. IEEE.

Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. (2012). MoCFI: a framework to mitigate Control-Flow Attacks on smartphones. In *Proceedings of Network & Distributed System Security Symposium (NDSS'12)*, volume 26, pages 27–40. Internet Society.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2011). ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the Symposium on Information, Computer and Communications Security (ASIACCS'11)*, pages 40–51. ACM.

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. (2013). On the feasibility of online malware detection with performance counters. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*, pages 559–570. ACM.

Deng, D. Y., Lo, D., Malysa, G., Schneider, S., and Suh, G. E. (2010). Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the International Symposium on Microarchitecture (MICRO'10)*, pages 137–148. IEEE.

Deng, D. Y. and Suh, G. E. (2012). High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12. IEEE.

Deng, L., Zeng, Q., and Liu, Y. (2015). ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security and Privacy Conference (IFIP SEC'15)*, pages 386–400. Springer.

Devietti, J., Blundell, C., Martin, M. M., and Zdancewic, S. (2008). 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'08)*, pages 103–114. ACM.

Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Knight Jr, T. F., Pierce, B. C., and DeHon, A. (2015). Architectural support for software-defined metadata processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, pages 487–502. ACM.

Digilent (2017). Zedboard zynq-7000 development board reference manual. [online] `https://bit.ly/3fEmRj2`. (Accessed on 28/02/2021).

Ding, R., Qian, C., Song, C., Harris, B., Kim, T., and Lee, W. (2017). Efficient protection of path-sensitive control security. In *Proceedings of the USENIX Security Symposium (Security'17)*, pages 131–148. USENIX Association.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al. (2014). The matter of heartbleed. In *Proceedings of the ACM on Internet Measurement Conference (IMC'13)*, pages 475–488. ACM.

Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., and Morris, R. (2005). Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review (OSR'05)*, 39(5):17–30.

Feldman, B. (2020). Is it safe to use zoom? [online] `https://nym.ag/3aSlk6p`. (Accessed on 02/06/2021).

Ford, B. and Cox, R. (2008). Vx32: lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference (ATC'08)*, pages 293–306. USENIX Association.

Frassetto, T., Gens, D., Liebchen, C., and Sadeghi, A.-R. (2017). Jitguard: hardening just-in-time compilers with SGX. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, pages 2405–2419. ACM.

Frassetto, T., Jauernig, P., Liebchen, C., and Sadeghi, A.-R. (2018). IMIX: in-process memory isolation extension. In *Proceedings of the USENIX Security Symposium (Security'18)*, pages 83–97. USENIX Association.

Fruhlinger, J. (2018). Marriott data breach faq: How did it happen and what was the impact? [online] `https://bit.ly/3aQy25n`. (Accessed on 02/06/2021).

Fytraki, S., Vlachos, E., Kocberber, O., Falsafi, B., and Grot, B. (2014). FADE: a programmable filtering accelerator for instruction-grain monitoring. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'14)*, pages 108–119. IEEE.

Gcov (2021). Gcov: Gnu coverage tool. [online] `https://bit.ly/3b2ynlL`. (Accessed on 28/02/2021).

Ge, X., Cui, W., and Jaeger, T. (2017). Griffin: Guarding control flows using Intel processor trace. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, pages 585–598. ACM.

Ghose, S., Gilgeous, L., Dudnik, P., Aggarwal, A., and Waxman, C. (2009). Architectural support for low overhead detection of memory violations. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*, pages 652–657.

Giffin, D. B., Levy, A., Stefan, D., Terei, D., Mazieres, D., Mitchell, J. C., and Russo, A. (2012). Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 47–60. USENIX Association.

Google Corporation (2017). OSS-Fuzz: five months later, and rewarding projects. [online] `https://bit.ly/3pOdcI9`. (Accessed on 02/24/2021).

Google Corporation (2020a). The chromium projects. [online] `https://www.chromium.org/Home`. (Accessed on 03/14/2021).

Google Corporation (2020b). What is v8? [online] `https://v8.dev/`. (Accessed on 03/14/2021).

Graham-Cumming, J. (2015). Searching for the prime suspect: how heartbleed leaked private keys. [online] `https://bit.ly/3pPPRpD`. (Accessed on 02/24/2021).

