The Pennsylvania State University

The Graduate School

ENFORCING EXECUTION INTEGRITY FOR SOFTWARE SYSTEMS

A Dissertation in

Computer Science and Engineering

by

Xinyang Ge

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

December 2016

The dissertation of Xinyang Ge was reviewed and approved* by the following:

Trent R. Jaeger
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Anand Sivasubramaniam
Professor of Computer Science and Engineering

Gang Tan
Associate Professor of Computer Science and Engineering

Danfeng Zhang
Assistant Professor of Computer Science and Engineering

Peng Liu
Professor of Information Science and Technology

Mathias Payer
Assistant Professor of Computer Science at Purdue
Special Member

Mahmut Kandemir
Professor of Computer Science and Engineering
Graduate Program Chair for Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Memory corruption bugs have been the most common cause of security vulnerabilities for decades. Adversaries exploit such program flaws to launch code-injection and code-reuse attacks. Code-injection attacks execute instructions that are injected by an adversary at runtime while code-reuse attacks redirect the execution to existing code of her choice. Both attacks can lead to severe security breaches because they often enable arbitrary code execution. To prevent adversaries from exploiting such vulnerabilities, researchers have proposed mitigation techinques such as data execution prevention (DEP), which prohibits execution over data memory hence injected code, and control-flow integrity (CFI), which limits adversaries' choices when reusing existing code. Despite the effectiveness of these defenses, they can be circumvented by a compromised operating system kernel, which has full privileges over the running system and is capable of negating the deployed defenses. In this dissertation, we explore solutions to protecting an entire software system with both DEP and CFI, which we refer as *execution integrity*. We take three steps to approach this goal. First, we propose a lightweight system to enforce DEP for the operating system kernel from an isolated computation environment. We mediate the kernel's memory management operations and ensure they always comply with a set of general principles for lifetime code integrity. Second, we demonstrate a systematic approach to enforce fine-grained CFI for the operating system kernel comprehensively and efficiently. We find that fine-grained CFI can be enforced as efficiently as (or even outperform) comparable coarse-grained CFI. Third, we examine the effectiveness of applying recent hardware mechanisms that log complete control-flow traces to enforce CFI system-wide. We construct an online CFI enforcement system that is capable of enforcing various types of CFI policies over all running, unmodified user-space applications. We intensively optimize the implementation to maximize the performance using the current hardware, and study alternative hardware logging schemes for further performance improvements. The dissertation shows that execution integrity can be enforced efficiently, flexibly and comprehensively, thus is practical to mitigate memory corruption attacks for the whole software system.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would express my deepest gratitude towards my advisor Trent Jaeger. Trent leads me through this wonderful journey and makes me who I am today. From him, I learnt how to think big. He always allows me to freely explore areas that are of interest to me, even for those he does not have past experiences with. I truly appreciate his generosity for being willing to take on risks and learn new things along with me. I feel fortunate to be one of his students.

I am grateful to my dissertation committee. Anand Sivasubramaniam offers me many insightful suggestions from systems' perspective to the dissertation. Gang Tan is an expert in programming language, and we have many fruiteful discussions that substantially changed my understandings on several topics in this area. Danfeng Zhang shares many of his detailed experiences on giving presentations with me. Peng Liu shows me how nice a professor can ever become. Finally, I have collaborated with Mathias Payer on several projects, and he is always willing to spend time and offer detailed and constructive suggestions for improving our work. I thank all of you for serving on my committee.

I want to reserve my special thanks to my great mentor Weidong Cui. From him, I learnt how to be courageous to solve challenging problems. He closely works with me and shows me how enjoyable and exciting it can become to explore in the darkness searching for solutions to a problem that I originally thought was not even possible.

I would also thank my lab mates for their accompany for many days and nights: Hayawardh Vijayakumar, Divya Muthukumaran, Stephen McLaughlin, Devin Pohley, Nirupama Talele, Yuqiong Sun, Giuseppe Petracca, Frank Capobianco, and Berkey Celik. Particularly, I worked with Hayawardh when I started in Penn State. He taught me specific things in doing research, and I have been benefiting from them ever since.

Finally, I want to thank my parents Peili Yang and Shien Ge, and my important life partner Yu Pu for their selfless and unconditional supports. They give me strength when I feel weak. I cannot imagine the life without them.

Dedication

感谢父母给予我生命，让我体会这美好纷繁的世界。

Chapter 1

Introduction

1.1 Overview

System programming languages such as C/C++ are widely used to implement critical software. While these languages offer performance benefits by delegating memory management to programmers, they also invite memory corruption bugs to programs. As a result, adversaries exploit these bugs to redirect programs' control flows to injected code and/or existing code for arbitrary code execution. People have proposed defenses to prevent exploitation of memory corruption, such as data execution prevention (DEP) [1] to mitigate code-injection attacks and control-flow integrity (CFI) [2] to mitigate code-reuse attacks. DEP sets memory pages as either writable or executable, preventing an adversary from executing code that is written at runtime. CFI restricts a program's execution to its control-flow graph (CFG), limiting the choices an adversary has when reusing existing code.

