Enforcing Low-Cost Security for ARM

by

Zhuojia Shen

Submitted in Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor John Criswell

Department of Computer Science
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2023

To my wife, Xiaowan, and my family.

Table of Contents

Biographical Sketch

Zhuojia Shen (沈卓佳) was born in Ningbo, Zhejiang, China. He attended Beijing Institute of Technology from 2012 to 2016, during which he co-founded with his college classmates a campus e-commerce start-up company and served as CTO. After graduating in 2016 with a Bachelor of Science degree in Computer Science and Technology, he started his doctorate in the Department of Computer Science at the University of Rochester, advised by Professor John Criswell. His doctoral study is in the broad area of systems security, with an initial focus on software defenses against side-channel attacks. His research interests later shifted to low-cost security enforcement for ARM systems. During his doctorate, he received a Master of Science degree in Computer Science in 2019 and interned twice at VMware in the summer of 2019 and 2020, respectively.

The following publications and preprints were a result of the work conducted during his doctoral study:

- *InversOS: Efficient Control-Flow Protection for AArch64 Applications with Privilege Inversion*. Zhuojia Shen and John Criswell. arXiv e-Print 2304.08717. April 2023.

- *Randezvous: Making Randomization Effective on MCUs*. Zhuojia Shen, Komail Dharsee, and John Criswell. In Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22). Austin, TX, USA. December 2022.

- *Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage*. Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. In Proceedings of the 31st USENIX Security Symposium (Security '22). Boston, MA, USA. August 2022.

- *Fast Execute-Only Memory for Embedded Systems*. Zhuojia Shen, Komail Dharsee, and John Criswell. In Proceedings of the 2020 IEEE Secure Development Conference (SecDev '20). ~~Atlanta, GA, USA~~ (Virtual). September 2020.

- *Silhouette: Efficient Protected Shadow Stacks for Embedded Systems*. Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. In Proceedings of the 29th USENIX Security Symposium (Security '20). ~~Boston, MA, USA~~ (Virtual). August 2020.

- *Restricting Control Flow During Speculative Execution with Venkman*. Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. arXiv e-Print 1903.10651. March 2019.

- *POSTER: Restricting Control Flow During Speculative Execution*. Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Toronto, ON, Canada. October 2018.

- *Shielding Software from Privileged Side-Channel Attacks*. Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. In Proceedings of the 27th USENIX Security Symposium (Security '18). Baltimore, MD, USA. August 2018.

- *Spectres, Virtual Ghosts, and Hardware Support*. Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. In Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18). Los Angeles, CA, USA. June 2018.

Acknowledgments

First, I would like to thank my advisor, Professor John Criswell, who brought me into the systems security research area and taught me how to do research independently. Throughout my PhD journey, I have always been impressed by John, not only for his technical knowledge on computer systems and security but also for his rigorous attitude towards research. These two qualities have also profoundly influenced me, shaping me into a better engineer and researcher. John has also been a great mentor; I would not have accomplished any milestones without his guidance and support, and I feel fortunate and honored to have worked with him.

I would like to thank my committee members, Professors Sandhya Dwarkadas, Michael L. Scott, and Ziming Zhao. Their valuable comments and suggestions during six-month reviews have made my research work ever better. Sandhya collaborated in two research projects in my early PhD years, through which I got to admire her great attention to miniscule technical details that turned out to have a large impact. Michael has always been a role model for me; I appreciate his quick response to all my questions and requests. Ziming is very knowledgeable on ARM systems and has consistently provided critical feedback to my research projects.

I would like to thank all my other collaborators, including Professor Robert J. Walls, Professor Alan L. Cox, Komail Dharsee, Jie Zhou, Yufei Du, Xiaowan Dong, Lele Ma, and Divya Ojha. It was a great pleasure to work with these brilliant minds, and I learned a lot from all of them.

I would like to acknowledge all the help provided by the rest of the department's faculty and staff members. In particular, thanks to our past and current Graduate Coordinators, Michelle Kiso, Emily Tevens, and Robin Clark, for working behind the scenes and making the long journey as smooth as possible.

I would like to extend my thanks to my managers, mentors, and colleagues at VMware during my internships. Thanks to Joyce Spencer and Dr. Jiajun Cao for giving me the first internship opportunity and an invaluable in-person industry experience. Thanks to Rakesh Agarwal and Dr. Zheng Cui for creating a remote working environment in which I was able to learn and grow as much as in a face-to-face scenario. Special thanks to Dr. Karen Zee for her kindness and help during my time at VMware.

I would also like to thank all my friends throughout my PhD life, including but not limited to, Vojtěch Aschenbrenner, Wentao Cai, Sayak Chakraborti, Dong Chen, Jie Chen, Lele Chen, Tianlang Chen, Alan Po-Chun Chiu, Komail Dharsee, Xiaowan Dong, Mingzhe Du, Yufei Du, Mohammad Hossein Faghihi Sereshgi, Xui Feng, Yang Feng, Yu Feng, Yiming Gan, Ning Gu, Shupeng Gui, Md. Kamrul Hasan, Tianran Hu, Weijian Li, Wenbin Li, Haotian Liang, Fangzhou Liu, Shuyang Liu, Menglin Lou, Xinwei Lu, Fanyang Meng, Divya Ojha, Georgiy Platonov, Xiaochang Peng, Yiyun Peng, Mengshi Qi, Parker Riley, Samiha Samrose, Jing Shi, Chi Shu, Qiuyuan Song, Yapeng Tian, Haosen Wen, Tianxin Xie, Wei Xiong, Tiancheng Xu, Zhengyuan Yang, Quanzeng You, Chen Yu, Anis Zaman, Shuang Zhai, Tianqin Zhao, Shuwen Zhang, Songyang Zhang, Wei Zhang, Xiong Zhang, Zheng Zhang, Hao Zhou, Jie Zhou, Xiaofei Zhou, and Wei Zhu. I cherish all the great time we had together.

Lastly, I would like to express my gratitude to my family, especially my wife Xiaowan Dong and my mother Hong Shen. Their love, support, and encouragement through all the ups and downs have been a great source of power for me to take the challenges heads-on and to pursue my dreams.

Abstract

A great share of today's computer systems, including most embedded and microcontroller (MCU) systems and an increasing number of desktops and servers, run a processor based on ARM architectures. Securing these systems against runtime attacks that exploit common software vulnerabilities poses great challenges; effective mitigations usually incur prohibitive overhead, while practical defenses typically only provide less-than-ideal security guarantees.

This dissertation makes four contributions to low-cost security policy enforcement for ARM-based systems. First, we develop a novel compiler-based intra-address space isolation technique and use it to efficiently enforce return address integrity for bare-metal embedded applications running on ARM-based MCUs. Second, we leverage ARM's hardware debugging facilities to enforce execute-only memory with negligible overhead. Third, utilizing the execute-only memory we developed and combining compiler- and hardware-based techniques, we measurably strengthen layout randomization defenses on ARM-based MCUs to effectively and efficiently resist leakage-equipped attacks and their brute force variants. Lastly, we extend hardware-assisted write-protected shadow stacks to AArch64 (64-bit ARM) user-space applications, using a novel combination of commonly available AArch64 hardware features, operating system kernel modifications, and compiler transformations.

Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors John Criswell (advisor) and Michael L. Scott of the Department of Computer Science at the University of Rochester, Professor Sandhya Dwarkadas of the Department of Computer Science at the University of Virginia (a former professor of the Department of Computer Science at the University of Rochester), and Professor Ziming Zhao of the Department of Computer Science and Engineering at the University at Buffalo. Professor John Criswell collaborated in all the projects presented in this dissertation. Professor Robert J. Walls of the Department of Computer Science at Worcester Polytechnic Institute contributed to the initial design of Silhouette (Chapter 3). Jie Zhou, Yufei Du, and Lele Ma collaborated in the implementation and evaluation of Silhouette; Jie Zhou implemented the initial prototype of store hardening and led the security evaluation of Silhouette, Yufei Du implemented the shadow stack, and Lele Ma implemented the privileged code scanner. Komail Dharsee contributed to the idea of using ARM debug registers for address range monitoring (Chapter 4) and implemented the initial prototype of local-to-global variable promotion (Chapter 5). The rest of the work conducted for the dissertation was completed by the author independently.

This material is based upon work supported by the National Science Foundation Awards CNS-1618213, CNS-1629770, CNS-1652280, CNS-1955498, and CNS-2154322, and by the Office of Naval Research Award N00014-17-1-2996. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In today's world, computer systems that run a processor based on ARM architectures are everywhere. Not only has ARM maintained a dominant market share on embedded systems [24] and mobile platforms [103], but also ARM is gradually gaining popularity on personal computers [14] and high-performance servers and data centers [11, 111, 176, 194]. There have been hundreds of billions of ARM processors in the world that we currently depend on [220].

As ARM-based systems become more ubiquitous in production and in our daily lives, a new realm has been opened in which adversaries leverage software flaws to attack ARM-based systems. In particular, most of the software stack on ARM-based embedded microcontrollers and a large portion of user-space applications running on ARM processors are still written in memory-unsafe programming languages such as C [131] and C++ [242]. As a result, such software suffers from memory safety vulnerabilities (e.g., buffer overflows [275] and buffer overreads [241]) that could be exploited by attackers to launch powerful runtime attacks. These attacks, aiming at stealing intellectual properties or executing arbitrary code, pose serious threats to the safety and security of modern computer systems.

In this dissertation, we explore efficient ways to enforce security policies that defend against runtime attacks on software stacks of ARM systems. Our work tackles

this problem for both tiny, resource-constrained microcontrollers (i.e., systems based on M-profile ARM architectures [21, 23]) and powerful general-purpose application processors (i.e., systems based on A-profile ARM architectures [22]). While there exists a plethora of approaches to mitigating the attacks in question, a common limitation of these approaches is that they either incur high performance overhead or provide weak security guarantees. Therefore, our work focuses on developing novel methods to lower the costs of security policy enforcement that achieve the same or better efficacy against the attacks. Lowering the costs of a security defense enables a greater chance for it to be deployed in practice.

1.1 Low-Cost Security Enforcement for ARM Systems

Efficient Protected Shadow Stacks for Embedded Systems Embedded systems based on microcontrollers (MCUs) are increasingly used for applications that can have serious and immediate consequences if compromised—including automobile control systems, smart locks, drones, and implantable medical devices. Due to resource and execution-time constraints, C [131] is the primary language used for programming these devices. Unfortunately, C is neither type-safe nor memory-safe, and control-flow hijacking remains a prevalent threat [35, 213, 223].

We present *Silhouette*: a compiler-based defense that efficiently guarantees the integrity of return addresses, significantly reducing the attack surface for control-flow hijacking. Silhouette combines an incorruptible shadow stack for return addresses with checks on forward control flow and memory protection to ensure that all functions return to the correct dynamic caller. To protect its shadow stack, Silhouette uses *store hardening*, an efficient intra-address space isolation technique targeting various ARM architectures that leverages special store instructions found on ARM processors.

We implemented Silhouette for the ARMv7-M architecture [21], but our techniques are applicable to other common embedded ARM architectures. Our evaluation shows

that Silhouette incurs a geometric mean of 1.3% and 3.4% performance overhead on two benchmark suites. Furthermore, we prototyped *Silhouette-Invert*, an alternative implementation of Silhouette, which incurs just 0.3% and 1.9% performance overhead, at the cost of a minor hardware change.

Fast Execute-Only Memory for Embedded Systems Remote code disclosure attacks [230, 241] threaten embedded systems as they allow attackers to steal intellectual property or to find reusable code for use in control-flow hijacking attacks. Execute-only memory (XOM) [147] prevents remote code disclosures, but existing XOM solutions either require a memory management unit that is not available on ARM embedded systems [40, 52, 65, 106, 107, 112, 280] or incur significant overhead [141].

We present *PicoXOM*: a fast and novel XOM system for ARMv7-M [21] and ARMv8-M [23] devices which leverages ARM's Data Watchpoint and Tracing unit along with the processor's simplified memory protection hardware. On average, PicoXOM incurs 0.33% performance overhead and 5.89% code size overhead on two benchmark suites and five real-world applications.

Leakage-Resistant Randomization for Microcontrollers Internet-of-Things devices such as autonomous vehicular sensors, medical devices, and industrial cyber-physical systems commonly rely on small, resource-constrained MCUs. MCU software is typically written in C [131] and is prone to memory safety vulnerabilities that are exploitable by remote attackers to launch code reuse attacks [35, 213, 223, 248] and code/control data leakage attacks [95, 101, 230, 241].

We present *Randezvous*, a highly performant diversification-based mitigation to such attacks and their brute force variants on ARM MCUs [21, 23]. Atop code/data layout randomization [59] and an efficient execute-only code approach [227], Randezvous creates *decoy pointers* to camouflage control data in memory; code pointers in the stack are then protected by a *diversified shadow stack*, *local-to-global variable pro-*

motion, and *return address nullification*. Moreover, Randezvous adds a novel *delayed reboot* mechanism to slow down persistent attacks and mitigates control data spraying attacks via *global guards*. We demonstrate Randezvous's security by statistically modeling leakage-equipped brute force attacks under Randezvous, crafting a proof-of-concept exploit that shows Randezvous's efficacy, and studying a real-world CVE [75]. Our evaluation of Randezvous shows low overhead on three benchmark suites and two applications.

Efficient Control-Flow Protection for AArch64 Applications With the increasing popularity of AArch64 (64-bit ARM) processors in general-purpose computing, securing software running on AArch64 systems against control-flow hijacking attacks has become a critical part toward secure computation. Shadow stacks [43, 55] keep shadow copies of function return addresses and, when protected from illegal modifications and coupled with forward-edge control-flow integrity [1, 2], form an effective and proven defense against such attacks [46, 90]. However, AArch64 lacks native support for write-protected shadow stacks, while software alternatives either incur prohibitive performance overhead or provide weak security guarantees.

We present *InversOS*, the first hardware-assisted write-protected shadow stacks for AArch64 user-space applications, utilizing commonly available features of AArch64 [22] to achieve efficient intra-address space isolation (called *Privilege Inversion*) required to protect shadow stacks. Privilege Inversion adopts unconventional design choices that run protected applications in the kernel mode and mark operating system (OS) kernel memory as user-accessible; InversOS therefore uses a novel combination of OS kernel modifications, compiler transformations, and another AArch64 feature to ensure the safety of doing so and to support legacy applications. We show that InversOS is secure by design, effective against various control-flow hijacking attacks, and performant on selected benchmarks and applications (incurring overhead of 7.0% on LMBench [172], 7.1% on SPEC CPU 2017 [232], and 3.0% on Nginx web

server [244]).

1.2 Contributions

Thesis Statement: Advanced runtime attacks that exploit memory safety vulnerabilities found in C/C++ code for intellectual property theft or remote code execution plague ARM systems. We show that effective and low-cost mitigations to such attacks can be achieved on ARM systems by utilizing novel combinations of modern hardware/architectural features and OS/compiler techniques.

This dissertation makes the following contributions:

- We developed *store hardening*, a novel compiler transformation technique for enforcing efficient intra-address space isolation on ARMv7-M [21]. We designed and implemented a system called *Silhouette* to enforce return address integrity for bare-metal MCU applications with low costs, by using store hardening to protect the integrity of shadow stacks.

- We proposed using ARM's hardware debug features to enforce security policies. We designed and implemented *PicoXOM*, a novel and efficient XOM realization which utilizes debug registers on ARMv7-M [21] and ARMv8-M [23] for monitoring against read accesses to code regions.

- We developed a set of novel compiler- and hardware-based techniques to enhance the security of randomization and XOM on MCUs with limited entropy against runtime attacks that leverage memory disclosure. On a diversification-based system called *Randezvous*, we show that our techniques achieved the desired security goals and incurred low performance costs.

- We designed and implemented *Privilege Inversion*, a novel and efficient intra-address space isolation technique for AArch64 user-space applications, combin-

ing OS kernel modifications, compiler transformations, and widely available architectural features of AArch64. Utilizing Privilege Inversion, we prototyped an OS-kernel-compiler co-design named *InversOS* that provides hardware-assisted shadow stack protection on AArch64 and is compatible with legacy application binaries.

1.3 Organization

The rest of the dissertation is organized as follows. Chapter 2 provides background information on ARM architectures. Chapter 3 is derived from our Silhouette paper [282] and describes how we enforce write-protected shadow stacks for bare-metal MCU applications on ARMv7-M [21]. Chapter 4 is derived from our PicoXOM paper [227] and presents how we utilize ARM debug registers to create execute-only memory for ARMv7-M [21] and ARMv8-M [23]. Chapter 5, based on our Randezvous paper [228], extends the system in Chapter 4 with a set of novel techniques to strengthen randomization on ARM MCUs against leakage-equipped code reuse attacks. Chapter 6, based on our InversOS paper [226], elaborates how we extend write-protected shadow stacks to AArch64 user-space applications. Chapter 7 concludes our current research and discusses directions for future work.

Chapter 2

Background

In this chapter, we provide background knowledge and information of ARM architectures, including ARMv7-M [21], ARMv8-M [23], and AArch64 [22]. We introduce architectural features that are relevant to our work described in Chapters 3–6.

2.1 ARMv7-M and ARMv8-M Architectures

A great portion of our work (Chapters 3–5) targets ARMv7-M [21] and/or ARMv8-M [23] architectures, which cover a wide range of embedded systems on the market. Our work leverages unique features of the two architectures to fulfill low-cost security enforcement for microcontrollers. We briefly summarize the instruction sets, privilege and execution modes, address space layout, memory protection features, and on-chip debug support of the two architectures.

2.1.1 Instruction Sets and Execution Modes

ARMv7-M [21] and ARMv8-M [23] are the mainstream M-profile ARM instruction set architectures (ISAs) for embedded microcontrollers (MCUs), in which software is commonly compiled into a single native code executable that contains all application, library, and/or operating system kernel code. Unlike ARM's A and R profiles, M-profile

architectures only support the Thumb instruction set which is a mixture of 16-bit and 32-bit densely-encoded Thumb instructions.

ARMv7-M [21] supports two execution modes: *unprivileged* mode and *privileged* mode. An ARMv7-M processor always executes exception handlers in privileged mode, while application code is allowed to execute in either mode. Code running in unprivileged mode raises the current execution mode to privileged mode either involuntarily by traps and interrupts or voluntarily by executing a supervisor call (`SVC`) instruction. The latter is typically how ARMv7-M realizes system calls. However, embedded applications usually run in privileged mode to reduce the cost of system calls. Such applications also frequently use a *Hardware Abstraction Layer* (HAL) to provide a software interface to device-specific hardware. HAL code is often generated by a manufacturer-provided tool (e.g., HALCOGEN [246]), is linked directly into an application, and runs within its address space.

ARMv8-M inherits all the features of ARMv7-M and adds a security extension called TrustZone-M [23] that isolates software into a secure world and a non-secure world; this effectively doubles the execution modes as software can be executing in either world, privileged or unprivileged.

A unique feature of ARMv7-M and ARMv8-M architectures is special unprivileged store instructions for storing 32-bit values (`STRT`), 16-bit values (`STRHT`), and 8-bit values (`STRBT`) [21, 23]. When a program is running in the processor's privileged mode, these store instructions are treated as though they are executed in unprivileged mode, i.e., the processor always checks the unprivileged access permission when executing an `STRT`, `STRHT`, or `STRBT` instruction regardless of whether the processor is executing in privileged or unprivileged mode.

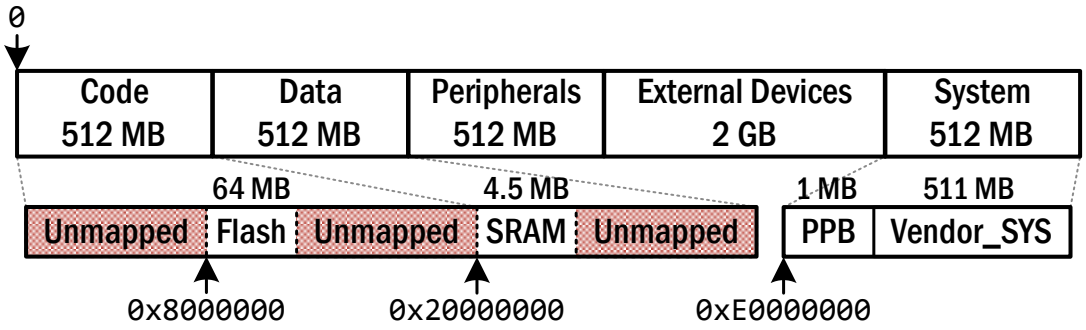

Figure 2.1: General Address Space of ARMv7-M and ARMv8-M Architectures and Usable Memory of NXP MIMXRT685-EVK Board (Remapping of the Same Memory at Different Locations Excluded)

2.1.2 Address Space Layout

Both ARMv7-M [21] and ARMv8-M [23] architectures operate on a single 32-bit physical address space and use memory-mapped I/O to access external devices and peripherals. Figure 2.1 depicts the general address space of the two architectures [21, 23], as well as all usable memory mapped on an NXP MIMXRT685-EVK board [190] that we use. While the exact layout varies between hardware, the address space is generally divided into eight consecutive 512 MB regions; the `Code` region maps flash memory or ROM that contains code and read-only data, the `Data` region typically holds runtime mutable data (e.g., heaps and stacks), and the `System` region contains memory-mapped system registers including a Private Peripheral Bus (PPB) subregion. The PPB subregion contains all critical system registers such as memory protection configuration registers and the Vector Table Offset Register `VTOR`. All other regions are for memory-mapped peripherals and external devices. Note that ARMv7-M and ARMv8-M do not have special privileged instructions to access system registers mapped in the `System` region; instead, they can be modified by regular load and store instructions. Also note that, though code and data regions are up to 512 MB each, there is only 64 MB flash memory for code and 4.5 MB SRAM for data on our board. As neither architecture supports virtual memory [21, 23], code/data layout randomization techniques are limited to moving code/data within the physical address space and have much lower entropy than

general-purpose machines with a 64-bit virtual address space and gigabytes of RAM.

2.1.3 Memory Protection Unit

ARMv7-M and ARMv8-M systems do not have a memory management unit (MMU) that supports virtual memory; instead, they support an optional memory protection unit (MPU) that privileged code can configure to enforce region-based access control on physical memory [21, 23]. A typical ARMv7-M system supports up to eight MPU regions, each of which is configurable with a base address, a power-of-two size from 32 bytes to 4 GB, and separate access permissions (read, write, and execute) for both privileged and unprivileged modes. With TrustZone-M, ARMv8-M has separate MPU configurations for both secure and non-secure worlds [23]. All MPU configuration registers reside in the PPB region.

To prevent code injection [193], typically code is placed in non-writable and executable MPU regions, and data is placed in writable and non-executable MPU regions. There are, however, limitations on how one can configure access permissions for an MPU region. First, the privileged access permission cannot be more restrictive than the unprivileged one; this prohibits an MPU region with, for example, unprivileged read-write and privileged read-only permissions. Second, the MPU assumes that the PPB region is always privileged-accessible, unprivileged-inaccessible, and non-executable regardless of the MPU configuration. Third, the MPU does not have the execute-only permission necessary to support XOM; an MPU region is executable only if it is configured as both readable and executable.

2.1.4 Debug Support

Debug support is another processor feature that ARMv7-M and ARMv8-M systems can optionally support. Of all components in the architecture's debug support, we focus on the DWT unit [21, 23] which provides groups of debug registers called *DWT*

comparators that support instruction/data address matching, PC value tracing, cycle counters, and many other functionalities. Most importantly, a DWT comparator enables monitoring of read accesses over a specified address range; if the processor reads from or writes to an address within a specified range, the DWT comparator will halt the software execution or generate a debug monitor exception. If, instead, the access does not fall into the specified range, execution proceeds as normal, and performance is unaffected. When multiple DWT comparators are configured for data address range matching, an access that hits any of them will trap.

On ARMv7-M, a DWT comparator can be configured to match an address range by programming its base address with a mask that specifies a power-of-two range size [21]. ARMv8-M implements DWT address range matching by using two consecutively numbered DWT comparators [23], where the first one specifies the lower bound of the address range and the second one specifies the upper bound.

2.2 AArch64 Architecture

AArch64 [22] is the 64-bit A-profile ARM ISA for general-purpose systems, targeted by the last piece of our work (Chapter 6). We briefly introduce features of AArch64 pertinent to the design and implementation of our work, including exception levels, address space and page tables, unprivileged load and store instructions, and architectural extensions.

2.2.1 Exception Levels

AArch64 [22] provides four Exception Levels from EL0 to EL3, with increasing execution privileges. Typically user-space software executes in EL0 and OS kernels execute in EL1. EL2 and EL3 are for hypervisors and a secure monitor, respectively. A processor core enters from a lower Exception Level to a same or higher non-EL0 Exception

Level via taking synchronous exceptions (e.g., traps, system calls) or asynchronous exceptions (e.g., interrupts) and returns via executing an `ERET` instruction. Each Exception Level ELx has a dedicated stack pointer register `SP_EL`x. Software running in ELx ($x \geq 1$) can select `SP_EL0` or `SP_EL`x as the current stack pointer, referred to as running in ELxt or ELxh (i.e., thread or handler mode). The two modes are different only in the stack pointer register in use, which also determines the set of exception vectors to use when an exception occurs that targets the same Exception Level. The Linux kernel, as of v4.19.219, executes in EL1h and leaves EL1t (and thus the corresponding set of exception vectors) unused [152]. Within the context of AArch64, hereafter we only focus on EL0 and EL1(t/h) and refer to them as the unprivileged and privileged (thread/handler) modes, respectively.

2.2.2 Address Space and Page Tables

AArch64 [22] uses hierarchical page tables and a hardware memory management unit (MMU) to provide virtual memory, with two Translation Table Base Registers `TTBR0_EL1` and `TTBR1_EL1` holding the root page table addresses. `TTBR0_EL1` is for the lower half of the virtual address space (which typically corresponds to the user space), while `TTBR1_EL1` is for the upper half (which typically corresponds to the kernel space). Not all 64 bits of an virtual address are used in address translation; AArch64 supports a virtual address space up to 52 bits, thus leaving a gap between the two halves, as Figure 2.2 shows.

AArch64 [22] supports page-level access permissions, controlled by the `UXN` (Unprivileged eXecution Never) bit, the `PXN` (Privileged eXecution Never) bit, and two `AP[2:1]` (Access Permission) bits in last-level page table entries (PTEs). As the names imply, `UXN` and `PXN`, when set, disable unprivileged and privileged instruction access of the corresponding page, respectively. `AP[1]` disables unprivileged data access when cleared, and `AP[2]` disables write access when set.

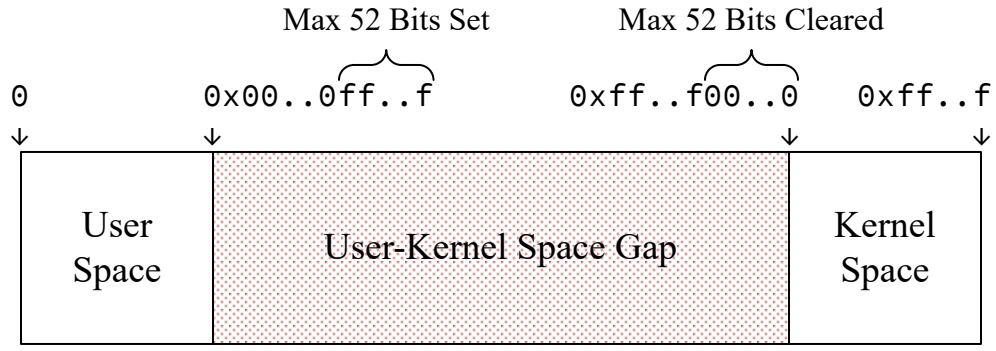

Figure 2.2: AArch64 Virtual Address Space

In addition to the above PTE bits, AArch64 [22] also supports hierarchical access permission control via the `UXNTable` bit, the `PXNTable` bit, and two `APTable[1:0]` bits in top- and mid-level PTEs (PTEs that point to a next-level page table rather than a page). Unlike their last-level PTE counterparts, these bits can apply access restrictions to the whole corresponding address range on top of the permission of subsequent levels. When set, `UXNTable` and `PXNTable` disallow unprivileged and privileged instruction access, respectively. `APTable[0]` disallows unprivileged data access when set, and `APTable[1]` disallows write access when set. The Linux kernel, as of v4.19.219, always keeps these bits cleared and instead only controls access permissions at page level [152].

2.2.3 Unprivileged Load/Store Instructions

Similar to unprivileged store instructions (`STRT`, `STRHT`, and `STRBT`) on ARMv7-M [21] and ARMv8-M [23] (which we introduced in Section 2.1.1), AArch64 also has corresponding unprivileged load and store (LSU) instructions [22]. These instructions, with mnemonics starting with `LDTR` or `STTR`, check unprivileged memory access permissions even when executed in the privileged mode. This makes LSU instructions useful in accessing user-space memory inside the OS kernel (e.g., Linux's `get_user()` and `put_user()` functions [37]).

2.2.4 Architecture Extensions

AArch64 [22] has architecture extensions; the initial ISA is called ARMv8.0-A, and subsequent releases (e.g., ARMv8.1-A) are based on the previous ISA with new hardware features. Specifically, we focus on the following hardware features: Privileged Access Never (PAN), User Access Override (UAO), Hierarchical Permission Disable (HPDS), and Preventing EL0 access to halves of address maps (E0PD).

Privileged Access Never PAN [22] is an ARMv8.1-A feature which prevents privileged code from accessing unprivileged-accessible data memory, similar to x86's Supervisor Mode Access Prevention (SMAP) [4, 123]. When PAN is enabled via setting the `PAN` bit in the processor state `PSTATE`, all loads and stores (except LSU instructions) executed in the privileged mode that try to access memory accessible in the unprivileged mode will generate a permission fault.

User Access Override UAO [22] is an ARMv8.2-A feature which, when enabled via setting the `PSTATE.UAO` bit, allows LSU instructions executed in the privileged mode to act as regular loads/stores.

Hierarchical Permission Disable HPDS [22], introduced in ARMv8.1-A, allows disabling hierarchical access permission bits (`UXNTable`, `PXNTable`, and `APTable[1:0]`) during page table lookups. Software running in the privileged mode can set the `HPD{0,1}` bits in Translation Control Register `TCR_EL1` to disable hierarchical access permission checks in address translation from `TTBR{0,1}_EL1`. However, as AArch64 allows caching `TCR_EL1.HPD{0,1}` in translation lookaside buffers (TLBs), flipping either bit may require a local TLB flush to take effect.

Preventing EL0 access to halves of address maps E0PD [22], introduced in ARMv8.5-A as a hardware mitigation to side-channel attacks that leverage fault timing

(e.g., Meltdown [153]), prevents code running in the unprivileged mode from accessing (lower or upper or both) halves of the virtual address space and generates faults in constant time. Similar to HPDS, there are two bits `TCR_EL1.E0PD{0,1}` that privileged software can use to control whether unprivileged access to which half of the address space is disabled.

Chapter 3

Efficient Protected Shadow Stacks for Embedded Systems

3.1 Introduction

Microcontroller-based embedded systems are typically developed in C [131], meaning they suffer from the same memory errors that have plagued general-purpose systems [193, 213, 245]. Indeed, hundreds of vulnerabilities in embedded software have been reported since 2017.[1] Exploitation of such systems can directly lead to physical consequences in the real world. For example, the control system of a car is crucial to passenger safety; the security of programs running on a smart lock is essential to the safety of people's homes. As these systems grow in importance,[2] their vulnerabilities become increasingly dangerous [127, 178, 216].

Past work on control-flow hijacking attacks highlights the need to protect return addresses, even when the software employs other techniques such as forward-edge control-flow integrity (CFI) [45, 46, 62, 80, 109]. Saving return addresses on a separate shadow stack [43] is a promising approach, but shadow stacks themselves reside

[1] Examples include CVE-2017-8410 [69], CVE-2017-8412 [70], CVE-2018-3898 [74], CVE-2018-16525 [71], CVE-2018-16526 [72], and CVE-2018-19417 [73].

[2] Both Amazon and Microsoft have recently touted operating systems targeting microcontroller-based embedded devices [10, 175].

in the same address space as the exploitable program and must be protected from corruption [43, 62]. Traditional memory isolation that utilizes hardware privilege levels can be adapted to protect the shadow stack [255], but it incurs high overhead as there are frequent crossings between protection domains (e.g., once for every function call). Sometimes *information hiding* is used to approximate intra-address space isolation as it does not require an expensive context switch. In information hiding, security-critical data structures are placed at a random location in memory to make it difficult for adversaries to guess the exact location [140]. Unfortunately, information hiding is poorly suited to embedded systems as most devices have a limited amount of memory that is directly mapped into the address space—e.g., the board used in this work has just 384 KB of SRAM and 16 MB of SDRAM [239].

This chapter presents *Silhouette*: an efficient write-protected shadow stack [77] system that guarantees that a return instruction will always return to its dynamic legal destination. To provide this guarantee, Silhouette combines a shadow stack, an efficient intra-address space isolation mechanism that we call *store hardening*, a control-flow integrity [2] implementation to protect forward-edge control flow, and a corresponding memory protection unit (MPU) configuration to enforce memory access rules. Utilizing the unprivileged store instructions on modern embedded ARM architectures, store hardening[3] creates a logical separation between the code and memory used for the shadow stack and that used by application code. Unlike hardware privilege levels, store hardening does not require expensive switches between protection domains. Also, unlike the probabilistic protections of information hiding, protections based on store hardening are hardware-enforced. Further, the forward-edge control-flow protection prevents unexpected instructions from being executed to corrupt the shadow stack or load return addresses from illegal locations. Finally, the MPU configuration enforces memory access rules required by Silhouette.

[3] *u*XOM [141] independently developed a similar technique for implementing execute-only memory. We compare the implementation differences between store hardening and that of *u*XOM in Section 3.5.2.

We focus on the ARMv7-M architecture [21] given the architecture's popularity and wide deployment; however, our techniques are also applicable to a wide range of ARM architectures, including ARMv7-A [19] and the new ARMv8-M Main Extension [23]. We also explore an alternative, inverted version of Silhouette that promises significant performance improvements at the cost of minor hardware changes; we call this version *Silhouette-Invert*. We summarize our contributions as follows:

- We built a compiler and runtime system, Silhouette, that leverages store hardening and coarse-grained CFI to provide embedded applications with efficient intra-address space isolation and a protected shadow stack.
- We have evaluated Silhouette's performance and code size overhead and found that Silhouette incurs a geometric mean of 1.3% and 3.4% performance overhead, and a geometric mean of 8.9% and 2.3% code size overhead on the CoreMark-Pro and the BEEBS benchmark suites, respectively. We also compare Silhouette to two highly-related defenses: RECFISH [255] and μRAI [9].
- We prototyped and evaluated the Silhouette-Invert variant and saw additional improvements with an average performance overhead measured at 0.3% and 1.9% by geometric mean and code size overhead measured at 2.2% and 0.5%, again, on CoreMark-Pro and BEEBS.

In addition to the above contributions, we observe that store hardening could be extended to protect other security-critical data, making Silhouette more flexible than other approaches. For example, Silhouette could be extended to isolate the sensitive pointer store for Code-Pointer Integrity (CPI) [140]. Similarly, it could be used to protect kernel data structures within an embedded operating system (OS) such as Amazon FreeRTOS [10].

The rest of the chapter is organized as follows: Section 3.2 introduces the threat model. Section 3.3 explains how we can provide two protection domains efficiently on

ARMv7-M systems. Section 3.4 presents how we use our intra-address space isolation to build Silhouette, and Section 3.5 describes our prototype implementation. Section 3.6 explains how Silhouette significantly reduces the control-flow hijacking attack surface. Section 3.7 presents the experimental results, Section 3.8 elaborates the extensibility of Silhouette, and Section 3.9 discusses related work.

3.2 Threat Model and System Assumptions

While embedded code can be conceptually divided into application code, libraries, kernel code, and the hardware abstraction layer, there is often little distinction *at runtime* between these logical units. Due to performance, complexity, and real-time considerations, it is quite common for all of this code to run in the same address space, without isolation, and with the same privilege level [59, 135, 141]. For example, under the default configuration of Amazon FreeRTOS (v1.4.7), *all* code runs as privileged in ARMv7-M [10]. These embedded characteristics heavily inform our threat model and the design decisions for Silhouette.

Our threat model assumes a strong adversary that can exploit a memory error in the *application code* to create a *write-what-where* style of vulnerability. That is, the adversary can attempt to write to any location in memory at any time. The adversary's goal is to manipulate the control flow of a program by exploiting the aforementioned memory error to overwrite memory (e.g., a return address). Non-control data attacks [49, 120] are out of scope of this work. Further, we assume the adversary has full knowledge of the memory contents and layout; we do not rely on information hiding for protection. Our threat model is consistent with past work on defenses against control-flow hijacking.

We assume the target system runs a single bare-metal application statically linked with all the library code and the HAL—the latter provides a device-specific interface to the hardware. We assume the HAL is part of the trusted computing base (TCB) and is

either compiled separately from the application code or annotated, allowing Silhouette to forgo transformations on the HAL that might preclude privileged hardware operations. Similarly, we assume that exception handlers are part of the TCB. Further, we assume the whole binary runs in privileged mode for the reasons mentioned previously.

Finally, we assume the target device includes a memory protection unit (or similar hardware mechanism) for configuring coarse-grained memory permissions, i.e., Silhouette is able to configure read, write, and execute permissions for five regions (summarized in Section 3.5.4) of the address space.

3.3 Intra-Address Space Isolation

Many security enforcement mechanisms rely on intra-address space isolation to protect security-critical data; in other words, the defenses are built on the assumption that application code, under the influence of an attacker,i cannot modify security-critical regions of the address space. For example, defenses with shadow stacks [43] need a safe region to store copies of return addresses, and CPI [140] needs a protected region of the address space to place its safe stack and sensitive pointer store. Complicating matters, defenses often intersperse accesses to the protected region with regular application code; the former should be able to access the protected region while the latter should not. Consequently, existing mechanisms to switch between protection domains—e.g., system calls between unprivileged and privileged mode—are often too inefficient for implementing these security mechanisms for microcontroller-based embedded systems. Rather than incurring the performance penalty of true memory isolation, some defenses hide the security-critical data structures at random locations in the address space [43, 59, 140, 228]. Embedded systems have limited entropy sources for generating random numbers and only kilobytes or megabytes of usable address space; information hiding in general is unlikely to provide effective shadow stack protection on such systems with limited entropy.

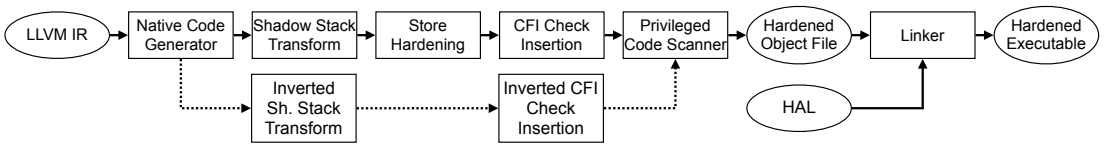

Figure 3.1: Architecture of Silhouette and the Silhouette-Invert Variant

We devise a protection method, *store hardening*, for embedded ARM systems utilizing unprivileged store instructions. We leverage this feature to create two protection domains. One *unprivileged domain* contains regular application code and only uses the unprivileged `STRT`, `STRHT`, and `STRBT` instructions for writing to memory. The other *privileged domain* uses regular (i.e., privileged) store instructions. As code from both domains runs in the same, privileged, processor mode, this method allows us to enforce memory isolation without costly context switching.