Greathouse, J. L., Xin, H., Luo, Y., and Austin, T. (2012). A case for unlimited watchpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, pages 159–172. ACM.

Gu, J., Wu, X., Li, W., Liu, N., Mi, Z., Xia, Y., and Chen, H. (2020). Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *Proceedings of the Annual Technical Conference (ATC'20)*, pages 401–417. USENIX Association.

Gu, Y., Zhao, Q., Zhang, Y., and Lin, Z. (2017). PT-CFI: transparent backward-edge control flow violation detection using Intel Processor Trace. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY'17)*, pages 173–184. ACM.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Annual International Workshop on Workload Characterization (WWC'01)*, pages 3–14. IEEE.

Heartbleed (2020). The heartbleed bug. [online] `https://heartbleed.com`. (Accessed on 02/24/2021).

Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K., and Marty, M. (2019). Hodor: intra-process isolation for high-throughput data plane libraries. In *Proceedings of USENIX Annual Technical Conference (ATC'19)*, pages 1221–1238. USENIX Association.

Henning, J. L. (2000). SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7):28–35.

Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Hex Five Corporation (2020). Multizone hex five security. [online] `https://hex-five.com/`. (Accessed on 03/14/2021).

Holst, A. (2021). Number of internet of things (iot) connected devices worldwide from 2019 to 2030. [online] `https://bit.ly/3qT8B8P`. (Accessed on 02/24/2021).

HP Corporation (2014). HP study reveals 70 percent of internet of things devices vulnerable to attack. [Online] https://bit.ly/3pPfZ3R. (Accessed on 02/24/2021).

Hu, H., Qian, C., Yagemann, C., Chung, S. P. H., Harris, W. R., Kim, T., and Lee, W. (2018). Enforcing unique code target property for control-flow integrity. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, pages 1470–1486. ACM.

IBM Corporation (2017). Power ISA version 3.0b. [online] https://bit.ly/3c1anif. (Accessed on 02/24/2021).

Intel Corporation (2006). Intel trusted execution technology. [online] https://intel.ly/3pQbrKK. (Accessed on 02/06/2021).

Intel Corporation (2013). Introduction to Intel memory protection extensions. [online] https://intel.ly/3byfNB3. (Accessed on 02/06/2021).

Intel Corporation (2016). Intel 64 and IA-32 architectures software developer's manual. [online] https://intel.ly/3lbch3R. (Accessed on 02/24/2021).

Intel Corporation (2017). Control-flow enforcement technology preview. [online] https://intel.ly/3dVraWD. (Accessed on 02/24/2021).

Intel Corporation (2019). Intel 64 and IA-32 architectures software developers manual. [online] https://intel.ly/3bi6HJS. (Accessed on 02/24/2021).

Intel Corporation (2020). Intel 64 and IA-32 architectures software developer's manual. [online] https://intel.ly/39Hysu6. (Accessed on 02/24/2021).

Johnson, J. (2021). Annual number of data breaches and exposed records in the united states from 2005 to 1st half 2020. [online] https://bit.ly/3swi2vA. (Accessed on 02/06/2021).

Karlsson, H. (2020). Openmz: a c implementation of the multizone api. M.S., KTH Royal Institute of Technology, Stockholm, Sweden. [online] https://bit.ly/38EZhyj. (Accessed on 03/14/2021).

Kazdagli, M., Reddi, V. J., and Tiwari, M. (2016). Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, pages 1–13. IEEE.

Khasawneh, K. N., Ozsoy, M., Donovick, C., Abu-Ghazaleh, N., and Ponomarev, D. (2015). Ensemble learning for low-level hardware-supported malware detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, pages 3–25. Springer.

Kim, H., Lee, J., Pratama, D., Awaludin, A. M., Kim, H., and Kwon, D. (2020). RIMI: instruction-level memory isolation for embedded systems on RISC-V. In *Proceedings of the ACM International Conference on Computer-Aided Design (ICCAD'20)*, pages 1–9. ACM.

Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. (2014). Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*, pages 361–372. IEEE.

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)*, pages 1–19. IEEE.