Despite the proposed strong defenses, they can be circumvented by a compromised operating system kernel in practice. This is because an operating system kernel is in the trusted computing base (TCB) in almost all commodity systems, hence a kernel vulnerability can undermine the security of an entire system. Researchers have demonstrated that, by controlling the return value of system calls, a compromised kernel can

cause memory corruption in almost any application [3] even under the deployment of a strong reverse sandbox [4, 5].

Given the importance of the operating system kernel, researchers have proposed techniques to protect the kernel space with both DEP and CFI. NICKLE [6] and SecVisor [7] leverage the more privileged virtual machine monitor (VMM) to trap the guest kernel's MMU operations and build their enforcement of kernel code integrity inside the VMM. However, the use of a VMM increases the trusted code running on the system. Several VMM vulnerabilities have been found that enable attacks against the VMM itself [8]. Additionally, researchers have also brought CFI to the kernel space. KCoFI [9] implements the complete control-flow integrity for a conventional operating system kernel (FreeBSD). However, the implementation requires non-trivial manual effort to port the kernel that has millions of SLoC to a lower-level software layer (Secure Virtual Architecture [10]). This not only thwarts the real-world adoption in practice, but also incurs significant performance slowdown. The CFI policy enforced by KCoFI is also too coarse-grained to prevent recent, advanced code-reuse attacks [11, 12, 13]. Finally, although researchers have explored techniques to protect the kernel alone, few have looked at utilizing the protected kernel as a foundation for offering further security protections over the user-space applications. The operating system is capable of integrating various defenses to offer strong yet flexible user-level protections to balance the security and performance.

In the dissertation, we aim at enforcing both DEP and strong CFI (referred as *execution integrity* when combined) for the operating system kernel, and further building an operating system mechanism to extend the same defense to the entire user space flexibly and efficiently. Achieving such a goal is not without challenges. First, given current VMM-based systems for enforcing kernel code integrity, it is crucial to minimize checks during kernel execution and enforce simple and verifiable policies with as small a runtime as possible with low overheads. Second, to protect the operating system with strong CFI, we must be able to compute a fine-grained CFG for the kernel. In addition, the kernel has to handle asynchronous system events (e.g., interrupts, system calls) that affect its control flows in complicated ways, so efficiently enforcing the CFG

despite these system events is key. Third, to protect the user space with execution integrity comprehensively, the system must offer flexible enforcement. It must support legacy programs that do not have protections (e.g., CFI) built in place, and enable customized enforcement by supporting different policies for the same programs to strike the balance between security and performance based on the deployment's need.

We tackle these challenges with the following insights. First, we propose five simple invariants for restricting the kernel's MMU operations to enforce lifetime kernel code integrity. We implement a novel instruction-level instrumentation primitive to mediate such MMU operations on the TrustZone architecture [14] to enforce the invariants. We leverage the isolation directly enforced by the TrustZone hardware to reduce the size of TCB significantly compared to VMM-based systems. Second, we find that operations on function pointers are often limited to assignment and dereference in the kernel source code. Based on this observation, we design a simple static taint analysis to track how function pointers are propagated in the kernel and compute its fine-grained CFG accordingly. To enforce the fine-grained CFG, we eliminate unpredictable context switches that may occur in the kernel space by configuring the kernel to run non-preemptively, and further propose four system invariants to restrict how the kernel can be entered and exited. Finally, to construct a flexible enforcement mechanism, we leverage the recent hardware feature called Intel Processor Trace (PT) [15] to log the control-flow trace of user-space applications and compare the execution to a separately specified CFI policy for enforcement. This separation between the program execution and CFI enforcement enables us to enforce arbitrary policies for unmodified programs, and even change the enforced policy at runtime.

1.2 Towards a System with Execution Integrity

We cast the goal of enforcing whole-system execution integrity to two steps. The first step is to enforce execution integrity for the operating system kernel. Protecting the kernel space presents additional challenges for both code integrity, because the kernel has full privilege over all system resources including the MMU, and control-flow

integrity, because the kernel handles asynchronous system events (e.g., interrupts) that can change the runtime control flows in complicated ways. The second step is to enforce execution integrity for the user space. Enforcing code integrity for most user-space applications can be trivially done from the kernel by restricting the use of system calls that change page permissions (e.g., `mprotect`). However, in terms of control-flow integrity, while many research proposals focus on strong and efficient protection over user-level applications, practically achieving such protections system wide requires solving additional challenges, such as supporting legacy programs and customized enforcement to meet the deployment's needs.

1.3 Challenges to Whole-System Execution Integrity

Ideally, we should minimize the amount of trusted code running on a system and enforce the most constrained execution integrity for both user space and kernel space efficiently and flexibly. We classify the challenges based on whether they are applied to kernel space or user space. We first discuss the challenges of enforcing execution integrity for kernel space.

- **Minimal TCB.** To enforce kernel code integrity for conventional operating systems, a system must intercept *all* MMU operations that either update the page tables in place or switch to a different set of page tables. Existing approaches leverage the heavyweight hypervisors. *The key challenge here is to achieve a full mediation of such operations with a minimal tamper-proof runtime.*

- **Fine-grained CFI.** The security of CFI depends on the enforced CFG. Typically, the more fine-grained the enforced CFG is, the fewer choices an adversary will have when hijacking control flows. Kernel software is complicated and has assembly code. *The key challenge here is to compute a minimal set of targets for each indirect branch in the kernel.*

- **Lightweight protection over system events.** System events like interrupts and exceptions can change the kernel's runtime control flows in complicated ways.