To completely isolate the data memory used by the unprivileged and privileged domains, two additional features are needed. First, there needs to be a mechanism to prevent unprivileged code from jumping into the middle of privileged code; doing so could allow unprivileged code to execute a privileged store instruction with arbitrary inputs. We can use forward-edge CFI checks to efficiently prevent such attacks. Second, a trusted code scanner must ensure that the code contains no *system instructions* that could be used to modify important program state without the use of a store instruction. For example, an adversary could use the `MSR` instruction [21] to change the value of the main or process stack pointer registers (`MSP` and `PSP`, respectively), effectively changing the location of the shadow stack and potentially moving it to an unprotected memory region. We discuss a defense that leverages these techniques in the next section.

3.4 Silhouette Design

Silhouette is a compiler and run-time system that leverages our memory isolation scheme to efficiently protect embedded systems from control-flow hijacking attacks. As Figure 3.1 shows, Silhouette transforms application code with four new compiler passes placed after native code generation but before linking the hardened object code with the hardware abstraction layer (HAL). We also explore an alternative, inverted version of these passes that promises significant performance improvements at the cost of minor hardware changes; we call this version *Silhouette-Invert* (see Section 3.4.6). Silhouette's new compiler passes are as follows:

1. **Shadow Stack Transformation**: The shadow stack transformation modifies the native code to save return values on a shadow stack and to use the return value stored in the shadow stack in return instructions.

2. **Store Hardening**: The store hardening pass modifies all store instructions, except those used in the shadow stack instrumentation and Store-Exclusive instructions [21] (see Section 3.4.2 for the reasons), to use variants that check the unprivileged-mode permission bits.

3. **CFI Transformation**: The CFI transformation instruments indirect function calls and other computed branches (aside from returns) to ensure that program execution follows a pre-computed control-flow graph. Consequently, this instrumentation prevents the execution of gadgets that could, for example, be used to manipulate protected memory regions.

4. **Privileged Code Scanner**: The privileged code scanner analyzes the native code prior to emitting the final executable to ensure that application code is free of privileged instructions that an adversary might seek to use to disable Silhouette's protections.

In addition to the above transformations, Silhouette employs mechanisms to prevent memory safety errors from disabling the hardware features that Silhouette uses to provide its security guarantees. In the context of ARMv7-M, it means that the MPU cannot be reconfigured to allow unprivileged accesses to restricted memory regions. Also note that the HAL library is not transformed with Silhouette as it may contain I/O functions that need to write to memory-mapped I/O registers that are only accessible to privileged store instructions. We also forbid inlining HAL functions into application code.

Moreover, Silhouette specially handles variable-length arrays on the stack and `alloca()` calls with argument values that cannot be statically determined by the compiler. For these two types of memory allocation, Silhouette adopts the method from SAFECode [86] and SVA [67] that promotes the allocated data from stack to heap. As Section 3.6.1 explains, such stack allocations (while rare in C code) can cause stack register spills, endangering the integrity of the shadow stack.

3.4.1 Shadow Stack

In unprotected embedded systems, programs store return addresses on the stack, leaving return addresses open to corruption by an adversary. To mitigate such attacks, some compilers transform code to use shadow stacks. A *shadow stack* [43] is a second stack, stored in an isolated region of memory, on which a program saves the return address. Only the code that saves the return address should be able to write to the shadow stack; it should be otherwise inaccessible to other store instructions in the program. If the shadow stack cannot be corrupted by memory safety errors, then return addresses are not corrupted. Furthermore, if the function epilogue uses the correct return address stored on the shadow stack, then the function always returns to the correct dynamic call site.

Silhouette's shadow stack transformation pass modifies each function's prologue to save the return address on a shadow stack and each function's epilogue to use the shadow stack return address on function return. Once the transformation is complete, the program uses a shadow stack, but the shadow stack is not protected. For that, Silhouette employs the store hardening pass and the CFI pass.

3.4.2 Protection via Store Hardening

Silhouette leverages the MPU and the intra-address space isolation mechanism described in Section 3.3 to efficiently protect the shadow stack. This protection is comprised of two parts. First, during compilation, Silhouette's *store hardening* pass transforms all store instructions in application code from privileged instructions to equivalent unprivileged store instructions (`STRT`, `STRHT`, and `STRBT`). As discussed previously, these unprivileged variants always check the MPU's unprivileged-mode permission bits. Second, when loading the program, Silhouette instrumentation configures the MPU so that the shadow stack is readable and writeable in privileged mode but only readable in unprivileged mode. This ensures that store instructions executed in unprivileged mode and unprivileged stores (`STRT`, `STRHT`, and `STRBT`) executed in privileged mode cannot modify values on the shadow stack. Together, these mechanisms ensure shadow stack isolation, *even if the entire program is executed in privileged mode*.

Store hardening transforms all stores within the application code except for two cases. First, store hardening does not transform stores used as part of Silhouette's shadow stack instrumentation as they must execute as privileged instructions so that they can write to the shadow stack. The shadow stack pass marks all stores to the shadow stack with a special flag, making them easily identifiable. Second, store hardening cannot transform atomic stores (Store-Exclusive [21]) because they do not have unprivileged counterparts. Silhouette utilizes Software Fault Isolation (SFI) [254] to prevent those stores from writing to the shadow stack region.

As discussed in Section 3.2, Silhouette does not transform the HAL code; thus, the stores in the HAL code are left unmodified. This is because the HAL contains hardware I/O and configuration code that must be able to read and write the `System`, `Device`, and `Peripheral` memory regions. To prevent attackers from using privileged stores within the HAL code, Silhouette employs CFI as Section 3.4.3 explains.

3.4.3 Forward Branch Control-Flow Integrity

Shadow stacks protect the integrity of function returns, but memory safety attacks can still corrupt data used for forward-edge control flow branches, e.g., function pointers. If left unchecked, these manipulations would allow an attacker to redirect control flow to anywhere in the program, making it trivial for the attacker to corrupt the shadow stack with an arbitrary value or to load a return address from an arbitrary location. Consequently, Silhouette must restrict the possible targets of forward-edges to ensure return address integrity.

There are two types of forward branches: indirect function calls and forward indirect jumps. For the former, Silhouette uses label-based CFI checks [2, 41] to restrict the set of branch targets and ensure that the remaining privileged store instructions cannot be leveraged by an attacker to corrupt the shadow stack. Silhouette-protected systems use privileged store instructions only in the HAL library and in function prologues to write the return address to the shadow stack. The HAL library is compiled separately and has no CFI labels in its code; even coarse-grained CFI ensures that no store instructions within the HAL library can be exploited via an indirect call (direct calls to HAL library functions are permitted as they do not require CFI label checks). For a function call, ARM processors automatically put the return address in the `lr` register. Silhouette's shadow stack transformation pass modifies function prologues to store `lr` to the shadow stack. Label-based CFI guarantees an indirect function call can only jump to the beginning of a function, ensuring that attackers cannot use the function prologue to

write arbitrary values to the shadow stack.

There are three types of constructs in C that may cause a compiler to generate forward indirect jumps: indirect tail function calls, large `switch` statements, and computed `goto` statements ("Label as Values" in GNU's nomenclature [99]). Silhouette's CFI forces indirect tail function calls to jump to the beginning of a function. Restricting large `switch` statements and computed `goto` statements is implementation-dependent. We explain how Silhouette handles them in Section 3.5.3.

3.4.4 `setjmp()` and `longjmp()` Support

A special case for Silhouette to handle is `setjmp()` and `longjmp()` in the C library. `setjmp()` saves the current execution context to a memory location specified by its argument, and `longjmp()` recovers the saved context from the specified memory location as if the execution was just returned from a previous call to `setjmp()`. Calls to `setjmp()` and `longjmp()` can undermine Silhouette's return addresses integrity guarantees because `longjmp()` uses a return address from its `jmp_buf` argument which could be located in corruptible global, heap, or stack memory. Applications might also misuse `setjmp()` and `longjmp()`, such as calling `longjmp()` after the function that called `setjmp()` with the corresponding `jmp_buf` returns, leading to undefined behaviors exploitable by attackers. Silhouette modifies the implementation of `setjmp()` and `longjmp()` to support them while maintaining its return address integrity guarantees.

Specifically, Silhouette reserves part of the protected shadow stack region to store a map of active `jmp_buf` records in use by the program. Figure 3.2 shows the format of a map entry; the address of a `jmp_buf` passed to `setjmp()`/`longjmp()` serves as a key, and all callee-saved registers plus `sp` and `lr` are values. Algorithms 3.1 and 3.2 depict the design of our custom `setjmp()` and `longjmp()`, respectively. When the application calls `setjmp()`, instead of saving the execution context to the application-

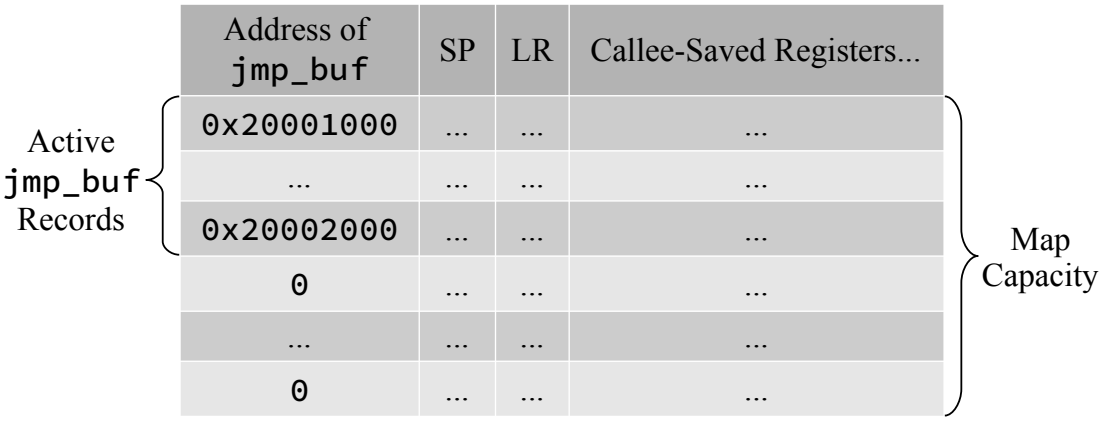

Figure 3.2: Format of Silhouette's `jmp_buf` Records

specified `jmp_buf`, Silhouette's `setjmp()` saves it to the map by inserting a new entry or overriding an existing entry, based on the address of `jmp_buf`. If we are inserting a new entry and the number of active `jmp_buf` records reaches the map's capacity, Silhouette's `setjmp()` reports an error and aborts the program; this is not a practical problem as we expect the program to have only a few `jmp_buf`s. We can also provide an option for the application developer to specify a desired size of the map. Our store hardening pass will recognize this safe version of `setjmp()` and generate regular stores (instead of unprivileged stores) for it to access the map. Saving the execution context in the protected region ensures the integrity of saved stack pointer values and return addresses.

Silhouette's `longjmp()` checks if the address of the supplied `jmp_buf` matches an entry in the map. If no matched entry is found, either the supplied `jmp_buf` is invalid or the supplied `jmp_buf` has expired due to function returns or a call to `longjmp()` on an outer-defined `jmp_buf` (both explained below). In both cases, execution is aborted. If a matched entry is found, Silhouette's `longjmp()` first invalidates all entries in the map that have a smaller `sp` value than that of the matched entry; these `jmp_buf`s become expired when the control flow is unwound to an outer call site of `setjmp()`. The execution context stored in the matched entry is then recovered.

The remaining case is that, when a function that calls `setjmp()` returns, the

Algorithm 3.1: Silhouette `setjmp()`

```
   Input: A jmp_buf buf
 1 foreach entry e in map do
 2 |   if e.buf == &buf then
 3 |   |   e.{sp, lr, ...} ← {sp, lr, ...};
 4 |   |   return 0;
 5 |   end
 6 end
 7 if map.size < map.capacity then
 8 |   Insert a new entry {&buf, sp, lr, ...} into map;
 9 |   map.size ← map.size +1;
10 |   return 0;
11 else
12 |   Error("Map reached its capacity");
13 end
```

`jmp_bufs` used in the function and in its callees become obsolete. Silhouette handles this case by inserting code in the epilogue of such functions to invalidate all the map entries whose `sp` value is smaller than or equal to the current `sp` value. This ensures that future calls to `longjmp()` do not use obsolete `sp` and `lr` values.

3.4.5 Privileged Code Scanner

As Silhouette executes all code within the processor's privileged mode, Silhouette uses a code scanner to ensure the application code is free of privileged instructions that could be used by an attacker to disable Silhouette's protections. If the scanner detects such instructions, it presents a message to the application developer warning that the security guarantees of Silhouette could be violated by the use of such instructions. It is the application developer's decision whether to accept the risk or modify the source code to avoid the use of privileged instructions.

On ARMv7-M [21], there is only one type of privileged instructions that must be removed: `MSR` (Move to Special register from Register). One other, `CPS` (Change Processor State), must be rendered safe through hardware configuration. Specifically, the

Algorithm 3.2: Silhouette `longjmp()`

```
   Input: A jmp_buf buf
   Input: An integer val
 1 buf_entry ← null;
 2 foreach entry e in map do
 3 |   if e.buf == &buf then
 4 |   |   buf_entry ← e;
 5 |   |   break;
 6 |   end
 7 end
 8 if buf_entry == null then
 9 |   Error("Invalid jmp_buf");
10 end
11 foreach entry e in map do
12 |   if e.sp < buf_entry.sp then
13 |   |   Invalidate e;
14 |   |   map.size ← map.size −1;
15 |   end
16 end
17 {sp, lr, ...} ← buf_entry.{sp, lr, ...};
18 if val == 0 then
19 |   return 1;
20 else
21 |   return val;
22 end
```

`MSR` instruction can change special register values in ways that can subvert Silhouette. For example, MPU protections on the shadow stack could be bypassed by changing the stack pointer registers (`MSP` or `PSP` on ARMv7-M) to move the shadow stack to a memory region writeable by unprivileged code. The `CPS` instruction can change the execution priority, and the MPU will elide protection checks if the current execution priority is less than 0 and the `HFNMIENA` bit in the MPU Control Register (`MPU_CTRL`) is set to 0 [21]. However, Silhouette disables this feature by setting the `HFNMIENA` bit to 1, rendering the `CPS` instruction safe. A third type, `MRS` (Move to Register from Special register), can read special registers [21] but cannot be used to compromise the integrity of Silhouette.

Finally, as Silhouette provides control-flow integrity, an attacker cannot use misaligned instruction sequences to execute unintended instructions [2]. Therefore, a linear scan of the assembly is sufficient for ensuring that the application code is free of dangerous privileged instructions.

3.4.6 Improvements with Silhouette-Invert

Swapping a privileged store with a single equivalent unprivileged store introduces no overhead. However, as Section 3.5.2 explains, Silhouette must add additional instructions when converting some privileged stores to unprivileged stores. For example, transforming floating-point stores and stores with a large offset operand adds time and space overhead.

However, we can minimize store hardening overhead by *inverting* the roles of hardware privilege modes. Specifically, if we can invert the permissions of the shadow stack region to disallow writes from privileged stores but allow writes from unprivileged stores, then we can leave the majority of store instructions unmodified. In other words, this design would allow all stores (except shadow stack writes) to remain unmodified, thereby incurring negligible space and time overhead for most programs. We refer to this variant as *Silhouette-Invert*.

Silhouette-Invert is similar in design to ILDI [56] which uses the Privileged Access Never (PAN) feature on ARMv8-A [22, 39] to prevent privileged stores from writing to user-space memory. Unfortunately, the ARMv7-M architecture lacks PAN support and provides no way of configuring memory to be writeable by unprivileged stores but inaccessible to privileged stores [21]. We therefore reason about the potential performance benefits using a prototype that mimics the overhead of a real Silhouette-Invert implementation. Section 3.5.5 discusses two potential hardware extensions to ARMv7-M to enable development of Silhouette-Invert.

3.4.7 Hardware Configuration Protection

As all code on our target system resides within a single address space and, further, as Silhouette executes application code in privileged mode to avoid costly context switching, we must use both the code transformations described above and load-time hardware configurations to ensure that memory safety errors cannot be used to reconfigure privileged hardware state. For example, such state would include the interrupt vector table and memory-mapped MPU configuration registers; on ARMv7-M, most of this privileged hardware state is mapped into the physical address space and can be modified using store instructions [21]. If application code can write to these physical memory locations, an adversary can reconfigure the MPU to make the shadow stack writable or can violate CFI by changing the address of an interrupt handler and then waiting for an interrupt to occur. Therefore, Silhouette makes sure that the MPU prevents these memory-mapped registers from being writable by unprivileged store instructions. As Section 2.1.3 explains, the ARMv7-M MPU is automatically configured this way.

3.5 Implementation

We implemented Silhouette by adding three new `MachineFunction` passes to the LLVM 9.0 compiler [142]: one that transforms the prologue and epilogue code to use a shadow stack, one that inserts CFI checks on all computed branches (except those used for returns), and one that transforms stores into `STRT`, `STRHT`, or `STRBT` instruction sequences. Silhouette runs our new passes after instruction selection and register allocation so that subsequent code generator passes do not modify our instrumentation. Finally, we implemented the privileged code scanner using a Bourne Shell script which disassembles the final executable binary and searches for privileged instructions. Writing a Bourne shell script made it easier to analyze code within inline assembly statements; such statements are translated into strings within special instructions in the LLVM code generator. We measured the size of the Silhouette passes and code scanner

```
mov.w ip, #0xe00000 ; ip is the intra-procedure scratch register
str.w lr, [sp, ip]  ; Save lr to mem[sp + ip]
```

Figure 3.3: Instructions to Update the Silhouette Shadow Stack

using SLOCCount 2.26. Silhouette adds 2,416 source lines of C++ code to the code generator; the code scanner is 95 source lines of Bourne shell code.

3.5.1 Shadow Stack Transformation

Our prototype implements a parallel shadow stack [77] which mirrors the size and layout of the normal stack. By using parallel shadow stacks, the top of the shadow stack is always a constant offset from the regular stack pointer. Figure 3.3 shows the two instructions inserted by Silhouette in a function's prologue for our STM32F469 Discovery board [234, 239]. The constant moved to the `ip` register may vary across different devices based on the available address space. Note that the transformed prologue writes the return address into both the regular stack and the shadow stack.

Silhouette transforms the function epilogue to load the saved return address to either `pc` (program counter) or `lr`, depending on the instructions used in the original epilogue code. The instructions added by the shadow stack transformation are marked with a special flag so that a later pass (namely, the *store hardening* pass) knows that these instructions implement the shadow stack functionality.

Silhouette also handles epilogue code within `IT` blocks [21]. An `IT` (short for If-Then) instruction begins a block of up to 4 instructions called an `IT` *block*. An `IT` block has a condition code and a mask to control the conditional execution of the instructions contained within the block. A compiler might generate an `IT` block for epilogue code if a function contains a conditional branch and one of the branch targets contains a `return` statement. For each such epilogue `IT` block, Silhouette removes the `IT` instruction, applies the epilogue transformation, and inserts new `IT` instruction(s) with the correct condition code and mask to cover the new epilogue code.

3.5.2 Store Hardening

Silhouette transforms all possible variations of regular stores to one of the three unprivileged store instructions: `STRT` (store word), `STRHT` (store halfword), and `STRBT` (store byte) [21]. When possible, Silhouette swaps the normal store with the equivalent unprivileged store. However, some store instructions are not amenable to a direct one-to-one translation. For example, some store instructions use an offset operand larger than the offset operand supported by the unprivileged store instructions; Silhouette will insert additional instructions to compute the target address in a register so that the unprivileged store instructions can be used. ARMv7-M also supports instructions that store multiple values to memory [21]; Silhouette converts such instructions to multiple unprivileged store instructions. For Store-Exclusive instructions [21], Silhouette adds two `BIC` (bit-masking) instructions before the atomic store to force the address operand to point into the global, heap, or regular stack regions.

Silhouette handles store instructions within `IT` [21] blocks in a similar way to how it handles epilogue code within `IT` blocks. If an `IT` block has at least one store instruction, Silhouette removes the `IT` instruction, applies store hardening for each store instruction within the `IT` block, and adds new `IT` instruction(s) to cover newly inserted instructions as well as original non-store instructions within the old `IT` block. This guarantees store hardening generates semantically equivalent instructions for every store in an `IT` block.

Silhouette sometimes adds code that must use a scratch register. For example, when transforming floating-point store instructions, Silhouette must create code that moves the value from a floating-point register to one or two integer registers because unprivileged store instructions cannot access floating-point registers. Our prototype uses LLVM's `LivePhysRegs` class [166] to find free registers to avoid adding register spill code. This optimization significantly reduces store hardening's performance overhead on certain programs; for example, we observed a reduction from 39% to 4.9%

for a loop benchmark. Section 3.7.3 presents detailed data of our experiments.

Comparison with *u*XOM's Store Transformation There are two major differences between Silhouette's implementation of store hardening and the corresponding store transformation of *u*XOM [141]. First, Silhouette performs store hardening near the end of LLVM's backend pass pipeline (after register allocation and right before the `ARMConstantIslandPass` [162]). We made this choice to avoid situations wherein later compiler passes (potentially added by other developers) either generate new privileged stores or transform instructions inserted by Silhouette's shadow stack, store hardening, and CFI passes. As mentioned above, Silhouette avoids register spilling by utilizing LLVM's `LivePhysRegs` class to find free registers. In contrast, *u*XOM transforms store instructions prior to register allocation to avoid searching for scratch registers. As a consequence, subsequent passes, such as prologue/epilogue insertion or passes added by future developers, must ensure that they do not add any new privileged store instructions. Second, our store hardening pass transforms all privileged stores (sans Store-Exclusives) while *u*XOM optimizes its transformation by eliding transformation of certain stores (such as those whose base register is `sp`) when it is safe to do so. The *u*XOM optimization is safe when used with *u*XOM's security policy but may not be safe if store hardening is used to enforce a new security policy that does not protect the integrity of the stack pointer register. Implementing store hardening and optimization in a single pass makes the compiler efficient. However, by adhering to the Separation of Concerns principle in compiler implementation [25], our code is more easily reused: to use store hardening for a new security policy, one simply changes the compiler to run our store hardening pass and then implements any optimization passes that are specific to that security policy.

Code Patterns	How Silhouette Handles Them
Large `switch` statements	Compiled to bounds-checked `TBB` or `TBH` instructions
Indirect tail function calls	Restricted by CFI
Computed `goto` statements	Transformed to `switch` statements

Table 3.1: C Code Patterns That May Be Compiled to Indirect Jumps

3.5.3 Forward Branch Control-Flow Integrity

Indirect Function Calls With link-time optimization enabled, Silhouette inserts a CFI label at the beginning of every address-taken function. Silhouette also inserts a check before each indirect call to ensure that the control flow transfers to a target with a valid label.

Our prototype uses coarse-grained CFI checks, i.e., the prototype uses a single label for all address-taken functions. We picked `0x4600` for the CFI label as it encodes the Thumb instruction `mov r0, r0` and therefore has no side effect when executed. With the addition of static call graph analysis [143], it is possible to extend the Silhouette prototype to use multiple labels with no increase in runtime overhead.

Forward Indirect Jumps Table 3.1 summarizes the three types of constructs of C that may cause a compiler to generate a forward indirect jump and how they are handled by Silhouette. The compiler may insert indirect jumps to implement large `switch` statements. LLVM lowers large `switch` statements into PC-relative jump-table jumps using `TBB` or `TBH` instructions [21]; for each such instruction, LLVM places the jump table immediately after the instruction and inserts a bounds check on the register holding the jump-table index to ensure that it is within the bounds of the jump table. As jump-table entries are immutable and point to basic blocks that are valid targets, such indirect jumps are safe. Tail-call optimization transforms a function call preceding a return into a jump to the target function. Silhouette's CFI checks ensure that tail-call optimized indirect calls jump only to the beginning of a function. The last construct that can generate indirect jumps is the computed `goto` statement. Fortunately, LLVM com-

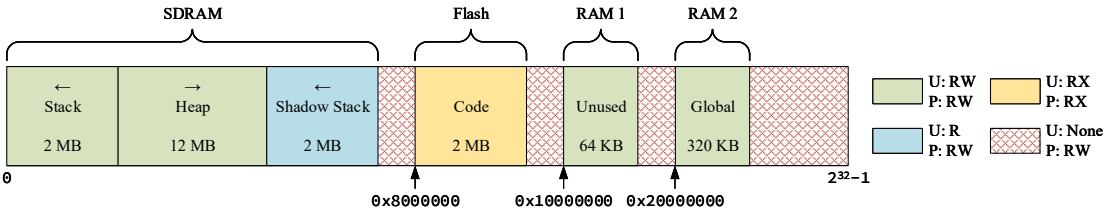

Figure 3.4: Silhouette's Address Space and MPU Configuration on STM32F469 Discovery Board

piles computed `goto` statements into `indirectbr` IR instructions [165]. Silhouette uses LLVM's existing `IndirectBrExpandPass` [163] to turn `indirectbr` instructions into `switch` instructions. We can then rely upon LLVM's existing checks on `switch` instructions, described above, to ensure that indirect jumps generated from `switch` instructions are safe. In summary, Silhouette guarantees that no indirect jumps can jump to the middle of another function.

3.5.4 MPU Configuration

Our prototype also includes code that configures the MPU before an application starts. Figure 3.4 shows the address space and the MPU configuration for each memory region of a Silhouette-protected system on our STM32F469 Discovery board [234, 239]. Silhouette uses five MPU regions to prevent unprivileged stores from corrupting the shadow stack, program code, and hardware configuration. First, Silhouette sets the code region to be readable, executable, and non-writable for both privileged and unprivileged accesses. No other regions are configured executable; this effectively enforces W⊕X [200]. Second, Silhouette configures the shadow stack region to be writable only by privileged code. All other regions of RAM are set to be readable and writable by both privileged and unprivileged instructions. Our prototype restricts the stack size to 2 MB; this should suffice for programs on embedded devices.[4] Note that Silhouette swaps the normal positions of the stack and the heap to detect shadow stack overflow: a

[4]The default stack size of Android applications, including both Java code and native code, is only around 1 MB [12].

stack overflow will decrement the stack pointer to point to the inaccessible region near the top of the address space; a trap will occur when the prologue attempts to save the return address there. An alternative to preventing the overflow is to put an inaccessible guard region between the stack and the heap; however, it costs extra memory and an extra MPU configuration region. Finally, Silhouette enables the default background region which disallows any unprivileged reads and writes to address ranges not covered by the above MPU regions, preventing unprivileged stores from writing the MPU configuration registers and the `Peripheral`, `Device`, and `System` regions.

3.5.5 Silhouette-Invert

Our Silhouette-Invert prototype assumes that the hardware supports the hypothetical inverted-design described in Section 3.4.6, i.e., the MPU can be configured so that the shadow stack is only writable in unprivileged mode. We briefly propose two designs to change the hardware to support the memory access permissions required by Silhouette-Invert.

One option is to use a reserved bit in the Application Program Status Register (`APSR`) [21] to support the PAN state mentioned in Section 3.4.6. In ARMv8-A processors, PAN is controlled by the `PAN` bit in the Current Program Status Register (`CPSR`) [22]. Currently, 24 bits of `APSR` are reserved [21] and could be used for PAN on ARMv7-M.

The second option is to add support to the MPU. In ARMv7-M, the permission configuration of each MPU region is defined using three Access Permission (AP) bits in the MPU Region Attribute and Size Register (`MPU_RASR`) [21]. Currently, binary value `0b100` is reserved, so one could map this reserved value to read and write in unprivileged mode and no access in privileged mode, providing support to the permissions required by Silhouette-Invert without changing the size of AP or the structure of `MPU_RASR`.

In the Silhouette-Invert prototype, the function prologue writes the return address to the shadow stack using an unprivileged store instruction, and CFI uses regular store instructions to save registers to the stack during label checks; all other store instructions remain unchanged. The MPU is also configured so that the shadow stack memory region is writable in unprivileged mode, and other regions of RAM are accessible only in privileged mode. As configuring memory regions to be writable in unprivileged mode only would require a hardware change, the Silhouette-Invert prototype instead configures the shadow stack region to be writable by both unprivileged and privileged stores. We believe both of the potential hardware changes proposed above would add negligible performance overhead. Section 3.7 shows that Silhouette-Invert reduces overhead considerably.

3.5.6 Implementation Limitations

Our Silhouette and Silhouette-Invert prototypes share a few limitations. First, they currently do not transform inline assembly code. The LLVM code generator represents inline assembly code within a C source file as a special "inline asm" instruction with a string containing the assembly code. Consequently, inline assembly code is fed directly into the assembler without being transformed by `MachineFunction` passes. Fortunately, hand-written inline assembly code in applications is rare; our benchmarks contain no inline assembly code. Future implementations could implement store hardening within the assembler which would harden stores in both compiler-generated and hand-written assembly code. Second, our current prototypes do not instrument the startup code or the `newlib` library [184]. These libraries are provided with our development board as pre-compiled native code. In principle, a developer can recompile the startup files and `newlib` from source code to add Silhouette and Silhouette-Invert protections. Third, we have not implemented the "stack-to-heap" promotion (discussed in Section 3.4) for dynamically-sized stack data. Only one of our benchmarks allocates a variable-length local array; we manually rewrote the code to allocate the variable on

the heap. Lastly, we opted not to implement Silhouette's `setjmp()` and `longjmp()` support, described in Section 3.4.4, as none of our benchmarks use `setjmp()` and `longjmp()`.

3.6 Security Analysis

This section explains how Silhouette hinders control-flow hijacking attacks. We first discuss how Silhouette's protected shadow stack, combined with the defenses on forward control-flow, ensure that each return instruction transfers control back to its dynamic caller. We then explain why these security mechanisms provide strong protection against control-flow hijacking attacks.

3.6.1 Integrity of Return Addresses

Silhouette ensures that functions return control flow to their dynamic callers when executing a return instruction by enforcing three invariants at run-time:

Invariant 3.1. *A function stores the caller's return address on the shadow stack, or never spills the return address in register* `lr` *to memory.*

Invariant 3.2. *Return addresses stored on the shadow stack cannot be corrupted.*

Invariant 3.3. *If a function stores the return address on the shadow stack, its epilogue will always retrieve the return address from the correct memory location in the shadow stack, i.e., the location into which its prologue stored the return address.*

As the prologue and epilogue code use the stack pointer to compute the shadow stack pointer, maintaining all the invariants requires maintaining the integrity of the stack pointer. Invariants 3.1 and 3.3 require the function prologue and epilogue to keep the stack pointer within the stack region. Additionally, for Invariant 3.3, Silhouette must ensure that the stack pointer is restored to the *correct* location on the stack to

ensure that the shadow stack pointer is pointing to the correct return address. For Invariant 3.2, besides being inside the stack region, any function call's stack pointer must be guaranteed to stay lower than its frame pointer; otherwise, the valid return addresses on the shadow stack may be corrupted.

To maintain the invariants, Silhouette prevents programs from *loading corrupted values into the stack pointer* by ensuring that application code never spills/reloads the stack pointer to/from memory. In particular, functions that have dynamically-sized stack allocations or that allocate stack memory within a loop may trigger the code generator to spill and reload the stack pointer. As Section 3.4 explains, Silhouette promotes such problematic `alloca` instructions into heap allocations, ensuring that all functions have constant-sized stack frames and therefore have no need to spill the stack pointer.

The next issue is ensuring that the remaining fixed-size stack memory allocations and deallocations cannot be used to violate the invariants. To prevent *stack overflow*, Silhouette positions the regular stack at the bottom of the address space as Figure 3.4 shows. If a stack overflow occurs, the stack pointer will point to a location near the top of the address space; if any function prologue subsequently executes, it will attempt to write the return address into an inaccessible location, causing a trap that will allow the TCB to respond to the overflow.

To ensure that stack deallocation does not cause *stack underflow*, Silhouette ensures that deallocation frees the same amount of stack memory that was allocated in the function prologue. Several Silhouette features ensure this. First, the checks on forward control flow ensure that control is never transferred into the middle of a function (as Section 3.5.3 describes). Second, if Invariants 3.1, 3.2, and 3.3 hold prior to the underflow, then the shadow stack ensures that a function returns to the correct caller, preventing mismatched prologues and epilogues. Finally, since the function prologue dominates all code in the function, and since the function epilogue post-dominates all code in the function, the epilogue will always deallocate the memory allocated in the

prologue.

In summary, Silhouette maintains Invariants 3.1 and 3.3 by ensuring that the stack pointer stays within the stack region during the function prologue and epilogue and that the epilogue will always deallocate stack memory correctly. Silhouette also ensures that the stack pointer will always be lower than the frame pointer, maintaining Invariant 3.2.

3.6.2 Reduced Attack Surface

Recent work has shown the importance of protecting return addresses to increase the precision, and thus strength, of CFI-based defenses [45, 46, 62, 80, 109]. In particular, without a protected shadow stack or other mechanisms to ensure the integrity of return addresses, CFI with static labels cannot ensure that a function returns to the correct caller at runtime; instead, a function is typically allowed to return to a *set* of possible callers. Attacks against CFI exploit this imprecision.

Most attacks against CFI target programs running on general-purpose systems. Some attacks exploit features specific to certain platforms, and it is not clear if they can be ported to attack embedded devices. For example, Conti et al. [62] showed how to corrupt return addresses saved by unprotected context switches on Windows on 32-bit x86 processors. However, many attacks involve generic code patterns that can likely be adapted to attack CFI-protected programs on embedded systems. We now discuss generic control-flow hijacking code patterns discovered by recent work [45, 46, 80, 109]. As we discuss below, Silhouette is robust against these attacks.

Göktaş et al. [109] evaluated the effectiveness of coarse-grained CFI that allows two types of gadgets: *Call-site* (CS) gadgets that start after a function call and end with a return, and *Entry-point* (EP) gadgets that start at the beginning of a function and end with any indirect control transfer. CS gadgets are a result of corrupted return addresses, and EP gadgets stem from corrupted function pointers or indirect jumps if the CFI policy does not distinguish indirect calls and jumps. The authors proposed four

methods of chaining the gadgets: CS to CS (i.e., return-oriented programming), EP to EP (call-oriented programming), EP to CS, and CS to EP. Three of these four methods require a corrupted return address. Their proof-of-concept exploit uses both types of the gadgets. Similarly, Carlini et al. [45] and Davi et al. [80] showed how to chain *call-preceded* gadgets (instruction sequences starting right after a call instruction) to launch code-reuse attacks against CFI. As Silhouette prevents return address corruption, only attacks that chain EP gadgets are possible.

Carlini et al. [46] also demonstrated the weaknesses of CFI and emphasized the importance of a shadow stack. They proposed a *Basic Exploitation Test* (BET)—i.e., a minimal program for demonstrating vulnerabilities—to quickly test the effectiveness of a CFI policy. Their work identifies five dangerous gadgets that allow arbitrary reads, writes, and function calls in the BET under a coarse-grained CFI policy. However, all of these are call-preceded gadgets, and Silhouette's protected shadow stack stymies call-preceded gadgets.

Additionally, Carlini et al. [46] demonstrated a fundamental limitation of CFI defenses when used *without* another mechanism to provide return address integrity. Specifically, they showed that even fully-precise static CFI cannot completely prevent control-flow hijacking attacks, concluding that, regardless of the precision of the computed call graph, protection for return addresses is needed.

In summary, with the protection of Silhouette, control-flow hijacking attacks are restricted to only call-oriented programming. Although there are still potential dangers [96], Silhouette significantly reduces the control-flow hijacking attack surface for embedded programs.

3.7 Experimental Results

Below, we evaluate the performance and code size overhead of our Silhouette and Silhouette-Invert prototypes. We also compare Silhouette to an orthogonal approach,

SSFI, which uses software fault isolation (SFI), instead of store hardening, to isolate the shadow stack from application code. In summary, we find that Silhouette and Silhouette-Invert incur low runtime overhead (1.3% and 0.3% on average for CoreMark-Pro, respectively) and small increases in code size (8.9% and 2.2%, respectively). In addition, we compare Silhouette with the two most closely related defenses, RECFISH [255] and μRAI [9]; they both protect return addresses of programs running on microcontroller-based embedded devices but leverage different mechanisms than Silhouette.

3.7.1 Methodology

We evaluated Silhouette on an STM32F469 Discovery board [234, 239] that can run at speeds up to 180 MHz. The board encapsulates an ARM Cortex-M4 processor [20] and has 384 KB of SRAM (a 320 KB main SRAM region and a 64 KB CCM RAM region), 16 MB of SDRAM, and 2 MB of flash memory. As some of our benchmarks allocate megabytes of memory, we use the SDRAM as the main memory for all programs; global data remains in the main SRAM region.

We used unmodified Clang 9.0 to compile all benchmark programs as the baseline, and we compare this baseline with programs compiled by Silhouette, Silhouette-Invert, and SSFI for performance and code size overhead. We also measured the overhead incurred for each benchmark program when transformed with only the shadow stack (SS) pass, only the store hardening (SH) pass, and only the CFI pass. For all experiments, we used the standard `-O3` optimizations, and we used LLVM's `lld` linker with the `-flto` option to do link-time optimization.

As Silhouette-Invert requires a hardware enhancement for a fully-functional implementation, the numbers we present here are an estimate of Silhouette-Invert's performance. However, as Sections 3.4.6 and 3.5.5 discuss, the hardware changes needed by Silhouette-Invert should have minor impact on execution time and no impact on code

size. Therefore, our evaluation of the Silhouette-Invert prototype should provide an accurate estimate of its performance and memory overhead.

We discuss the implementation of SSFI and compare it with Silhouette and Silhouette-Invert in Section 3.7.5.

3.7.2 Benchmarks

We chose two benchmark suites for our evaluation: CoreMark-Pro [92] and BEEBS [195]. The former is the de facto industry standard benchmark for embedded processors; the latter has been used in the evaluation of other embedded defenses [59, 141, 255].

CoreMark-Pro The CoreMark-Pro [92] benchmark suite is designed for both low-end microcontrollers and high-end multicore processors. It includes five integer workloads (including JPEG compression and SHA-256) and four floating-point workloads such as fast Fourier transform (FFT) and a neural network benchmark. One of the workloads is a more memory-intense version of the original CoreMark benchmark [91]; note, ARM recommends the use of the original CoreMark benchmark to test Cortex-M processors [18]. We used commit `d15927b` of the CoreMark-Pro repository on GitHub.

The execution time of CoreMark-Pro is reported by benchmarks themselves, which is by calling `HAL_GetTick()` [237] to mark the start and the end of benchmark workload execution and printing out the time difference in milliseconds. We added code before the main function starts to initialize the HAL, set up the clock speed, configure the MPU, and establish a serial output. We run each CoreMark-Pro benchmark in different number of iterations so that the baseline execution time is between 5 to 500 seconds.

BEEBS The BEEBS benchmark suite [195] is designed for measuring the energy consumption of embedded devices. However, it is also useful for evaluating perfor-

mance and code size overhead because it includes a wide range of programs, including a benchmark based on the Advanced Encryption Standard (AES), integer and floating-point matrix multiplications, and an advanced sorting algorithm.

The major drawback of BEEBS is that many of its programs either are too small or process too small inputs, resulting in insufficient execution time. For example, `fibcall` is intended to compute the 30th Fibonacci number, but Clang computes the result during compilation and returns a constant directly. To account for this issue, we exclude programs with a baseline execution time of less than one second with 10,240 iterations. We also exclude `mergesort` because it failed the `verify_benchmark()` check when compiled with unmodified Clang. For all the other programs, all of our transformed versions passed this function, if it was implemented. We used commit `049ded9` of the BEEBS repository on GitHub.