Koning, K., Chen, X., Bos, H., Giuffrida, C., and Athanasopoulos, E. (2017). No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the European Conference on Computer Systems (EuroSys'17)*, pages 437–452. ACM.

Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. (2007). Information flow control for standard os abstractions. *ACM SIGOPS Operating Systems Review (OSR'07)*, 41(6):321–334.

Kuznetzov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. (2018). Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. ACM and Morgan & Claypool.

Larabel, M. (2018a). Intel MPX support will be removed from Linux. [online] `https://bit.ly/3qQFx1L`. (Accessed on 02/24/2021).

Larabel, M. (2018b). Intel MPX support removed from GCC 9. [online] `https://bit.ly/2Nz4rEI`. (Accessed on 02/24/2021).

LCOV (2021). Lcov - the ltp gcov extension. [online] `https://bit.ly/3sxp6YP`. (Accessed on 28/02/2021).

Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., and Song, D. (2020). Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'20)*, pages 1–16. ACM.

Lee, Y. (2015). RISC-V "Rocket Chip" SoC generator in Chisel. [Online] `https://bit.ly/38BvF5a`. (Accessed on 02/24/2021).

Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanović, V., and Asanović, K. (2014). 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'14)*, pages 199–202. IEEE.

Li, P. S., Izraelevitz, A. M., and Bachrach, J. (2016). Specification for the FIRRTL language. EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2016-9. [online] `https://bit.ly/3rQJOCX`. (Accessed on 02/24/2021).

Lindemer, S., Midéus, G., and Raza, S. (2020). Real-time thread isolation and trusted execution on embedded RISC-V. In *Proceedings of the International Workshop on Secure RISC-V Architecture Design Exploration (SECRISC-V'20)*, pages 1–4.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: reading kernel memory from user space. In *Proceedings of the USENIX Security Symposium (Security'18)*, pages 973–990. USENIX Association.

Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. (2016). Light-weight contexts: an OS abstraction for safety and performance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 49–64. USENIX Association.

Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., and Guan, H. (2017). Transparent and efficient CFI enforcement with Intel Processor Trace. In *Proceedings of the IEEE International Symposium on High performance computer architecture (HPCA'17)*, pages 529–540. IEEE.

Liu, Y., Zhou, T., Chen, K., Chen, H., and Xia, Y. (2015). Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, pages 1607–1619. ACM.

Lo, D., Chen, T., Ismail, M., and Suh, G. E. (2015). Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'15)*, pages 662–674. IEEE.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. a., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200. ACM.

McCamant, S. and Morrisett, G. (2006). Evaluating SFI for a CISC architecture. In *Proceedings of the USENIX Security Symposium (Security'06)*, volume 10, pages 1267336–1267351. USENIX Association.

Microsoft (2020). ChakraCore. [online] `https://bit.ly/2P89rAO`. (Accessed on 03/14/2021).

Microsoft Corporation (2017). Microsoft security development lifecycle. [online] `https://bit.ly/3dKhnlY`. (Accessed on 02/24/2021).

Mijat, R. (2010). Better trace for better software: introducing the new ARM CoreSight system trace macrocell and trace memory controller. ARM, White Paper. [online] `https://bit.ly/3vwKccf`. (Accessed on 03/14/2021).

Mogosanu, L., Rane, A., and Dautenhahn, N. (2018). Microstache: a lightweight execution context for in-process safe region isolation. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'18)*, pages 359–379. Springer.

Moon, H. (2017). *Hardware techniques against memory corruption attacks*. PhD thesis, Seoul National University. [online] `https://bit.ly/3eGVgNI`. (Accessed on 28/02/2021).

Mozilla (2020). SpiderMonkey: The Mozilla JavaScript runtime. [online] `https://mzl.la/3pQdnTp`. (Accessed on 03/14/2021).

Myers, A. C. (1999). JFlow: practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241. ACM.

Nagarajan, V., Kim, H.-S., Wu, Y., and Gupta, R. (2008). Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures (INTERACT'08)*, pages 1–10.

Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2012). Watchdog: hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the International Symposium on Computer Architecture (ISCA'12)*, pages 189–200. ACM.

Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2009). Softbound: highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 245–258. ACM.