For instance, the kernel leverages page faults to implement efficient user-kernel memory copy. We must restrict where the page fault handling can return while still allowing the kernel to function properly. *The key challenge here is to come up with a systematic approach to handle these events in a lightweight and efficient way.*

Next, we discuss the challenges for user space protection.

- **Supporting a wide range of CFI policies.** Our protection over user-space applications should support a wide range of CFI policies such as coarse-grained, fine-grained, and stateful policies (e.g., shadow stack) to allow tradeoffs between security and performance for deployment. *The key challenge here is to design one general mechanism that supports arbitrary CFI policies as efficiently as possible.*

- **Flexible enforcement.** Typical CFI systems inline CFI checks into the protected programs for effective enforcement. This prevents the deployment from choosing the right defense for different programs to balance the security and performance after they are shipped. A system-wide protection mechanism should grant a user the flexibility to choose the desired protection over her selected programs at load time or even change the type of protection at runtime. *The key challenge here is to separate CFI policies from their enforcement.*

- **Supporting unmodified binaries.** Most software systems come with pre-compiled programs that do not have CFI protection in place. To have a comprehensive protection over all running programs on a system, *the key challenge here is to support both new and legacy programs.*

1.4 Thesis Statement

> *Execution integrity can be enforced efficiently, flexibly and comprehensively for software systems.*

We find that to satisfy the thesis requires the following three efforts:

- In Chapter 3, we present SPROBES, a TrustZone-based instrumentation mechanism, to enforce lifetime kernel code integrity for a commodity operating system (e.g., Linux). The key insight behind this approach is to instrument the memory management operations in the commodity operating system kernel at runtime, and verify the validity of each operation in an isolated, trusted environment called TrustZone before allowing them to occur. Compared to the hypervisor-based solutions, the isolation of TrustZone is directly enforced by the processor hardware, thus SPROBES have a smaller trusted computing base (TCB). In addition, our evaluation shows that it has comparable performance to prior solutions.

- In Chapter 4, we present an approach for retrofitting kernel software that leverages features of such software to enable comprehensive, efficient, fine-grained CFI enforcement. We design a static taint analysis to derive a fine-grained CFG for kernel software by tracking how function pointers are propagated, and propose four invariants of the kernel execution to enable lightweight comprehensive CFI enforcement. We further optimize the CFI instrumentation based on the type and the target set size of indirect branches, and show that fine-grained CFI enforcement can be as efficient as (or even outperform) coarse-grained CFI.

- In Chapter 5, we present GRIFFIN, an operating system mechanism that is capable of enforcing various CFI policies for user-space applications without instrumenting the program binaries. We collect program's runtime control flows using the recent Intel Processor Trace (PT) feature, and design GRIFFIN to utilize the collected trace to execute CFI checks efficiently and in parallel. We evaluate a range of CFI policies, from the most efficient coarse-grained CFI policies to the most restrictive but less efficient stateful CFI policies (e.g., shadow stack), and find that GRIFFIN can protect user-space applications in a flexible manner with comparable performance to traditional software-based approaches. Finally, we examine alternative hardware logging schemes and show that GRIFFIN's performance can be further improved with more targeted logging approaches.

Chapter 2

Background

In this chapter, we first introduce memory corruption bugs. Then, we describe exploitations over these bugs. Finally, we discuss various mitigation techniques.

2.1 Memory Corruption

System programming languages such as C/C++ are widely used to implement critical software such as operating systems and browsers. These languages allow the programmers to manage the memory themselves by supporting arbitrary pointer operations. It offers performance benefits, but also makes the programs prone to memory corruption when a pointer is used to access an unintentional memory location (e.g., buffer overflow, use-after-free). An adversary can exploit these bugs to leak sensitive information and/or execute arbitrary code.

```
int main(int argc, char *argv[])
{
    char buf[256];
    strcpy(buf, argv[1]);
}
```

Figure 2.1. An example of stack-based buffer overflow

We show an example of the stack-based buffer overflow bug in Figure 2.1. The program allocates an 256-byte buffer on the stack and copies the command line argument into the buffer. However, the program does not check if its length exceeds the size of the