To record the execution time of an individual BEEBS benchmark, we wrapped 10,240 iterations of benchmark workload execution with calls to `HAL_GetTick()` [237] and added code to print out the time difference in milliseconds. We also did the same initialization sequence for each BEEBS benchmark as we did for CoreMark-Pro.

3.7.3 Runtime Overhead

Tables 3.2 and 3.3 show the performance overhead that Silhouette and Silhouette-Invert induce on CoreMark-Pro and BEEBS, respectively; overhead is expressed as execution time normalized to the baseline. The **SS** column shows the overhead of just the shadow stack transformation, **SH** shows the overhead induced when only store hardening is performed, and **CFI** shows the overhead of the CFI checks on forward branches. The **Silhouette** and **Invert** columns show the overhead of the complete Silhouette and Silhouette-Invert prototypes, respectively. The **SSFI** column denotes overhead incurred by a version of Silhouette that uses software fault isolation (SFI) in place of store hard-

	Baseline (ms)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
cjpeg-rose7-preset	12,765	1.002	1.004	1.001	1.006	1.003	1.041
core	137,385	1.013	1.002	1.000	1.017	1.015	1.024
linear_alg-mid-100x100-sp	18,278	1.000	1.010	1.000	1.010	1.000	1.015
loops-all-mid-10k-sp	35,241	1.000	1.049	1.000	1.049	1.000	1.016
nnet_test	222,461	1.000	1.013	1.000	1.013	1.000	1.023
parser-125k	9,985	1.004	1.001	1.001	1.005	1.004	1.009
radix2-big-64k	17,270	1.000	1.007	1.000	1.007	1.000	1.019
sha-test	40,725	1.002	1.005	0.999	1.007	1.005	1.046
zip-test	19,955	1.000	1.000	1.000	1.001	1.000	1.006
Min	9,985	1.000	1.000	0.999	1.001	1.000	1.006
Max	222,461	1.013	1.049	1.001	1.049	1.015	1.046
Geomean	—	1.002	1.010	1.000	1.013	1.003	1.022

Table 3.2: Silhouette's Performance Overhead on CoreMark-Pro (Lower is Better)

ening; Section 3.7.5 describes that experiment in more detail.

Silhouette Performance As Tables 3.2 and 3.3 show, Silhouette incurs a geometric mean overhead of only *1.3%* on CoreMark-Pro and *3.4%* on BEEBS. The highest overhead is 4.9% from CoreMark-Pro's loops benchmark and 24.8% from BEEBS's `bubblesort` benchmark. The `bubblesort` benchmark exhibits high overhead because it spends most of its execution in a small loop with frequent stores; to promote these stores, Silhouette adds instructions to the loop that compute the target address. Another BEEBS program with high overhead is `levenshtein`. The reason is that one of its functions has a variable-length array on the stack and that function is called in a loop; Silhouette promotes the stack allocation to the heap with `malloc()` and `free()`. Without this promotion, Silhouette incurs 2.2% overhead on `levenshtein`. Nearly all (8 of 9) CoreMark-Pro benchmarks slow down by less than 2%, and 5 programs have less than 1% overhead. For BEEBS, 24 of the 29 programs slow down by less than 5%; 16 programs have overhead less than 1%. Tables 3.2 and 3.3 also show that the primary source of the overhead is typically store hardening, though for some programs (e.g., `core` and `sglib-rbtree`) the shadow stack induces more overhead due to extensive function calls. CFI overhead is usually negligible

	Baseline (ms)	**SS** (×)	**SH** (×)	**CFI** (×)	**Silhouette** (×)	**Invert** (×)	**SSFI** (×)
bubblesort	2,755	1.001	1.247	1.000	1.248	1.000	1.510
ctl-string	1,393	1.015	1.011	0.999	1.027	1.016	1.035
cubic	28,657	1.002	1.002	1.000	1.002	1.001	1.005
dijkstra	40,580	1.002	1.001	1.000	1.003	1.002	1.117
edn	2,677	1.000	1.004	1.000	1.004	1.000	1.058
fasta	16,274	1.000	1.000	1.000	1.000	1.000	1.001
fir	16,418	1.000	1.000	1.000	1.000	1.000	1.021
frac	8,846	1.000	1.003	1.000	1.000	1.000	1.009
huffbench	46,129	1.000	1.005	1.000	1.005	1.000	1.017
levenshtein	7,835	1.005	1.019	1.000	1.207	1.186	1.248
matmult-int	5,901	1.000	1.011	1.000	1.012	1.000	1.048
nbody	124,578	1.000	0.997	1.000	0.997	1.000	1.003
ndes	1,938	1.010	1.008	1.000	1.016	1.011	1.039
nettle-aes	7,030	1.000	1.003	1.000	1.003	1.000	1.111
picojpeg	43,010	1.037	1.057	0.997	1.098	1.037	1.380
qrduino	43,564	1.000	1.036	1.000	1.036	1.000	1.072
rijndael	78,849	1.001	1.008	1.000	1.008	1.005	1.146
sglib-dllist	1,327	1.001	1.006	1.000	1.007	1.001	1.268
sglib-listinsertsort	1,359	1.001	1.000	1.000	1.001	1.001	1.054
sglib-listsort	1,058	1.001	0.999	1.000	1.000	1.001	1.233
sglib-queue	2,135	1.000	1.029	1.000	1.030	1.000	1.122
sglib-rbtree	7,802	1.092	1.017	1.000	1.110	1.093	1.157
slre	4,163	1.031	1.013	1.000	1.045	1.035	1.112
sqrt	55,894	1.000	1.002	1.000	1.006	1.002	1.002
st	20,036	1.002	1.002	1.002	1.002	1.002	1.008
stb_perlin	3,168	1.073	1.052	1.000	1.049	1.073	1.045
trio-sscanf	1,335	1.037	1.006	1.022	1.073	1.063	1.115
whetstone	97,960	1.000	1.001	1.000	1.001	1.000	1.002
wikisort	160,307	1.011	1.013	1.016	1.039	1.029	1.180
Min	1,058	1.000	0.997	0.997	0.997	1.000	1.001
Max	160,307	1.092	1.247	1.022	1.248	1.186	1.510
Geomean	—	1.011	1.018	1.001	1.034	1.019	1.102

Table 3.3: Silhouette's Performance Overhead on BEEBS (Lower is Better)

	Baseline (bytes)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
cjpeg-rose7-preset	98,316	1.017	1.082	1.094	1.193	1.113	1.315
core	51,516	1.006	1.038	1.002	1.046	1.009	1.106
linear_alg-mid-100x100-sp	58,772	1.007	1.062	1.002	1.071	1.010	1.135
loops-all-mid-10k-sp	99,156	1.009	1.103	1.002	1.113	1.011	1.225
nnet_test	69,580	1.007	1.111	1.002	1.118	1.009	1.226
parser-125k	57,712	1.009	1.038	1.002	1.049	1.012	1.095
radix2-big-64k	58,220	1.007	1.060	1.002	1.069	1.010	1.121
sha-test	52,428	1.005	1.028	1.002	1.036	1.008	1.071
zip-test	82,924	1.009	1.096	1.007	1.112	1.017	1.280
Min	51,516	1.005	1.028	1.002	1.036	1.008	1.071
Max	99,156	1.017	1.111	1.094	1.193	1.113	1.315
Geomean	—	1.008	1.068	1.012	1.089	1.022	1.172

Table 3.4: Silhouette's Code Size Overhead on CoreMark-Pro (Lower is Better)

because our benchmarks seldom use indirect function calls.

Silhouette-Invert Performance Silhouette-Invert greatly decreases the overhead because it only needs to convert the single privileged store instruction in the prologue of a function to a unprivileged one and leave all other stores unchanged. It incurs only *0.3%* geometric mean overhead on CoreMark-Pro. Seven of the 9 programs show overhead less than 0.5%. For BEEBS, the geometric mean overhead is 1.9%. When excluding the special case of `levenshtein`, the average overhead is 1.3%. Twenty of the 29 programs slow down by less than 1%. Only three programs, `sglib-rbtree`, `stb_perlin`, and `trio-sscanf`, again, except `levenshtein`, slow down by over 5%, and all of them have very frequent function calls.

3.7.4 Code Size Overhead

Small code size is critical for embedded systems with limited memory. We therefore measured the code size overhead incurred by Silhouette by measuring the code size of the CoreMark-Pro and BEEBS benchmarks. Tables 3.4 and 3.5 present the baseline code size and the overhead Silhouette and Silhouette-Invert incurred on CoreMark-Pro

	Baseline (bytes)	**SS** (×)	**SH** (×)	**CFI** (×)	**Silhouette** (×)	**Invert** (×)	**SSFI** (×)
bubblesort	30,716	1.004	1.019	1.001	1.023	1.005	1.026
ctl-string	30,976	1.005	1.009	1.001	1.014	1.006	1.021
cubic	43,468	1.003	1.008	1.000	1.011	1.003	1.011
dijkstra	30,900	1.004	1.017	1.001	1.022	1.005	1.041
edn	32,536	1.003	1.010	1.000	1.014	1.004	1.035
fasta	30,600	1.003	1.007	1.001	1.011	1.004	1.014
fir	30,164	1.003	1.005	1.001	1.009	1.004	1.011
frac	30,776	1.004	1.009	1.001	1.014	1.005	1.017
huffbench	34,156	1.003	1.033	1.000	1.036	1.004	1.067
levenshtein	31,408	1.004	1.016	1.001	1.013	1.000	1.025
matmult-int	31,344	1.004	1.009	1.001	1.014	1.005	1.024
nbody	33,636	1.003	1.015	1.000	1.019	1.003	1.019
ndes	33,660	1.004	1.025	1.000	1.030	1.005	1.042
nettle-aes	32,312	1.004	1.010	1.000	1.015	1.005	1.028
picojpeg	45,084	1.006	1.061	1.000	1.068	1.008	1.201
qrduino	46,108	1.003	1.058	1.000	1.062	1.004	1.125
rijndael	39,268	1.003	1.025	1.000	1.029	1.004	1.091
sglib-dllist	30,740	1.003	1.008	1.001	1.012	1.004	1.026
sglib-listinsertsort	30,144	1.003	1.005	1.001	1.010	1.004	1.013
sglib-listsort	30,524	1.003	1.007	1.001	1.011	1.004	1.022
sglib-queue	31,272	1.003	1.023	1.001	1.027	1.004	1.070
sglib-rbtree	30,748	1.006	1.012	1.001	1.019	1.007	1.032
slre	33,420	1.005	1.015	1.000	1.020	1.006	1.042
sqrt	30,428	1.003	1.006	1.001	1.010	1.004	1.009
st	35,300	1.003	1.014	1.001	1.017	1.004	1.023
stb_perlin	31,124	1.004	1.010	1.001	1.014	1.005	1.010
trio-sscanf	35,468	1.005	1.030	1.013	1.048	1.019	1.079
whetstone	40,984	1.003	1.010	1.000	1.013	1.004	1.019
wikisort	40,108	1.005	1.053	1.001	1.060	1.007	1.156
Min	30,144	1.003	1.005	1.000	1.009	1.000	1.009
Max	46,108	1.006	1.061	1.013	1.068	1.019	1.201
Geomean	—	1.004	1.018	1.001	1.023	1.005	1.044

Table 3.5: Silhouette's Code Size Overhead on BEEBS (Lower is Better)

and BEEBS, respectively. In summary, Silhouette incurs a geometric mean of 8.9% and 2.3% code size overhead on CoreMark-Pro and BEEBS, respectively.

For Silhouette, most of the code size overhead comes from store hardening. As Section 3.5.2 explains, Silhouette transforms some regular store instructions into a sequence of multiple instructions. Floating-point stores and stores that write multiple registers to contiguous memory locations bloat the code size most. In BEEBS, `picojpeg` incurs the highest code size overhead because an unrolled loop contains many such store instructions, and the function that contains the loop is inlined multiple times. For Silhouette-Invert, because it leaves nearly all stores unchanged, its code size overhead is only 2.2% on CoreMark-Pro and 0.5% on BEEBS.

3.7.5 Store Hardening versus SFI

An alternative to using store hardening to protect the shadow stack is to use software fault isolation (SFI) [254]. To compare the performance and code size overhead of store hardening against SFI, we built a system that provides the same protections as Silhouette but that uses SFI in place of store hardening. We dub this system *Silhouette-SFI* (*SSFI*). Our SFI pass instruments all store instructions within a program other than those introduced by the shadow stack pass and those in the HAL. Specifically, our SSFI prototype adds the same `BIC` [21] (bit-masking) instructions as what Silhouette does for Store-Exclusives (discussed in Section 3.5.2) before each store to restrict them from writing to the shadow stack.

SSFI incurs much higher performance and code size overhead compared to Silhouette. On CoreMark-Pro, SSFI incurs a geometric mean of 2.2% performance overhead, nearly doubling Silhouette's average overhead of 1.3%; on BEEBS, SSFI slows down programs by 10.2%, three times of Silhouette's 3.4%. Only on one program, the loops benchmark in CoreMark-Pro, SSFI performs better than Silhouette. For code size, SSFI incurs an average of 17.2% overhead on CoreMark-Pro and 4.4% on BEEBS; the

highest overhead is 31.5% and 20.1%, respectively, while on Silhouette it is 19.3% and 6.8%. The specific implementation of SFI will vary on different devices due to different address space mappings, so it is possible to get different overhead on different boards for the same program. In contrast, Silhouette's performance overhead on the same program should be more predictable across different boards because the instructions added and replaced by Silhouette remain the same.

3.7.6 Comparison with RECFISH and μRAI

RECFISH [255] and μRAI [9] are both recently published defenses that provide security guarantees similar to Silhouette but via significantly different techniques. Like Silhouette, they provide return address integrity coupled with coarse-grained CFI protections for ARM embedded architectures. As each defense has distinct strengths and weaknesses, the choice of defense depends on the specific application to be protected. To compare Silhouette with RECFISH and μRAI more directly and fairly, we also evaluated Silhouette with BEEBS and the original CoreMark benchmark using only SRAM and present their performance numbers.

RECFISH [255], which is designed for real-time systems, runs code in unprivileged mode and uses supervisor calls to privileged code to update the shadow stack. Due to frequent context switching between privilege levels, RECFISH incurs higher overhead than Silhouette or μRAI. For the 24 BEEBS benchmarks that RECFISH and Silhouette have in common,[5] RECFISH incurs a geometric mean of 21% performance overhead, and approximately 30% on CoreMark whereas Silhouette incurs just 3.6% and 6.7%, respectively. Unlike the other two defenses, RECFISH patches binaries; no application source code or changes to the compiler are needed.

μRAI [9] protects return addresses, in part, by encoding them into a single reserved register and guaranteeing this register is never corrupted. This approach is more com-

[5] We obtained RECFISH's detailed performance data on BEEBS via direct correspondence with the RECFISH authors.

plicated but requires no protected shadow stack. Consequently, μRAI is very efficient for most function calls, incurring three to five cycles for each call-return. However, there are cases, such as calling a function from an uninstrumented library, where μRAI needs to switch hardware privilege levels to save/load the reserved register to/from a safe region, which is expensive.

μRAI [9] reports an average of 0.1% performance overhead on CoreMark and five IoT applications. The μRAI authors observed that one IoT program, `FatFs_RAM`, saw a 8.5% speedup because their transformation triggered the compiler to perform a special optimization that was not performed on the baseline code. When accounting for this optimization, μRAI incurred an overhead of 6.9% on `FatFs_RAM` and 2.6% on average for all benchmarks. We measured the performance of CoreMark using Silhouette; the result is 6.7% overhead compared to μRAI's reported 8.1% [9].

Finally, we observe that Silhouette's store hardening is a general technique for intra-address space isolation. Thus, Silhouette can be extended to protect other security-critical data in memory, which Section 3.8 discusses. In contrast, μRAI only protects a small amount of data by storing it within a reserved register; its approach cannot be as easily extended to protect arbitrary amounts of data. μRAI does rely on SFI-based instrumentation in exception handlers for memory isolation, but our results in Section 3.7.5 show that store hardening is more efficient than SFI and could therefore be used to replace SFI in μRAI.

3.8 Extensibility

Although Silhouette focuses on providing control-flow and return address integrity for bare-metal applications, it can also be extended to other use cases. For example, with minimal modification, Silhouette can be used to protect other security-critical data in memory, such as CPI's sensitive pointer store [140] or the kernel data structures within an embedded OS like Amazon FreeRTOS [10].

With moderate modification, Silhouette can also emulate the behavior of running application code in unprivileged mode on an embedded OS. First, the kernel of the embedded OS would need to configure the MPU to disable unprivileged write access to all kernel data. Second, the embedded OS kernel's scheduler would need to disable unprivileged write access to memory of background applications. Third, in addition to store hardening, Silhouette would need to transform loads in the application code into unprivileged loads in order to protect the confidentiality of OS kernel data structures. It would also need to ensure that the embedded OS kernel code contains no CFI labels used by user-space applications. Fourth, the privileged code scanner must be adjusted to forbid all privileged instructions (as opposed to only those that can be used to bypass Silhouette's protections) in application code, forbid direct function calls to internal functions of the kernel, and allow privileged instructions in the embedded OS kernel. Fifth, since the stack pointer of background applications needs to be spilled to memory during context switch, the embedded OS kernel must protect the stack pointer of applications from corruption in order to enforce Silhouette's security guarantee of return address integrity. One simple solution would be storing application stack pointers to a kernel data structure not writable by application code. Finally, system calls require no changes. In ARMv7-M [21], application code calls a system call using the `SVC` instruction, which generates a supervisor call exception. A pointer to the exception handler table (which stores the address of exception handler functions) is stored in a privileged register within the `System` region; Silhouette can protect both the `System` region and the exception handler table to ensure that the `SVC` instruction always transfers control to a valid system call entry point. Also, regardless of current privilege mode, exception handlers in ARMv7-M, including the supervisor call handler, will execute in privileged mode and switch the stack pointer to use the kernel stack [21]. Therefore, system calls require no change for Silhouette to work as intended.

3.9 Related Work

Control-Flow Hijacking Defenses for Embedded Systems Besides REC-FISH [255] and μRAI [9], which Section 3.7.6 discusses, there are several other control-flow hijacking defenses for embedded devices. CaRE [191] uses supervisor calls and TrustZone-M technology, available on the ARMv8-M [23] architecture but not on ARMv7-M, to provide coarse-grained CFI and a protected shadow stack. CaRE's performance overhead on CoreMark is 513%. SCFP [264] provides fine-grained CFI by extending the RISC-V architecture. Unlike Silhouette, SCFP is a pure CFI defense and does not provide a shadow stack. Therefore, it cannot mitigate attacks such as control-flow bending [46] while Silhouette can, as Section 3.6.2 shows.

Use of Unprivileged Loads/Stores Others [56, 141] have explored the use of ARM's unprivileged loads and stores to provide security guarantees; however, their work differs from Silhouette's store hardening in both implementation and application. *u*XOM [141] transforms regular load instructions to unprivileged ones to implement execute-only memory on embedded systems. Aside from differences in the provided security guarantees—i.e., execute-only memory versus control-flow and return address integrity—these systems differ in how they handle dangerous instructions that could be manipulated to bypass protections. In particular, *u*XOM inserts verification routines before unconverted load/store instructions to ensure that they will not access security-critical memory regions while Silhouette leverages CFI and other forward branch protections to prevent unexpected instructions from being executed. ILDI [56] combines unprivileged loads and stores on the ARMv8-A architecture along with the PAN state and `hyp` mode to isolate data within the Linux kernel—the latter two features are not available on the ARMv7-M systems targeted by Silhouette.

Intra-Address Space Isolation Silhouette protects the shadow stack by leveraging store hardening. Previous work has explored other methods of intra-address space iso-

lation which could be used to protect the shadow stack. Our evaluation in Section 3.7.5 compares Silhouette to software fault isolation (SFI) [254], so we focus on other approaches here.

ARM Mbed μVisor [17], MINION [135], and ACES [60] enforce memory compartmentalization on embedded systems using the MPU. They all dynamically reconfigure the MPU at runtime but target different scenarios; Mbed μVisor and MINION isolate processes from each other at context switches, and ACES dissects a bare-metal application at function boundaries for intra-application isolation. As discussed previously, isolation that requires protection domain switching is poorly-suited to security instrumentation that requires frequent crossing of the isolation boundaries—such as Silhouette's shadow stack accesses.

ARMlock [283] uses ARM domains to place pages into different protection domains; a privileged register controls access to pages belonging to different domains. ARM domains are only available for CPUs with MMUs [19, 21] and therefore cannot be used in ARMv7-M systems. Additionally, access to ARM domains can only be modified in the privileged mode; software running in user-space must context switch to the privileged mode to make changes.

Information Hiding Given the traditionally high cost of intra-address space isolation, many defenses hide security-critical data by placing it at a randomly chosen address. This class of techniques is generally referred to as information hiding. For example, EPOXY [59] includes a backward-edge control-flow hijacking defense that draws inspiration from CPI [140]—relying on information hiding to protect security-critical data stored in memory. Consequently, an adversary with a write-what-where vulnerability (as assumed in our threat model) can bypass EPOXY protections.

In general, information hiding is unlikely to be a strong defense on embedded systems as such systems tend to use only a fraction of the address space (and the memory

is directly mapped) which limits the entropy attainable.[6] For example, our evaluation board only has 2 MB of memory for code; if each instruction occupies two bytes, randomizing the code segment provides at most 20 bits of entropy. In contrast, Silhouette's defenses are effective even if the adversary has full knowledge of the memory layout and contents.

Memory Safety Memory safety provides strong protection but incurs high overhead. Solutions using shadow memory [7, 8, 87, 156, 222] may consume too much memory for embedded systems. Other solutions [84, 86, 129, 183, 215] incur too much performance overhead. nesCheck [177] is a memory safety compiler for TinyOS [118] applications which induces 6.3% performance overhead on average. However, nesCheck cannot support binary code libraries as it adds additional arguments to functions. Furthermore, nesCheck's performance relies heavily on static analysis. We believe that, due to their simplicity, the benchmarks used in the nesCheck evaluation are more amenable to static analysis than applications for slightly more powerful embedded systems (such as ours). In contrast, Silhouette's performance does not depend on static analysis's precision.

[6] Chapter 5 presents a study on the use of randomization on microcontrollers and how to improve the limited entropy on such embedded systems.

Chapter 4

Fast Execute-Only Memory for Embedded Systems

4.1 Introduction

Remote code disclosure attacks threaten computer systems. Remote attackers exploiting buffer overread vulnerabilities [241] can not only steal intellectual property (e.g., proprietary application code, for reverse engineering), but also leak code to locate gadgets for advanced code reuse attacks [230], thwarting code layout diversification defenses like Address Space Layout Randomization (ASLR) [201]. Embedded Internet-of-Things (IoT) devices exacerbate the situation; many of these microcontroller-based systems have the same Internet connectivity as desktops and servers but rarely employ protections against attacks [127, 216]. Given the ubiquity of these embedded devices in industrial production and in our lives, making them immune to code disclosure attacks is crucial.

Recent research [27, 38, 40, 52, 65, 106, 107, 112, 141, 205, 280] implements *execute-only memory (XOM)* to defend against code disclosure attacks. Despite being unable to prevent code pointer leakage from data regions such as heaps and stacks, XOM enforces memory protection on the code region so that instruction fetching is allowed but reading or writing instructions as data is disallowed. This simple and ef-

fective defense, however, is not natively available on low-end microcontrollers. For example, the ARMv7-M and ARMv8-M architectures used in mainstream devices support memory protection but not execute-only (XO) permissions [21, 23]. uXOM [141] implements XOM on ARM embedded systems but incurs significant performance and code size overhead (7.3% and 15.7%, respectively) as it transforms most load instructions into special unprivileged load instructions. Given embedded systems' real-time constraints and limited memory resources, a practically ideal XOM solution should have *close-to-zero performance penalty* and *minimal memory overhead*.

This chapter presents *PicoXOM*, a fast and novel XOM system for ARMv7-M and ARMv8-M devices using a memory protection unit (MPU) and the Data Watchpoint and Tracing (DWT) unit [21, 23]. PicoXOM uses the MPU to enforce *write* protection on code and uses the unique *address range matching capability* of the DWT unit to control *read* access to the code region. On a matched access, the DWT unit generates a debug monitor exception indicating an illegal code read, while unmatched accesses execute normally without slowdown. As PicoXOM disallows all read accesses to the code segment, it includes a minimal compiler change that removes all data embedded in the code segment.

We built a prototype of PicoXOM and evaluated it on an ARMv7-M board with two benchmark suites and five real-world embedded applications. Our results show that PicoXOM adds *negligible performance overhead* of 0.33% and only has a *small code size increase* of 5.89% while providing strong protection against code disclosure attacks. We open-sourced our prototype at `https://github.com/URSec/PicoXOM`.

To summarize, our contributions are:

- PicoXOM: a novel method of utilizing the ARMv7-M and ARMv8-M debugging facilities to implement XOM. To the best of our knowledge, this is the first use of ARM debug features for security purposes.

- A prototype implementation of PicoXOM on ARMv7-M.
- An evaluation of PicoXOM's performance and code size impact on the BEEBS benchmark suite, the CoreMark-Pro benchmark suite, and five real-world embedded applications, showing that PicoXOM only incurs 0.33% run-time overhead and 5.89% code size overhead.

The rest of the chapter is organized as follows. Section 4.2 describes our threat model and assumptions. Sections 4.3 and 4.4 present the design and implementation of PicoXOM, respectively. Section 4.5 reports on our evaluation of PicoXOM, and Section 4.6 discusses related work.

4.2 Threat Model and System Assumptions

We assume a buggy but unmalicious application running on an embedded device with memory safety vulnerabilities that allow a remote attacker to read or write arbitrary memory locations. The attacker wants to either steal proprietary application code for purposes like reverse engineering or learn the application code layout in order to launch code reuse attacks such as Return-into-libc [248] and Return-Oriented Programming (ROP) [213] attacks. Physical and offline attacks are out of scope as we believe such attacks can be stopped by orthogonal defenses [138, 216]. Our threat model also assumes the application code and data is diversified, using techniques such as those in EPOXY [59]. Therefore, remotely tricking the buggy application into reading its code content becomes a reasonable choice for the attacker.

We assume that the target embedded device supports MPU and DWT with enough configurable MPU regions and DWT comparators. We assume that the device is running a single bare-metal application statically linked with libraries, boot sequences, and exception handlers. The application is assumed to run in privileged mode, as Sec-

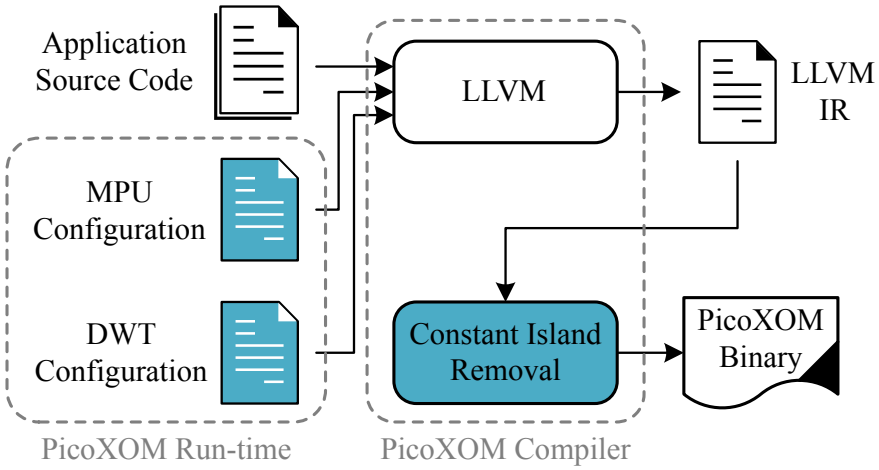

Figure 4.1: PicoXOM Workflow (PicoXOM Components Shown in Blue)

tion 2.1.1 dictates. For ARMv8-M devices with TrustZone-M, the application is assumed to reside in the non-secure world, while software in the secure world is trusted.

4.3 Design

Figure 4.1 shows PicoXOM's overall design. PicoXOM consists of three components that together implement a strong and efficient XOM on ARM embedded devices. First, PicoXOM uses a specially-configured DWT configuration to detect read accesses to program code. Second, it utilizes a special MPU configuration that prevents write access to the code region and prevents writeable memory from being executable. Third, it employs a small change to the LLVM compiler [142] to eliminate constant data embedded within the code region.

To use PicoXOM, embedded application developers merely compile their code with the PicoXOM compiler and install it on their embedded ARM device. On boot, the PicoXOM run-time configures MPU regions and DWT comparators using PicoXOM's MPU and DWT configurations and then passes control to the compiled embedded software.

4.3.1 W⊕X with MPU

PicoXOM requires that memory either be writeable or executable but not both i.e., the W⊕X policy [200]; otherwise, an attacker could simply inject code or overwrite code to achieve arbitrary code execution. To enforce W⊕X, PicoXOM configures the MPU regions at device boot time so that the code region is readable and executable, read-only data is read-only, and RAM regions are readable and writable. Note that the MPU *cannot* configure memory to be executable but unreadable; the MPU can configure a memory region as executable only if it is also configured as readable [21, 23].

PicoXOM runs application code in privileged mode and configures a background MPU region to allow read and write access to the remainder of the address space such as peripherals. This, however, leaves critical memory-mapped system registers in the PPB (such as MPU configuration registers and `VTOR`) open to modifications, which can be leveraged by an attacker to turn off MPU protections or, even worse, implant a custom exception handler. Section 4.3.2 discusses how PicoXOM prevents such cases.

4.3.2 R⊕X with DWT

PicoXOM leverages ARM's DWT comparators to watch over the whole code region for read accesses. As Section 2.1.4 states, each (pair) of DWT comparators available on an ARM microcontroller can be configured to generate a debug monitor exception when a memory access of a specified type to an address within a specified range occurs. PicoXOM therefore uses one (pair) of the available DWT comparators as follows:

1. At device boot time, PicoXOM configures a DWT comparator register (say `DWT_COMP<n>`) to hold the lower bound of the code region.

2. PicoXOM then sets the address-matching range by either writing the upper bound of the code region to the next DWT comparator register `DWT_COMP<n+1>` (for

ARMv8-M) or writing the correct mask to the corresponding DWT mask register DWT_MASK<n> (for ARMv7-M).

3. PicoXOM enables the DWT comparator (pair) by configuring the DWT function register DWT_FUNC<n> for data address reads. For ARMv8-M devices, DWT_FUNC<n+1> is also configured in order to form address range matching.

4. Finally, PicoXOM enables the debug monitor exception by setting the MON_EN bit (bit 16) of the Debug Exception and Monitor Control Register DEMCR.

With a DWT comparator (pair) set up for monitoring read accesses to the code region, R⊕X is effectively enforced. However, as Section 4.3.1 stated, the DWT registers and DEMCR are also memory-mapped system registers which could be modified by vulnerable application code. An attacker could leverage a buffer overflow vulnerability to reconfigure the debug registers to neutralize PicoXOM.

We can address the issue in two ways. One approach is to break the assumption that PicoXOM runs everything in privileged mode. As code running in unprivileged mode has no access to the PPB region regardless of the MPU configuration, the system registers that PicoXOM must protect (e.g., MPU configuration registers, DWT registers, DEMCR, and VTOR) are all in the PPB region and therefore inherently safe from unprivileged tampering. However, this approach requires PicoXOM to implement system calls that support privileged operations which application code could previously perform, incurring expensive context switching between privilege modes. The other approach is to use extra (pairs of) DWT comparators to prevent writes to critical system registers. For example, on ARMv7-M, we can configure one DWT comparator to write-protect the System Control Block SCB (0xE000ED00 – 0xE000ED8F) and DEMCR (0xE000EDFC) by setting the lower bound and the size to 0xE000ED00 and 256 bytes, respectively. Since MPU configuration registers are in the SCB, they are protected as well. DWT registers on ARMv7-M reside in a separate range (0xE0001000 – 0xE0001FFF), so we can use another DWT comparator to write-protect that range.

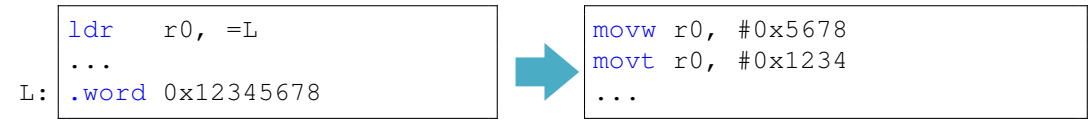

Figure 4.2: Constant Island Removal of a Load Constant

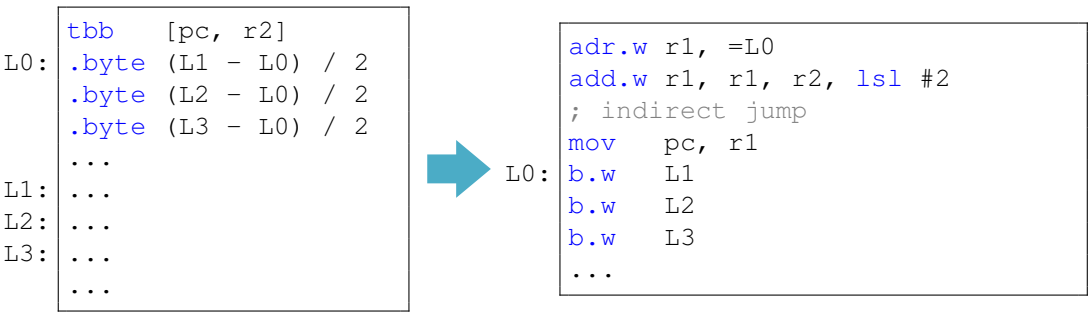

Figure 4.3: Constant Island Removal of a Jump-Table Jump

4.3.3 Constant Island Removal

By default, ARM compilers generate code that has constant data embedded in the code region (so-called "constant islands"). Since PicoXOM prevents the code from reading these constant islands, these programs will fail to execute when used with PicoXOM. PicoXOM therefore transforms these programs so that all data within the program is stored outside of the code region.

We have identified two cases of constant islands generated by LLVM's ARM code generator: *load constants* and *jump-table jumps*. Figures 4.2 and 4.3 show examples of the two cases, respectively, as well as their corresponding execute-only versions to which PicoXOM transforms them. Specifically, in the left part of Figure 4.2, a load constant instruction loads a constant from a PC-relative memory location `L` into register `r0`. Such instructions are usually generated to quickly load an irregular constant in light of the limited immediate encoding scheme of the Thumb instruction set [21, 23]. PicoXOM transforms such load constants into `MOVW` and `MOVT` instructions that encode the 32-bit constant in two 16-bit immediates, as the right part of Figure 4.2 shows. Jump-table jump instructions (`TBB` and `TBH`) [21, 23] are used to implement large

switch statements; the second register operand (`r2` in Figure 4.3) serves as an index into a jump table pointed to by the first register operand (`pc` in Figure 4.3), and a byte/half-word offset is loaded from the jump table to add to the program counter (`pc`) to calculate the target of the jump. Optimizing compilers like GCC and LLVM usually select `pc` as the first register operand in order to reduce register pressure, forcing the jump table to be located next to the jump-table jump itself. PicoXOM transforms such jump-table jumps into instruction sequences like that shown in the right part of Figure 4.3; it encodes the original jump table's contents into a sequence of branch instructions and expands the jump-table jump into a few explicit instructions that calculate which branch instruction to jump to and perform an indirect jump.

4.4 Implementation

We built our PicoXOM prototype for the ARMv7-M architecture. Our prototype provides MPU and DWT configurations as a run-time component written in C and executed at the end of the device boot sequence. We implemented constant island removal as a simple intermediate representation (IR) pass in the LLVM 10.0 compiler [142]. The constant island removal pass simply uses the existing `-mexecute-only` option in LLVM's Clang front-end and passes it along to the link-time optimization (LTO) code generator. Our prototype runs the constant island removal pass when linking the IR of the application, libraries (e.g., newlib and compiler-rt), and MPU and DWT configurations; this ensures that all code has no constant islands. Our prototype adds 88 source lines of C++ code to LLVM and has 177 source lines of C code in the PicoXOM runtime. Chapter 5 presents a PicoXOM implementation for ARMv8-M.

Different ARM microcontrollers support different numbers of MPU regions and DWT comparators, and the maximum ranges of their DWT comparators may vary. Our prototype runs on an STM32F469 Discovery board which supports up to 8 MPU regions [235] and 4 DWT comparators [239]. Each DWT comparator can only watch

over a maximum address range of 32 KB (a maximal mask value of 15), limiting our prototype to the following two options:

1. Use all 4 DWT comparators to support a maximum code size of 128 KB; the application must run in unprivileged mode in order for the critical system registers to be write-protected.

2. Configure one DWT comparator to write-protect the DWT registers (`0xE0001000 – 0xE0001FFF`) and another to write-protect the SCB (`0xE000ED00 – 0xE000ED8F`) and `DEMCR` (`0xE000EDFC`). This protects a maximum code size of 64 KB using the remaining 2 DWT comparators.

To accommodate a wider range of applications on our board with less performance loss, our prototype automatically chooses one option over the other based on the application code size. It rejects an application if the code size exceeds our board's 128 KB limit.

While our PicoXOM prototype only supports single bare-metal embedded applications, PicoXOM can also support multiple applications running on an embedded real-time operating system (RTOS) such as Amazon FreeRTOS [10]. On embedded systems, the application and RTOS kernel code is linked into a single shared code segment. PicoXOM can protect this code segment with little adaptation.

4.5 Evaluation

We evaluated PicoXOM on our STM32F469 Discovery board [239] which has an ARM Cortex-M4 processor implementing the ARMv7-M architecture that can run as fast as 180 MHz. The board comes with 2 MB of flash memory, 384 KB of SRAM, and 16 MB of SDRAM, and has an LCD screen and a microSD card slot. We configured the board to run at its fastest speed to understand the maximum impact that PicoXOM can incur on performance.

To evaluate PicoXOM's performance and code size overhead, we used the BEEBS [195] and CoreMark-Pro [92] benchmark suites and five embedded applications (FatFs-RAM, FatFs-uSD, LCD-Animation, LCD-uSD, and PinLock). **BEEBS** targets energy consumption measurement for embedded platforms and is widely used in evaluating embedded systems including uXOM [141], the state-of-the-art XOM implementation on ARM microcontrollers. It consists of a wide range of programs characterizing different workloads seen on embedded systems, including AES encryption, data compression, and matrix multiplication. Of all 80 benchmarks in BEEBS, we picked 42 benchmarks that have an execution time longer than 500 milliseconds when executed for 10,240 iterations. **CoreMark-Pro** is a processor benchmark suite that works on both high-performance processors and low-end microcontrollers, featuring five integer benchmarks (e.g., JPEG image compression, XML parser, and SHA-256) and four floating-point benchmarks (e.g., fast Fourier transform and neural network) that stress the CPU and memory. **FatFs-RAM** and **FatFs-uSD** operate a FAT file system on SDRAM and an SD card, respectively. **LCD-Animation** displays a single animated picture loaded from an SD card. **LCD-uSD** displays multiple static pictures from an SD card with fading transitions. **PinLock** simulates a smart lock reading user input from a serial port and deciding whether to unlock (send an I/O signal) based on whether the SHA-256 hashed input matches a precomputed hash. The above five applications represent real-world use cases of embedded devices and were also used to evaluate previous work [9, 59, 60].