Nangate Corporation (2008). 45nm open cell library. [online] `http://www.nangate.com/`. (Accessed on 28/02/2021).

Newsome, J. and Song, D. (2005). 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'05)*, pages 1–17. Internet Society.

Owaida, A. (2018). Meltdown and spectre. [online] `https://bit.ly/2NEpHcd`. (Accessed on 02/24/2021).

Owaida, A. (2020). Microsoft patch tuesday fixes 17 critical flaws, windows zero-day. [online] `https://bit.ly/2NEpHcd`. (Accessed on 02/24/2021).

Ozdoganoglu, H., Vijaykumar, T., Brodley, C. E., Kuperman, B. A., and Jalote, A. (2006). SmashGuard: a hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers (TC'06)*, 55(10).

Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. (2015). Malware-aware processors: a framework for efficient online malware detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'15)*, pages 651–661. IEEE.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the USENIX Security Symposium (Security'13)*, pages 447–462. USENIX Association.

Park, S., Lee, S., Xu, W., Moon, H., and Kim, T. (2019). libmpk: software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*, pages 241–254. USENIX Association.

Price, D. (2020). Is zoom safe to use? 6 privacy issues to consider. [online] `https://www.muo.com/is-zoom-safe/`. (Accessed on 02/06/2021).

Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. (2006). Lift: a low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE.

Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education (WCAE'04)*, pages 22–es. ACM.

Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC'00)*, 3(1).

Schrammel, D., Weiser, S., Steinegger, S., Schwarzl, M., Schwarz, M., Mangard, S., and Gruss, D. (2020). Donky: domain keys–efficient in-process isolation for RISC-V and x86. In *Proceedings of the USENIX Security Symposium (Security'20)*, pages 1677–1694. USENIX Association.

Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. (2017). kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium (Security'17)*, pages 167–182. USENIX Association.

Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., and Holz, T. (2014). Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, pages 88–108. Springer.

Sehr, D., Muth, R., Biffle, C. L., Khimenko, V., Pasko, E., Yee, B., Schimpf, K., and Chen, B. (2010). Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the USENIX Security Symposium (Security'10)*, pages 1–11. USENIX Association.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and communications security (CCS'07)*, pages 552–561. ACM.

Singh, B., Evtyushkin, D., Elwell, J., Riley, R., and Cervesato, I. (2017). On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS'17)*, pages 483–493. ACM.

Sinha, K. and Sethumadhavan, S. (2018). Practical memory safety with REST. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*, pages 600–611. IEEE.

Sinnadurai, S., Zhao, Q., and fai Wong, W. (2008). Transparent runtime shadow stack: protection against malicious return address modifications. Citeseer. [online] `https://bit.ly/2Ng6tJM`. (Accessed on 28/02/2021).

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. (2013). Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*, pages 574–588. IEEE.

Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., and Paek, Y. (2016). HDFI: hardware-assisted data-flow isolation. In *Proceedings of the Symposium on Security and Privacy (S&P)*, pages 1–17. IEEE.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*, pages 1–16. Internet Society.

Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. (2004). Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, pages 85–96. ACM.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). SoK: Eternal war in memory. In *Proceedings of the Symposium on Security and Privacy (S&P'13)*, pages 48–62. IEEE.

Tang, A., Sethumadhavan, S., and Stolfo, S. J. (2014). Unsupervised anomaly-based malware detection using hardware features. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, pages 109–129. Springer.

Thoziyoor, S., Muralimanohar, N., Ahn, J. H., and Jouppi, N. P. (2008). CACTI 5.1. HPL-2008-20, HP Labs. [online] `https://bit.ly/2Nm1S98`. (Accessed on 28/02/2021).

Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. (2019). ERIM: secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the USENIX Security Symposium (Security'19)*, pages 1221–1238. USENIX Association.

Venkataramani, G., Doudalis, I., Solihin, Y., and Prvulovic, M. (2008). Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'08)*, pages 173–184. IEEE.

Venkataramani, G., Roemer, B., Solihin, Y., and Prvulovic, M. (2007). Memtracker: efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'07)*, pages 273–284. IEEE.