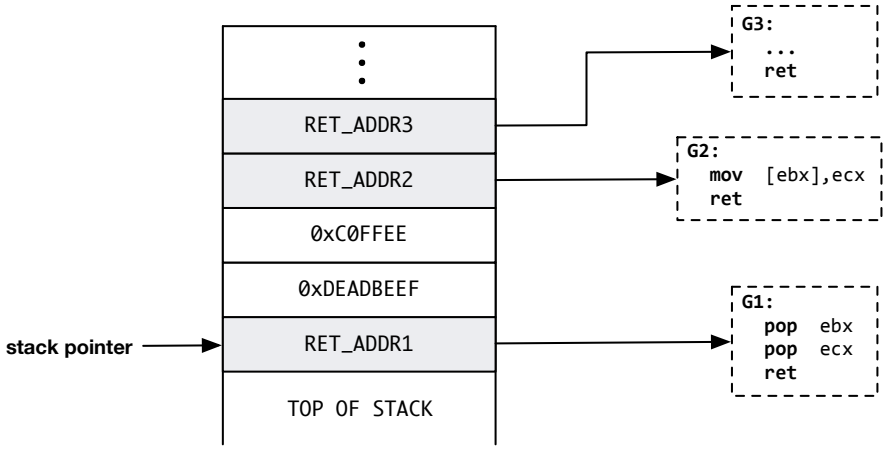

Figure 2.2. An example of return-oriented programming

allocated buffer. Consequently, by providing a well-crafted argument, an adversary can overwrite the return address on the stack and redirect the control flow to the location of her choosing. To further control the subsequent program behavior, the adversary can redirect the control flow either to her injected code (referred as *code-injection attacks*) or to existing code (referred as *code-reuse attacks*). Code-injection attacks can be mitigated by deployed defenses such as data execution prevention (DEP) (see Section 2.2).

In code-reuse attacks, the adversary executes existing code in an unintended way to perform desired actions. This is achieved by contiguously executing small pieces of code (called *gadget*) connected through indirect branches under her control. Researchers first proposed a generalized exploitation technique called *return-oriented programming* (ROP) based on function returns [16], then extended it to be capable of leveraging all indirect branches (e.g., indirect calls) [17, 18].

ROP is proven to be Turing-complete [16], and we show an example of ROP attack that writes an arbitrary value into a specified memory location in Figure 2.2. Suppose the stack layout after memory corruption is shown in the figure, and the stack pointer points to the overwritten `RET_ADDR1`. Once the return address is taken, the program execution will be redirected to `G1`. `G1` pops up value `0xDEADBEEF` into register `ebx` and value `0xC0FFEE` into register `ecx`, respectively, and increments the stack pointer by two words. When `G1` returns, the execution will be further redirected to `G2` through `RET_ADDR2`,

which was written by the adversary on memory corruption. `G2` stores the value in register `ecx` (`0xC0FFEE`) to the memory location specified by register `ebx` (`0xDEADBEEF`), and then continues the attack by executing subsequent code gadgets in a similar fashion.

2.2 Mitigations

In this section, we discuss various mitigation techniques that prevent an adversary from effectively exploiting memory corruption bugs such as buffer overflow and use after free. We also examine the deficiencies of these techniques when applied to both kernel space and/or user space to motivate the dissertation. We focus on code integrity and control-flow integrity in this section and present a more complete discussion on related work in Chapter 6.

2.2.1 Data Execution Prevention

Early software systems did not differentiate code from data due to the lack of executable bit in the page table, so a memory page is executable by default. This enables the adversary to inject the code into the address space at runtime [19] and hijack the control flow to execute injected code. To prevent the adversary from launching code-injection attacks, researchers propose the technique of data execution prevention (DEP) [1]. When DEP is deployed, a program memory page is mapped as either writable or executable, but never both. Therefore, writable data pages are no longer executable, defeating the adversary's attempt to execute her injected code at runtime.

The deployment of DEP turns adversary's attention to code-reuse attacks. Though code-reuse attacks have been shown as Turing-complete [16], the actual exploitation is more complex and restrictive than code-injection attacks due to the space limit to store a ROP chain and the availability of code gadgets. Therefore, many real-world exploits launch code-reuse attacks as the first stage, whose goal is to disable DEP and enable the second-stage code-injection attack [11].

While implementing DEP for the user space is straightforward, enforcing DEP for the kernel space is challenging. This is because the kernel has full privilege over all

system resources including the MMU, hence a kernel rootkit can subvert the deployed DEP by tampering with the memory protection settings (i.e., page tables). Previous systems [6, 7] leverage the more privileged hypervisor to enforce the code integrity for conventional operating systems. For example, SecVisor implements its own hypervisor to virtualize the physical memory, and trap the kernel's writes to code memory. However, the use of a VMM increases the size of TCB. And researchers have found many security vulnerabilities inside the VMMs [8].

2.2.2 Control-Flow Integrity

Control-Flow Integrity (CFI) is a technique to mitigate code-reuse attacks by restricting the execution of a program to its control-flow graph (CFG) [2]. CFI fundamentally limits attacks that need to hijack the program's runtime control flows as in traditional code-injection attacks and modern code-reuse attacks.

There are two types of branches in programs. Direct branches have fixed targets and hence are not controlled by an adversary. However, the targets of indirect branches are computed at runtime, thus they are susceptible to memory corruption attacks as demonstrated in Section 2.1. CFI focuses on restricting the targets of indirect control transfers at runtime to prevent adversaries from hijacking the control flows.

CFI techniques can be further classified as *coarse-grained* and *fine-grained*. In the enforced CFG, if all indirect branches share one or two sets of allowed targets, it is coarse-grained; otherwise (i.e., each indirect branch has its own set of allowed targets), it is fine-grained. Researchers have shown that sophisticated code-reuse attacks can bypass CFI with the use of CFI-compliant code gadgets [20, 13, 11]. This motivates the exploration of finer-grained CFI techniques to make exploitation harder [21, 22].