We used the LLVM compiler infrastructure [142] to compile benchmarks and applications into the default non-XO format, with MPU and DWT disabled; this is our baseline. We then used PicoXOM's configuration, i.e. enabling MPU, DWT, and constant island removal. Note that with PicoXOM, none of the benchmarks and applications exceeds the code size limitation (128 KB) on our board. Only `cjpeg-rose7-preset` in CoreMark-Pro has a code size larger than 64 KB and thereby has to run in unprivileged mode; nevertheless, it does not require source code modifications as it does not

	Baseline (ms)	PicoXOM (×)		Baseline (ms)	PicoXOM (×)
aha-compress	821	1.0000	nettle-arcfour	814	1.0000
aha-mont64	856	0.9988	picojpeg	43,864	1.0027
bubblesort	4,392	1.0000	qrduino	40,877	1.0030
crc32	956	1.0000	rijndael	70,024	1.0018
ctl-string	630	1.0000	sglib-arraybinsearch	808	1.0000
ctl-vector	786	0.9987	sglib-arrayheapsort	1,039	1.0000
cubic	35,140	1.0005	sglib-arrayquicksort	735	1.0000
dijkstra	36,582	1.0000	sglib-dllist	1,800	1.0000
dtoa	631	1.0127	sglib-hashtable	1,302	1.0000
edn	3,167	1.0003	sglib-listinsertsort	2,030	1.0000
fasta	16,900	0.9999	sglib-listsort	1,265	1.0008
fir	16,048	1.0000	sglib-queue	1,177	1.0000
frac	5,858	1.0323	sglib-rbtree	4,808	1.0025
huffbench	20,682	0.9995	slre	2,761	0.9873
levenshtein	2,685	1.0000	sqrt	38,506	1.0748
matmult-float	1,150	0.9991	st	20,906	1.0252
matmult-int	4,532	1.0000	stb_perlin	5,132	1.0306
mergesort	24,353	1.0062	trio-snprintf	697	1.0100
nbody	128,126	1.0090	trio-sscanf	1,064	0.9915
ndes	2,039	0.9995	whetstone	112,754	1.0092
nettle-aes	5,687	0.9998	wikisort	113,195	1.0008
Min (×)					0.9873
Max (×)					1.0748
Geomean (×)					1.0046

Table 4.1: PicoXOM's Performance Overhead on BEEBS (Lower is Better)

perform privileged operations.

4.5.1 Performance

We measured PicoXOM's performance on our benchmarks and applications. We configured each BEEBS benchmark to print the time, in milliseconds, for executing its workload 10,240 times. We ran each BEEBS benchmark 10 times and report the average execution time. Each CoreMark-Pro benchmark is pre-programmed to print out the execution time in a similar way; the difference is that we configure each benchmark to run a minimal number of iterations so that the program takes at least 10 seconds to run for each experimental trial. Again, we ran each benchmark 10 times and report the average execution time. For the real-world applications, we ran FatFs-RAM 10 times and report the average execution time. The other applications exhibit higher variance

	Baseline (ms)	PicoXOM (×)		Baseline (ms)	PicoXOM (×)
cjpeg-rose7-preset	10,200	1.0001	parser-125k	12,363	1.0012
core	83,160	0.9918	radix2-big-64k	21,955	0.9961
linear_alg-mid-100x100-sp	22,962	1.0000	sha-test	25,463	0.9995
loops-all-mid-10k-sp	33,830	0.9995	zip-test	23,227	1.0000
nnet_test	282,398	1.0017			
Min (×)					0.9918
Max (×)					1.0017
Geomean (×)					0.9989

Table 4.2: PicoXOM's Performance Overhead on CoreMark-Pro (Lower is Better)

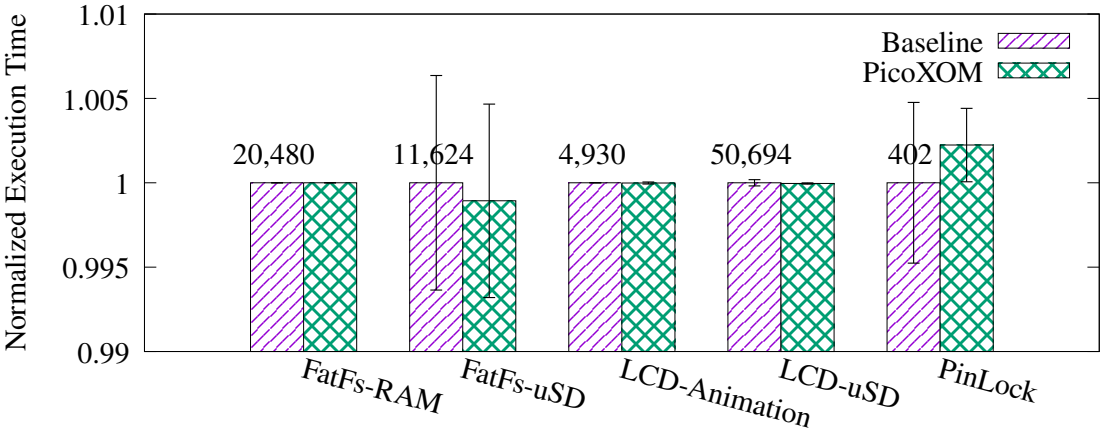

Figure 4.4: PicoXOM's Performance Overhead on Applications (Lower is Better)

in their execution times as they access peripherals like an SD card, an LCD screen, and a serial port, so we ran them 20 times and report the average with a standard deviation. All other programs exhibit a standard deviation of zero.

Tables 4.1 and 4.2 and Figure 4.4 present PicoXOM's performance on BEEBS, CoreMark-Pro, and the five real-world applications, respectively; Figure 4.4 shows baseline execution time in milliseconds on top of the Baseline bars. Overall, PicoXOM incurs negligible performance overhead of 0.33%: 0.46% on BEEBS with a maximum of 7.48%, −0.11% on CoreMark-Pro with a maximum of 0.17%, and 0.02% on the applications with a maximum of 0.22%. Thirteen programs exhibit a minor speedup with PicoXOM. We re-ran our experiments with the MPU and DWT disabled so that the only change to performance is due to constant island removal and the alignment of the code segment (the DWT on ARMv7-M requires the monitored address range

	Baseline (bytes)	PicoXOM (×)		Baseline (bytes)	PicoXOM (×)
aha-compress	30,164	1.0646	nettle-arcfour	29,988	1.0649
aha-mont64	31,236	1.0624	picojpeg	36,620	1.0599
bubblesort	29,868	1.0650	qrduino	37,228	1.0529
crc32	29,804	1.0654	rijndael	37,460	1.0516
ctl-string	30,668	1.0631	sglib-arraybinsearch	29,828	1.0654
ctl-vector	30,892	1.0624	sglib-arrayheapsort	29,956	1.0651
cubic	42,428	1.0329	sglib-arrayquicksort	30,036	1.0649
dijkstra	30,220	1.0644	sglib-dllist	30,364	1.0641
dtoa	36,204	1.0552	sglib-hashtable	30,164	1.0644
edn	30,940	1.0633	sglib-listinsertsort	30,052	1.0649
fasta	29,956	1.0650	sglib-listsort	30,100	1.0648
fir	29,884	1.0651	sglib-queue	29,988	1.0650
frac	30,468	1.0626	sglib-rbtree	30,564	1.0639
huffbench	30,988	1.0628	slre	32,284	1.0603
levenshtein	30,140	1.0647	sqrt	30,372	1.0641
matmult-float	30,108	1.0644	st	31,124	1.0602
matmult-int	30,060	1.0650	stb_perlin	31,140	1.0627
mergesort	30,852	1.0604	trio-snprintf	33,724	1.0675
nbody	30,684	1.0633	trio-sscanf	34,156	1.0668
ndes	31,028	1.0630	whetstone	40,164	1.0371
nettle-aes	31,756	1.0614	wikisort	34,332	1.0541
Min (×)					1.0329
Max (×)					1.0675
Geomean (×)					1.0614

Table 4.3: PicoXOM's Code Size Overhead on BEEBS (Lower is Better)

to be aligned by its power-of-two size). In this configuration, we observed the same speedups, so either constant island removal and/or code alignment is causing the slight performance improvement.

4.5.2 Code Size

We measured the code size of benchmarks and applications by using the `size` utility on generated binaries and collecting the `.text` segment size.

Table 4.3 and Figures 4.5 and 4.6 show the baseline code size and the overhead incurred by PicoXOM on BEEBS, CoreMark-Pro, and the five real-world applications, respectively. On average, PicoXOM increases the code size by 6.14% on BEEBS, 4.39% on CoreMark-Pro, and 6.52% on the real-world applications, with a 5.89% overall overhead. We studied PicoXOM's code size overhead and discovered that constant

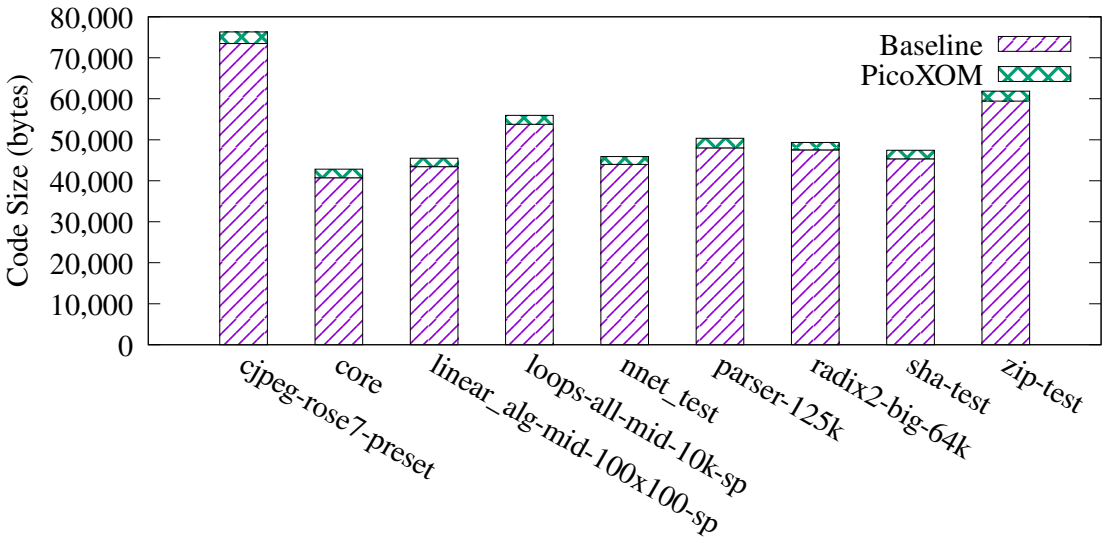

Figure 4.5: PicoXOM's Code Size Overhead on CoreMark-Pro (Lower is Better)

island removal caused the majority of the code size overhead, especially for programs with relatively large code bases like CoreMark-Pro. In fact, the additional code that sets up the MPU and DWT only contributes a minor part of the overhead (1.22% and 0.53% on average, respectively).

4.6 Related Work

Two other XOM implementations exist for ARM microcontrollers. uXOM [141] provides XOM for ARM Cortex-M systems by transforming loads into special unprivileged load instructions and configuring the MPU to make the code region unreadable by unprivileged loads. uXOM similarly transforms stores to protect the memory-mapped MPU configuration registers. Since some loads and stores do not have unprivileged counterparts, transforming them requires the compiler to insert additional instructions, causing the majority of uXOM's overhead. PicoXOM is more efficient in both performance (0.33% compared to uXOM's 7.3%) and code size (5.89% compared to uXOM's 15.7%) as no such transformation is needed. A trade-off for PicoXOM is the code size limit on some ARMv7-M devices; we envision no such limit on ARMv8-

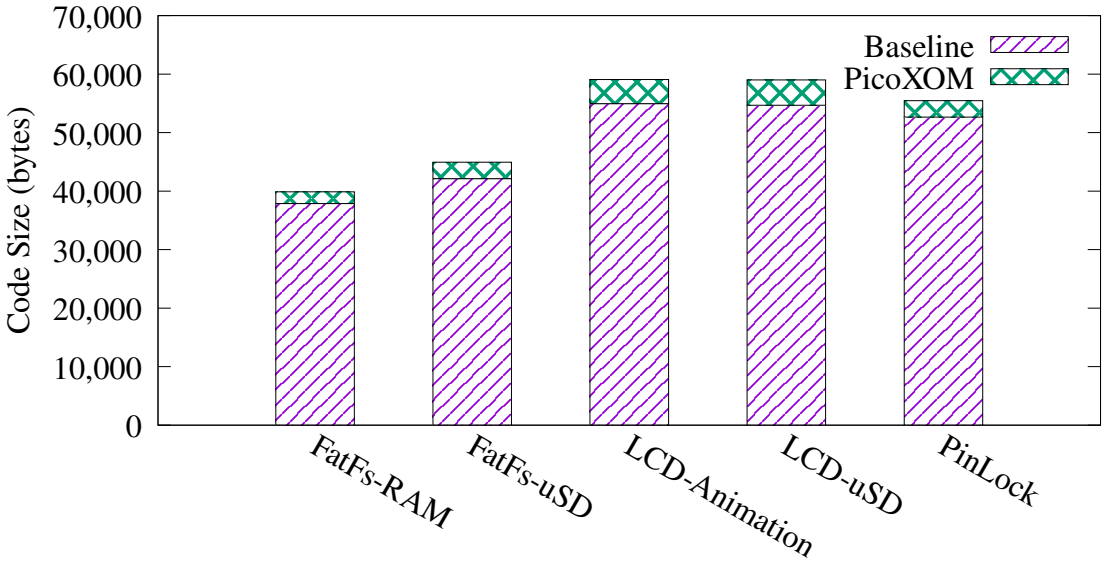

Figure 4.6: PicoXOM's Code Size Overhead on Applications (Lower is Better)

M. PCROP [233] is a programmable feature of the flash memory which prevents the flash memory from being read out and modified by application code but still allows code in the flash memory to execute. However, PCROP is only available on some STMicroelectronics devices and cannot be used for other types of memory. In contrast, PicoXOM relies on the MPU and DWT features [21, 23] which can be found on most conforming devices and can protect code stored in any type of memory.

Hardware-assisted XOM has been explored on other architectures. The AArch64 [22] and RISC-V [212] page tables natively support XO permissions. NORAX [52] enables XOM for commercial-off-the-shelf binaries on AArch64 that have constant islands using static binary instrumentation and runtime monitoring. Various approaches [40, 65, 106, 107, 112, 280] leverage features of the MMU on Intel x86 processors [123] to implement XOM. None of these approaches are applicable on ARM embedded devices lacking an MMU. Lie et al. [147] proposed an architecture with memory encryption to mimic XOM, but it only provides probabilistic guarantees and cannot be directly applied to current embedded systems. Compared to solutions for systems lacking native hardware XOM support, PicoXOM is faster as it has nearly no

overhead.

Software can emulate XOM. XnR [27] maintains a sliding window of currently executing code pages and keeps only these pages accessible. It still allows read accesses to a subset of code pages and may incur higher overhead for a smaller sliding window size due to frequent page permission changes. LR^2 [38] and kRˆX [205] instrument all load instructions to prevent them from reading the code segment. While these software XOM approaches can generally be ported to embedded devices, they can be bypassed by attacker-manipulated control flow and are less efficient than hardware-assisted XOM [141].

There are also methods of hardening embedded systems. Early versions of SAFECode [85] enforced spatial and temporal memory safety on embedded applications, and nesCheck [177] uses static analysis to build spatial memory safety for simple nesC [102] applications running on TinyOS [118]. PicoXOM enforces weaker protection than memory safety but supports arbitrary C programs (unlike SAFECode and nesCheck) and does not rely on heavy static analysis like nesCheck. RECFISH [255], μRAI [9], and Silhouette [282] (which we present in Chapter 3) mitigate control-flow hijacking attacks on embedded systems. They protect forward-edge control flow using coarse-grained CFI [2] and backward-edge control flow by using either a protected shadow stack [43] or a return address encoding mechanism. EPOXY [59] randomizes the order of functions and the location of a modified safe stack from CPI [140] to resist control-flow hijacking attacks on bare-metal microcontrollers. These systems do not enforce XOM and are still vulnerable to forward-edge corruptions; they can incorporate PicoXOM's techniques to mitigate forward-edge attacks with negligible additional overhead.

Chapter 5

Leakage-Resistant Randomization for Microcontrollers

5.1 Introduction

The increasing prevalence of Internet-of-Things devices challenges the security of embedded microcontroller (MCU) systems. MCU software is commonly written in C [131] and, consequently, suffers from memory safety vulnerabilities. These vulnerabilities can be exploited by attackers to launch control-flow hijacking attacks [33, 45, 46, 80, 109, 213, 248] which corrupt control data (e.g., return addresses and function pointers) so that control flow is diverted to existing code in the program. Worse yet, MCU software is typically executed in the processor's privileged mode alongside or without an operating system kernel. Successful exploitation of MCU software means that the attacker controls the *entire* system.

To mitigate control-flow hijacking attacks on MCUs, previous work [9, 130, 191, 255, 264, 282] has explored control-flow integrity (CFI) [2] which protects or checks the integrity of control data used in indirect control-flow transfers. However, CFI is vulnerable to advanced attacks [46, 96] even with a fully precise static control-flow graph (CFG) and a protected shadow stack. Also, CFI implementations on MCUs incur high runtime overhead (8.1%–513% [9, 191, 282]), leaving them less likely to be

deployed in practice.[7]

Randomization [201] with execute-only memory (XOM) [141, 227] is another potential solution: by randomizing the location of code and preventing buffer over-reads [241] from reading the code segment, attackers no longer know where reusable code is located and therefore cannot divert control flow to the chosen code. However, such approaches have two key limitations on MCUs. First, the address space on MCUs is limited: there is no virtual memory [21, 23], so the entropy of randomization is limited by the physical memory size (typically on the order of kilobytes to megabytes). Brute force attacks [224] which simply guess the location of reusable code can therefore succeed in short amounts of time. Second, and worse yet, previous solutions [3, 59, 116, 199] do not mitigate control data leakage in which a buffer over-read [241] leaks control data to learn the location of reusable code [65, 82, 204, 214]. In fact, a proof-of-concept exploit we built shows that even a large-sized MCU protected with randomization and XOM can be breached in less than an hour with the help of control data disclosure. On the other hand, control data leakage defenses [38, 65, 167, 205] do exist in general-purpose systems, which hide control data using indirection or encryption. However, they still leave control data identifiable and usable by attackers.

This chapter presents *Randezvous*, a system that mitigates control-flow hijacking attacks against ARMv7/8-M MCUs which utilize brute force attacks and attacks which leak control data. Built on top of previous work that randomizes code and global data layouts [59] and enforces XOM [227], Randezvous protects control data with a set of novel techniques we developed. At the center of Randezvous is a new concept called a *decoy pointer*, which is a code pointer that points to a random unused trap instruction; by filling unused data memory with decoy pointers, real code pointers are camouflaged and thus protected. Leveraging decoy pointers in global data segments, Randezvous moves return addresses into a *diversified shadow stack* and *promotes local variables*

[7] Silhouette [282] incurs 11.2%–12.1% runtime overhead on our NXP MIMXRT685-EVK board, much higher than the reported 1.3%–3.4% on STM32F469 Discovery board. Section 5.7 discusses the difference in more detail.

containing function pointers into globals to protect them. To further reduce the danger of return address leakage, Randezvous introduces *return address nullification* which overwrites stale return addresses with decoy pointers so that leakage is limited to return addresses of currently executing functions. In addition to control data protection, Randezvous improves the limited entropy on MCUs against attacks with *delayed reboot* and *global guards*. The former adds an artificial reboot delay to slow down brute force attacks when an attack attempt is detected. The latter is an adaptation of memory guards [64] to mitigate spraying attacks that massively corrupt a memory region in order to guarantee control-flow hijacking. Collectively, Randezvous builds a holistic probabilistic (but measurably strong) defense against control-flow hijacking attacks on MCUs, which is, to our best knowledge, the first to mitigate both control data leakage and lack of entropy on MCUs. Compared to randomization with XOM alone, Randezvous significantly enhances the protection of in-memory control data and improves the entropy against attacks.

We built a prototype of Randezvous for ARMv8-M MCUs at `https://github.com/URSec/Randezvous`, upon the LLVM/Clang compiler [142]. We evaluated Randezvous's security by statistically modeling brute force attacks with control data leakage, building a real exploit that demonstrates the necessity of Randezvous's security features, and analyzing how Randezvous can stop exploitations of a real-world CVE. We also evaluated Randezvous's overhead on three benchmark suites and two real-world applications. On average, Randezvous incurred 5.9% performance overhead, 15.4% code size overhead, and 22.0% data size overhead.

To summarize, our contributions are as follows:

- We developed a set of novel control data protection techniques for MCUs that strengthen randomization and XOM, centered on decoy pointers and including a diversified shadow stack, return address nullification, and local-to-global variable promotion.

- We devised delayed reboot, a mechanism that mitigates brute force attacks exploiting the limited entropy on MCUs.
- We designed and implemented Randezvous, a strong holistic software diversity approach to securing MCUs against control-flow hijacking attacks.
- We built the first mathematical model of brute force attacks with control data leakage on MCUs to evaluate the strength of Randezvous's defenses and demonstrated the efficacy of Randezvous with a proof-of-concept exploit and a study on a real-world CVE.
- We evaluated Randezvous on our benchmarks and applications and found that it incurs, on average, 5.9% performance overhead, 15.4% code size overhead, and 22.0% data size overhead.

The rest of the chapter is organized as follows. Section 5.2 defines our threat model. Sections 5.3 and 5.4 describe the design and implementation of Randezvous, respectively. Section 5.5 evaluates the security of Randezvous, Section 5.6 evaluates the performance of Randezvous, and Section 5.7 discusses related work.

5.2 Threat Model

We assume a benign but potentially buggy bare-metal MCU application with memory safety errors that allow a remote attacker to write (and optionally read) arbitrary memory locations. We assume the attacker wants to launch a control-flow hijacking attack, such as return-into-libc [224, 248] or return-oriented programming (ROP) [213, 223], against the system. Memory safety attacks that do not corrupt control data (e.g., non-control data attacks [49]) are out of scope. We further assume that the attacker has a copy of the source code and can generate native code of the same instruction set as used on the system (though layouts may be different due to randomization). The attacker can

therefore locate exploitable vulnerabilities and find reusable code in the program's code segment for the aforementioned attacks. As Randezvous uses randomization to thwart code reuse attacks, it faces several threats that may undermine its defenses:

Threat 5.1. *An attacker may attempt to use a buffer overread [241] to read the code segment and locate reusable code.*

Threat 5.2. *An attacker may attempt to use a buffer overread [241] to read control data (pointers to code like return addresses and function pointers) out of memory to locate reusable code.*

Threat 5.3. *An attacker may attempt to guess the location of reusable code or the location of a control data slot (a memory location containing control data) in a brute force attack.*

Threat 5.4. *An attacker may corrupt a control data slot to hijack the control flow.*

Threat 5.5. *An attacker may "spray" control data across a memory region [231] to corrupt all control data slots within that region.*

5.3 Design

Randezvous is a compiler that transforms code installed on an ARMv7/8-M MCU and a set of runtime support routines used by the MCU's reset and exception handlers. Our design requires the memory protection unit (MPU) support, the set of debug registers needed by PicoXOM [227], and a hardware-based cryptographically secure pseudorandom number generator (CSPRNG). These features are available on many real-world MCUs, from low-end (e.g., STM32L412R8 [240]) to high-end (e.g., MIMXRT685-EVK [190]) and across manufacturers (e.g., STMicroelectronics [238], Microchip [174], and Renesas [210]).

In principle, Randezvous protects control data by destroying it when possible and hiding it with improved entropy when destruction is infeasible. We break down Randezvous's design components into three categories:

1. randomization and code protection,
2. control data protection, and
3. entropy improvements.

We first describe the randomization and code protection schemes that Randezvous employs, which previous work [59, 227] explored. We then explain how Randezvous's control data protection and entropy-improving techniques mitigate the additional threats described in Section 5.2. Though orthogonal to issues that Randezvous addresses, we also discuss how to deploy compile-time diversified binaries at scale.

5.3.1 Randomization and Code Protection

Traditional code reuse attacks [213, 248] require the attacker to know a priori the location of reusable code in memory. Randezvous therefore utilizes randomization and XOM to force the attacker to either use a buffer overread to leak control data [65, 82, 204, 214] or use brute force attacks that guess the location of reusable code.

Specifically, Randezvous performs the following randomized permutations of code at compile time:

1. *Function layout reordering*: Randezvous places each function in the program at a random location in the code segment.
2. *Basic block layout reordering*: In each function, Randezvous shuffles the order of basic blocks. If a basic block can fall through to a successor, they are kept contiguous in memory to avoid adding extra branch instructions to the code.

3. *Trap instruction insertion*: Randezvous fills *unused* code segment memory (between functions and between basic blocks that do not fall through) with trap instructions. These instructions are never executed during benign executions and only detect attack probes that jump to unused code. When that happens, Randezvous's trap handler responds by rebooting the system and optionally alerting a system administrator that a potential attack attempt has been thwarted.

Randezvous also randomizes the layout of global data segments (i.e., `.rodata`, `.data`, and `.bss`) at compile time by placing each memory object at a random location in its segment. The reason to use compile-time randomization rather than runtime rerandomization is that, compared to the former, the latter requires significantly more MCU resources (e.g., separate memory for storing the original program to be randomized) while only adding one extra bit of entropy against brute force attacks [224].

To mitigate Threat 5.1, Randezvous employs XOM on the code segment. As the ARMv7/8-M MPU does not support XOM [21, 23], Randezvous employs a software alternative named PicoXOM, as we present in Chapter 4. PicoXOM [227] configures the ARM debug registers, called Data Watchpoint and Trace (DWT) comparators [21, 23], to generate a trap if a read is performed from the code segment. Furthermore, since the debug registers are memory-mapped [21, 23], PicoXOM uses additional DWT comparators to ensure that XOM cannot be disabled by writing to the debug registers.

5.3.2 Control Data Protection

Randomization plus XOM defeats Threat 5.1. However, an attacker can attack the system by leaking control data (Threat 5.2) or by guessing the location of code (Threat 5.3) and then corrupting control data in memory (Threats 5.4 and 5.5). We now describe how Randezvous protects the confidentiality and integrity of control data.

Decoy Pointers To mitigate Threats 5.2 and 5.4, we developed *decoy pointers*, which are code pointers that point to random trap instructions and are used to fill unused data memory. Unlike other techniques used in code/data layout randomization, decoy pointers are novel as they, when combined with randomization and XOM, can camouflage genuine control data (slots): attackers leaking data via a buffer overread [241] cannot distinguish actual control data from decoy pointers; neither can they distinguish control data slots from unused data memory. Even if leaked, decoy pointers are lethal and using them in control-flow hijacking risks trapping the system.

By default, Randezvous only fills unused memory in the global data segments with decoy pointers. Subsequent paragraphs explain how Randezvous protects control data on the stack by moving it to the global data segments. As many MCU heap implementations simply manage a statically allocated chunk of memory in a global data segment as the heap, such a heap as a whole benefits from decoy pointers placed around it. A more overhead-tolerant implementation could camouflage in-heap control data by providing a custom `free()` that refills freed memory with decoy pointers.

Diversified Shadow Stack Return addresses on the stack pose two challenges in our threat model. First, return addresses are the most common target to corrupt in code reuse attacks (Threat 5.4). Second, return addresses are relatively easy to leak (Threat 5.2) via stack-based buffer overreads [241]. Randezvous must protect return addresses to mitigate these threats.

Randezvous protects return addresses by using a *diversified shadow stack*, which is a compact shadow stack [43] with random per-function strides. Randezvous employs four methods of randomizing the shadow stack. First, Randezvous places the shadow stack in the `.data` segment so that its location is randomized at compile time. Second, Randezvous initializes the shadow stack with decoy pointers, camouflaging real return addresses. Third, Randezvous selects a static random stride value for each function at compile time. Fourth, Randezvous selects a dynamic global random stride value at

boot time from the CSPRNG. For each non-leaf function, the static and dynamic stride values are added to a shadow stack pointer in its prologue to determine the location for saving the return address for the next function call. Likewise, the stride values are subtracted from the shadow stack pointer in its epilogue before loading its own return address from the shadow stack. Since the dynamic stride value is selected at boot time, the memory locations to which return addresses are stored get rerandomized for each reboot. Randezvous further encodes the static stride values in code and keeps the shadow stack pointer and dynamic stride value in reserved registers to prevent leakage and corruption.

Local-to-Global Variable Promotion Local variables that hold function pointers are susceptible to leakage (Threat 5.2) or corruption (Threat 5.4) as they are stored on the regular stack. An attacker can use a buffer overflow to corrupt them with addresses of reusable code and can also use a buffer overread [241] to leak them and use them to learn the location of reusable code. To mitigate such threats, we developed a simple *local-to-global variable promotion* transformation in the Randezvous compiler. This transformation converts local variables that may contain function pointers into global variables, enabling global data layout randomization to randomize their locations and decoy pointers to camouflage them.

The transformation is safe as long as the function containing the promoted variable is not called recursively or concurrently by multiple threads. To support local function pointers in recursive functions, Randezvous promotes each of such function pointers to an array and requires a maximum recursion depth specified as the array length. The function is then instrumented to use a copy of the function pointer for each recursion. As Randezvous targets bare-metal single-threaded MCU applications, multithreading is not an issue. To support multithreading, all promoted variables (as well as the diversified shadow stack described earlier in this section) must be placed in thread-local storage. We leave multithreading support for future work.

Return Address Nullification Our diversified shadow stack described earlier in this section mitigates return address leakage. However, a buffer overread [241] may still allow an attacker to leak large amounts of the shadow stack at a time. To further reduce the danger of such leakage, we developed a new compiler transformation called *return address nullification* which overwrites the stale return address on the shadow stack with a null value before a function returns. This transformation ensures that a single buffer overread can only leak the return addresses of actively executing functions, limiting the number of return addresses a particular buffer overread can disclose.

When nullifying a return address, instead of zeroing it out, Randezvous overwrites it with a distinct decoy pointer statically chosen and encoded in code for each nullification site. In this way, Randezvous ensures that memory used for return addresses always appears to be decoy pointers, sustaining its initial state.

5.3.3 Entropy Improvements

Despite Randezvous's randomization and control data protection schemes, the entropy they provide on MCUs with small memory size may not effectively resist brute force and control data spraying attacks (as Section 5.5 will discuss). This section discusses how Randezvous improves the limited entropy on MCUs to mitigate such attacks.

Delayed Reboot Randezvous's code reuse defenses are probabilistic: each time an attacker tries to attack the system by guessing where reusable code is located or by guessing which chunk of memory contains control data, there is a small chance that the attacker will guess correctly. Consequently, if the attacker repeatedly tries different values and has no bound on the number of attack attempts (Threat 5.3), there is an amount of time by which the attacker is expected to guess correctly and succeed. It is then natural to ask how long a system is expected to resist such brute force attacks. If the time is sufficiently long, then probabilistic defenses suffice.

Our security analysis in Section 5.5 models such attacks and computes the time by which we expect an attacker to succeed. Our analysis shows that the entropy provided by the aforementioned Randezvous defenses alone may not effectively resist such attacks for a reasonable length of time for all MCUs; small-sized MCUs simply have too few places in which to hide code and/or camouflage control data.

As the number of possible locations of a single piece of reusable code or control data is too small, the only other recourse is to make each failed attack attempt take longer. Hence, we devised a technique called *delayed reboot* which artificially delays a system's reboot. Whenever it detects a trap caused by a violation of Randezvous's security policies, Randezvous reboots the system. Successive reboots are incrementally slowed, artificially reducing the number of failed attempts an attacker can feasibly perpetrate in a given amount of time. For the i-th successive reboot caused by a violation, an artificial delay in time D_i is added to the boot sequence. D_i increases as i increases until the number of such reboots reaches a predetermined value R, after which D_i remains constant.

Delayed reboot exchanges availability for confidentiality and integrity through configuration of the parameter R and the delay function incrementing D_i: a smaller value of R and larger values of D_i provide more integrity and confidentiality at the expense of availability. Section 5.5 quantifies the security gain and availability loss of using delayed reboot, and we use our analysis results to inform concrete configurations to meet specific system requirements. However, we note that delayed reboot may not be appropriate for systems with hard real-time requirements or that cannot tolerate service disruptions. For example, a car's engine control unit may not be suitable for delayed reboot due to real-time constraints while, in contrast, a network of monitoring sensor devices can tolerate delayed reboot, especially if multiple devices monitor overlapping areas to provide redundancy. Systems that cannot use delayed reboot will need to use more memory to gain the entropy needed to stay secure.

Global Guards Randomizing global data and camouflaging control data with decoy pointers together hinder an attacker from corrupting control data in the global data segments. However, the attacker can still corrupt control data in these regions via spraying attacks (Threat 5.5). If corrupting non-control data does not crash the program, a buffer overflow that writes to the whole `.data` segment is *guaranteed* to corrupt control data in the `.data` segment, neutralizing the already limited entropy of randomization.

To mitigate this threat, we repurposed guard memory [64], which was originally meant to detect stack smashing only. At boot time, Randezvous uses the CSPRNG to randomly select one or more randomly sized pieces of unused data memory as *global guards* and configures the MPU to disallow writes to them (the specific number depends on how many regions the MPU can support). Attempted writes to them cause a trap and trigger a reboot.

Global guards establish the entropy against spraying attacks; successful attacks must avoid writing to any global guards, which becomes much less probable. Since global guards are randomly selected at boot time, their location information from previous failed attack attempts cannot be used to inform future attacks.

5.3.4 Diversified Binary Deployment

Deploying and updating diversified MCU application binaries provides challenges to software developers. However, we believe that such challenges can be readily addressed. When a device manufacturer releases a new version of software for an MCU, they can first translate all compilation units to LLVM intermediate representation (IR) bitcode and link the files into a single LLVM IR file containing all the code using LLVM's link-time optimization (LTO) features [142]. They can then, for each device, generate random seeds using a CSPRNG or a true random number generator (TRNG), have the compiler's code generator translate the LLVM IR into a randomized binary using those seeds, and then record in a database the hash of the generated binary and

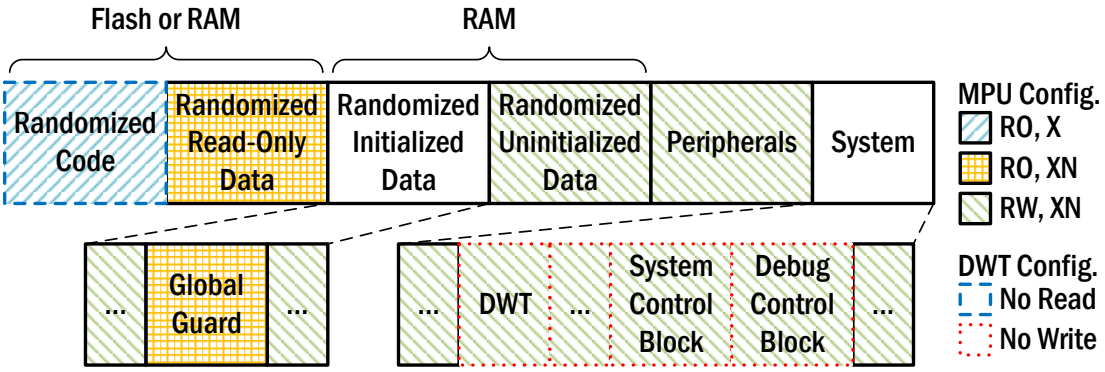

Figure 5.1: Memory Protection of Randezvous Prototype

the seeds that were used to create it.

When a customer submits a crash dump or requests a service from the device manufacturer that requires knowing which diversified binary the customer is using, the customer can simply supply the hash of their binary file. The device manufacturer can then feed the corresponding random seeds from the database into the code generator and regenerate the randomized binary. In this way, the device manufacturer can always re-create the randomized binary without having to store a copy of each binary given to a customer.

5.4 Implementation

We built a prototype of Randezvous for the ARMv8-M architecture with all of our design components except delayed reboot; we opted not to implement it as it does not impact our evaluation. We added four compiler passes to LLVM/Clang 11.0.1's ARM code generator [142], totaling 4,336 source lines of code using Tokei 12.1.2 [269]. We now describe our prototype's memory layout and then describe implementation details of our compiler passes.

5.4.1 PicoXOM Enhancements and Memory Configuration

PicoXOM, Randezvous's XOM component, was only implemented for ARMv7-M [227], as Chapter 4 described. We extended its MPU and DWT configuration code to support ARMv8-M. Unlike the previous implementation on ARMv7-M, which can only protect code segments up to 128 KB, we have verified that PicoXOM on ARMv8-M can support an arbitrary code size. This allows our prototype to support larger code bases. It also increases the entropy for code layout randomization if the usable memory for code is larger than 128 KB. Our extended PicoXOM implementation contains 361 source lines of C code.

Figure 5.1 shows our Randezvous prototype's memory protection. It requires five MPU regions to cover code, read-only data, RAM, a single global guard, and peripherals. Note that the System region requires no separate MPU region because it is readable, writable, and execute-never for privileged code regardless of the MPU configuration [23]. ARMv8-M DWT comparators must be used in pairs to monitor memory address ranges [23], so our prototype uses four DWT comparators (two pairs) to read-protect the code segment and write-protect critical memory-mapped system registers.

5.4.2 Code Layout Randomization

The code layout randomization pass, as Section 5.3.1 describes, randomizes the code layout by shuffling the order of functions and basic blocks and inserting trap instructions between them. It takes a size option for developers to specify the maximum code size and a seed option to be able to generate different code layouts. We used LLVM's `RandomNumberGenerator` [160] to make our experimental results more reproducible.

```
push {r4, lr}
...
pop  {r4, pc}
```

(a) Original Code

```
str  lr, [r8], #32
add  r8, r8, r9
push {r4}
...
pop  {r4}
sub  r8, r8, r9
ldr  pc, [r8, #-32]!
```

(b) Diversified Shadow Stack w/ Static Stride of 32

```
str  lr, [r8], #32
add  r8, r8, r9
push {r4}
...
pop  {r4}
sub  r8, r8, r9
ldr  lr, [r8, #-32]!
movw ip, #decoy-lo16
movt ip, #decoy-hi16
str  ip, [r8]
bx   lr
```

(c) Return Address Nullification

Figure 5.2: Example of Randezvous Prologue/Epilogue Transformations

5.4.3 Global Data Layout Randomization

The global data layout randomization pass randomizes the layout of global data segments by shuffling the order of global variables and inserting an unused memory object (called a garbage object) of random size between each two of them. Similarly, it takes three size options for the maximum size of the three global data segments and also comes with a seed option. We opted to implement decoy pointers and global guards in this pass as well, as they reside in the global data segments. The former is by initializing `.rodata` and `.data` garbage objects with addresses of random trap instructions. For the latter, we opted to implement support for a single global guard by randomly picking a developer-specified number of `.data` garbage objects and encoding their addresses and sizes in a runtime function. This function randomly picks a garbage object from those encoded ones as the global guard and returns its address and size; the source of randomness used in the function is the CSPRNG whose address is specified by developers. Randezvous's MPU configuration code calls this function and sets up a read-only MPU region for the global guard.