Vilanova, L., Ben-Yehuda, M., Navarro, N., Etsion, Y., and Valero, M. (2014). CODOMs: Protecting software with code-centric memory domains. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*, pages 469–480. ACM.

Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. (1993). Efficient software-based fault isolation. In *Proceedings of the ACM symposium on Operating systems principles (SOSP'93)*, pages 203–216. ACM.

Wang, X., Chai, S., Isnardi, M., Lim, S., and Karri, R. (2016). Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization (TACO'16)*, 13(1):1–23.

Warren, T. (2020). Microsoft patches windows 10 security flaw discovered by the nsa. [online] `https://bit.ly/3pRLryz`. (Accessed on 02/24/2021).

Waterman, A., Asanovic, K., and SiFive Inc. (2019a). The RISC-V instruction set manual, Volume I: unprivileged ISA, document version 20191213. [online] `https://bit.ly/3aQn14f`. (Accessed on 03/14/2021).

Waterman, A., Asanovic, K., and SiFive Inc. (2019b). The RISC-V instruction set manual Volume II: Privileged architecture, document version 20190608-priv-msu-ratified. [online] `https://bit.ly/3aQn14f`. (Accessed on 03/14/2021).

Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K. (2011). The RISC-V instruction set manual, volume i: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley. [online] `https://bit.ly/3rOwC1w`. (Accessed on 03/14/2021).

Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., et al. (2015). Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*, pages 20–37. IEEE.

Wikipedia (2017). 2017 equifax data breach — Wikipedia, The Free Encyclopedia. [online] `https://bit.ly/3bQ9I39`. (Accessed on 02/06/2021).

Wikipedia (2021). Heartbleed — Wikipedia, The Free Encyclopedia. [online] `https://en.wikipedia.org/wiki/Heartbleed`. (Accessed on 02/24/2021).

Wu, Y., Liu, Y., Liu, R., Chen, H., Zang, B., and Guan, H. (2018). Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'18)*, pages 441–453. IEEE.

Xu, Y., Dunn, A. M., Hofmann, O. S., Lee, M. Z., Mehdi, S. A., and Witchel, E. (2014). Application-defined decentralized access control. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, pages 395–407. USENIX Association.

Xu, Y., Ye, C., Solihin, Y., and Shen, X. (2020). Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *Proceedings*

of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'20), pages 680–692. IEEE.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. (2009). Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the Symposium on Security and Privacy (S&P'09)*, pages 79–93. IEEE.

Yuan, E. S. (2020). A message to our users. [online] `https://bit.ly/3dI3Lrj`. (Accessed on 02/06/2021).

Yuan, P., Zeng, Q., and Ding, X. (2015). Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, pages 66–85. Springer.

Zalewski, M. (2017). American fuzzy lop (AFL) fuzzer. [online] `http://lcamtuf.coredump.cx/afl/`. (Accessed on 02/24/2021).

Zeldovich, N., Boyd-Wickizer, S., and Mazieres, D. (2008). Securing distributed systems with information flow control. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, volume 8, pages 293–308. USENIX Association.

Zhang, M., Qiao, R., Hasabnis, N., and Sekar, R. (2014). A platform for secure static binary instrumentation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE'14)*, pages 129–140. ACM.

Zhao, L., Li, G., De Sutter, B., and Regehr, J. (2011). ARMor: fully verified software fault isolation. In *Proceedings of the ACM International Conference on Embedded software (EMSOFT'11)*, pages 289–298. ACM.

Zhou, B., Gupta, A., Jahanshahi, R., Egele, M., and Joshi, A. (2018). Hardware performance counters can detect malware: myth or fact? In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS'18)*, pages 457–468. ACM.

Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., and Walls, R. J. (2020). Silhouette: efficient protected shadow stacks for embedded systems. In *Proceedings of the USENIX Security (Security'20)*, pages 1219–1236. USENIX Association.

Zhou, P., Qin, F., Liu, W., Zhou, Y., and Torrellas, J. (2004). iWatcher: efficient architectural support for software debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*, pages 224–235. IEEE.

Zhou, P., Teodorescu, R., and Zhou, Y. (2007). HARD: hardware-assisted lockset-based race detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'07)*, pages 121–132. IEEE.

CURRICULUM VITAE