Current CFI systems instrument programs to inline checks for effective enforcement [21, 9, 23, 22]. For example, in the original CFI proposal [2], it assigns a *label* for each indirect branch and target. Before each indirect branch (e.g., function return), it inserts a check to compare the labels of the source and destination, and reports a violation if they do not match.

However, CFI enforcement methods using instrumentation have several deficien-

cies. First, once instrumentation is added to a program, the instrumentation is essentially hardcoded in the program. Although we aim to enforce a restrictive CFI policy for all programs, deployments may wish to balance security and performance, which is impractical for instrumented programs. Second, enforcing CFI over uninstrumented legacy programs can be a concern. While researchers have proposed various binary rewriting techniques, they may suffer from compatibility issues [24], imperfect disassembling [25, 26, 27] and/or incur high performance cost [28]. In addition, combining instrumented and uninstrumented binaries for a dynamically-linked program may also break functionalities or weaken security [22].

Researchers have proposed CFI enforcement for the operating system kernel to mitigate code-reuse attacks [29, 30, 31]. Enforcing restrictive CFI for the kernel space can be more challenging than for user space for two reasons. First, computing fine-grained CFG for the kernel can be non-trivial because of the prevalence of function pointers and assembly code. Second, the kernel has to handle asynchronous system events such as interrupts and system calls that change its runtime control flows in complicated ways.

Existing CFI implementations [9] for conventional operating system kernels have suffered from the following limitations. First, recent proposals have shown that code-reuse attacks are still possible against systems protected by coarse-grained CFI [11, 13, 12]. While researchers have proposed techniques to compute fine-grained CFI based on function pointer signatures [21, 22], these techniques may cause false violations when directly applied to the kernel because assembly functions do not often have a strict type and signature casts are common in the kernel code. Existing CFI implementations for operating system kernels are still coarse-grained. Second, a comprehensive CFI enforcement for the kernel must ensure that the handling of asynchronous system events do not divert the control flow in unexpected ways. Previous approaches propose to modify the kernel to run on a special middle layer (SVA [10]) and apply memory safety to critical data structures (e.g., thread context) to protect the return of the event handling. This requires manual porting efforts and incurs non-trivial performance slowdown.

2.2.3 Summary

To summarize, current defenses assume a large TCB for kernel code integrity, provide weak and expensive CFI protection for kernel space, and lack the deployment flexibility to choose desired defenses for different programs at different times when enforcing CFI system-wide, thus fail to enforce execution integrity effectively for the entire software system. We present a detailed assessment of related work in Chapter 6.

Chapter 3

Enforcing Kernel Code Integrity

We present our approach to enforce lifetime kernel code integrity based on the design of a novel introspection primitive called SPROBES for the ARM TrustZone architecture. Though SPROBES are implemented specifically for TrustZone, we note that the principles to mediate memory management operations for kernel code integrity are generally applicable.

ARM introduced hardware extensions for security called ARM TrustZone technology [14, 32] in ARMv6. Intuitively, TrustZone physically partitions all system resources (e.g., physical memory, peripherals, etc.) into two worlds: a *secure world* for security-sensitive resources and a *normal world* for conventional processing. TrustZone protects the secure world resources from the normal world, but the secure world can access resources in the normal world. This hardware separation protects the confidentiality and integrity of any computation in the secure world while permitting the secure world to view the normal world. The secure world is often used as an isolated trusted computing environment, however, it has not been used to implement a security monitor because of the lack of an effective introspection mechanism. Each world has full discretion over its own resources, and the normal world can use its resource (e.g., modify MMU) without mediation by the secure world.

We utilize the TrustZone extensions to develop SPROBES, a novel instrumentation mechanism that enables the secure world to cause the normal world to trap on arbitrary normal world instructions and provide an unforgeable view of the normal world's

processor state. This property of SPROBES helps facilitate monitoring over the normal world, as the secure world can choose the normal world instructions for which it wants to be notified, receive the current processor state of the normal world, and perform its desired monitoring actions before returning control to the normal world. Other than the placement of instrumentation, SPROBES are invisible to the normal world, so no changes to the operating system are required to utilize SPROBES.

We demonstrate SPROBES by developing a methodology for restricting the normal world's kernel execution to approved kernel code memory. To do this, we define a set of five invariants that when enforced imply that the supervisor mode of the normal world complies with the data execution prevention (DEP) invariant [1] for an approved set of immutable kernel code pages, even if a rootkit has control of the normal world kernel. We show that these invariants can be enforced comprehensively using only 12 SPROBES for the Linux 2.6.38 kernel. We find that each SPROBE hit causes 5611 instructions to be executed using the ARM Fast Models emulator, but that most SPROBES are never hit in normal execution and those that are hit are either those enforced in normal VM introspection or account for less than 2% of the instructions executed. As a by-product, SPROBES protect themselves from modification, even from a live rootkit, enabling the enforcement of richer security policies over the conventional operating system kernel.