5.4.4 Diversified Shadow Stack

The diversified shadow stack pass transforms function prologues and epilogues so that they access the shadow stack for their return addresses. The pass takes a seed option and creates the shadow stack as a global variable whose size can be specified by developers. The shadow stack's location in the `.data` segment is randomized by the global data layout randomization pass. To improve performance and avoid leakage of the shadow stack address and stride, we reserve two callee-saved registers `r8` and `r9` for the shadow stack pointer and stride value, respectively. Our pass generates a runtime function to initialize the two registers: it sets `r8` to point to the shadow stack, loads a random number to `r9` from the CSPRNG, and clears a developer-specified number of high bits in `r9` to limit the stride length. The system's boot code calls this function before making any other function calls. During transformation, our pass generates a random static stride for each function, whose length is also limited by the same number of bits. As the pass already finds and transforms instructions in function epilogues, we opted to implement return address nullification in the pass as well. For each instrumented function epilogue, our pass randomly picks a trap instruction and generates code that writes its address back to the shadow stack. Figure 5.2 illustrates the two transformations performed on a function's prologue and epilogue.

5.4.5 Local-to-Global Variable Promotion

The local-to-global variable promotion pass promotes local variables whose type contains function pointer types to globals, as Section 5.3.2 describes. The pass operates on LLVM IR bitcode before it is lowered to machine code. As none of our benchmarks and applications uses local function pointers in recursive functions, we elided implementing support for it.

Symbol	Definition
S_C	Size of randomized code segment
S_{C_O}	Size of original application code
S_T	Size of control flow target
S_D	Size of randomized `.data` segment
$S_{D'}$	Size of memory in `.data` that does not resemble control data
S_{D_0}	Size of zeroed memory in `.data`
S_G	Total size of all global guards
S_W	Size of memory in `.data` that attacker chooses to corrupt
N	Number of control data slots in `.data`
$p_{S,x}$	Probability of success w/ Strategy x
$p_{T,x}$	Probability of finding/hitting a trap w/ Strategy x
F_x	Search space size of Strategy x
p_i	Probability of success at i-th guess regardless of strategy
A	# of bin permutations (only for Equ. 5.5, 5.6, and 5.7)
C_S	# of bin combinations where attacker succeeds (only for Equ. 5.5, 5.6, and 5.7)
A_S	# of bin permutations where attacker succeeds (only for Equ. 5.5, 5.6, and 5.7)
p_S	Probability of success
p_T	Probability of trapping the system
P_x	Number of guesses for a success in Strategy x
P	Number of brute force attacks for a success
$E(X)$	Expected value of random variable X
t_B	Time from booting to reaching an exploitable vulnerability
t_N	Time for attacker to send and receive data over network
T_n	Expected time to resist brute force attacks w/o delayed reboot
T_d	Total time of delay provided by delayed reboot
T_{min}	Expected time to resist brute force attacks w/ delayed reboot
D_i	Time of delay at i-th reboot caused by security violation
R	Number of reboots after which reboot delay stops increasing

Table 5.1: Mathematical Symbol Definitions in Randezvous Security Evaluation

5.5 Security Evaluation

We now evaluate Randezvous's security by measuring the entropy it adds to three different-sized MCUs and computing the amount of reboot delay needed to protect these systems from attacks for a given amount of time. We then provide a proof-of-concept exploit that experimentally demonstrates the security of Randezvous and a study on how Randezvous could mitigate attacks exploiting a real-world CVE. Table 5.1 lists all mathematical symbols used in this section for quick reference.

5.5.1 Attack Procedure

We model a return-into-libc [224, 248] control-flow hijacking attack as it is the simplest. Other types of attacks (e.g., ROP [213, 223] and JIT-ROP [230]) require locating additional reusable code and therefore require leaking or guessing more code locations. Consequently, if Randezvous can resist return-into-libc attacks, it should be able to resist these more sophisticated attacks as well. In the return-into-libc attack, the attacker follows the two steps below:

1. Locate the control flow target to which to jump. For a return-into-libc attack, this is the address of a function.

2. Find a control data slot and corrupt it with the address of the control flow target acquired in Step 1. This is usually where a return address or a function pointer is stored, which will be used in a future control flow transfer.

On an unprotected system, Step 1 can be skipped because the attacker has a priori knowledge of the code layout. However, as Randezvous randomizes the code and data layouts and forbids code reads via PicoXOM [227] (Section 5.3.1), the attacker is forced to

1a) guess the location of the control flow target, or

1b) try leaking a return address using a buffer overread, or

1c) try leaking a function pointer (if any) using a buffer overread.

Similarly, in Step 2, finding the address of a control data slot is no longer straightforward for the attacker; in Randezvous, control data is stored in the `.data` segment (Section 5.3.2), randomized to unknown locations (Section 5.3.1), camouflaged among numerous decoy pointers (Section 5.3.2), and protected by randomly-picked non-writable global guards (Section 5.3.3). As a result, the attacker must either

2a) guess the location of a control data slot to corrupt, or

2b) massively corrupt part of the `.data` segment, aiming to corrupt a desired control data slot while hitting none of the global guards, i.e., a control data spraying attack [231].

5.5.2 Attack Probe Analysis

Our analysis assumes that the attacker knows the boundaries of randomized memory regions and makes no out-of-bounds guesses in Steps 1 and 2. While not always true in practice, this assumption biases the analysis in the attacker's favor and simplifies our analysis.

We first analyze the expected number of attempts the attacker needs for each strategy to complete Step 1. For Strategy 1a, let S_C be the size of the randomized code segment, S_{C_O} be the size of the original application code, and S_T be the size of the control flow target. Assuming the control flow target is 2-byte aligned (typical for Thumb instructions [21, 23]) and has an equal chance to appear in each eligible location, then the probability of success is $p_{S,1a} = \frac{2}{S_C - S_T + 2}$; the chance of finding a trap instruction can be approximated as $p_{T,1a} = \frac{S_C - S_{C_O}}{S_C} \cdot \frac{S_C - S_T}{S_C}$. For a brute force attack, an attacker can simply retry the attack repeatedly with different values for the control flow target until the attack works, excluding previously guessed values each time a new guess is made. If P_x is a random variable representing the number of guesses for a success in Strategy x ($x \in \{1a, 1b, 1c, 2a, 2b\}$), then the expected number of guesses for completing Step 1 with Strategy 1a is

$$E(P_{1a}) = \frac{S_C - S_T + 4}{4}. \tag{5.1}$$

To derive Equation 5.1, let F_{1a} be the search space size of Strategy 1a. We have

$$F_{1a} = \frac{S_C - S_T + 2}{2}.$$

We hereafter use p_i to represent the probability of success at i-th guess regardless of the strategy. In Strategy 1a, we have

$$p_i = \frac{\cancel{F_{1a}-1}}{F_{1a}} \cdot \frac{\cancel{F_{1a}-2}}{\cancel{F_{1a}-1}} \cdots \frac{\cancel{F_{1a}-i+1}}{\cancel{F_{1a}-i+2}} \cdot \frac{1}{\cancel{F_{1a}-i+1}} = \frac{1}{F_{1a}}.$$

So the expected number of guesses for a success in Strategy 1a, $E(P_{1a})$, can be expressed by:

$$E(P_{1a}) = \sum_{i=1}^{F_{1a}} i \cdot p_i = \sum_{i=1}^{F_{1a}} i \cdot \frac{1}{F_{1a}} = \frac{S_C - S_T + 4}{4}.$$

For Strategies 1b and 1c, our analysis assumes the attacker's best case scenario: a function pointer or return address pointing into the desired function exists in a single memory location; if leaked, the attacker can locate the function. In this scenario, the attacker first uses a buffer overread [241] to leak the entire contents of the `.data` segment and then examines it for the desired control data. Even so, the attacker can at most eliminate values that do not look like control data and still must guess which of the remaining ones can be used. This is because Randezvous randomizes the `.data` segment layout and camouflages control data with decoy pointers. Let S_D be the size of the randomized `.data` segment, $S_{D'}$ be the size of memory in the `.data` segment that does not look like control data, and N be the number of control data slots in the `.data` segment. N can be approximated by the current call chain depth (due to return address nullification in Section 5.3.2) plus the number of function pointers in the program. Assuming the desired control data has an equal chance to appear in each possible location, the attacker must try every memory location that appears to contain control data. The probability of success is $p_{S,1b} = p_{S,1c} = \frac{4}{S_D - S_{D'}}$, and the probability of finding a decoy pointer is approximately $p_{T,1b} = p_{T,1c} = \frac{S_D - S_{D'} - 4N}{S_D - S_{D'}}$. The difference between Strategies 1b and 1c is whether the attacker can exclude a previously incorrect guess: return addresses might be stored in different memory locations as the dynamic shadow stack stride is randomized on each boot, while function pointers always reside in the same address across reboots as the `.data` segment is randomized once at

compile time. This leads to a difference in the expected number of guesses, shown in Equations 5.2 and 5.3, respectively:

$$E(P_{1b}) = \frac{S_D - S_{D'}}{4} \tag{5.2}$$

$$E(P_{1c}) = \frac{S_D - S_{D'} + 4}{8} \tag{5.3}$$

To derive Equation 5.2, we first have

$$p_i = (1 - p_{S,1b})^{i-1} p_{S,1b}.$$

So

$$E(P_{1b}) = \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i(1 - p_{S,1b})^{i-1} p_{S,1b}.$$

According to geometric distribution,

$$E(P_{1b}) = \frac{1}{p_{S,1b}} = \frac{S_D - S_{D'}}{4}.$$

Similar to the derivation of Equation 5.1, to derive Equation 5.3, let F_{1c} be the search space size of Strategy 1c. We have

$$F_{1c} = \frac{S_D - S_{D'}}{4}$$

and

$$p_i = \frac{1}{F_{1c}}.$$

So

$$E(P_{1c}) = \sum_{i=1}^{F_{1c}} i \cdot p_i = \sum_{i=1}^{F_{1c}} i \cdot \frac{1}{F_{1c}} = \frac{S_D - S_{D'} + 4}{8}.$$

We now consider the expected number of attempts needed to complete Step 2. For Strategy 2a, the attacker can also leverage a buffer overread [241] on the `.data` seg-

ment to filter out memory that does not resemble control data except for zeroed memory, which might be uninitialized control data slots. Let S_{D_0} be the size of zeroed memory in the `.data` segment and S_G be the total size of all global guards. The chance of success is $p_{S,2a} = \frac{4N}{S_D - S_{D'} + S_{D_0}}$, and the chance of hitting any of the global guards is $p_{T,2a} = \frac{S_G}{S_D - S_{D'} + S_{D_0}}$. Similar to Strategy 1b, the attacker cannot exclude previously incorrect guesses due to both the dynamic shadow stack stride and the global guards, so the expected number of guesses is

$$E(P_{2a}) = \frac{S_D - S_{D'}}{4N}. \tag{5.4}$$

Similar to the derivation of Equation 5.2, to derive Equation 5.4, we have

$$p_i = (1 - p_{S,2a})^{i-1} p_{S,2a}.$$

and

$$E(P_{2a}) = \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i(1 - p_{S,2a})^{i-1} p_{S,2a}.$$

According to geometric distribution,

$$E(P_{2a}) = \frac{1}{p_{S,2a}} = \frac{S_D - S_{D'}}{4N}.$$

For Strategy 2b, let S_W be the size of memory in the `.data` segment that the attacker chooses to corrupt. Equations 5.5 and 5.6 give the probability of success and of hitting a global guard, respectively, and Equation 5.7 computes the expected number of attempts:

$$p_{S,2b} = \frac{\sum_{i=1}^{\min(N, \frac{S_W}{4})} C(N,i) C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4} - i)}{C(\frac{S_D}{4}, \frac{S_W}{4})} \tag{5.5}$$

$$p_{T,2b} = 1 - p_{S,2b} - \frac{C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4})}{C(\frac{S_D}{4}, \frac{S_W}{4})} \tag{5.6}$$

$$E(P_{2b}) = \frac{1}{p_{S,2b}} \tag{5.7}$$

To help derive Equations 5.5, 5.6, and 5.7, note that the condition of success is by corrupting at least one of N control data slots while hitting none of the current global guards, and note that the condition of trapping the system is by hitting any of the current global guards. This can be modeled as the following situation:

- The attacker picks $\frac{S_W}{4}$ consecutive bins out of $\frac{S_D}{4}$ bins sorted in a certain order, N of which are black (representing control data slots) and $\frac{S_G}{4}$ of which are red (representing the current global guards).
- The attacker succeeds if the $\frac{S_W}{4}$ bins she picks contain no red bin and at least one black bin.
- The attacker traps the system if the $\frac{S_W}{4}$ bins she picks contain at least one red bin.

The total number of different bin permutations (denoted as A) is $\frac{S_D}{4}!$. The number of bin combinations in which the attacker succeeds (denoted as C_S) is the number of all possible combinations of i black bins and $\frac{S_W}{4} - i$ non-black non-red bins ($i \in \{1, 2, \ldots, \min(N, \frac{S_W}{4})\}$), which can be calculated by

$$C_S = \sum_{i=1}^{\min(N, \frac{S_W}{4})} C(N, i) C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4} - i).$$

This number can then be used to calculate the number of bin permutations in which the attacker succeeds (denoted as A_S), by multiplying it with the number of all possible permutations with the starting location of the $\frac{S_W}{4}$ bins fixed (as the bins the attacker picks must be consecutive). So we have

$$A_S = C_S \cdot \frac{S_W}{4}! \cdot \frac{S_D - S_W}{4}!$$

and therefore

$$p_{S,2b} = \frac{A_S}{A} = \frac{\sum_{i=1}^{\min(N,\frac{S_W}{4})} C(N,i)C(\frac{S_D-S_G}{4}-N, \frac{S_W}{4}-i)}{C(\frac{S_D}{4}, \frac{S_W}{4})}.$$

We can calculate $p_{T,2b}$ indirectly by first calculating the probability of the attacker picking all $\frac{S_W}{4}$ bins as non-black non-red bins and then doing a subtraction from 1. Since the number of bin combinations of $\frac{S_W}{4}$ non-black non-red bins is $C(\frac{S_D-S_G}{4}-N, \frac{S_W}{4})$, we can easily get

$$p_{T,2b} = 1 - p_{S,2b} - \frac{C(\frac{S_D-S_G}{4}-N, \frac{S_W}{4})}{C(\frac{S_D}{4}, \frac{S_W}{4})}.$$

Finally, $E(P_{2b})$ is derived in a similar way to that in $E(P_{1b})$ and in $E(P_{2a})$. We have

$$p_i = (1 - p_{S,2b})^{i-1} p_{S,2b}$$

and

$$E(P_{2b}) = \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i(1 - p_{S,2b})^{i-1} p_{S,2b}.$$

According to geometric distribution,

$$E(P_{2b}) = \frac{1}{p_{S,2b}}.$$

Combining the two steps, there are three outcomes:

1) success, only when an attacker makes a correct guess in both steps;

2) nothing happening, due to an incorrect guess in Step 2 that hits none of the global guards;

3) trap, which can be caused by an incorrect guess in either Step 1 (finding a decoy pointer) or Step 2 (hitting any of the global guards).

As none of the two unsuccessful outcomes gives information about which control data

(slot) is (in)correct, the attacker can only guess blindly in both steps. Let P be a random variable of the number of brute force attacks for a success. Since the two steps are independent of each other, we have the chance of success $p_S = p_{S,x} \cdot p_{S,y}$, the chance of trapping the system $p_T = p_{T,x} \cdot p_{S,y} + p_{T,y}$, and *the expected number of brute force attacks for a success* $E(P) = E(P_x) \cdot E(P_y)$ if the attacker adopts Strategies x and y ($x \in \{1a, 1b, 1c\}$ and $y \in \{2a, 2b\}$).

5.5.3 Time Analysis

Entropy measures a system's randomness, but it fails to measure the system's strength against brute force attacks as it fails to consider the frequency at which attacks are launched. We therefore analyze how long a Randezvous-protected system, with different sizes, can resist brute force attacks. This analysis informs the configuration of delayed reboot and thus controls the security/availability trade-off.

Let t_B be the time from booting to reaching a vulnerability that an attacker can exploit and t_N be the time for the attacker to send an attack payload and receive its execution result over the network. Without Randezvous's delayed reboot, we can expect the system to withstand brute force attacks by an amount of time $T_n = (p_T \cdot t_B + t_N) \cdot E(P)$. With delayed reboot providing a total delay of T_d, we wish the whole system to resist brute force attacks for at least an amount of time T_{min} before the attacker finishes the expected number of attack payloads to succeed. So we have $T_d \leq \sum_{i=1}^{R} D_i$, $T_n + T_d = T_{min}$, and $R \leq p_T \cdot E(P)$, where R is the number of reboots after which the delay stops increasing and $\{D_i\}_{i=1}^{R}$ is the sequence of the delay Randezvous adds to the i-th reboot, as Section 5.3.3 describes.

We aim to protect the system from brute force attacks for three or more days. Three days give the system time to alert an administrator about the attack and for the administrator to respond, even if the attack commences during a short period in which the administrator is unavailable (e.g., a weekend). Table 5.2 lists three sets

	Small	Medium	Large		All Systems
S_C	32 KB	1 MB	16 MB	S_G	32 bytes
S_{C_O}	16 KB	128 KB	1 MB	S_T	16 bytes
S_D	32 KB	256 KB	4 MB	S_W	128 bytes
$S_{D'}$	1 KB	4 KB	32 KB	t_B	1 second
S_{D_0}	128 bytes	512 bytes	1 KB	t_N	0.6 seconds
N	8	32	64	T_{min}	3 days

Table 5.2: Common Values for Randezvous Time Analysis

System	Case	Strategies	$E(P)$	p_T	T_n
Small	Worst	$\{1a, 2a\}$	8,155,248.0	0.151%	56.8 days
Small	Best	$\{1c, 2b\}$	132,651.0	6.073%	1.0 days
Medium	Worst	$\{1a, 2a\}$	529,522,800.0	0.557%	10.1 years
Medium	Best	$\{1c, 2b\}$	2,087,482.7	1.934%	15.0 days
Large	Worst	$\{1a, 2a\}$	68,199,318,000.0	0.007%	1,297.7 years
Large	Best	$\{1c, 2b\}$	266,649,737.1	0.220%	5.1 years

Table 5.3: Randezvous Time Analysis Results (Best/Worst for the Attacker)

of common values representing three MCUs of different sizes (STM32L412R8 [240], STM32F469NIH6 [236], and MIMXRT685-EVK [189, 190]) and values we pick to evaluate attacks (latencies are based on a wireless network [229]) and Randezvous's protections. By substituting all variables with their corresponding values in each set, we can estimate whether delayed reboot is needed (i.e., whether $T_n < T_{min}$) and, if so, how much delay can be scattered throughout all R reboots. Our results, summarized in Table 5.3, show that the medium and large systems do not need delayed reboot; Randezvous's other protections can mitigate all the modeled attacks for at least half a month. The small system, however, requires an average per-reboot delay of 21.3 seconds to keep it probabilistically secure for three days against all possible attack strategies that we evaluated.

While our results necessitate delayed reboot for certain systems, we note that using an exponentially-growing delay will still provide reasonable availability when an attack commences while maintaining our target of three days worth of resilience. For example, our best case for the attacker expects the system to trap 8055.3 times; Randezvous could be configured with $\{D_i\}_{i=1}^{R}$ as an exponential sequence with $D_1 = 100$ ms, $R = 8055$,

and a ratio of 1.001. Even at the 2000-th reboot (at which point an administrator should have been notified), the delay on a single boot is just around 738 ms.

5.5.4 Exploit Analysis

Proof-of-Concept Exploit We built a proof-of-concept exploit to showcase Randezvous's security. The exploit consists of a script representing an attacker and a vulnerable application that can run on an NXP MIMXRT685-EVK board [190]. The application contains both arbitrary memory read and write vulnerabilities, matching our threat model in Section 5.2. To favor the attacker, it also contains a global function pointer pointing to the attacker's desired function. We compiled the application with three different configurations: one unprotected, one protected with only randomization and PicoXOM (as in Section 5.3.1), and one protected with full Randezvous. For the two protected configurations, we further configured the application to match each of the three different-sized MCUs in Table 5.2 as closely as possible. When running the application, the script communicates with the board via a serial port and sends attack payloads generated from the best strategies for the attacker for each configuration: direct return-into-libc for the unprotected, Strategy 1c with return address corruption for randomization plus PicoXOM, and Strategies 1c and 2b for Randezvous.

Our exploit finished immediately for the unprotected system as no guessing is needed in Step 1 or 2. With randomization plus PicoXOM, the exploit finished in 15 seconds, 68 seconds, and 2,821 seconds for the small, medium, and large systems, respectively. Most of time was spent trying out leaked values that resemble control data. In contrast, the exploit failed in all three Randezvous-protected systems after continuously sending attack payloads for three days.

Real-World CVE To demonstrate its efficacy against real-world exploits, we analyzed how Randezvous could stop attacks exploiting CVE-2021-27421 [75]. We picked this CVE because

1) it can both read from and write to arbitrary heap locations,

2) it affects applications using the NXP MCUXpresso SDK library, and

3) we can exploit it on our NXP MIMXRT685-EVK board [190].

CVE-2021-27421 [75] overflows a heap buffer. We built a demonstrative application with the CVE for the small system in Table 5.2 and compiled it with similar configurations to those used in our proof-of-concept exploit. We then launched a return-into-libc attack on our board, which exploits the CVE to corrupt a pointer in the heap to point to a memory location of our choice. The overwritten pointer is eventually dereferenced. Our attack utilizes attack strategies that do not use buffer overread or spraying (Strategy 1a with return address corruption for randomization plus PicoXOM and Strategies 1a and 2a for Randezvous) because the application stores no return address or function pointer to the attacker's desired control flow target in memory and because the exploit only corrupts four bytes of memory. Our attack exploited the unprotected system immediately and the system protected by randomization plus PicoXOM in 23.6 hours. In contrast, the attack failed on the Randezvous-protected system after running for three days; we therefore expect Randezvous to resist the attack for more than three days for the larger systems in Table 5.2.

5.6 Performance Evaluation

We evaluated Randezvous's performance on an NXP MIMXRT685-EVK board which has an ARM Cortex-M33 processor implementing the ARMv8-M Mainline architecture that can run up to 300 MHz [190]. It comes with 4.5 MB of SRAM, 64 MB of flash memory, a true random number generator (TRNG) that fulfills Randezvous's CSPRNG requirement, and an SD card slot [189, 190].

We used three benchmark suites and two real-world applications to evaluate Randezvous. **BEEBS** [195] is a benchmark suite to measure embedded systems' energy

usage. It includes a wide range of common MCU workloads (e.g., packet routing, sorting, and hashing). As many BEEBS programs are too small or perform too little computation, we picked 54 of its 80 programs that run longer than 0.1 seconds on our board for 10,240 iterations. **CoreMark-Pro** [92] is a benchmark suite that includes and enhances CoreMark [91], an industry standard benchmark for embedded processors, with more CPU- and memory-intensive programs. It consists of five integer benchmarks and four floating-point benchmarks that together characterize processor performance. **MbedTLS-Benchmark** [16] is a test program of the Mbed TLS library [15]. It measures the latency and throughput of various cryptographic algorithms (e.g., SHA, AES, and RSA). **PinLock** [58] is an application that emulates a password-based lock. It reads a 4-digit passphrase from a serial port, computes a SHA-256 hash of the input, and activates an LED if the hash matches the stored passphrase hash. **FatFs-SD** is an application from the board manufacturer. It operates a FAT file system on an SD card with file system creation, mounting, and file I/O. Previous work [9, 59, 227, 229] used PinLock and FatFs-SD.

We compiled each program into an ELF executable and loaded its code into the SRAM for execution, using two configurations: Baseline and Randezvous. In Baseline, we used the LLVM/Clang compiler [142] to compile programs with all Randezvous passes and runtime components disabled. In Randezvous, we enabled everything; all Randezvous's randomization seeds are set to zero, and all memory size options for Randezvous passes are set appropriately to allow execution in the SRAM while still adding entropy to the program. In particular, the shadow stack size and stride length were tailored to add one bit of entropy. Both configurations use the `-Os` and `-fomit-frame-pointer` options and perform link-time optimization (LTO) via the `-flto` and `-fuse-ld=lld` options.

5.6.1 Performance Overhead

To measure Randezvous's performance overhead, we configured each BEEBS benchmark to execute for 10,240 iterations of its workload and print out its execution time in milliseconds. Each benchmark in CoreMark-Pro was configured to execute for a minimal number of iterations that is a power of 10 and yields an execution time of at least 10 seconds. MbedTLS-Benchmark measures latency and throughput with 1,024 iterations and 1 or 3 seconds, respectively. All benchmarks produced identical numbers over multiple runs, yielding zero standard deviations. As PinLock and FatFs-SD access slow peripherals, we ran each of them 10 times and report the average execution time with a standard deviation.

Tables 5.4, 5.5, 5.7, 5.6, and 5.8 present Baseline performance in absolute numbers as well as the overhead Randezvous incurs relative to Baseline on BEEBS, CoreMark-Pro, MbedTLS-Benchmark, and the applications, respectively. Overall, Randezvous incurs minor performance overhead of 5.9%: 6.9% in BEEBS, 7.0% in CoreMark-Pro, 4.5% in MbedTLS-Benchmark's throughput, 5.5% in MbedTLS-Benchmark's latency, and 0.6% in the applications.

We studied the overhead by enabling only one of Randezvous's features at a time. We discovered that the diversified shadow stack and return address nullification transformations are the major sources of overhead in BEEBS and CoreMark-Pro. Specifically, the former reserves two registers and adds a few instructions in the prologue and epilogue(s) of every non-leaf function. The latter adds a few more instructions in those function epilogues. As a result, Randezvous incurred more overhead on benchmarks with higher register pressure and more frequent function calls. MbedTLS-Benchmark's latency overhead on each algorithm roughly matches its throughput overhead. The highest (in `DES` and `3DES`) also comes from these transformations. `ECDSA-secp521r1` saw a miniscule speedup in signature verification, likely caused by caching. Randezvous exhibits negligible runtime overhead in the applications. We

Benchmark	Baseline (ms)	Randezvous (×)	Benchmark	Baseline (ms)	Randezvous (×)
aha-compress	627	1.011	nettle-cast128	103	1.019
aha-mont64	555	0.984	nettle-sha256	309	1.055
bubblesort	2,164	1.017	newlib-sqrt	104	1.010
crc32	463	1.000	ns	268	1.000
ctl-stack	381	1.037	nsichneu	207	1.005
ctl-string	341	1.117	picojpeg	28,879	1.065
ctl-vector	547	1.007	prime	304	1.056
cubic	20,807	1.168	qrduino	25,309	1.084
dijkstra	21,798	1.034	rijndael	24,462	1.005
dtoa	347	1.133	sglib-arraybinsearch	509	1.051
edn	2,071	1.020	sglib-arrayheapsort	609	1.002
fasta	14,502	1.000	sglib-arrayquicksort	381	1.097
fdct	110	1.018	sglib-dllist	1,111	1.094
fir	8,643	1.030	sglib-hashtable	687	1.262
frac	3,998	1.220	sglib-listinsertsort	1,247	1.124
huffbench	13,310	1.031	sglib-listsort	808	1.069
jfdctint	116	1.017	sglib-queue	713	1.000
levenshtein	1,757	1.085	sglib-rbtree	2,830	1.207
ludcmp	104	1.010	slre	1,465	1.146
matmult-float	781	1.013	sqrt	24,751	1.171
matmult-int	3,623	1.004	st	15,691	1.192
mergesort	15,058	1.147	stb_perlin	3,736	1.207
miniz	336	1.006	stringsearch1	352	1.011
nbody	98,435	1.117	trio-snprintf	385	1.086
ndes	1,180	1.018	trio-sscanf	687	1.086
nettle-aes	1,893	1.130	whetstone	84,383	1.215
nettle-arcfour	579	1.040	wikisort	70,699	1.138
Min (×)					0.984
Max (×)					1.262
Geomean (×)					1.069

Table 5.4: Randezvous BEEBS Execution Time (Lower is Better)

believe this is due to I/O dominating the execution time.

5.6.2 Memory Overhead

Memory usage is critical for MCUs. We therefore measured how much memory Randezvous uses to provide its protections by calculating code and global data segment sizes (without unused memory) before and after its transformations during compilation.

Benchmark	Baseline (ms)	Randezvous (×)	Benchmark	Baseline (ms)	Randezvous (×)
cjpeg-rose7-preset	21,172	1.023	parser-125k	41,700	1.069
core	33,813	1.112	radix2-big-64k	15,363	1.177
linear_alg-mid-100x100-sp	45,177	1.001	sha-test	17,220	1.046
loops-all-mid-10k-sp	73,085	1.010	zip-test	37,097	1.014
nnet_test	183,048	1.195			
Min (×)					1.001
Max (×)					1.195
Geomean (×)					1.070

Table 5.5: Randezvous CoreMark-Pro Execution Time (Lower is Better)

Cryptographic Algorithm	Baseline (cycle/byte)	Randez-vous (×)	Cryptographic Algorithm	Baseline (cycle/byte)	Randez-vous (×)
MD5	15.84	1.009	AES-GCM-256	113.37	1.084
SHA-1	3.47	1.009	AES-CCM-128	72.75	1.087
SHA-256	3.30	1.009	AES-CCM-192	75.54	1.081
SHA-512	109.40	1.022	AES-CCM-256	78.33	1.070
3DES	298.72	1.145	CTR_DRBG (NOPR)	27.22	1.092
DES	117.32	1.145	CTR_DRBG (PR)	47.12	1.071
AES-CBC-128	3.12	1.013	HMAC_DRBG SHA-1 (NOPR)	181.58	1.039
AES-CBC-192	3.44	1.012	HMAC_DRBG SHA-1 (PR)	200.18	1.040
AES-CBC-256	3.95	1.013	HMAC_DRBG SHA-256 (NOPR)	153.25	1.034
AES-GCM-128	110.58	1.087	HMAC_DRBG SHA-256 (PR)	153.25	1.034
AES-GCM-192	111.97	1.085			
Min (×)					1.009
Max (×)					1.145
Geomean (×)					1.055

Table 5.6: Randezvous MbedTLS-Benchmark Latency (Lower is Better)

Tables 5.9 and 5.10 and Figure 5.3 show Randezvous's code and data size overhead on BEEBS, CoreMark-Pro, MbedTLS-Benchmark, and the two applications, respectively. Overall, Randezvous incurs moderate overhead on both code and data sizes: a geometric mean of 15.8% on code size and 21.2% on data size in BEEBS, 14.2% and 27.5% in CoreMark-Pro, 10.8% and 11.9% in MbedTLS-Benchmark, and 13.6% and 24.5% in the applications. We note that `parser-125k` in CoreMark-Pro exhibits the highest data size overhead because its shadow stack is more than twice the size of its original global data size to accommodate a function that calls itself over 2,000 times. Correspondingly, its stack usage decreases as none of its recursive stack frames

Cryptographic Algorithm	Baseline	Randezvous (×)	Cryptographic Algorithm	Baseline	Randezvous (×)
MD5 (KB/s)	14,630.25	0.981	HMAC.. SHA-1 (NOPR) (KB/s)	1,339.78	0.962
SHA-1 (KB/s)	56,491.74	0.958	HMAC.. SHA-1 (PR) (KB/s)	1,215.91	0.961
SHA-256 (KB/s)	58,746.89	0.956	HMAC.. SHA-256 (NOPR) (KB/s)	1,585.92	0.967
SHA-512 (KB/s)	2,216.21	0.978	HMAC.. SHA-256 (PR) (KB/s)	1,585.94	0.967
3DES (KB/s)	816.31	0.874	RSA-1024 (public/s)	1,420.58	0.987
DES (KB/s)	2,067.83	0.873	RSA-1024 (private/s)	14.92	0.975
AES-CBC-128 (KB/s)	61,610.23	0.950	DHE-2048 (handshake/s)	0.94	0.979
AES-CBC-192 (KB/s)	56,954.36	0.954	DH-2048 (handshake/s)	1.18	0.975
AES-CBC-256 (KB/s)	50,861.98	0.958	ECDSA-secp521r1 (sign/s)	7.69	0.986
AES-GCM-128 (KB/s)	2,192.97	0.919	ECDSA-secp384r1 (sign/s)	13.05	0.968
AES-GCM-192 (KB/s)	2,165.85	0.921	ECDSA-secp256r1 (sign/s)	28.39	0.960
AES-GCM-256 (KB/s)	2,139.37	0.922	ECDSA-secp521r1 (verify/s)	5.66	1.011
AES-CCM-128 (KB/s)	3,319.83	0.919	ECDSA-secp384r1 (verify/s)	12.21	0.965
AES-CCM-192 (KB/s)	3,198.46	0.924	ECDSA-secp256r1 (verify/s)	27.09	0.957
AES-CCM-256 (KB/s)	3,085.67	0.934	ECDHE-secp521r1 (handshake/s)	4.46	0.991
CTR_DRBG (NOPR) (KB/s)	8,699.62	0.913	ECDHE-secp384r1 (handshake/s)	7.52	0.975
CTR_DRBG (PR) (KB/s)	5,092.05	0.932	ECDHE-secp256r1 (handshake/s)	16.44	0.970
			ECDH-secp521r1 (handshake/s)	8.62	0.991
			ECDH-secp384r1 (handshake/s)	14.71	0.975
Min (×)					0.873
Max (×)					1.011
Geomean (×)					0.955

Table 5.7: Randezvous MbedTLS-Benchmark Throughput (Higher is Better)

Application	Baseline (ms)	Stdev (ms)	Randezvous (×)	Stdev (×)
PinLock	46,429.5	108.8	1.009	0.001
FatFs-SD	14,965.3	47.6	1.003	0.003
Geomean	—	—	1.006	—

Table 5.8: Randezvous Application Execution Time (Lower is Better)

contains a return address slot.

Breaking down the overhead, the code size overhead comes from PicoXOM (3.2%–5.5%), function prologue/epilogue transformations (4.6%–7.0%), and runtime components that set up the shadow stack and a single global guard (1,356 bytes). The data size overhead comes from string literals used in additional code (1,389 bytes), a diversified shadow stack (48–19,040 bytes), and promoted local variables containing function pointers (0–5.0%).

Benchmark	Baseline Code (bytes)	Baseline Data (bytes)	Randezvous Code (×)	Randezvous Data (×)
aha-compress	60,834	4,867	1.160	1.295
aha-mont64	61,914	4,652	1.158	1.309
bubblesort	60,616	5,453	1.161	1.264
crc32	60,474	5,672	1.161	1.253
ctl-stack	60,962	12,851	1.160	1.112
ctl-string	61,352	13,188	1.162	1.109
ctl-vector	61,518	12,855	1.157	1.112
cubic	62,864	4,619	1.161	1.318
dijkstra	60,906	14,139	1.161	1.102
dtoa	67,260	15,745	1.148	1.091
edn	61,476	7,852	1.160	1.183
fasta	60,710	5,063	1.162	1.284
fdct	61,070	5,027	1.160	1.286
fir	60,544	7,565	1.161	1.190
frac	61,030	4,723	1.160	1.304
huffbench	61,700	12,860	1.159	1.112
jfdctint	61,090	5,159	1.160	1.279
levenshtein	60,808	4,709	1.161	1.305
ludcmp	61,156	5,321	1.159	1.270
matmult-float	60,784	6,252	1.161	1.230
matmult-int	60,730	11,050	1.161	1.130
mergesort	61,476	6,288	1.162	1.229
miniz	75,260	19,584	1.133	1.079
nbody	60,952	5,284	1.160	1.272
ndes	61,648	6,158	1.158	1.233
nettle-aes	62,338	15,899	1.157	1.090
nettle-arcfour	60,654	5,023	1.162	1.286
nettle-cast128	64,616	8,882	1.152	1.162
nettle-sha256	63,392	5,060	1.156	1.284
newlib-sqrt	60,932	4,704	1.161	1.305
ns	60,520	9,641	1.161	1.149
nsichneu	71,336	4,715	1.137	1.305
picojpeg	67,436	7,698	1.149	1.187
prime	60,636	4,656	1.160	1.309
qrduino	68,156	14,446	1.145	1.099
rijndael	67,854	9,912	1.144	1.145
sglib-arraybinsearch	60,508	5,063	1.161	1.284
sglib-arrayheapsort	60,628	5,433	1.161	1.264
sglib-arrayquicksort	60,724	5,434	1.160	1.264
sglib-dllist	61,172	13,226	1.160	1.110
sglib-hashtable	60,844	13,309	1.162	1.110
sglib-listinsertsort	60,748	13,259	1.161	1.110
sglib-listsort	60,866	13,253	1.160	1.108
sglib-queue	60,682	5,050	1.160	1.285
sglib-rbtree	61,386	13,251	1.160	1.162
slre	63,144	5,019	1.158	1.293
sqrt	60,642	4,643	1.161	1.309
st	61,266	6,269	1.161	1.229
stb_perlin	61,240	7,328	1.160	1.196
stringsearch1	61,354	9,385	1.159	1.153
trio-snprintf	64,246	4,992	1.157	1.295
trio-sscanf	64,842	5,578	1.155	1.264
whetstone	62,652	4,743	1.159	1.306
wikisort	64,868	7,862	1.155	1.183
Min	60,474	4,619	1.133	1.079
Max	75,260	19,584	1.162	1.318
Geomean	—	—	1.158	1.212

Table 5.9: Randezvous BEEBS Memory Usage (Lower is Better)

Benchmark	Baseline Code (bytes)	Baseline Data (bytes)	Randezvous Code (×)	Randezvous Data (×)
cjpeg-rose7-preset	104,978	51,802	1.135	1.039
core	72,852	8,451	1.151	1.184
linear_alg-mid-100x100-sp	75,046	8,839	1.148	1.176
loops-all-mid-10k-sp	84,440	12,624	1.146	1.125
nnet_test	75,218	48,734	1.147	1.033
parser-125k	80,808	7,266	1.137	3.855
radix2-big-64k	74,196	1,383,731	1.149	1.001
sha-test	77,368	5,887	1.139	1.264
zip-test	91,910	20,347	1.128	1.084
Min	72,852	5,887	1.128	1.001
Max	104,978	1,383,731	1.151	3.855
Geomean	—	—	1.142	1.275

Table 5.10: Randezvous CoreMark-Pro Memory Usage (Lower is Better)

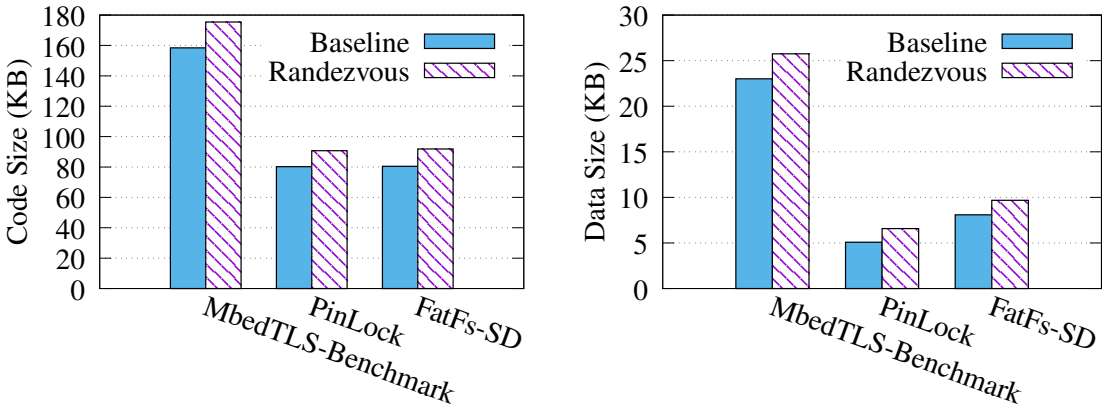

Figure 5.3: Randezvous MbedTLS-Benchmark and Application Memory Usage (Lower is Better)

5.7 Related Work

5.7.1 Randomization on General-Purpose Systems

Randomization on general-purpose systems is well studied. The original ASLR [30, 201] loads memory sections at random addresses and is widely deployed. Due to its coarse granularity and lack of entropy on 32-bit systems, researchers have focused on fine-grained code randomization at the level of pages [26], functions [31, 65, 108, 134], basic blocks [137, 263], instructions [83, 119, 196], register allocation [65, 196], ex-

ecution paths [82], or tunable sizes [206]. Fine-grained data randomization has been explored as well, including global data object reordering [31], data representation encryption [29, 44], structure field randomization [48, 66, 108, 150], stack randomization [5, 31, 50, 145], and heap randomization [28, 188]. While most of these techniques can be used on MCUs, Randezvous leverages just a few of them with the best efficacy and the least performance impact.