3.1 Problem Definition

We assume that the kernel initially is enforcing DEP over an approved set of kernel code pages[1], but that an adversary may still be capable of executing some form of a ROP attack to launch their rootkit [29]. We observe that rootkits could compromise the integrity of kernel execution in the following ways. First, as DEP is the common technique used by the kernel to prevent code injection attacks, a rootkit can simply disable the DEP protection. For ARM processors, this is done by disabling the *Write eXecute-Never* (`WXN`) bit. When the `WXN` bit is set, writable pages are never executable, regardless of how the page table is configured. In this case, we assume that the kernel has the

[1]Note that even code pages may have read-only data embedded in them.

`WXN` bit set initially, but a rootkit may execute existing code, if available, to disable that protection, enabling the rootkit to inject code in the kernel.

Second, to bypass the DEP protection, rootkits can modify page table entries. Suppose the adversary wants to modify an initially read-only code page. Firstly, she alters the permission bits of that page from executable to writable, as they are mutually exclusive. Then, she may write to that page arbitrarily. Finally, she changes the permission of that page from writable back to executable.

Third, an alternative approach from modifying page table entries in place is to duplicate a page table elsewhere and reset the page table base (e.g., `TTBR` on ARM and `CR3` on x86). Following similar steps of the second approach, i.e., mark a page as writable and revert it to executable once the write is complete, the adversary would be able to inject code into the kernel.

Fourth, if the adversary can disable the MMU, she can bypass all the existing memory protections (e.g., page permission and DEP) as all of them are relied on virtual memory system. Note that disabling MMU can limit what a rootkit can do as well. Disabling the MMU reduces the adversary to utilizing physical memory addresses, and she may not know the physical memory addresses of code necessary to continue her ROP attack. However, if the operating system maps the virtual addresses of kernel space to the exactly same physical addresses, the rootkit could benefit from disabling MMU without limiting its capabilities.

Lastly, an adversary may direct the kernel to execute instructions in user space instead. Since an adversary often controls user space (e.g., a root process) prior kernel exploitation, she can simply prepare the malicious instructions there and invoke them from kernel. This is possible because most operating systems (e.g., Linux) map kernel space and user space into one unified virtual address space, and page permissions are set in such a way that the kernel space has a one-way view of the user space.

To summarize, we cast the problem into the following security invariants:

- **S1**: Execution of user space code from the kernel must never be allowed.
- **S2**: DEP protection employed by the operating system must always be enabled.

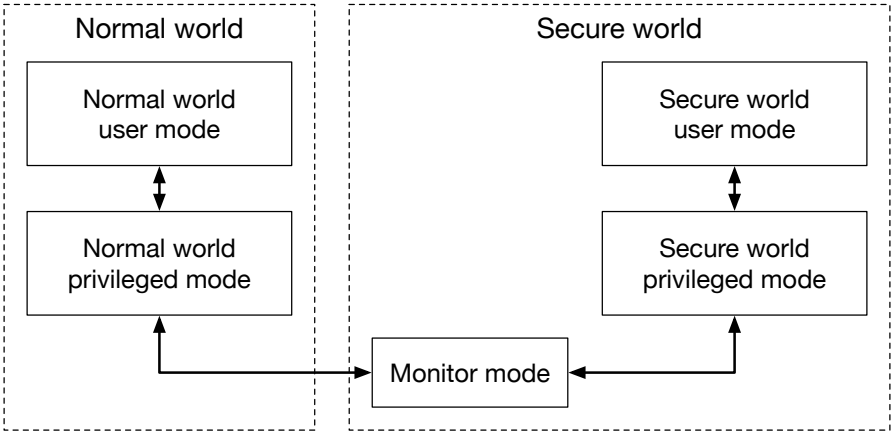

Figure 3.1. The architecture of TrustZone hardware

- **S3**: The page table base address must always correspond to a valid page table.
- **S4**: Any modification to the page table entry must be mediated and verified.
- **S5**: MMU must be on to ensure all existing memory protections function properly.

3.2 TrustZone Architecture

TrustZone [14] is a set of security extensions first added to ARMv6 processors. Its goal is to provide a secure, isolated environment that protects the confidentiality and integrity of critical computation from conventional computation. TrustZone partitions both hardware and software resources into two worlds - the *secure world* for assets that are security-sensitive and the *normal world* for conventional processing.

The TrustZone hardware architecture is illustrated as Figure 3.1. The processor core implements the two separate worlds, i.e., the normal world and the secure world. And it can be in one world at a time, meaning the two worlds are running in a time-sliced fashion. To maintain the processor state during the world switch, TrustZone adds a monitor mode, which resides only in the secure world. The software in the monitor mode ensures the state of the world (e.g., registers) that processor is leaving is saved, and the state of the world that processor is switching to is correctly restored. This procedure is similar to a context switch between processes except there are some banked

registers that are not required to be saved.

The mechanisms by which the processor can enter monitor mode are tightly controlled. Interrupts can be configured to be handled in either the normal world or the secure world. In addition, the normal world may proactively execute a Secure Monitor Call (`smc`), which is a dedicated instruction that can trigger the entry to monitor mode (i.e., the secure world). For the ease of understanding, the `smc` instruction is similar to `int` (i.e., software interrupt) on Intel x86 and `svc` on ARM in terms of privilege mode switch.

The current world in which the processor runs is determined by the Non-Secure (`NS`) bit. In addition, almost all the system resources (e.g., memory, peripherals) are tagged with their own `NS` bits, determining the world they belong to. A general access control policy enforced by TrustZone is that the processor can access all the resources when running in the secure world, while it can only access normal world resources (i.e., those with the `NS` bit set) when running in the normal world. For example, memory hardware is partitioned into the two worlds. When the processor is in the normal world, it can only access the physical memory of its own world. After entering the secure world, the processor can access all the physical memory in the system.

Unfortunately, the secure world as provided by the TrustZone architecture does not directly enable protecting the kernel running in the normal world from adversary. Unlike a VMM, the secure world is *not* more privileged than the normal world regarding the normal world resources. Once a hardware resource is assigned to the normal world, the secure world cannot control its access (e.g., by managing all physical memory, as a VMM would). For instance, the normal world has full privilege over its memory management system, meaning it can arbitrarily set its virtual memory environment (i.e., virtual address mappings and page permissions) and access its physical memory without requiring any permission from the secure world. Another example is that interrupts assigned to the normal world are handled locally, and the secure world cannot intervene in this procedure. By relinquishing this control to the normal world, an adversary can then tamper with the normal world's virtual memory environment without being detected by the secure world.

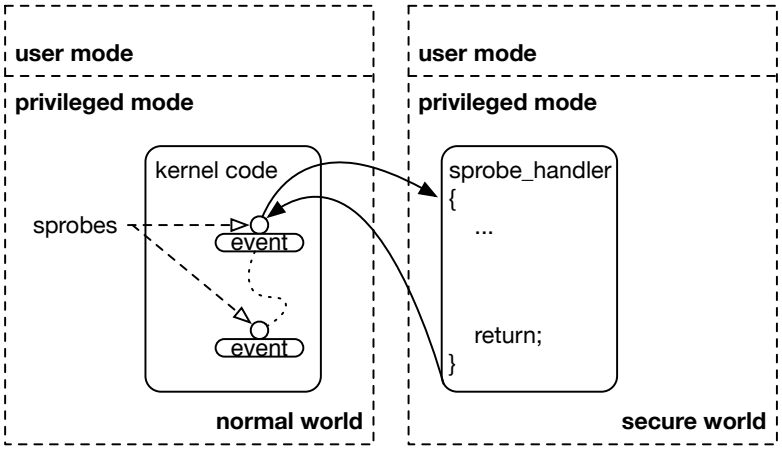

Figure 3.2. The runtime control flow when an SPROBE is triggered

3.3 SPROBES Mechanism

To enable the secure world to control the execution of normal world events of its choice, we present SPROBES, an instrumentation mechanism that can transparently break on any instruction in the normal world. The control flow of an SPROBE is shown in Figure 3.2. When an SPROBE is hit, the secure world immediately takes over the control, switches the context and invokes the specified SPROBE handler. The processor state of the normal world (e.g., register values when SPROBE is hit) is packed as a structure by the monitor mode and passed to the invoked handler. Since this parameter cannot be forged by the normal world, the handler obtains a true view of the normal world from it. Finally, the control returns to the location where SPROBE is hit.

To trigger the secure world from a normal world instruction, we rewrite the specified instruction in the normal world as illustrated in Figure 3.3. In this example, a secure world process (called a "trustlet") inserts an SPROBE at the `pop` instruction in the normal world by rewriting it to be the `smc` instruction. The `smc` instruction is the instruction for triggering entry to the secure world. Although code pages in the normal world might be write-protected, the secure world may rewrite any normal world instruction without causing an exception because the secure world uses a different set of page tables. This implies all the virtual memory protections (e.g., DEP) employed in the normal world are not applicable to the secure world, and the secure world is authorized

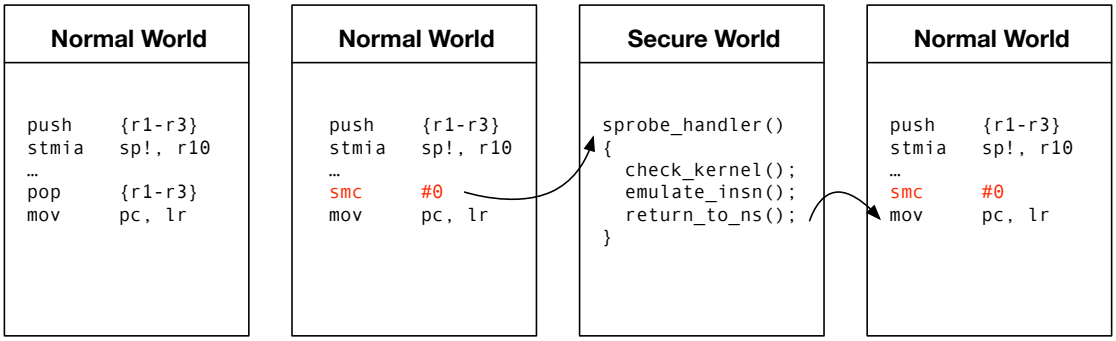

Figure 3.3. The involved world switches when an SPROBE is triggered

to access all physical memory without intervention of the normal world virtual memory system. Later, when the SPROBE is hit, meaning the program counter reaches the `smc` instruction, the processor state will be switched to the secure world immediately. Within the secure world, the SPROBE handler is invoked, which can perform monitoring operations such as checking the state of the normal world kernel prior to emulating the original `pop` instruction that was substituted by the `smc` instruction[2]. Lastly, the processor exits the secure world and resumes execution starting from the next instruction. It is worth noting that, if the instrumented instruction is a control transfer instruction (e.g., conditional branch), the next instruction will depend on the effect of emulation of such instruction.