Leakage-resistant randomization for general-purpose systems, such as Readactor [65], ASLR-Guard [167], LR^2 [38], and kRˆX [205], hide code pointers via indirection or encryption. These systems are still susceptible to control data leakage; despite not knowing *where* code is located, attackers can identify indirect or encrypted code pointers from disclosed memory and reuse them to corrupt control data slots. Randezvous's decoy pointers, in contrast, prevent attackers from identifying real code pointers from decoy pointers; using a leaked pointer risks causing a trap.

Runtime rerandomization shortens the window for successful exploitation and can be done manually at runtime [53], periodically [6, 100, 108, 209, 266], at certain system calls [32, 168, 259], and when detecting suspicious probes [260]. Randezvous uses no runtime rerandomization as its additional resource consumption outweighs its security gain (as Section 5.3.1 describes).

5.7.2 Randomization on MCUs

Previous work has employed randomization for MCUs. μArmor [3] and EPOXY [59] employ compile-time code layout randomization; EPOXY [59] also randomizes data layout at compile time. AVRAND [199] and MAVR [116] proposed boot-time code layout randomization for AVR MCUs. Both solutions randomize code and reprogram the flash memory at every reboot, using a trusted bootloader reading metadata from EEPROM or a separate processor with extra flash memory. Compared to Randezvous, all of the above systems assume a weaker threat model and, there-

fore, do not mitigate information leakage. Consequently, attackers can still locate code and launch code reuse attacks on these systems using information leaked from code [230] or data [65, 82, 204, 214]. It is also unclear if these systems can resist brute force attacks effectively; they omitted modeling such attacks [3, 59, 199] or yielded an outrageously large number of guesses by incorrectly assuming that attackers have to guess the locations of all functions in the program before launching an attack [116]. HARM [229] implements function-level periodical code rerandomization using TrustZone-M on ARMv8-M [23], requiring more than twice the memory. fASLR [169] uses TrustZone-M to dynamically load functions to random addresses in RAM when being called and unload finished ones when out of RAM, thus reducing memory usage of rerandomization. Unlike HARM and fASLR, Randezvous requires no TrustZone-M and thus supports ARMv7-M systems. While runtime rerandomization reduces the window of code reuse attacks, a successful exploit equipping memory disclosure to learn the code layout is still possible, especially where rerandomization may not take place frequently (e.g., fASLR [169]).

As to performance, AVRAND [199] and MAVR [116] only present startup overhead in absolute numbers; comparing to Randezvous is impossible. EPOXY shows better performance in BEEBS (1.6% on average) than Randezvous because its safe stack [140] improves locality. For BEEBS programs that both Randezvous and HARM [229] evaluate (all 19 programs by HARM), Randezvous outperforms HARM (8.8% vs. 25%, on average). Similarly, in BEEBS programs shared between Randezvous and fASLR [169] (5 programs out of 9 by fASLR), Randezvous is slightly faster (2.3% vs. 3.7%, on average).

5.7.3 CFI on MCUs

An alternative to randomization is to use CFI [2] and/or protected shadow stacks [43]. To protect shadow stacks, CaRE [191] and TZmCFI [130] leverage TrustZone-M [23],

RECFISH [255] utilizes privilege mode switching, and Silhouette [282] and Kage [90] utilize ARM's unprivileged store instructions. μRAI [9] encodes return addresses in a reserved register and uses system calls to extend the encoding space. All these solutions enforce return address integrity and use coarse-grained forward-edge CFI [2], while SCFP [264] extends a RISC-V MCU with a stateful instruction encryption scheme for fine-grained CFI. However, even with a fully precise static CFG and a protected shadow stack, CFI is still vulnerable to advanced forward-edge corruptions that adhere to the CFG [46, 96]. In contrast, Randezvous provides probabilistic guarantees but is not susceptible to such attacks without identifying both a control flow target and a control data slot.

Performance-wise, Randezvous outperforms all the above CFI implementations except Silhouette and Kage. We believe Silhouette's low overhead (3.4% on BEEBS and 1.3% on CoreMark-Pro) is due to the high latency of the SDRAM in its STM32F469 Discovery board [282]; we evaluated Silhouette on our NXP MIMXRT685-EVK board (which uses SRAM), and its overhead increases to 12.1% on BEEBS and 11.2% on CoreMark-Pro.

Chapter 6
Efficient Control-Flow Protection for AArch64 Applications

6.1 Introduction

AArch64 (64-bit ARM) processors are becoming increasingly popular, not only in embedded and mobile platforms but also in personal computers [14] and high-performance servers and data centers [11, 111, 176, 194]. Given the popularity of AArch64 processors used in production and in our daily lives, securing software on such systems is critical. In particular, a large portion of AArch64 application code is written in memory-unsafe programming languages (e.g., C and C++) and is vulnerable to control-flow hijacking attacks [213, 248] that exploit memory safety errors. While basic code injection attacks are prevented by the wide deployment of the W$\oplus$X [200] policy, which disallows memory to be writable and executable at the same time, advanced code-reuse attacks like return-oriented programming (ROP) [213, 223] and jump-oriented programming (JOP) [35] are still possible. These attacks hijack a program's control flow by corrupting code pointers (e.g., return addresses and function pointers) to point to reusable code of the attacker's choosing. Worse yet, recent research [61] has demonstrated automation of ROP attacks on AArch64, necessitating effective and practical defenses to be deployed.

Control-flow integrity (CFI) [1, 2], a seminal mitigation to control-flow hijacking attacks, restricts a program's control flow to follow its intended control-flow graph. While ineffective by itself [45, 62, 80, 109], CFI necessitates a mechanism that protects the integrity of return addresses, such as write-protected shadow stacks [43, 55], to form an effective defense [46]. However, software approaches to protecting return address integrity either suffer from high performance overhead (e.g., software-based shadow stacks [55, 63, 98, 255, 282]) or only provide probabilistic guarantees (e.g., information hiding [43, 202, 228, 284]). Hardware-assisted shadow stack protection, such as Control-flow Enforcement Technology (CET) [225] on x86, offers the best security and performance but is not natively available on AArch64.

In this chapter, we present *InversOS*, a system that provides AArch64 user-space applications with hardware-assisted write-protected shadow stacks. InversOS does so without requiring the most recent hardware security features on AArch64 or modifying hardware. Instead, InversOS uses two widely available AArch64 features [22], namely *unprivileged load/store instructions* and *Privileged Access Never*, in a novel way to create an efficient domain-based instruction-level intra-address space isolation technique which we call *Privilege Inversion*. With Privilege Inversion, InversOS runs protected applications in the same privilege mode as an operating system (OS) kernel, sets up incorruptible shadow stack memory accessible only by unprivileged load/store instructions, and ensures the safety of running privileged user-space code via a combination of OS kernel modifications and compiler transformations. To keep compatibility with legacy untransformed application binaries, InversOS repurposes another AArch64 feature to support coexistence of legacy and protected applications securely and efficiently.

We built a prototype implementation of InversOS based on the Linux kernel v4.19.219 [152] and the LLVM/Clang compiler v13.0.1 [142]. We analyzed the security of InversOS and assessed the strength of its defense against different types of control-flow hijacking attacks. Our evaluation of InversOS on a real AArch64 system and a comprehensive set of benchmarks and applications (LMBench [172], SPEC CPU

2017 [232], and Nginx [244]) shows low performance overhead (7.0% on LMBench, 7.1% on SPEC CPU 2017, and 3.0% on Nginx), indicating that InversOS is practical for deployment. We open-sourced InversOS at https://github.com/URSec/InversOS.

To summarize, we make the following contributions:

- We present Privilege Inversion, the first domain-based intra-address space isolation technique for AArch64 user-space applications, using only widely available features on commodity hardware.
- We designed and implemented InversOS, an OS-kernel-compiler co-design that provides the first hardware-assisted protected shadow stacks on AArch64 utilizing Privilege Inversion and is compatible with existing binaries.
- We evaluated the security and performance of InversOS and showed that InversOS is both efficacious and efficient.

The rest of the chapter is organized as follows. Section 6.2 provides background information on protected shadow stacks. Section 6.3 defines our threat model. Sections 6.4 and 6.5 describe the design and implementation of InversOS, respectively. Section 6.6 analyzes the security of InversOS, Section 6.7 presents the performance evaluation of InversOS, and Section 6.8 discusses related work.

6.2 Protected Shadow Stacks

Control-flow hijacking attacks like ROP [213, 223] corrupt saved return addresses on the stack. One way to mitigate such attacks is to use shadow stacks [43], which keep copies of return addresses in separate memory regions. When calling a function, a return address is pushed onto both the regular stack and the shadow stack; on return, the program loads the return address from the shadow stack and either compares it

to the one on the regular stack to ensure its validity [55, 77, 78] or jumps to the value loaded from the shadow stack directly [2, 112, 228, 282, 284]. To enforce return address integrity, however, shadow stacks themselves require protection that disallows illegal modifications. Prior approaches to protecting shadow stack integrity rely on system calls [55, 98, 255], software fault isolation (SFI) [63, 282], information hiding [43, 202, 228, 284], or special hardware such as segmentation [2], Memory Protection Extensions (MPX) [43, 122, 128], Memory Protection Keys (MPK) [43, 112], and CET [225]). To the best of our knowledge, no hardware-assisted shadow stack protection exists on AArch64.

6.3 Threat Model

We assume a powerful attacker trying to achieve arbitrary code execution on a benign but potentially buggy application by exploiting arbitrary memory read/write vulnerabilities to hijack the control flow. We assume that the underlying OS kernel and hardware are trusted and unexploitable, providing the user space with the basic W$\oplus$X protection [200]. Non-control data attacks [49] (such as data-oriented programming [120] and block-oriented programming [125]), side-channel attacks, and physical attacks are out of scope. This threat model is in line with recent work on user-space control-flow hijacking attacks [61, 62] and defenses [43, 146, 149, 258].

6.4 Design

In this section, we present the design of InversOS. The goal of InversOS is to provide low-cost return address integrity to user-space applications running on commodity AArch64 systems, which may or may not come with the most recent hardware security features such as Pointer Authentication (PAuth), Branch Target Identification (BTI), and Memory Tagging Extension (MTE) [22]. To do so, InversOS must only rely on

AArch64 features from the early ISA versions. We therefore require InversOS's target platform to support at least PAN and HPDS (i.e., conforming to ARMv8.1-A [22]); this allows InversOS to be deployed on most of AArch64 systems released since 2017 [265].

Overall, we devise InversOS as a co-design between an OS kernel and a compiler. The InversOS-compliant OS kernel utilizes *Privilege Inversion*, a novel intra-address space isolation technique we invented, to provide user-space applications an extra protection domain accessible only by LSU instructions. The InversOS-compliant compiler then instruments user-space code to leverage the protection domain for efficient protected shadow stacks as well as to enforce forward-edge CFI [1, 2], allowing InversOS to protect user-space applications without modifying their source code. The nature of Privilege Inversion dictates running user-space applications in the privileged mode; we therefore combine CFI, a compile-time bit-masking compiler pass, a load-time code scanner in the OS kernel, and a set of kernel modifications to together ensure the safety and security of doing so. Lastly, InversOS supports running legacy untransformed applications to keep compatibility with existing binaries via a novel use of HPDS or E0PD (if available).

6.4.1 Privilege Inversion

LSU instructions in AArch64, as described in Section 2.2.3, show a great potential in implementing efficient intra-address space isolation; previous work [56] has explored their usage in kernel-level data isolation. However, using these instructions to compartmentalize user-space applications poses challenges as they act like regular loads/stores when executed in the unprivileged mode. Essentially the underlying hardware only supports one protection domain for unprivileged software.

We devise Privilege Inversion, a novel intra-address space isolation technique that creates a separate protection domain for AArch64 user-space applications. With Privilege Inversion, the OS kernel runs a user-space application needing an extra protection

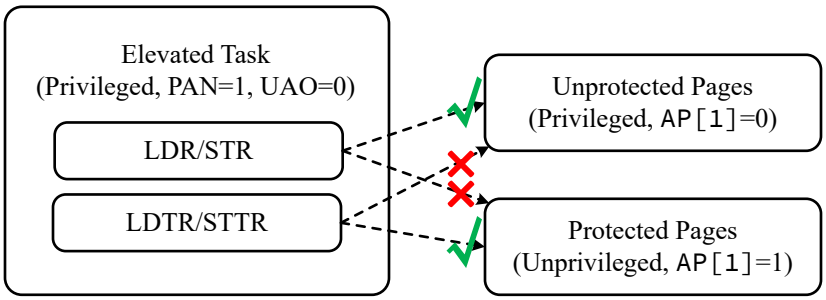

Figure 6.1: Compartmentalization by Privilege Inversion

domain in the privileged mode. We dub such an application as an *elevated task*. When launching an elevated task, the OS kernel configures its memory pages as unprivileged-inaccessible (i.e., with `AP[1]` cleared in PTEs), marks its code pages as privileged-executable (i.e., with `PXN` cleared and `UXN` set in PTEs), and enables PAN during its execution. Then, pages that the elevated task wants to place in the separate protection domain are marked as unprivileged-accessible (i.e., with `AP[1]` set in PTEs). Note that the elevated task's pages are still mapped to the user space (translated by `TTBR0_EL1`); the above changes only apply to their access permission bits in the PTEs. This configuration allows LSU instructions in elevated task code to access the protected pages but forbids accesses to them made by all regular loads/stores due to PAN. In the meanwhile, it leaves all other unprotected pages in the elevated task accessible by regular loads/stores but inaccessible by LSU instructions, effectively compartmentalizing the elevated task into two separate protection domains (one for regular loads/stores and the other for LSU instructions), as Figure 6.1 shows. Note that in systems with UAO support, UAO has to be turned off during elevated task execution; otherwise LSU instructions would act just like regular loads/stores.

However, in order to make Privilege Inversion safe and useful, we need to address the following challenges:

Challenge 6.1. *As elevated tasks run in the privileged mode, kernel memory becomes accessible by their regular loads/stores.*

Challenge 6.2. *As elevated tasks run in the privileged mode, their control-flow transfer*

instructions can jump to the kernel space to execute arbitrary kernel code (i.e., kernel memory with `PXN` *cleared).*

Challenge 6.3. *As elevated tasks run in the privileged mode, they may contain and execute special privileged instructions that would only be allowed to execute in kernel code (e.g., instructions that flip* `PSTATE.PAN`*).*

To address Challenge 6.1, we incorporate a set of kernel modifications that mark all kernel memory as unprivileged-accessible and disable PAN during kernel execution. Such modifications, while radical in idea, effectively stop regular loads/stores in elevated tasks from accessing kernel memory and still keep the OS kernel functional. The ramifications of modifying the OS kernel in this way are two folds. First, LSU instructions in elevated tasks can now access kernel memory. We therefore require that elevated tasks not contain LSU instructions by themselves (which is the case in C/C++ code compiled by GCC or LLVM/Clang) and use a compiler pass to insert vetted LSU instructions for enforcing the desired protection policies. Our shadow stack pass described in Section 6.4.2 provides a good example. Second, if we are to support running legacy untransformed applications in the unprivileged mode still, they can access kernel memory as well; Section 6.4.3 discusses how we tackle this problem.

To address Challenge 6.2, we use a bit-masking compiler pass, which instruments all indirect control-flow transfer instructions (i.e., indirect calls, indirect jumps, and returns) in elevated tasks by preceding them with a bit-masking instruction that clears the top bit of the target register.[8] This limits the control-flow transfer target to be within the user space or to become an invalid pointer pointing to the user-kernel space gap. Such instrumentation alone, however, can be bypassed by attacker-manipulated control flow that jumps over the bit-masking instruction; we therefore combine it with CFI to ensure its execution, which we discuss in Section 6.4.2. Note that direct control-flow

[8] AArch64 returns via the `RET` instruction, which uses the link register `LR` (by default) or another explicitly specified register as the return address [22].

transfer instructions (i.e., direct calls and jumps) do not need such instrumentation; their target is PC-relative and always points to a known location within the user space.

To address Challenge 6.3, we add to the OS kernel a load-time code scanner which scans for privileged instructions that unprivileged software should never execute. Whenever a page in an elevated task is being marked as executable, the OS kernel invokes our code scanner to scan the whole page; if the page contains any forbidden privileged instruction, the execution permission of the whole page is denied. As AArch64 instructions are 4-byte sized and aligned [22], a linear non-overlapping scan should suffice.

6.4.2 Protected Shadow Stacks and Forward-Edge CFI

With Privilege Inversion creating an extra protection domain, we can now leverage the protection domain to enforce efficient shadow stack protection for the user space. Specifically, the OS kernel allocates unprivileged memory for a shadow stack when a new elevated task is launched via `exec()` or when a new thread in an elevated task is created via `clone()`. The compiler utilizes a shadow stack pass to instrument elevated task code; a copy of the return address is saved onto a shadow stack via an `STTR` instruction inserted into the prologue of functions that save the return address to the regular stack, and the return address is loaded from the shadow stack via an `LDTR` instruction inserted into the epilogue(s) of these functions. A special case for shadow stacks to handle is irregular control flow such as `setjmp()`/`longjmp()` in C and exception handling in C++. Since support for such irregular control flow depends on the specific shadow stack scheme used [43], we discuss how our InversOS prototype supports such code constructs in Section 6.5.2.

To form a complete control-flow protection, we couple our shadow stacks with forward-edge CFI [1, 2], which ensures that the target of indirect calls and jumps is within a set of allowed code locations. Specifically, we use a label-based CFI pass in

the compiler. For each indirect call or tail-call indirect jump in elevated task code, the pass inserts a CFI label at the beginning of every function that might be the call target and inserts a CFI check before the call. Similarly, for each non-tail-call indirect jump in elevated task code, the pass inserts a CFI label at the beginning of every successor basic block and inserts a CFI check before the jump. The CFI check ensures that a proper CFI label is present at the control-flow target; otherwise it generates a fault and traps the execution.

6.4.3 Compatibility

Not all AArch64 user-space applications need a separate protection domain, nor can all of them be recompiled. InversOS must therefore allow existing application and library binaries that are not compiled by the InversOS-compliant compiler to run without compromising its security.

We propose two methods to allow safe execution of legacy applications in the unprivileged mode (dubbed as *legacy tasks*), depending on hardware feature availability. In systems with E0PD support (ARMv8.5-A and onward), the OS kernel can directly enable E0PD via setting `TCR_EL1.E0PD1` during legacy task execution. This way, even though kernel memory is marked unprivileged-accessible, legacy tasks running in the unprivileged mode still cannot access kernel memory translated by `TTBR1_EL1`.

In pre-ARMv8.5-A systems without E0PD support, however, we rely on HPDS to provide a less-efficient solution. Specifically, the OS kernel first sets `APTable[0]` in all top- and mid-level PTEs of kernel memory when establishing page tables for the kernel space. This effectively marks all kernel pages as unprivileged-inaccessible even if `AP[1]` in their last-level PTEs is set. Then, the OS kernel enables HPDS via setting `TCR_EL1.HPD1` before running an elevated task, disables HPDS via clearing `TCR_EL1.HPD1` before running a legacy task, and flushes the local TLBs every time after flipping `TCR_EL1.HPD1`. This way, legacy and elevated tasks will possess

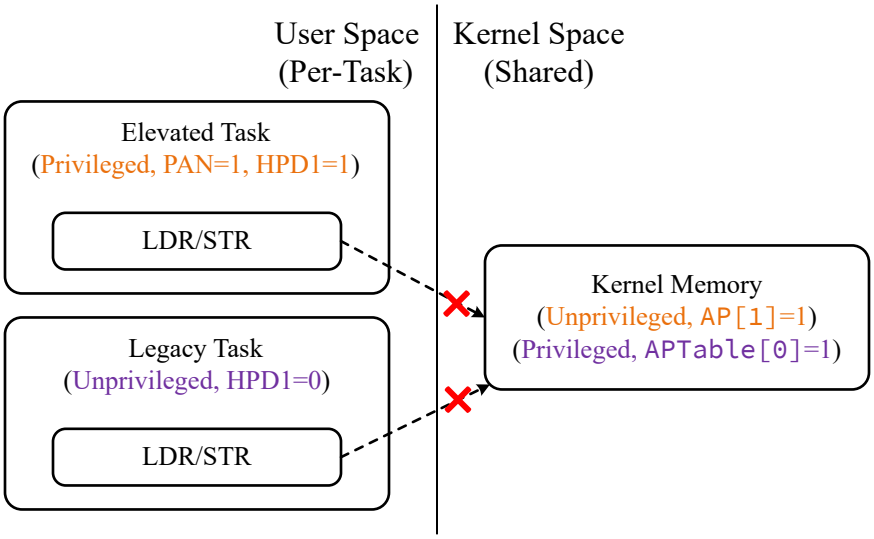

Figure 6.2: Different "Views" of Kernel Memory Due to HPDS

different "views" of kernel memory, as Figure 6.2 depicts. Specifically, legacy tasks see kernel memory as unprivileged-inaccessible due to `APTable[0]` being set, while elevated tasks see kernel memory as unprivileged-accessible because HPDS disables `APTable[0]` in top- and mid-level PTEs and `AP[1]` in last-level PTEs takes effect. As a result, both types of tasks cannot access kernel memory.

Note that relying on HPDS prevents the OS kernel from mapping kernel memory with the largest huge pages on certain systems (e.g., 1 GB huge pages with a page size of 4 KB and a 39-bit virtual address space), because such pages have no top- or mid-level PTEs for setting `APTable[0]`. However, we believe this has no practical impact on the OS kernel's address translation and memory usage; the use of the largest huge pages is rare and infrequent.

6.5 Implementation

We implemented a prototype of InversOS on the Linux kernel v4.19.219 [152] and the LLVM/Clang compiler v13.0.1 [142]. Using Tokei v12.1.2 [269], our kernel modifications include 1,815 lines of C code and 207 lines of assembly code, and our changes

to LLVM contain 1,003 lines of C++ code. To provide complete and transparent InversOS support for user-space applications, we also modified the musl libc v1.2.2 [97] and LLVM's LLD linker [164], compiler-rt builtin runtime library [161], and libunwind [159], totalling 27 lines of C code, 131 lines of C++ code, and 299 lines of assembly code.

6.5.1 OS Kernel Modifications

Privilege Inversion requires running elevated tasks in the privileged mode. As Linux does not use the privileged thread mode (as Section 2.2.1 describes), our prototype therefore utilizes it to run elevated tasks. This way, the Linux kernel can keep using the privileged handler mode for its own operations without interference from elevated tasks. It also greatly simplifies our implementation. To enable the privileged thread mode, our prototype enables an unused set of exception vectors that are responsible for taking exceptions from the privileged thread mode to the privileged handler mode. Changes were also made to Linux's existing AArch64 exception handler code so that our prototype can reuse most of the code to handle exceptions from the privileged thread mode and to resume elevated task execution properly. Note that elevated tasks in our prototype still use the `SVC` instruction for system calls, which is unnecessary because elevated tasks are already privileged; we leave system call optimizations as future work.

Apart from the architectural usage of `AP[1]`, Linux also uses `AP[1]` to distinguish whether a page is kernel or user memory. As InversOS marks kernel memory unprivileged-accessible, `AP[1]` can no longer serve for that purpose. Our prototype therefore utilizes an unused bit (bit 63) in last-level PTEs to differentiate between kernel and user memory; the hardware MMU ignores this bit automatically [22].

When launching a new task, InversOS must decide whether it should be run as a legacy or elevated task. For simplicity and ease of implementation, our prototype checks the presence of an environment variable `INVERSOS=1` to make such a decision;

Instruction	Description
MRS*/MSR*	Read/Write System Register
IC*/DC*	Invalidate Instruction/Data Cache
TLBI	Invalidate Translation Lookaside Buffer
HVC	Hypervisor Call
SMC	Secure Monitor Call
AT	Address Translation
ERET	Exception Return
CFP/CPP/DVP	Prediction Restriction
LDGM/STGM/STZGM	Load/Store Tag Multiple (MTE)
BRB	Branch Record Buffer
SYS/SYSL	Other System Instructions

* Instructions with Certain Operands Allowed

Table 6.1: Forbidden Privileged Instructions by InversOS Code Scanner

if it is present, the task is started as an elevated task. Production systems can use a more enhanced mechanism (e.g., checking the presence of a code signature generated by an InversOS-compliant compiler) to qualify an elevated task.

The load-time code scanner, as part of our kernel modifications, scans for illegal privileged instructions in elevated task code. Instead of directly scanning a user-space code page, our prototype maps the page to the kernel space for scanning in order to avoid frequently calling `get_user()`. Table 6.1 lists all types of privileged instructions that our prototype forbids, which roughly correspond to instructions that would generate a fault when executed in the unprivileged mode but might not when executed in the privileged mode [22]. In particular, `MRS`/`MSR`/`IC`/`DC` instructions with certain operands (e.g., reading the unprivileged thread ID register `TPIDR_EL0` via `MRS`) are allowed in unprivileged software, so these instructions are also permitted in elevated tasks.

Our kernel modifications take responsibility of setting up and tearing down memory for protected shadow stacks in elevated tasks, as Section 6.4.2 describes. Each shadow stack region in an elevated task can grow as much as a regular stack can grow, supporting both parallel and compact shadow stack schemes [43]. To prevent shadow stack overflow and underflow, each shadow stack region is surrounded by two guard

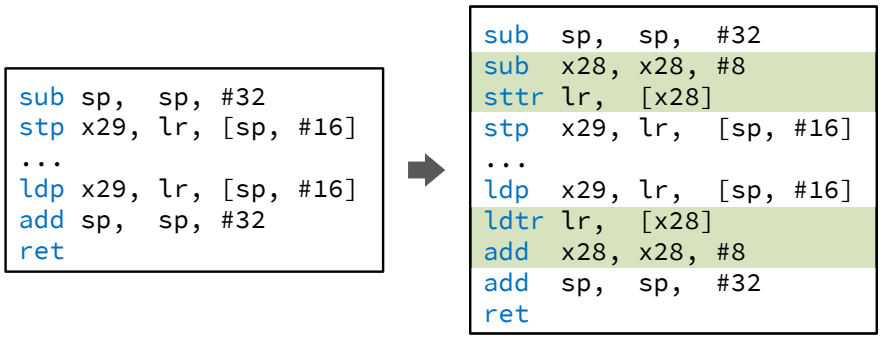

Figure 6.3: InversOS's Shadow Stack Transformations

regions inaccessible by both regular loads/stores and LSU instructions. Mappings of shadow stack and guard regions are unmodifiable by `munmap()`, `mremap()`, and `mprotect()` requests from the user space.

Lastly, our prototype implements the HPDS support for running legacy tasks, as described in Section 6.4.3. We omitted implementing the E0PD alternative due to the lack of hardware that supports E0PD. As Linux has introduced support for E0PD since v5.6 [151] (which is enabled by default), a simple backport of the relevant changes would suffice.

6.5.2 Compiler, Linker, and Library Modifications

We implemented the shadow stack, forward-edge CFI, and bit-masking compiler passes in a single LLVM pass that transforms LLVM machine intermediate representation (IR).

Our shadow stack transformations adopt the compact shadow stack scheme [43] and reserve the `X28` register (a callee-saved register) as the shadow stack pointer register. Figure 6.3 demonstrates our shadow stack transformations performed on a function's prologue and epilogue. Our prototype supports C's `setjmp()`/`longjmp()` functions and C++ exception handling via modifications to the musl libc and LLVM's libunwind, respectively. Instead of directly guaranteeing the integrity of return address saved by `setjmp()` or `__unw_getcontext()`, our prototype provides

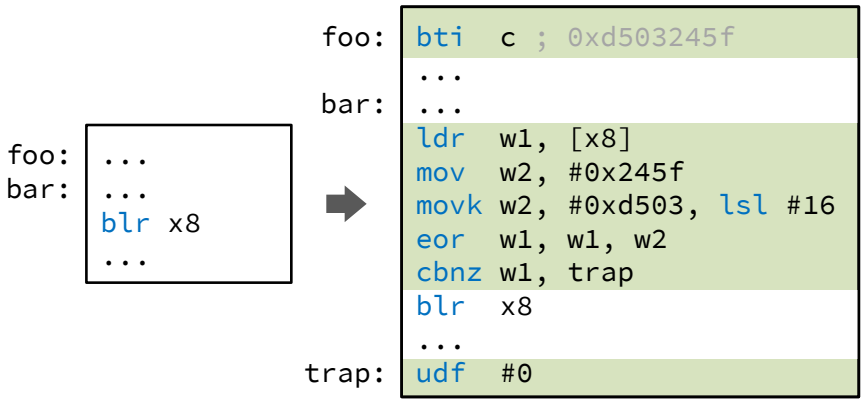

Figure 6.4: InversOS's Forward-Edge CFI Transformations

shadow stack pointer integrity when restoring the saved context in `longjmp()` or `__libunwind_Registers_arm64_jumpto()`. Specifically, rather than overriding `X28` with the saved value, we unwind `X28` step by step until a matched return address is found or it reaches a guard region to cause shadow stack underflow.

Our forward-edge CFI transformations use the `BTI` instructions as CFI labels to keep forward compatibility with ARMv8.5-A's BTI [22], a hardware-assisted forward-edge CFI mechanism rolling out to new AArch64 processors. Processors not supporting BTI execute a `BTI` instruction as a no-operation. An appropriate CFI check is inserted before every indirect call or jump to ensure that the target contains a correct CFI label (`BTI C` for indirect calls and tail-call indirect jumps and `BTI J` for non-tail-call indirect jumps). Figure 6.4 illustrates our forward-edge CFI transformations performed on an indirect call and one of its target functions. On AArch64, a non-tail-call indirect jump can only be generated from a `switch` or computed `goto` statement; the former is bounds-checked against a read-only jump table, and our prototype restricts the latter by transforming it to a `switch` statement using the `IndirectBrExpandPass` [163]. Consequently, a non-tail-call indirect jump is limited to jump within its function and cannot branch to other functions.

Our bit-masking transformation inserts an `AND` instruction before every indirect call, indirect jump, or return to clear the top bit of control-flow transfer target. For

indirect calls and jumps, the instruction is placed after the CFI check.

While our all-in-one LLVM machine IR pass transforms most of elevated task code, it fails to cover certain pieces of code in the user space when compiling the application. One piece of untransformed code is the procedure linkage table (PLT) generated by the linker. We therefore also modified LLD to be able to generate CFI-checked and bit-masked PLT code. Another piece of untransformed code is Linux's virtual dynamic shared object (vDSO); it is compiled with the Linux kernel and stored within the kernel's read-only data. We therefore applied our compiler transformations to the vDSO as well during kernel compilation. The last case is assembly code (including assembly files and inline assembly statements). We manually instrumented assembly code in the musl libc and compiler-rt builtin runtime library.

6.5.3 Discussion

Virtualization Host Support ARMv8.1-A adds Virtualization Host Extensions (VHE) [22] to accelerate hosted (Type 2) hypervisors such as Linux's KVM [76] and FreeBSD's bhyve [93]. In pre-VHE systems, a host OS kernel (running in EL1) needs to partition its hypervisor into a "high-visor" (running in EL1) and a "low-visor" (running in EL2) and thus incurs heavy overhead when context-switching between the two parts. VHE allows the host OS kernel to run entirely in EL2 to reduce the cost. The Linux kernel, as of v4.19.219 [152], stays in EL2h for execution when having detected VHE support during early boot. Our prototype therefore transparently supports running elevated tasks in EL2t in such a case.

AArch32 Support Quite a few AArch64 processors still allow running AArch32 (32-bit ARM) applications for compatibility. While there are no technical difficulties to support an elevated task running in the AArch32 state (i.e., LSU instructions and PAN are also available on AArch32), we opted not to implement AArch32 support for the sake of time.

6.6 Security Analysis

In this section, we analyze the security of InversOS by providing answers to the following security questions:

SQ1 Why is InversOS secure (to run *instrumented* elevated tasks in the privileged mode and *arbitrary* legacy tasks in the unprivileged mode)?

SQ2 How well does InversOS mitigate control-flow hijacking attacks on elevated tasks?

6.6.1 Security by Design

To answer **SQ1**, we examine *all* potential ways to compromise InversOS from a legacy or elevated task:

1. A task may try to read from/write to memory of other tasks to break their confidentiality/integrity.
2. A task may try to read from/write to kernel memory to break the confidentiality/integrity of the OS kernel.
3. A task may try to allocate an excessive amount of resources (e.g., time, memory) to break the availability of InversOS.
4. A task may try to execute detrimental instructions that could undermine the security of InversOS.
5. A task may try to jump to kernel code and use kernel code as a "confused deputy" for the above goals.

As each task's memory (sans shared memory) is mapped exclusively to the task's own address space, reading and writing other tasks' memory can only be carried out by accessing kernel memory or jumping to kernel code. Since kernel memory has `AP[1]`

(and `APTable[0]`, if using HPDS) set, accessing kernel memory is disabled via PAN for elevated tasks and via HPDS or E0PD for legacy tasks. Jumping to kernel code is also impossible; having `UXN` set for kernel code prevents legacy tasks from executing kernel code, while InversOS's CFI and bit-masking instrumentation ensures that control-flow transfers in elevated tasks never reach the kernel space. As for attacks on availability, we argue that InversOS does not introduce new availability problems; running an elevated task in the privileged mode does not prioritize it on resource allocation over all other legacy or elevated tasks and the OS kernel. The remaining case is privileged instructions, the execution of which is restricted by hardware automatically for legacy tasks and by InversOS's load-time code scanner for elevated tasks. Conclusively, InversOS does not introduce new security flaws and is secure by design.

6.6.2 Efficacy against Control-Flow Hijacking

To answer **SQ2**, we first define and explain a list of invariants that InversOS maintains for guaranteeing return address integrity of elevated tasks and then reason about why return address integrity significantly reduces the control-flow hijacking attack surface. Specifically, InversOS maintains the following invariants for elevated tasks:

Invariant 6.1. *A function in an elevated task either pushes its return address in* `LR` *to a shadow stack, or never spills the return address to memory.*

Invariant 6.2. *If a function in an elevated task pushed its return address to a shadow stack, its epilogue will always load the return address from the shadow stack location in which its prologue saved the return address.*

Invariant 6.3. *An elevated task cannot corrupt shadow stacks by itself or by using a system call as a "confused deputy" (e.g., calling* `read(fd, buf, size)` *where* `buf` *points to shadow stack memory [261]).*

Invariant 6.1 is easily upheld by our shadow stack pass, which instruments `LR`-saving function prologues to push `LR` to the shadow stack. With the counterpart in-

strumentation on epilogue(s) of these functions to pop `LR` from the shadow stack, our shadow stack pass guarantees that only a function's prologue and epilogue(s) can update the shadow stack pointer with a matched decrement/increment, contributing to Invariant 6.2. Since our forward-edge CFI pass ensures that all indirect calls and tail-call indirect jumps target the beginning of a function and all non-tail-call indirect jumps are restricted within their containing function, shadow stack pointer decrements and increments are guaranteed to occur in a matched order, sustaining Invariant 6.2. Finally, Invariant 6.3 is maintained because the shadow stacks are unprivileged and no existing/new LSU instructions can be exploited/introduced to corrupt the shadow stacks (due to CFI/W$\oplus$X), and because of the benign nature of elevated tasks assumed by our threat model in Section 6.3.

With return address integrity, control-flow hijacking attacks that require corrupting return addresses (such as return-into-libc [248] and ROP [213, 223]) are effectively prevented. Furthermore, as non-tail-call indirect jumps cannot break the "jail" of their containing function, attacks that exploit indirect jumps (such as JOP [35]) no longer work. The remaining attack surface requires attackers to do purely *call-oriented programming* (i.e., using only corrupted function pointers); while such attacks are possible [96, 219], they are limited by forward-edge CFI and can be further restrained if InversOS refines CFI's granularity. In short, InversOS greatly reduces the control-flow hijacking attack surface for elevated tasks.

6.7 Performance Evaluation

We evaluated the performance of InversOS on a Station P2 mini-PC which has an RK3568 quad-core Cortex-A55 processor implementing the ARMv8.2-A architecture that can run up to 2.0 GHz. The mini-PC comes with 8 GB of LPDDR4 DRAM up to 1,600 MHz, 64 GB of internal eMMC storage (unused), and 1 TB of SATA SSD. It runs Ubuntu 20.04 LTS modified by the manufacturer.

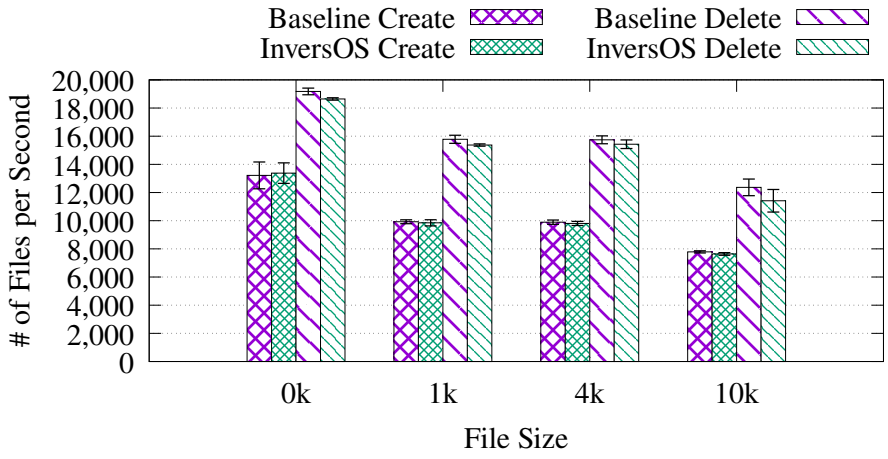

Figure 6.5: InversOS LMBench File Operation Rate (Higher is Better)

We ran all our experiments using two configurations: Baseline and InversOS. In Baseline, we compiled program and library code using LLVM/Clang v13.0.1 [142] without the InversOS compiler transformations and ran the generated binary executables on a Linux v4.19.219 kernel [152] without our kernel modifications. In InversOS, all program and library code was compiled with the InversOS compiler transformations (i.e., shadow stack, forward-edge CFI, and bit-masking transformations) and executed on the same version of the Linux kernel modified with our kernel changes. When running an InversOS executable, we set an environment variable `INVERSOS=1` to inform the OS kernel that the program should be started as an elevated task, as Section 6.5.1 describes. As the processor lacks E0PD support, we rely on HPDS to prevent legacy tasks from accessing kernel memory. Both configurations used `-O2` optimizations and performed static linking against the musl libc v1.2.2 [97] and LLVM's compiler-rt builtin runtime library v13.0.1 [161]. C++ code in our experiments was compiled with and statically linked against libc++ [157], libc++abi [158], and libunwind [159] from LLVM v13.0.1. Libraries for Baseline and InversOS are compiled without and with our modifications described in Section 6.5.2, respectively.

Microbenchmark	**Baseline** (μs)	**stdev** (μs)	**InversOS** ($\times$)	**stdev** ($\times$)
null syscall	0.148	0.000	1.047	0.007
read	0.482	0.001	1.054	0.004
write	0.351	0.002	0.991	0.003
stat	4.928	0.023	1.066	0.003
fstat	0.422	0.003	1.052	0.005
open/close	9.744	0.017	0.989	0.003
select 500 fd	24.365	0.017	1.002	0.001
signal install	0.375	0.001	1.059	0.003
signal catch	3.801	0.009	1.493	0.002
protection fault	0.408	0.005	0.980	0.029
pipe	16.115	0.067	0.948	0.004
AF_UNIX stream	27.314	0.618	1.051	0.008
AF_UNIX connect	99.329	0.733	1.012	0.009
fork+exit	266.767	6.945	1.256	0.012
fork+exec	562.585	7.046	1.188	0.009
fork+shell	2,878.983	12.869	4.007	0.015
page fault	0.910	0.016	1.038	0.009
mmap 1 MB	42.700	3.318	1.019	0.007
udp	76.490	0.214	1.018	0.005
tcp	63.472	0.200	1.011	0.002
connect	102.196	0.503	1.004	0.006
context switch	59.318	0.880	0.993	0.014
fcntl	8.772	1.643	0.992	0.219
semaphore	3.083	0.515	0.954	0.162
usleep	78.661	1.579	0.995	0.020
Geomean	—	—	1.103	—

Table 6.2: InversOS LMBench Latency (Lower is Better)

6.7.1 Microbenchmarks

To understand the performance impact of the InversOS Linux kernel modifications, we used LMBench v3.0-alpha9 [172], a microbenchmark suite that measures the latency and bandwidth of various OS services. For each microbenchmark that supports parallelism, we ran four parallel workloads to reduce variance. We report an average and a standard deviation of 10 rounds of execution for each microbenchmark.

Tables 6.2 and 6.3 and Figure 6.5 show LMBench performance of both Baseline and InversOS. Overall, InversOS incurred a geometric mean of 7.0% overhead: 10.3% on latency, 1.1% on bandwidth, and 2.2% on file operation rate. In most microbenchmarks the overhead is miniscule. Most notably, `fork+shell` exhibited a 4$\times$ slowdown

Microbenchmark	**Baseline** (MB/s)	**stdev** (MB/s)	**InversOS** (×)	**stdev** (×)
pipe	1,096.147	72.703	0.991	0.049
AF_UNIX stream	931.933	6.753	1.003	0.011
read 1 MB	3,706.665	65.823	0.978	0.013
read 1 MB open2close	3,474.633	45.699	0.990	0.015
mmap 1 MB	10,689.636	36.243	1.006	0.001
mmap 1 MB open2close	6,365.563	43.215	0.972	0.008
tcp	720.056	48.645	0.987	0.013
Geomean	—	—	0.989	—

Table 6.3: InversOS LMBench Bandwidth (Higher is Better)

because InversOS had to scan every code page of a newly executed shell. The same goes with `fork+exec`, in which the executed program is much smaller than the shell and thus incurred much less overhead (18.8%). In `fork+exit`, the 25.6% overhead comes from an optimization of copying code page PTEs upfront; Linux by default only sets up shared page table mappings of a child process at page faults (i.e., when the child first accesses the page), which, however, would cause redundant code scanning in InversOS as InversOS invokes the code scanner whenever a page in an elevated task is marked executable. We therefore optimized InversOS to avoid redundant code scanning by copying an elevated task's code page PTEs during `fork()` and enabled this optimization in all InversOS experiments. InversOS incurred 49.3% overhead in signal catching because of additional flipping of `PSTATE.UAO` (due to PAN being disabled) when setting up and tearing down a signal frame; this could be optimized away by simply disabling UAO support in the Linux kernel, which we opted not to in order to avoid introducing less relevant changes.

6.7.2 Macrobenchmarks and Applications

To see how InversOS performs on real workloads, we used SPEC CPU 2017 v1.1.9 [232] and Nginx v1.23.3 [244]. SPEC CPU 2017 is a comprehensive benchmark suite containing CPU- and memory-intensive programs written in C, C++, and/or Fortran that stress a computer system's performance. Nginx is a high performance web

Benchmark (Rate)	Baseline (s)	Benchmark (Speed)	Baseline (s)
500.perlbench_r	135.795	600.perlbench_s	135.289
502.gcc_r	268.035	602.gcc_s	268.294
505.mcf_r	431.810	605.mcf_s	428.423
520.omnetpp_r	354.081	620.omnetpp_s	353.981
523.xalancbmk_r	242.465	623.xalancbmk_s	242.501
525.x264_r	96.540	625.x264_s	96.527
531.deepsjeng_r	203.713	631.deepsjeng_s	227.060
541.leela_r	216.941	641.leela_s	217.306
557.xz_r	128.610	657.xz_s	127.926
508.namd_r	157.894		
510.parest_r	330.373		
511.povray_r	25.722		
519.lbm_r	231.428	619.lbm_s	1,718.814
526.blender_r	533.649		
538.imagick_r	167.810	638.imagick_s	168.136
544.nab_r	396.789	644.nab_s	397.586

Table 6.4: Baseline SPEC CPU 2017 Execution Time (Lower is Better)

server written in C that has been widely used in the real world.

For SPEC CPU 2017, we evaluated 28 (out of 43) benchmark programs in C/C++ as LLVM/Clang cannot compile Fortran code. We used the `train` (instead of the larger `ref`) input set because `train` yielded execution time of at least 20 seconds in each benchmark already. We report average execution time with 10 rounds of execution for each benchmark; standard deviations are negligible (less than 1%).

For Nginx, we used Nginx to host randomly generated static files ranging from 1 KB to 512 MB with one worker process listening to port 8080 for HTTP requests. We then ran ApacheBench (`ab`) [13] on the same machine to measure Nginx's bandwidth of transferring files within a period of 10 seconds. We report an average and a standard deviation over 10 rounds of execution for each file size.

Table 6.4 and Figure 6.6 present the Baseline performance of SPEC CPU 2017 and Nginx, respectively. Figures 6.7 and 6.8 show the performance overhead InversOS incurred on SPEC CPU 2017 and Nginx, respectively. Overall, InversOS increased the execution time of SPEC CPU 2017 by a geometric mean of 7.1% and degraded the bandwidth of Nginx by a geometric mean of 3.0%. We studied the overhead on SPEC

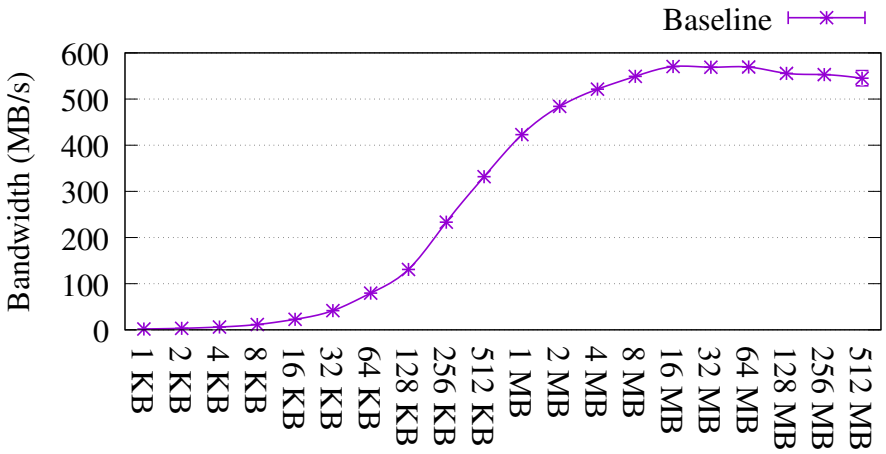

Figure 6.6: Baseline Nginx Bandwidth (Higher is Better)

CPU 2017 and discovered that our software-based forward-edge CFI caused most of the overhead; with that disabled, the overhead decreased to a geometric mean of 1.9% (in particular, `xalancbmk`'s overhead dropped down from more than 40% to less than 3%). This indicates that InversOS's shadow stack and bit-masking transformations and kernel modifications have minimal performance impact on SPEC CPU 2017, compared with software-based forward-edge CFI. Incorporating BTI [22], we expect InversOS's performance overhead to be greatly reduced; with BTI, no explicit CFI checks (as shown in Figure 6.4) are needed. However, as BTI does not provide protected shadow stacks by itself, (post-)ARMv8.5-A systems can still leverage InversOS's Privilege Inversion to protect the integrity of shadow stacks. Nginx saw significant variance especially on file sizes $\leq$ 128 KB. We suspect that the cause of high variance is caching and file system behaviors.

6.8 Related Work

6.8.1 Control-Flow Integrity

Since the introduction of the original CFI work [1, 2], a long line of research has been proposed to improve its precision, performance, and/or applicability [9, 34, 36, 41–

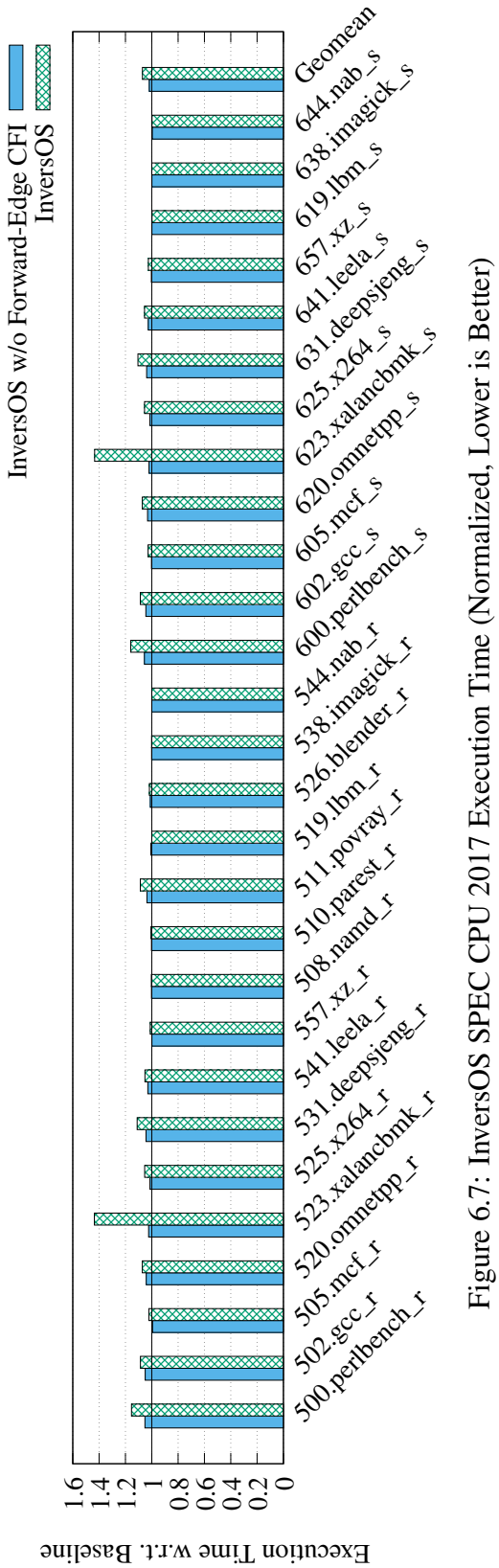

Figure 6.7: InversOS SPEC CPU 2017 Execution Time (Normalized, Lower is Better)

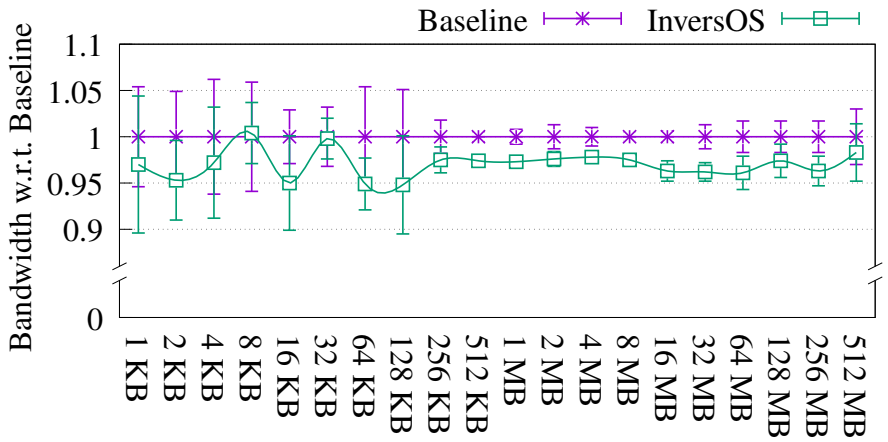

Figure 6.8: InversOS Nginx Bandwidth (Normalized, Higher is Better)

43, 47, 54, 57, 68, 77–79, 81, 88, 90, 94, 104, 105, 112, 115, 121, 122, 124, 126, 128, 130, 132, 133, 146, 148, 149, 155, 170, 179, 185–187, 191, 197, 202, 203, 225, 247, 250, 251, 255, 256, 258, 262, 270, 273, 274, 276–279, 282, 284]. As InversOS leverages label-based CFI for forward edges and protected shadow stacks for backward edges, we compare InversOS with various types of CFI schemes.

Stateless CFI The original CFI [1, 2] restricts forward-edge indirect control-flow targets via a coarse-grained context-insensitive analysis, which statically assigns a distinct label to allowed targets (an equivalence class or EC) of each indirect call or jump and inserts checks for a matched label at indirect call and jump sites. Subsequent research on stateless forward-edge CFI makes trade-offs between granularity and performance [34, 185–187, 202, 247, 251, 256, 277–279], strengthens other security policies [47, 94, 179, 276], or applies to new platforms [9, 36, 42, 68, 79, 90, 104, 126, 128, 130, 191, 203, 255, 262, 282]. Hardware support for stateless forward-edge CFI (such as HAFIX [81], HCFI [57], Intel CET [225], and ARM BTI [22]) has been proposed, which further lowers the performance overhead but only provides coarse-grained protection similar to the original CFI. InversOS's forward-edge CFI, while currently prototyped with two labels, can seamlessly adopt any of the above available finer-grained schemes for better security. It can also utilize BTI on newer processors for better per-

formance.

Stateful CFI Due to imprecision of context-insensitive CFI, researchers have focused on context-sensitive CFI policies that take previous execution history into account. Using a runtime monitor (inlined or as a separate process), these systems track executed branches [54, 115, 197, 270, 274], paths [88, 121, 250], call-sites [132, 133], code pointer origins [133], or complete control flows [105, 155] to reduce the size of ECs. However, such dynamic CFI schemes require hardware features only found on x86 processors, such as Branch Trace Store (BTS) [270], Last Branch Record (LBR) [54, 197, 250, 274], Performance Monitoring Unit (PMU) [274], Processor Trace (PT) [88, 105, 115, 121, 155], Transactional Synchronization Extensions (TSX) [132, 133], and MPX [133], limiting their applicability on AArch64. Compared with stateful CFI, InversOS offers a weaker protection on forward edges but provides the strongest security on backward edges with better performance and less resource consumption.

Shadow Stacks The original CFI [1, 2] uses shadow stacks for backward-edge protection; their debut dates back to RAD [55] and StackGhost [98], which all used the compact shadow stack design. Dang et al. [77] proposed the parallel shadow stack design, improving the performance but wasting more memory. As described in Section 6.2, in order to guarantee return address integrity, shadow stacks need a protection mechanism that forbids unauthorized tampering. A few systems [77, 78] simply leave shadow stacks unprotected, while some rely on system calls [55, 98, 255] or SFI [63, 282] for protection but incur prohibitive overhead. More commonly used is information hiding (i.e., ASLR [201]), which places shadow stacks at a random location in the address space to increase the difficulty for attackers to locate the shadow stacks [43, 202, 228, 284]. Though achieving the best performance among software-only solutions, information hiding provides the weakest guarantee and is vulnerable to

information disclosure attacks [33, 95, 101, 110, 192, 230, 241]. Hardware-assisted shadow stack protection significantly lowers the performance cost and can be fulfilled differently on different ISAs. On x86_32, segmentation [1, 2] provides the most efficient implementation. CET [225] offers native support for protected shadow stacks on x86_64 but is only available on most recent processors [4, 123]; a few solutions repurposed MPX [43, 122, 128] or MPK [43, 112] for non-CET-equipped Intel processors but reported vastly different overhead numbers. HCFI [57] implements an in-chip non-memory-mapped shadow stack on SPARC via a custom ISA extension. In the microcontroller world, Silhouette [282] (as we presented in Chapter 3) and Kage [90] transform regular store instructions into LSU stores on ARMv7-M [21], while CaRE [191] and TZmCFI [130] leverage TrustZone-M on ARMv8-M [23]. To the best of our knowledge, InversOS is the first to provide hardware-assisted protected shadow stacks on AArch64; our Privilege Inversion technique is inspired by Silhouette-Invert [282] (as described in Section 3.4.6).

Cryptographic CFI Mashtizadeh et al. [170] created Cryptographic CFI (CCFI), which uses message authentication codes (MACs) to sign and verify code pointers and leverages x86's AES-NI instructions to accelerate MAC calculation. ARMv8.3-A's PAuth [22] adds hardware support for pointer authentication codes (PACs) and places PACs in unused upper bits of pointers. Qualcomm has adopted PAuth to enforce CFI [208]. However, CCFI and plain PAuth suffer from pointer reuse attacks, in which attackers use buffer overread vulnerabilities [241] to harvest signed pointers for later reuse. Utilizing PAuth, PARTS [148] signs code pointers with type IDs; this limits reuse of signed return addresses within the same functions and signed function pointers within the same types. PACStack [149] and PACtight [124] are also based on PAuth; both solutions sign a return address with the PAC of the previous return address, creating an authenticated stack. PACtight further signs a function pointer with its address and a random tag. Studies on type-ID-based PACs [258] and authenticated chain of

return addresses [146] have also been explored on RISC-V as custom ISA extensions. PAL [273] uses PAuth to provide CFI for OS kernels.

As PACStack [149] and PACtight [124] share the most similar threat model, assumptions, and security guarantees with InversOS, we compare InversOS with them in more detail. PACStack claims that its authenticated stack "achieves security comparable to hardware-assisted shadow stacks *without requiring dedicated hardware*"; we show that InversOS achieves hardware-assisted shadow stacks *with even less hardware requirements* (ARMv8.1-A's PAN and HPDS vs. ARMv8.3-A's PAuth). Furthermore, PACStack requires forward-edge CFI but reported performance numbers without accounting its overhead. For an apples-to-apples comparison, InversOS without forward-edge CFI outperforms PACStack (1.9% vs. $\approx$3.0% on SPEC CPU 2017 and $\leq$3.0% vs. 6–13% on Nginx). PACtight enforces finer-grained forward-edge CFI than InversOS and its performance (4.0% on Nginx) is roughly on par with InversOS. However, PACtight maintains an in-memory metadata storage for the random tags at runtime and relies on ASLR [201] to hide its location. Essentially, PAC-based systems only offer probabilistic security even if the entropy they provide is large. In contrast, InversOS's shadow stacks are integrity-enforced, providing the strongest guarantees.

Other Approaches Kuznetsov et al. [140] developed code-pointer integrity (CPI), an approach to ensuring memory safety of all code pointers and data related to code pointers. CPI identifies such data via static analysis and instrumentation and places the data in isolated safe regions. Again, segmentation [4, 123] and ASLR [201] were used to protect the safe regions on x86_32 and x86_64, respectively. PACtight-CPI [124] implements CPI using PAuth, incurring 4.07% performance overhead on average. InversOS's Privilege Inversion provides an alternative option to protect CPI's safe regions with potentially less overhead. μRAI [9] enforces return address integrity on microcontrollers by encoding return addresses in a reserved register and ensuring that the register value is never corrupted; it relies on system calls to spill the register value to protected

memory when needing to fold a call chain longer than what a single register can hold. While μRAI is in theory applicable to general-purpose systems like x86 and AArch64, we believe such an approach provides poor scalability and may incur high performance overhead due to more nested function calls than on microcontrollers.

6.8.2 Intra-Address Space Isolation

InversOS uses Privilege Inversion for efficient intra-address space isolation. We omit discussing custom hardware modifications that compartmentalize software (e.g., CODOMs [252] and Mondrian [267, 268]) and limit our discussion on related work utilizing recent commodity hardware. Approaches used to enforce CFI are also not repeated here.

SFI [171, 254] instruments program loads and stores to prevent them from accessing certain memory regions and has been used to sandbox untrusted code [139, 221, 272]. While some systems [89, 136] accelerate SFI checks using MPX on x86, the overhead of SFI is still considered high (on both performance [261] and memory usage [43]) and grows as the number of isolated regions increases. Furthermore, SFI often requires CFI to ensure that SFI checks are not bypassed by attacker-manipulated control flow. Another address-based isolation technique is hardware-enforced address range monitoring. PicoXOM [227] (as described in Chapter 4) enforces execute-only memory (XOM) by configuring ARM debug registers to watch over a code segment against read accesses. Such approaches are limited by hardware resources available and cannot scale up.

Recent defenses enforce domain-based isolation; memory regions are associated with a protection domain, and different mechanisms are used to allow or disallow accesses to the protection domain at runtime. On x86, researchers have explored domain-based memory access control using hardware features such as Virtual Machine Extensions (VMX) [114, 117, 136, 154, 173, 182, 207, 261], MPK [113, 114, 117, 198,

217, 218, 243, 249, 253, 257], SMAP [261], and CET [271]. ARMlock [283] and Shreds [51] use ARM domains, which are only available on AArch32 [22]. Previous work has also used LSU instructions for isolation. ILDI [56] utilizes LSU instructions and PAN to protect a safe region inside the OS kernel; it relies on a more privileged hypervisor to moderate sensitive kernel operations. uXOM [141] transforms regular loads/stores to LSU instructions to enforce XOM on microcontrollers, where application code typically executes in the privileged mode already. InversOS, employing Privilege Inversion, is the first to extend domain-based isolation to AArch64 user space.

We notice that Privbox [139] and SEIMI [261], like InversOS, also proposed executing user-space code in the privileged mode (x86's ring 0). Privbox does so to accelerate system call invocation and uses SFI to safely run elevated code. The overhead of its heavy instrumentation, however, may outweigh its speedup from faster system calls on certain programs. InversOS can benefit from the idea of system call acceleration for elevated tasks, which we leave as future work. SEIMI flips SMAP (x86 equivalence to PAN) to create a safe region for trusted user-space code; its OS kernel is then elevated to run in ring -1 via VMX. Compared with SEIMI, InversOS's Privilege Inversion provides instruction-level isolation and requires no frequent domain switching.

Chapter 7

Conclusions and Future Work

This dissertation presented our work on enforcing low-cost security policies for ARM-based systems, whether they are based on tiny resource-constrained microcontrollers or powerful general-purpose application processors, and made four major contributions elaborated in each of Chapters 3–6, respectively. For each of the chapters, we conclude our work and discuss potential directions for future research as follows.

Efficient Protected Shadow Stacks for Embedded Systems We presented Silhouette: a software control-flow hijacking defense that guarantees the integrity of return addresses for embedded systems. To minimize overhead, we proposed Silhouette-Invert, a system which provides the same protections as Silhouette with significantly lower overhead at the cost of a minor hardware change. We implemented our prototypes for an ARMv7-M development board. Our evaluation showed that Silhouette incurs low performance overhead: a geometric mean of 1.3% and 3.4% on two benchmark suites, and Silhouette-Invert reduces the overhead to 0.3% and 1.9%. Silhouette is open-sourced at `https://github.com/URSec/Silhouette`.

We see two primary directions for future work. First, we can optimize Silhouette's performance. For example, currently Silhouette transforms all non-atomic stores into unprivileged stores and instruments atomic stores with SFI. As Silhouette ensures that the stack pointer stays within the stack region, store instructions using the `sp` register

and an immediate to compute target addresses are unexploitable. Consequently, Silhouette could elide store hardening and SFI on such stores. Second, we can use Silhouette to protect other memory structures, such as the safe region used in CPI [140] and the process state saved on interrupts and context switches (like previous work [68] does).

Fast Execute-Only Memory for Embedded Systems We presented PicoXOM: a fast and novel XOM system for ARMv7-M and ARMv8-M devices which leverages ARM's MPU and DWT unit. PicoXOM incurs an average performance overhead of 0.33% and an average code size overhead of 5.89% on the BEEBS and CoreMark-Pro benchmark suites and five real-world applications. A prototype of PicoXOM on ARMv7-M is open-sourced at `https://github.com/URSec/PicoXOM`.

In future work, we can explore how to leverage debug support like DWT to enforce other security policies with low overhead. In particular, hardware debug facilities on RISC-V [211] include a watchpoint trigger system similar to but more powerful than ARM's DWT [21, 23]; it supports any number of watchpoint triggers to be linked together to perform chained conjunctive matching of addresses, address ranges, instruction opcodes, and data values, and it supports a versatile collection of conditions under which a trigger fires (traps the execution) including inverse matching, less-than matching, and greater-than-or-equal-to matching. Utilizing RISC-V watchpoint triggers, we can implement more complicated access control mechanisms with performance overhead as negligible as PicoXOM.

Leakage-Resistant Randomization for Microcontrollers We presented Randezvous: a diversification-based control-flow hijacking defense enhanced with novel techniques that mitigate control data leakage and strengthen the low entropy on MCUs. We demonstrated Randezvous's efficacy and showed that Randezvous incurs low overhead on our benchmarks and applications. Randezvous is open-sourced at `https://github.com/URSec/Randezvous`.

In future work, we can explore randomization schemes that avoid code layouts with bad cache performance by leveraging theory on code locality [144]. We can also investigate whether an attacker can infer which control data (slot) is real by examining which control data mutates over time. Such attacks may be possible but require multiple buffer overreads [241] in multiple functions to be practical.

Efficient Control-Flow Protection for AArch64 Applications We presented InversOS, a hardware-assisted protected shadow stack implementation for AArch64, which utilizes common hardware features to create novel and efficient intra-address space isolation and safely executes user-space code in the privileged mode via OS kernel and compiler restraints. InversOS is backward-compatible with existing application binaries by a novel use of another AArch64 feature. Our analysis shows that InversOS is secure and effective in mitigating attacks, and our performance evaluation demonstrates the low costs of InversOS on real-world benchmarks and applications. Our prototype of InversOS is open-sourced at `https://github.com/URSec/InversOS`.

We see several directions for future work. First, we can explore system call optimizations (such as Privbox [139]) for elevated tasks; these tasks already run in the privileged mode and can accelerate system call invocation by avoiding the costly `SVC` instructions. Second, we can leverage Privilege Inversion to enforce other security policies such as CPI [140] and full memory safety [86, 180, 181, 281], reducing their overheads significantly. Finally, we intend to investigate potential performance improvements to InversOS by using more recent ISA features (e.g., BTI and E0PD) [22] on real hardware.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, Alexandria, VA, USA, 2005. ACM. ISBN 1-59593-226-7. doi: 10.1145/1102120.1102165. URL https://doi.org/10.1145/1102120.1102165.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, 13(1):4:1–4:40, November 2009. ISSN 1094-9224. doi: 10.1145/1609956.1609960. URL http://doi.org/10.1145/1609956.1609960.

[3] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in Designing Exploit Mitigations for Deeply Embedded Systems. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, EuroSP '19, pages 31–46, Stockholm, Sweden, 2019. IEEE Computer Society. ISBN 978-1-7281-1148-3. doi: 10.1109/EuroSP.2019.00013. URL https://doi.org/10.1109/EuroSP.2019.00013.

[4] AMD. *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices Inc., January 2023. URL https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5. 40332 Rev 4.06.

[5] Misiker Tadesse Aga and Todd Austin. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '19, pages 26–36, Washington, DC, USA, 2019. IEEE Computer Society. ISBN 978-1-7281-1436-1. doi: 10.1109/CGO.2019.8661202. URL https://doi.org/10.1109/CGO.2019.8661202.

[6] Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. Methodologies for Quantifying (Re-)Randomization Security and Timing under JIT-ROP. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1803–1820, Orlando, FL, USA, 2020. ACM. ISBN 978-1-4503-7089-9. doi: 10.1145/3372297.3417248. URL https://doi.org/10.1145/3372297.3417248.

[7] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Oakland, CA, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.30. URL https://doi.org/10.1109/SP.2008.30.

[8] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*, Security '09, pages 51–66, Montreal, QC, Canada, 2009. USENIX Association. ISBN 978-1-931971-69-0. URL https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/baggy-bounds-checking-efficient-and.

[9] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. μRAI: Securing Embedded Systems with Return Address Integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium*,

NDSS '20, San Diego, CA, USA, 2020. Internet Society. ISBN 1-891562-61-4. doi: 10.14722/ndss.2020.24016. URL https://doi.org/10.14722/ndss.2020.24016.

[10] Amazon Web Services. FreeRTOS: Real-Time Operating System for Microcontrollers, 2017. URL https://aws.amazon.com/freertos. [Accessed on: 2022-04-28].

[11] Amazon Web Services. Amazon EC2 A1 Instances: Optimized cost and performance for scale-out workloads, 2023. URL https://aws.amazon.com/ec2/instance-types/a1. [Accessed on: 2023-04-07].

[12] Android. Verifying App Behavior on the Android Runtime (ART), 2023. URL https://developer.android.com/guide/practices/verifying-apps-art#Stack_Size. [Accessed on: 2023-02-25].

[13] Apache. ab - Apache HTTP server benchmarking tool, 2023. URL https://httpd.apache.org/docs/current/programs/ab.html. [Accessed on: 2023-01-30].

[14] Apple. Apple unleashes M1, 2020. URL https://www.apple.com/newsroom/2020/11/apple-unleashes-m1. [Accessed on: 2023-04-17].

[15] ARM. SSL Library Mbed TLS, 2008. URL https://tls.mbed.org. [Accessed on: 2022-04-28].

[16] ARM. Mbed TLS Benchmark Demonstration Program, 2009. URL https://github.com/ARMmbed/mbedtls/blob/development/programs/test/benchmark.c. [Accessed on: 2022-04-28].

[17] ARM. Mbed μVisor, 2015. URL https://www.mbed.com/en/technologies/security/uvisor. [Accessed on: 2022-04-28].

[18] ARM. *CoreMark Benchmarking for ARM® Cortex® Processors: Application Note 350*. Arm Holdings, July 2013. URL https://developer.arm.com/documentation/106383/a. DAI 0350A.

[19] ARM. *ARM® Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Arm Holdings, March 2018. URL https://developer.arm.com/documentation/ddi0406/cd. DDI 0406C.d.

[20] ARM. *Arm® Cortex®-M4 Processor Technical Reference Manual*. Arm Holdings, May 2020. URL https://developer.arm.com/documentation/100166/0001. Issue 100166_0001_04_en.

[21] ARM. *Arm®v7-M Architecture Reference Manual*. Arm Holdings, February 2021. URL https://developer.arm.com/documentation/ddi0403/ee. DDI 0403E.e.

[22] ARM. *Arm® Architecture Reference Manual: for A-profile architecture*. Arm Holdings, August 2022. URL https://developer.arm.com/documentation/ddi0487/ia. DDI 0487I.a.

[23] ARM. *Arm®v8-M Architecture Reference Manual*. Arm Holdings, December 2022. URL https://developer.arm.com/documentation/ddi0553/bv. DDI 0553B.v.

[24] AspenCore. 2019 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments, 2019. URL https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. [Accessed on: 2023-04-25].

[25] Marc Auslander and Martin Hopkins. An Overview of the PL.8 Compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 22–31, Boston, MA, USA, 1982. ACM. ISBN 0-89791-074-5.

doi: 10.1145/800230.806977. URL https://doi.org/10.1145/800230.806977.

[26] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 433–447, San Diego, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/backes.

[27] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 1342–1353, Scottsdale, AZ, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660378. URL https://doi.org/10.1145/2660267.2660378.

[28] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, Ottawa, ON, Canada, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134000. URL https://doi.org/10.1145/1133981.1134000.

[29] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, Paris, France, 2008. Springer-Verlag. ISBN 978-3-540-70542-0. doi: 10.1007/978-3-540-70542-0_1. URL https://doi.org/10.1007/978-3-540-70542-0_1.

[30] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An

Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, Security '03, pages 105–120, Washington, DC, USA, 2003. USENIX Association. URL https://www.usenix.org/conference/12th-usenix-security-symposium/address-obfuscation-efficient-approach-combat-broad-range.

[31] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium*, Security '05, pages 255–270, Baltimore, MD, USA, 2005. USENIX Association. URL https://www.usenix.org/conference/14th-usenix-security-symposium/efficient-techniques-comprehensive-protection-memory-error.

[32] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 268–279, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813691. URL https://doi.org/10.1145/2810103.2813691.

[33] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, San Jose, CA, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.22. URL https://doi.org/10.1109/SP.2014.22.

[34] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, Orlando, FL, USA,

2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076783. URL https://doi.org/10.1145/2076732.2076783.

[35] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, Hong Kong, China, 2011. ACM. ISBN 978-1-4503-0564-8. doi: 10.1145/1966913.1966919. URL https://doi.org/10.1145/1966913.1966919.

[36] Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, 2016. Internet Society. ISBN 1-891562-41-X. doi: 10.14722/ndss.2016.23421. URL https://doi.org/10.14722/ndss.2016.23421.

[37] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates Inc, Sebastopol, CA, USA, 3rd edition, 2005. ISBN 0-5960-0565-2.

[38] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, 2016. Internet Society. ISBN 1-891562-41-X. doi: 10.14722/ndss.2016.23364. URL https://doi.org/10.14722/ndss.2016.23364.

[39] David Brash. The ARMv8-A Architecture and Its Ongoing Development, 2014. URL https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/the-armv8-a-architecture-and-its-ongoing-development. [Accessed on: 2023-02-25].

[40] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*, ICCWS '16, pages 56–64, Boston, MA, USA, 2016. ACPI. ISBN 978-1-910810-82-8.

[41] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1):16:1–16:33, April 2017. ISSN 0360-0300. doi: 10.1145/3054924. URL https://doi.org/10.1145/3054924.

[42] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, USA, 2018. Internet Society. ISBN 1-891562-49-5. doi: 10.14722/ndss.2018.23279. URL https://doi.org/10.14722/ndss.2018.23279.

[43] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, SP '19, pages 985–999, San Francisco, CA, USA, 2019. IEEE Computer Society. ISBN 978-1-5386-6660-9. doi: 10.1109/SP.2019.00076. URL https://doi.org/10.1109/SP.2019.00076.

[44] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. Data Randomization. Technical Report MSR-TR-2008-120, Microsoft Research, September 2008.

[45] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 385–399, San Diego, CA, USA, 2014. USENIX Association. ISBN 978-

1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[46] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*, Security '15, pages 161–176, Washington, DC, USA, 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[47] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 45–58, Big Sky, MT, USA, 2009. ACM. ISBN 978-1-6055-8752-3. doi: 10.1145/1629575.1629581. URL https://doi.org/10.1145/1629575.1629581.

[48] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. A Practical Approach for Adaptive Data Structure Layout Randomization. In *Proceedings of the 20th European Symposium on Computer Security*, ESORICS '15, pages 69–89, Vienna, Austria, 2015. Springer-Verlag. ISBN 978-3-319-24174-6. doi: 10.1007/978-3-319-24174-6_4. URL https://doi.org/10.1007/978-3-319-24174-6_4.

[49] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, Security '05, pages 177–191, Baltimore, MD, USA, 2005. USENIX Association. URL https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

[50] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, USA, 2015. Internet Society. ISBN 1-891562-38-X. doi: 10.14722/ndss.2015.23248. URL http://doi.org/10.14722/ndss.2015.23248.

[51] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 56–71, San Jose, CA, USA, 2016. IEEE Computer Society. ISBN 978-1-5090-0824-7. doi: 10.1109/SP.2016.12. URL https://doi.org/10.1109/SP.2016.12.

[52] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M. Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, SP '17, pages 304–319, San Jose, CA, USA, 2017. IEEE Computer Society. ISBN 978-1-5090-5532-6. doi: 10.1109/SP.2017.30. URL https://doi.org/10.1109/SP.2017.30.

[53] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-Demand Live Randomization. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 50–61, New Orleans, LA, USA, 2016. ACM. ISBN 978-1-4503-3935-3. doi: 10.1145/2857705.2857726. URL https://doi.org/10.1145/2857705.2857726.

[54] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS '14, San Diego, CA, USA, 2014. Internet Society. ISBN 1-

891562-35-5. doi: 10.14722/ndss.2014.23156. URL https://doi.org/10.14722/ndss.2014.23156.

[55] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 409–417, Mesa, AZ, USA, 2001. IEEE Computer Society. ISBN 0-7695-1077-9. doi: 10.1109/ICDSC.2001.918971. URL https://doi.org/10.1109/ICDSC.2001.918971.

[56] Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. Instruction-Level Data Isolation for the Kernel on ARM. In *Proceedings of the 54th ACM/EDAC/IEEE Annual Design Automation Conference*, DAC '17, Austin, TX, USA, 2017. ACM. ISBN 978-1-4503-4927-7. doi: 10.1145/3061639.3062267. URL https://doi.org/10.1145/3061639.3062267.

[57] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-Enforced Control-Flow Integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, New Orleans, LA, USA, 2016. ACM. ISBN 978-1-4503-3935-3. doi: 10.1145/2857705.2857722. URL https://doi.org/10.1145/2857705.2857722.

[58] Abraham A. Clements. PinLock, 2018. URL https://github.com/embedded-sec/ACES/tree/master/test_apps/pinlock. [Accessed on: 2022-06-23].

[59] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-Metal Embedded Systems with Privilege Overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, SP '17, pages 289–303, San Jose,

CA, USA, 2017. IEEE Computer Society. ISBN 978-1-5090-5532-6. doi: 10.1109/SP.2017.37. URL https://doi.org/10.1109/SP.2017.37.

[60] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic Compartments for Embedded Systems. In *Proceedings of the 27th USENIX Security Symposium*, Security '18, pages 65–82, Baltimore, MD, USA, 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/clements.

[61] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, pages 30–42, Limassol, Cyprus, 2022. ACM. ISBN 978-1-4503-9704-9. doi: 10.1145/3545948.3545997. URL https://doi.org/10.1145/3545948.3545997.

[62] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813671. URL https://doi.org/10.1145/2810103.2813671.

[63] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to Protect Return Addresses from Attack. *SIGARCH Computer Architecture News*, 33(1): 65–72, March 2005. ISSN 0163-5964. doi: 10.1145/1055626.1055636. URL https://doi.org/10.1145/1055626.1055636.

[64] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, Security '98, San Antonio, TX, 1998. USENIX Association. URL https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[65] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 763–780, San Jose, CA, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.52. URL https://doi.org/10.1109/SP.2015.52.

[66] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 243–255, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813682. URL https://doi.org/10.1145/2810103.2813682.

[67] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, Stevenson, WA, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294295. URL https://doi.org/10.1145/1294261.1294295.

[68] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, San Jose, CA, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.26. URL https://doi.org/10.1109/SP.2014.26.

[69] CVE. CVE-2017-8410, 2017. URL https://www.cve.org/CVERecord?id=CVE-2017-8410. [Accessed on: 2023-03-09].

[70] CVE. CVE-2017-8412, 2017. URL https://www.cve.org/CVERecord?id=CVE-2017-8412. [Accessed on: 2023-03-09].

[71] CVE. CVE-2018-16525, 2018. URL https://www.cve.org/CVERecord?id=CVE-2018-16525. [Accessed on: 2023-03-09].

[72] CVE. CVE-2018-16526, 2018. URL https://www.cve.org/CVERecord?id=CVE-2018-16526. [Accessed on: 2023-03-09].

[73] CVE. CVE-2018-19417, 2018. URL https://www.cve.org/CVERecord?id=CVE-2018-19417. [Accessed on: 2023-03-09].

[74] CVE. CVE-2018-3898, 2018. URL https://www.cve.org/CVERecord?id=CVE-2018-3898. [Accessed on: 2023-03-09].

[75] CVE. CVE-2021-27421, 2021. URL https://www.cve.org/CVERecord?id=CVE-2021-27421. [Accessed on: 2022-09-19].

[76] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, Salt Lake City, UT, USA, 2014.

ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541946. URL https://doi.org/10.1145/2541940.2541946.

[77] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '15, pages 555–566, Singapore, Republic of Singapore, 2015. ACM. ISBN 978-1-4503-3245-3. doi: 10.1145/2714576.2714635. URL https://doi.org/10.1145/2714576.2714635.

[78] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, Hong Kong, China, 2011. ACM. ISBN 978-1-4503-0564-8. doi: 10.1145/1966913.1966920. URL https://doi.org/10.1145/1966913.1966920.

[79] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 2012 Network and Distributed System Security Symposium*, NDSS '12, San Diego, CA, USA, 2012. Internet Society. URL https://www.ndss-symposium.org/ndss2012/ndss-2012-programme/mocfi-framework-mitigate-control-flow-attacks-smartphonesoverlay-contextmocfi-framework-mitigate-control-flow-attacks-smartphones.

[80] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 401–416, San Diego, CA, USA, 2014. USENIX Association.

ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[81] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-Assisted Flow Integrity Extension. In *Proceedings of the 52nd ACM/EDAC/IEEE Annual Design Automation Conference*, DAC '15, San Francisco, CA, USA, 2015. ACM. ISBN 978-1-4503-3520-1. doi: 10.1145/2744769.2744847. URL https://doi.org/10.1145/2744769.2744847.

[82] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, USA, 2015. Internet Society. ISBN 1-891562-38-X. doi: 10.14722/ndss.2015.23262. URL http://doi.org/10.14722/ndss.2015.23262.

[83] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me If You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIACCS '13, pages 299–310, Hangzhou, China, 2013. ACM. ISBN 978-1-4503-1767-2. doi: 10.1145/2484313.2484351. URL https://doi.org/10.1145/2484313.2484351.

[84] Dinakar Dhurjati and Vikram Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, Shanghai, China, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134309. URL https://doi.org/10.1145/1134285.1134309.

[85] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory Safety Without Garbage Collection for Embedded Applications. *ACM Transactions in Embedded Computing Systems*, 4(1):73–111, February 2005. ISSN 1539-9087. doi: 10.1145/1053271.1053275. URL https://doi.org/10.1145/1053271.1053275.

[86] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, Ottawa, ON, Canada, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133999. URL https://doi.org/10.1145/1133981.1133999.

[87] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy Bounds with Accurate Checking. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops*, ISSREW '12, pages 195–200, Dallas, TX, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4928-6. doi: 10.1109/ISSREW.2012.24. URL https://doi.org/10.1109/ISSREW.2012.24.

[88] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium*, Security '17, pages 131–148, Vancouver, BC, Canada, 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding.

[89] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software from Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium*, Security '18, pages 1441–1458, Baltimore, MD, USA, 2018. USENIX Association. ISBN 978-1-939133-

04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/dong.

[90] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *Proceedings of the 31st USENIX Security Symposium*, Security '22, pages 2281–2298, Boston, MA, USA, 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/du.

[91] EEMBC. CoreMark: An EEMBC Benchmark, 2018. URL https://www.eembc.org/coremark. [Accessed on: 2022-04-28].

[92] EEMBC. CoreMark-Pro: An EEMBC Benchmark, 2019. URL https://www.eembc.org/coremark-pro. [Accessed on: 2022-04-28].

[93] Alexandru Elisei. bhyvearm64: CPU and Memory Virtualization on Armv8.0-A. In *The BSDCan Conference*. Ottawa, ON, Canada, 2019. URL https://www.bsdcan.org/2019/schedule/events/1074.en.html. [Accessed on: 2023-04-12].

[94] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Seattle, WA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL https://www.usenix.org/conference/osdi-06/xfi-software-guards-system-address-spaces.

[95] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15,

pages 781–796, San Jose, CA, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.53. URL https://doi.org/10.1109/SP.2015.53.

[96] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813646. URL https://doi.org/10.1145/2810103.2813646.

[97] Rich Felker et al. musl libc, 2021. URL https://musl.libc.org. [Accessed on: 2023-01-30].

[98] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th USENIX Security Symposium*, Security '01, Washington, DC, USA, 2001. USENIX Association. URL https://www.usenix.org/conference/10th-usenix-security-symposium/stackghost-hardware-facilitated-stack-protection.

[99] Free Software Foundation, Inc. Labels as Values (Using the GNU Compiler Collection (GCC)), 2023. URL https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html. [Accessed on: 2023-02-25].

[100] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and*

Operating Systems, ASPLOS '19, pages 469–484, Providence, RI, USA, 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304037. URL https://doi.org/10.1145/3297858.3304037.

[101] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, 2016. Internet Society. ISBN 1-891562-41-X. doi: 10.14722/ndss.2016.23262. URL https://doi.org/10.14722/ndss.2016.23262.

[102] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, San Diego, CA, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781133. URL https://doi.org/10.1145/781131.781133.

[103] William Gayde. How Arm Came to Dominate the Mobile Market, 2020. URL https://www.techspot.com/article/1989-arm-inside. [Accessed on: 2023-04-25].

[104] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, EuroSP '16, pages 179–194, Saarbruecken, Germany, 2016. IEEE Computer Society. ISBN 978-1-5090-1752-2. doi: 10.1109/EuroSP.2016.24. URL https://doi.org/10.1109/EuroSP.2016.24.

[105] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, pages 585–598, Xi'an, China, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037716. URL https://doi.org/10.1145/3037697.3037716.

[106] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 325–336, San Antonio, TX, USA, 2015. ACM. ISBN 978-1-4503-3191-3. doi: 10.1145/2699026.2699107. URL https://doi.org/10.1145/2699026.2699107.

[107] Jason Gionta, William Enck, and Per Larsen. Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security*, CNS '16, Philadelphia, PA, USA, 2016. IEEE. ISBN 978-1-5090-3065-1. doi: 10.1109/CNS.2016.7860485. URL https://doi.org/10.1109/CNS.2016.7860485.

[108] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium*, Security '12, pages 475–490, Bellevue, WA, USA, 2012. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida.

[109] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, San Jose, CA, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.43. URL https://doi.org/10.1109/SP.2014.43.

[110] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (and What to Do about It). In *Proceedings of the 25th USENIX Security Symposium*, Security '16, pages 105–119, Austin, TX, USA, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goktas.

[111] Google Cloud. Arm VMs on Compute, 2023. URL https://cloud.google.com/compute/docs/instances/arm-on-compute. [Accessed on: 2023-04-07].

[112] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, pages 119–134, San Sebastian, Spain, 2021. ACM. ISBN 978-1-4503-9058-3. doi: 10.1145/3471621.3471849. URL https://doi.org/10.1145/3471621.3471849.

[113] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-Kernel Isolation and Communication. In *Proceedings of the 2020 USENIX Annual Technical Conference*, ATC '20, pages 401–417, Virtual Event, 2020. USENIX Association. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/gu.

[114] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and Efficient Memory Protection Keys. In *Proceedings of the 2022 USENIX Annual Technical Conference*, ATC '22, pages 609–624, Carlsbad, CA, USA, 2022. USENIX Association. ISBN 978-1-939133-29-8. URL https://www.usenix.org/conference/atc22/presentation/gu-jinyu.

[115] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, CODASPY '17, pages 173–184, Scottsdale, AZ, USA, 2017. ACM. ISBN 978-1-4503-4523-1. doi: 10.1145/3029806.3029830. URL https://doi.org/10.1145/3029806.3029830.

[116] Javid Habibi, Aditi Gupta, Stephen Carlsony, Ajay Panicker, and Elisa Bertino. MAVR: Code Reuse Stealthy Attacks and Mitigation on Unmanned Aerial Vehicles. In *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems*, ICDCS '15, pages 642–652, Columbus, OH, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-7214-5. doi: 10.1109/ICDCS.2015.71. URL https://doi.org/10.1109/ICDCS.2015.71.

[117] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC '19, pages 489–503, Renton, WA, USA, 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/hedayati-hodor.

[118] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '00, pages 93–104, Cambridge, MA, USA, 2000. ACM. ISBN 1-58113-317-0. doi: 10.1145/378993.379006. URL https://doi.org/10.1145/378993.379006.

[119] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, San Francisco,

CA, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.39. URL https://doi.org/10.1109/SP.2012.39.

[120] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 969–986, San Jose, CA, USA, 2016. IEEE Computer Society. ISBN 978-1-5090-0824-7. doi: 10.1109/SP.2016.62. URL https://doi.org/10.1109/SP.2016.62.

[121] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1470–1486, Toronto, ON, Canada, 2018. ACM. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243797. URL https://doi.org/10.1145/3243734.3243797.

[122] Wei Huang, Zhen Huang, Dhaval Miyani, and David Lie. LMP: Light-Weighted Memory Protection with Hardware Assistance. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 460–470, Los Angeles, CA, USA, 2016. ACM. ISBN 978-1-4503-4771-6. doi: 10.1145/2991079.2991089. URL https://doi.org/10.1145/2991079.2991089.

[123] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, December 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Order Number: 325462-078US.

[124] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly Seal Your Sensitive Pointers with PACTight. In *Pro-*

ceedings of the 31st USENIX Security Symposium, Security '22, pages 3717–3734, Boston, MA, USA, 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/ismail.

[125] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1868–1882, Toronto, ON, Canada, 2018. ACM. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243739. URL https://doi.org/10.1145/3243734.3243739.

[126] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS '14, San Diego, CA, USA, 2014. Internet Society. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23287. URL https://doi.org/10.14722/ndss.2014.23287.

[127] Yier Jin, Grant Hernandez, and Daniel Buentello. Smart Nest Thermostat: A Smart Spy in Your Home. In *Black Hat USA*. Las Vegas, NV, USA, 2014.

[128] Ethan Johnson, Colin Pronovost, and John Criswell. Hardening Hypervisors with Ombro. In *Proceedings of the 2022 USENIX Annual Technical Conference*, ATC '22, pages 415–435, Carlsbad, CA, USA, 2022. USENIX Association. ISBN 978-1-939133-29-8. URL https://www.usenix.org/conference/atc22/presentation/johnson.

[129] Richard W. M. Jones and Paul H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, AADEBUG '97, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press; Linköpings

universitet. URL https://www.ep.liu.se/ecp/article.asp?issue=001&article=002.

[130] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. TZm-CFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M. *International Journal of Parallel Programming*, 49:216–236, April 2021. ISSN 1573-7640. doi: 10.1007/s10766-020-00673-z. URL https://doi.org/10.1007/s10766-020-00673-z.

[131] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Saddle River, NJ, USA, 2nd edition, 1988. ISBN 0-13-110362-8.

[132] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive Call-Site Sensitive Control Flow Integrity. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, EuroSP '19, pages 95–110, Stockholm, Sweden, 2019. IEEE Computer Society. ISBN 978-1-7281-1148-3. doi: 10.1109/EuroSP.2019.00017. URL https://doi.org/10.1109/EuroSP.2019.00017.

[133] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-Sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 195–211, Santa Clara, CA, USA, 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/khandaker.

[134] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Miami Beach, FL, USA,

2006. IEEE Computer Society. ISBN 0-7695-2716-7. doi: 10.1109/ACSAC.2006.9. URL https://doi.org/10.1109/ACSAC.2006.9.

[135] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, USA, 2018. Internet Society. ISBN 1-891562-49-5. doi: 10.14722/ndss.2018.23107. URL https://doi.org/10.14722/ndss.2018.23107.

[136] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys '17, pages 437–452, Belgrade, Serbia, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064217. URL https://doi.org/10.1145/3064176.3064217.

[137] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-Assisted Code Randomization. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, SP '18, pages 461–477, San Francisco, CA, USA, 2018. IEEE Computer Society. ISBN 978-1-5386-4353-2. doi: 10.1109/SP.2018.00029. URL https://doi.org/10.1109/SP.2018.00029.

[138] Djamel Eddine Kouicem, Abdelmadjid Bouabdallah, and Hicham Lakhlef. Internet of Things Security: A Top-Down Survey. *Computer Networks*, 141:199–221, August 2018. ISSN 1389-1286. doi: 10.1016/j.comnet.2018.03.012. URL https://doi.org/10.1016/j.comnet.2018.03.012.

[139] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster System Calls Through

Sandboxed Privileged Execution. In *Proceedings of the 2022 USENIX Annual Technical Conference*, ATC '22, pages 233–247, Carlsbad, CA, USA, 2022. USENIX Association. ISBN 978-1-939133-29-8. URL https://www.usenix.org/conference/atc22/presentation/kuznetsov.

[140] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 147–163, Broomfield, CO, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov.

[141] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 231–247, Santa Clara, CA, USA, 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/kwon.

[142] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, CGO '04, Palo Alto, CA, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

[143] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, San Diego, CA, USA, 2007.

ACM. ISBN 978-1-5959-3633-2. doi: 10.1145/1250734.1250766. URL https://doi.org/10.1145/1250734.1250766.

[144] Rahman Lavaee, John Criswell, and Chen Ding. Codestitcher: Inter-Procedural Basic Block Layout Optimization. In *Proceedings of the 28th International Conference on Compiler Construction*, CC '19, pages 65–75, Washington, DC, USA, 2019. ACM. ISBN 978-1-4503-6277-1. doi: 10.1145/3302516.3307358. URL https://doi.org/10.1145/3302516.3307358.

[145] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout. *IEEE Transactions on Dependable and Secure Computing*, pages 2559–2575, July 2022. ISSN 1941-0018. doi: 10.1109/TDSC.2021.3063843. URL https://doi.org/10.1109/TDSC.2021.3063843.

[146] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. Zipper Stack: Shadow Stacks Without Shadow. In *Proceedings of the 25th European Symposium on Research in Computer Security*, ESORICS '20, pages 338–358, Guildford, UK, 2020. Springer-Verlag. ISBN 978-3-030-58951-6. doi: 10.1007/978-3-030-58951-6_17. URL https://doi.org/10.1007/978-3-030-58951-6_17.

[147] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '00, pages 168–177, Cambridge, MA, USA, 2000. ACM. ISBN 1-58113-317-0. doi: 10.1145/378993.379237. URL https://doi.org/10.1145/378993.379237.

[148] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ek-

berg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 177–194, Santa Clara, CA, USA, 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[149] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. PACStack: an Authenticated Call Stack. In *Proceedings of the 30th USENIX Security Symposium*, Security '21, pages 357–374, Virtual Event, 2021. USENIX Association. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[150] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. Polymorphing Software by Randomizing Data Structure Layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 107–126, Como, Italy, 2009. Springer-Verlag. ISBN 978-3-642-02918-9. doi: 10.1007/978-3-642-02918-9_7. URL https://doi.org/10.1007/978-3-642-02918-9_7.

[151] Linux. Linux Kernel Source Tree v5.6, 2020. URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v5.6. [Accessed on: 2023-03-11].

[152] Linux. Linux Kernel Stable Tree v4.19.219, 2021. URL https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v4.19.219. [Accessed on: 2023-02-23].

[153] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User

Space. In *Proceedings of the 27th USENIX Security Symposium*, Security '18, pages 973–990, Baltimore, MD, USA, 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[154] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1607–1619, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813690. URL https://doi.org/10.1145/2810103.2813690.

[155] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*, HPCA '17, pages 529–540, Austin, TX, USA, 2017. IEEE Computer Society. ISBN 978-1-5090-4985-1. doi: 10.1109/HPCA.2017.18. URL https://doi.org/10.1109/HPCA.2017.18.

[156] Zhengyang Liu and John Criswell. Flexible and Efficient Memory Object Metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM '17, pages 36–46, Barcelona, Spain, 2017. ACM. ISBN 978-1-4503-5044-0. doi: 10.1145/3092255.3092268. URL https://doi.org/10.1145/3092255.3092268.

[157] LLVM. "libc++" C++ Standard Library, 2021. URL https://libcxx.llvm.org. [Accessed on: 2023-01-30].

[158] LLVM. "libc++abi" C++ Standard Library Support, 2021. URL https://libcxxabi.llvm.org. [Accessed on: 2023-01-30].

[159] LLVM. libunwind LLVM Unwinder, 2021. URL https://bcain-llvm.readthedocs.io/projects/libunwind. [Accessed on: 2023-01-30].

[160] LLVM. llvm::RandomNumberGenerator Class Reference, 2022. URL https://llvm.org/doxygen/classllvm_1_1RandomNumberGenerator.html. [Accessed on: 2022-04-28].

[161] LLVM. "compiler-rt" runtime libraries, 2022. URL https://compiler-rt.llvm.org. [Accessed on: 2023-01-30].

[162] LLVM. lib/Target/ARM/ARMConstantIslandPass.cpp File Reference, 2023. URL https://llvm.org/doxygen/ARMConstantIslandPass_8cpp.html. [Accessed on: 2023-02-25].

[163] LLVM. lib/CodeGen/IndirectBrExpandPass.cpp File Reference, 2023. URL https://llvm.org/doxygen/IndirectBrExpandPass_8cpp.html. [Accessed on: 2023-02-25].

[164] LLVM. LLD - The LLVM Linker, 2023. URL https://lld.llvm.org. [Accessed on: 2023-01-30].

[165] LLVM. LLVM Language Reference Manual, 2023. URL https://llvm.org/docs/LangRef.html. [Accessed on: 2023-02-25].

[166] LLVM. llvm::LivePhysRegs Class Reference, 2023. URL https://llvm.org/doxygen/classllvm_1_1LivePhysRegs.html. [Accessed on: 2023-02-25].

[167] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 280–291, Denver, CO, USA,

2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813694. URL https://doi.org/10.1145/2810103.2813694.

[168] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, 2016. Internet Society. ISBN 1-891562-41-X. doi: 10.14722/ndss.2016.23173. URL http://doi.org/10.14722/ndss.2016.23173.

[169] Lan Luo, Xinhui Shao, Zhen Ling, Huaiyu Yan, Yumeng Wei, and Xinwen Fu. fASLR: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems. *IEEE Internet of Things Journal*, 9(18):17120–17135, September 2022. doi: 10.1109/JIOT.2022.3190374. URL https://doi.org/10.1109/JIOT.2022.3190374.

[170] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813676. URL https://doi.org/10.1145/2810103.2813676.

[171] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*, Security '06, pages 209–224, Vancouver, BC. Canada, 2006. USENIX Association. URL https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture.

[172] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*,

ATC '96, San Diego, CA, USA, 1996. USENIX Association. URL https://www.usenix.org/legacy/publications/library/proceedings/sd96/mcvoy.html.

[173] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems*, EuroSys '19, Dresden, Germany, 2019. ACM. ISBN 978-1-4503-6281-8. doi: 10.1145/3302424.3303946. URL https://doi.org/10.1145/3302424.3303946.

[174] Microchip. *32-bit Microcontroller Families: Industry's Broadest and Most Innovative 32-bit MCU Portfolio*. Microchip, August 2020. DS30009904V.

[175] Microsoft. Azure Sphere, 2023. URL https://azure.microsoft.com/en-us/services/azure-sphere. [Accessed on: 2023-02-25].

[176] Microsoft Azure. Azure Virtual Machines with Ampere Altra Arm–based processors—generally available, 2022. URL https://azure.microsoft.com/en-us/blog/azure-virtual-machines-with-ampere-altra-arm-based-processors-generally-available. [Accessed on: 2023-04-07].

[177] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory Safety for Embedded Devices with nesCheck. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 127–139, Abu Dhabi, United Arab Emirates, 2017. ACM. ISBN 978-1-4503-4944-4. doi: 10.1145/3052973.3053014. URL https://doi.org/10.1145/3052973.3053014.

[178] Charlie Miller and Chris Valasek. A Survey of Remote Automotive Attack Surfaces. In *Black Hat USA*. Las Vegas, NV, USA, 2014.

[179] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, USA, 2015. Internet Society. ISBN 1-891562-38-X. doi: 10.14722/ndss.2015.23271. URL https://doi.org/10.14722/ndss.2015.23271.

[180] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, Dublin, Ireland, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542504. URL https://doi.org/10.1145/1542476.1542504.

[181] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, Toronto, ON, Canada, 2010. ACM. ISBN 978-1-4503-0054-4. doi: 10.1145/1806651.1806657. URL https://doi.org/10.1145/1806651.1806657.

[182] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pages 157–171, Lausanne, Switzerland, 2020. ACM. ISBN 978-1-4503-7554-2. doi: 10.1145/3381052.3381328. URL https://doi.org/10.1145/3381052.3381328.

[183] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, Portland, OR, USA, 2002. ACM. ISBN 1-581-13450-9. doi:

10.1145/503272.503286. URL https://doi.org/10.1145/503272.503286.

[184] Newlib. Newlib, 1991. URL https://sourceware.org/newlib. [Accessed on: 2022-04-28].

[185] Ben Niu and Gang Tan. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 199–210, Berlin, Germany, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516649. URL https://doi.org/10.1145/2508859.2516649.

[186] Ben Niu and Gang Tan. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587, Edinburgh, UK, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594295. URL https://doi.org/10.1145/2594291.2594295.

[187] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813644. URL https://doi.org/10.1145/2810103.2813644.

[188] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584. ACM, 2010. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866371. URL https://doi.org/10.1145/1866307.1866371.

[189] NXP. *UM11159 User Manual: i.MX RT685 Evaluation Board User Manual*. NXP, June 2021. Rev. 2.

[190] NXP. *UM11147 User Manual: RT6xx User Manual*. NXP, September 2022. Rev. 1.5.

[191] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '17, pages 259–284, Atlanta, GA, USA, 2017. Springer-Verlag. ISBN 978-3-319-66332-6. doi: 10.1007/978-3-319-66332-6_12. URL https://doi.org/10.1007/978-3-319-66332-6_12.

[192] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *Proceedings of the 25th USENIX Security Symposium*, Security '16, pages 121–138, Austin, TX, USA, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/oikonomopoulos.

[193] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7, November 1996. URL http://www.phrack.org/issues/49/14.html.

[194] Oracle Cloud Infrastructure. Ampere A1 Compute, 2023. URL https://www.oracle.com/cloud/compute/arm. [Accessed on: 2023-04-07].

[195] James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *arXiv preprint arXiv:1308.5174*, August 2013. URL https://doi.org/10.48550/arXiv.1308.5174.

[196] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and*

Privacy, SP '12, pages 601–615, San Francisco, CA, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.41. URL https://doi.org/10.1109/SP.2012.41.

[197] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium*, Security '13, pages 447–462, Washington, DC, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas.

[198] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC '19, pages 241–254, Renton, WA, USA, 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/park-soyeon.

[199] Sergio Pastrana, Juan Tapiador, Guillermo Suarez-Tangil, and Pedro Peris-López. AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '16, pages 58–77, San Sebastián, Spain, 2016. Springer-Verlag. ISBN 978-3-319-40666-4. doi: 10.1007/978-3-319-40667-1_4. URL https://doi.org/10.1007/978-3-319-40667-1_4.

[200] PaX Team. Non-Executable Pages Design & Implementation, 2000. URL https://pax.grsecurity.net/docs/noexec.txt. [Accessed on: 2022-04-28].

[201] PaX Team. Address Space Layout Randomization, 2001. URL https://pax.grsecurity.net/docs/aslr.txt. [Accessed on: 2022-04-28].

[202] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '15, pages 144–164, Milan, Italy, 2015. Springer-Verlag. ISBN 978-3-319-20550-2. doi: 10.1007/978-3-319-20550-2_8. URL https://doi.org/10.1007/978-3-319-20550-2_8.

[203] Jannik Pewny and Thorsten Holz. Control-Flow Restrictor: Compiler-Based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 309–318, New Orleans, LA, USA, 2013. ACM. ISBN 978-1-4503-2015-3. doi: 10.1145/2523649.2523674. URL https://doi.org/10.1145/2523649.2523674.

[204] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. Breaking and Fixing Destructive Code Read Defenses. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC '17, pages 55–67, Orlando, FL, USA, 2017. ACM. ISBN 978-1-4503-5345-8. doi: 10.1145/3134600.3134626. URL https://doi.org/10.1145/3134600.3134626.

[205] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys '17, pages 420–436, Belgrade, Serbia, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064216. URL https://doi.org/10.1145/3064176.3064216.

[206] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Practical Fine-Grained Binary Code Randomization. In *Proceedings of the 36th Annual Computer Se-*

curity Applications Conference, ACSAC '20, pages 401–414, Austin, TX, USA, 2020. ACM. ISBN 978-1-4503-8858-0. doi: 10.1145/3427228.3427292. URL https://doi.org/10.1145/3427228.3427292.

[207] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP '20, pages 563–577, San Francisco, CA, USA, 2020. IEEE Computer Society. ISBN 978-1-7281-3497-0. doi: 10.1109/SP40000.2020.00041. URL https://doi.org/10.1109/SP40000.2020.00041.

[208] Qualcomm. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions. White paper, Qualcomm Technologies, Inc., January 2017. URL https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[209] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIACCS '20, pages 494–505, Taipei, China, 2020. ACM. ISBN 978-1-4503-6750-9. doi: 10.1145/3320269.3384757. URL https://doi.org/10.1145/3320269.3384757.

[210] Renesas. *RA Family Brochure*. Renesas, May 2022. Document No. R01CP0035EJ0300.

[211] RISC-V. *RISC-V External Debug Support*. RISC-V Foundation, March 2019. Document Version 0.13.2.

[212] RISC-V. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V Foundation, December 2021. Document Version 20211203.

[213] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012. ISSN 1094-9224. doi: 10.1145/2133375.2133377. URL https://doi.org/10.1145/2133375.2133377.

[214] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 2017 Network and Distributed System Security Symposium*, NDSS '17, San Diego, CA, USA, 2017. Internet Society. ISBN 1-891562-46-0. doi: 10.14722/ndss.2017.23477. URL http://doi.org/10.14722/ndss.2017.23477.

[215] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Network and Distributed System Security Symposium*, NDSS '04, San Diego, CA, USA, 2004. Internet Society. ISBN 1-891562-18-5. URL https://www.ndss-symposium.org/ndss2004/practical-dynamic-buffer-overflow-detector.

[216] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. Security and Privacy Challenges in Industrial Internet of Things. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, pages 54:1–54:6, San Francisco, CA, USA, 2015. ACM. ISBN 978-1-4503-3520-1. doi: 10.1145/2744769.2747942. URL https://doi.org/10.1145/2744769.2747942.

[217] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 546–558, Virtual

Event, 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446731. URL https://doi.org/10.1145/3445814.3446731.

[218] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium*, Security '20, pages 1677–1694, Boston, MA, USA, 2020. USENIX Association. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel.

[219] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 745–762, San Jose, CA, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.51. URL https://doi.org/10.1109/SP.2015.51.

[220] Simon Segars. Arm Partners Have Shipped 200 Billion Chips, 2021. URL https://www.arm.com/blogs/blueprint/200bn-arm-chips. [Accessed on: 2023-04-25].

[221] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium*, Security '10, Washington, DC, USA, 2010. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity10/adapting-software-fault-isolation-contemporary-cpu-architectures.

[222] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12, pages 309–318, Boston, MA, USA, 2012. USENIX Association. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[223] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, Alexandria, VA, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313. URL https://doi.org/10.1145/1315245.1315313.

[224] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, Washington, DC, USA, 2004. ACM. ISBN 1-58113-961-6. doi: 10.1145/1030083.1030124. URL https://doi.org/10.1145/1030083.1030124.

[225] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, Phoenix, AZ, USA, 2019. ACM. ISBN 978-1-4503-7226-8. doi: 10.1145/3337167.3337175. URL https://doi.org/10.1145/3337167.3337175.

[226] Zhuojia Shen and John Criswell. InversOS: Efficient Control-Flow Protection for AArch64 Applications with Privilege Inversion. *arXiv preprint*

arXiv:2304.08717, April 2023. URL https://doi.org/10.48550/arXiv.2304.08717.

[227] Zhuojia Shen, Komail Dharsee, and John Criswell. Fast Execute-Only Memory for Embedded Systems. In *Proceedings of the 2020 IEEE Secure Development Conference*, SecDev '20, pages 7–14, Atlanta, GA, USA, 2020. IEEE Computer Society. ISBN 978-1-7281-8388-6. doi: 10.1109/SecDev45635.2020.00017. URL https://doi.org/10.1109/SecDev45635.2020.00017.

[228] Zhuojia Shen, Komail Dharsee, and John Criswell. Randezvous: Making Randomization Effective on MCUs. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, pages 28–41, Austin, TX, USA, 2022. ACM. ISBN 978-1-4503-9759-9. doi: 10.1145/3564625.3567970. URL https://doi.org/10.1145/3564625.3567970.

[229] Jiameng Shi, Le Guan, Wenqiang Li, Dayou Zhang, Ping Chen, and Ping Chen. HARM: Hardware-Assisted Continuous Re-randomization for Microcontrollers. In *Proceedings of the 2022 IEEE European Symposium on Security and Privacy*, EuroSP '22, pages 520–536, Genoa, Italy, 2022. IEEE Computer Society. ISBN 978-1-6654-1614-6. doi: 10.1109/EuroSP53844.2022.00039. URL https://doi.org/10.1109/EuroSP53844.2022.00039.

[230] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, San Francisco, CA, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.45. URL https://doi.org/10.1109/SP.2013.45.

[231] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*. Amsterdam, Netherlands, 2007.

[232] Standard Performance Evaluation Corporation. SPEC CPU®2017, 2022. URL https://www.spec.org/cpu2017. [Accessed on: 2023-01-30].

[233] STMicroelectronics. *AN4701 Application Note: Proprietary Code Read-Out Protection on Microcontrollers of the STM32F4 Series*. STMicroelectronics, November 2016. DocID027893 Rev 3.

[234] STMicroelectronics. *RM0386 Reference Manual: STM32F469xx and STM32F479xx Advanced Arm®-Based 32-Bit MCUs*. STMicroelectronics, June 2018. RM0386 Rev 5.

[235] STMicroelectronics. *PM0214 Programming Manual: STM32 Cortex®-M4 MCUs and MPUs Programming Manual*. STMicroelectronics, March 2020. PM0214 Rev 10.

[236] STMicroelectronics. *DS11189 Datasheet: STM32F469xx*. STMicroelectronics, May 2021. DS11189 Rev 7.

[237] STMicroelectronics. *UM1725 User Manual: Description of STM32F4 HAL and Low-Layer Drivers*. STMicroelectronics, June 2021. UM1725 Rev 7.

[238] STMicroelectronics. *AN4230 Application Note: STM32 Microcontroller Random Number Generation Validation Using the NIST Statistical Test Suite*. STMicroelectronics, July 2022. Rev 7.

[239] STMicroelectronics. *UM1932 User Manual: Discovery Kit with STM32F469NI MCU*. STMicroelectronics, March 2022. UM1932 Rev 4.

[240] STMicroelectronics. *DS12469 Datasheet: STM32L412xx*. STMicroelectronics, December 2022. DS12469 Rev 9.

[241] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the Memory Secrecy Assumption. In *Proceedings of the 2nd European Workshop on System Security*, EuroSec '09, pages 1–8, Nuremburg, Germany, 2009. ACM. ISBN 978-1-60558-472-0. doi: 10.1145/1519144.1519145. URL https://doi.org/10.1145/1519144.1519145.

[242] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, Boston, MA, USA, 4th edition, 2013. ISBN 978-0-321-56384-2.

[243] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pages 143–156, Lausanne, Switzerland, 2020. ACM. ISBN 978-1-4503-7554-2. doi: 10.1145/3381052.3381326. URL https://doi.org/10.1145/3381052.3381326.

[244] Igor Sysoev et al. nginx, 2022. URL https://nginx.org/en. [Accessed on: 2023-01-30].

[245] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, San Francisco, CA, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.13. URL https://doi.org/10.1109/SP.2013.13.

[246] Texas Instruments, Inc. HALCOGEN: Hardware Abstraction Layer Code Generator for Hercules MCUs, 2023. URL https://www.ti.com/tool/HALCOGEN. [Accessed on: 2023-02-25].

[247] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow

Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 941–955, San Diego, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[248] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, RAID '11, pages 121–141, Menlo Park, CA, USA, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_7. URL https://doi.org/10.1007/978-3-642-23644-0_7.

[249] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 1221–1238, Santa Clara, CA, USA, 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[250] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, Denver, CO, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813673. URL https://doi.org/10.1145/2810103.2813673.

[251] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough `call`: Mitigating Advanced Code-Reuse Attacks at the

Binary Level. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 934–953, San Jose, CA, USA, 2016. IEEE Computer Society. ISBN 978-1-5090-0824-7. doi: 10.1109/SP.2016.60. URL https://doi.org/10.1109/SP.2016.60.

[252] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting Software with Code-Centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 469–480, Minneapolis, MN, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4394-4. doi: 10.1109/ISCA.2014.6853202. URL https://doi.org/10.1109/ISCA.2014.6853202.

[253] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the 17th European Conference on Computer Systems*, EuroSys '22, pages 266–282, Rennes, France, 2022. ACM. ISBN 978-1-4503-9162-7. doi: 10.1145/3492321.3519560. URL https://doi.org/10.1145/3492321.3519560.

[254] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, Asheville, NC, USA, 1993. ACM. ISBN 0-89791-632-8. doi: 10.1145/168619.168635. URL https://doi.org/10.1145/168619.168635.

[255] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-Flow Integrity for Real-Time Embedded Systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, ECRTS '19, pages 2:1–2:24, Stuttgart, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum füer Informatik. ISBN 978-3-95977-110-8. doi:

10.4230/LIPIcs.ECRTS.2019.2. URL https://doi.org/10.4230/LIPIcs.ECRTS.2019.2.

[256] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, pages 331–340, Los Angeles, CA, USA, 2015. ACM. ISBN 978-1-4503-3682-6. doi: 10.1145/2818000.2818017. URL https://doi.org/10.1145/2818000.2818017.

[257] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, pages 7–12, Heraklion, Greece, 2020. ACM. ISBN 978-1-4503-7523-8. doi: 10.1145/3380786.3391398. URL https://doi.org/10.1145/3380786.3391398.

[258] Yu Wang, Jinting Wu, Tai Yue, Zhenyu Ning, and Fengwei Zhang. RetTag: Hardware-Assisted Return Address Integrity on RISC-V. In *Proceedings of the 15th European Workshop on Systems Security*, EuroSec '22, pages 50–56, Rennes, France, 2022. ACM. ISBN 978-1-4503-9255-6. doi: 10.1145/3517208.3523758. URL https://doi.org/10.1145/3517208.3523758.

[259] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. ReRanz: A Light-Weight Virtual Machine to Mitigate Memory Disclosure Attacks. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 143–156, Xi'an, China, 2017. ACM. ISBN 978-1-4503-4948-2. doi: 10.1145/3050748.3050752. URL https://doi.org/10.1145/3050748.3050752.

[260] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-Randomization. In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 1239–1256, Santa Clara, CA, USA, 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/wang.

[261] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *Proceedins of the 2020 IEEE Symposium on Security and Privacy*, SP '20, pages 592–607, San Francisco, CA, USA, 2020. IEEE Computer Society. ISBN 978-1-7281-3497-0. doi: 10.1109/SP40000.2020.00087. URL https://doi.org/10.1109/SP40000.2020.00087.

[262] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Oakland, CA, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-6895-9. doi: 10.1109/SP.2010.30. URL https://doi.org/10.1109/SP.2010.30.

[263] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, Raleigh, NC, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382216. URL https://doi.org/10.1145/2382196.2382216.

[264] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-Based Control-Flow Protection for IoT Devices. In *Proceedings of the*

2018 IEEE European Symposium on Security and Privacy, EuroSP '18, pages 214–226, London, UK, 2018. IEEE Computer Society. ISBN 978-1-5386-4228-3. doi: 10.1109/EuroSP.2018.00023. URL https://doi.org/10.1109/EuroSP.2018.00023.

[265] Wikipedia. Comparison of ARM processors, 2023. URL https://en.wikipedia.org/wiki/Comparison_of_ARM_processors#ARMv8-A. [Accessed on: 2023-02-23].

[266] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 367–382, Savannah, GA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king.

[267] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '02, pages 304–316, San Jose, CA, USA, 2002. ACM. ISBN 1-58113-574-2. doi: 10.1145/605397.605429. URL https://doi.org/10.1145/605397.605429.

[268] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 31–44, Brighton, UK, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095814. URL https://doi.org/10.1145/1095810.1095814.

[269] XAMPPRocky et al. Tokei: Count your code, quickly, 2021. URL https://github.com/XAMPPRocky/tokei. [Accessed on: 2022-09-20].

[270] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, Boston, MA, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1625-5. doi: 10.1109/DSN.2012.6263958. URL https://doi.org/10.1109/DSN.2012.6263958.

[271] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 2989–3002, Los Angeles, CA, USA, 2022. ACM. ISBN 978-1-4503-9450-5. doi: 10.1145/3548606.3559344. URL https://doi.org/10.1145/3548606.3559344.

[272] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Oakland, CA, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.25. URL https://doi.org/10.1109/SP.2009.25.

[273] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. In *Proceedings of the 31st USENIX Security Symposium*, Security '22, pages 89–106, Boston, MA, USA, 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/yoo.

[274] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '15, pages 66–85, Kyoto, Japan, 2015. Springer-Verlag. ISBN 978-3-319-26362-5. doi: 10.1007/978-3-319-26362-5_4. URL https://doi.org/10.1007/978-3-319-26362-5_4.

[275] Peiter Zatko. How to Write Buffer Overflows, 1995. URL https://insecure.org/stf/mudge_buffer_overflow_tutorial.html. [Accessed on: 2023-04-25].

[276] Bin Zeng, Gang Tan, and Greg Morrisett. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 29–40, Chicago, IL, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046713. URL https://doi.org/10.1145/2046707.2046713.

[277] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *Proceedings of the 22nd USENIX Security Symposium*, Security '13, pages 369–382, Washington, DC, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/zeng.

[278] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, San Francisco, CA, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.44. URL https://doi.org/10.1109/SP.2013.44.

[279] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*, Security '13, pages 337–352, Washington, DC, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.

[280] Mingwei Zhang, Ravi Sahita, and Daiping Liu. XOM-Switch: Hiding Your Code from Advanced Code Reuse Attacks in One Shot. In *Black Hat Asia*. Singapore, Republic of Singapore, 2018.

[281] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 631–644, Providence, RI, USA, 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304017. URL https://doi.org/10.1145/3297858.3304017.

[282] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In *Proceedings of the 29th USENIX Security Symposium*, Security '20, pages 1219–1236, Boston, MA, USA, 2020. USENIX Association. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie.

[283] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-Based Fault Isolation for ARM. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS '14, pages 558–569, Scottsdale, AZ, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660344. URL https://doi.org/10.1145/2660267.2660344.

[284] Philipp Zieris and Julian Horsch. A Leak-Resilient Dual Stack Scheme for

Backward-Edge Control-Flow Integrity. In *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 369–380, Incheon, Republic of Korea, 2018. ACM. ISBN 978-1-4503-5576-6. doi: 10.1145/3196494.3196531. URL https://doi.org/10.1145/3196494.3196531.