There are several advantages of the SPROBES mechanism. First, it is independent on the software running in the normal world. All the features used by SPROBES are natively provided by the ARM hardware, such as the `smc` instruction and cross-world memory access. Second, an unforgeable state of the normal world can also be extracted from its hardware registers directly when an SPROBE is hit. Third, SPROBES are transparent to the normal world and thus do not require modifications to existing software. The original control flow in the normal world remains unchanged, so the operating system running in the normal world will not notice any overt differences caused by SPROBES. Fourth, SPROBES can break on any instruction in the normal world with-

[2]In our previous publication of this work [33], we restored the replaced instruction in the normal world. However, this may cause issues in a multi-processor environment – another thread can bypass the mediation before the same SPROBE is installed later. So we changed to emulate the replaced instruction within the secure world.

out any restrictions. This is because SPROBE does not cause side effects on the normal world. Fifth, other than the `smc` instruction, SPROBES are implemented in the secure world, so all of its code and data are isolated from the normal world. This limits the ability of a rootkit to affect any SPROBE execution. All these features combine to make SPROBES a powerful instruction-level instrumentation mechanism on TrustZone architecture.

3.4 Kernel Code Integrity

In this section, we propose an SPROBE placement strategy that can block all the approaches by which an adversary could violate the integrity of kernel code. With this placement strategy, we make a strong security guarantee that over the system's lifetime, any kernel rootkit cannot inject any code or modify approved kernel code. That is, even a rootkit in the normal world kernel is limited to running approved kernel code only. Further, since the SPROBES are inserted into kernel code, this placement strategy is sufficient to protect the SPROBES from modification as well.

3.4.1 Trust Model

We make the following assumptions for our trust model. In addition to trusting all the secure world code, we make the assumption that the normal world kernel code is free of rootkits prior to the execution of the first user-space process. That is, we assume that all rootkit threats originate from compromised root processes or any user-space process that has access to a kernel interface. We do not defend against malicious code running in the kernel prior to the first user-space process being initiated. In addition, we assume the use of hardware-based IOMMUs [34] to block malicious DMAs. Lastly, we trust the load-time integrity of kernel image is protected by utilizing technologies such as secure boot [14, 35].

3.4.2 Placement Strategy Overview

At a high level, our strategy is for the secure world to configure the normal world kernel such that the SPROBES can mediate runtime access to memory management. This strategy combines enforcement of high-level management settings for ARM (e.g., keeping DEP enabled to enforce S2 and keeping the MMU enabled to enforce S5) with VMM-like breakpoints to control kernel memory access (e.g., verifying page table integrity to enforce S3 and validating page table updates to enforce S4), leveraging an ARM hardware feature to prevent the execution of user-space code from supervisor mode (e.g., to enforce S1). This strategy is implemented in two phases. First, the secure world must configure the kernel for booting in such a way that invariants $S1 - S5$ are satisfied initially and ensure configuration of certain memory protections *prior to running the first user-space process* (see Section 3.4.3). Second, we describe how to place SPROBES such that they can implement policies necessary to enforce the invariants $S1 - S5$ throughout the system runtime (see Section 3.4.4).

3.4.3 Boot Configuration

We require that the secure world has some specific knowledge about the normal world kernel memory in order to establish the enforcement of $S1 - S5$ during booting. First, the secure world must know both the virtual addresses and physical addresses of kernel code pages, so it can validate the page table mappings and associated page permissions necessary to enforce S3 and S4. Second, the secure world must write-protect the kernel page tables to continue enforcement of S4. Other invariants will be established after booting the kernel.

In addition, the secure world must know that all code pages for the normal world kernel are approved for execution prior to booting. One issue with this assumption is that many kernels support loadable kernel modules (LKM), which may change the code in the kernel legitimately. A principled approach is to intercept `init_module` system call by inserting an SPROBE in the system call handler and validate the module before actually loading it into the kernel. In addition, kernel modules may contain security-sensitive instructions that would require SPROBE mediation as well. In our prototype,

we do not support LKM in the normal world kernel and leave the more general problem for future work.

Once the normal world kernel is booted then further work is necessary to establish invariants S2 and S5. We require that DEP is set (S2) and the MMU is enabled (S5) prior to running the first user-space process. These are typically set early in the boot sequence, and SPROBES are placed to mediate access to these settings as described below. If they are not set prior to changing the page table base address for the first time, indicating that a new process is running, then the secure world can halt the system. We note that although preventing the execution of user-space code in supervisor mode (S1) is not relevant until the first new process is initiated, we require the operating system to set the Privileged eXecute-Never (`PXN`) bit[3] on all user-space pages before actually context switching to the first user-space process.

3.4.4 Runtime Invariants Enforcement

The above boot configuration ensures invariants $S1 - S5$ are satisfied initially by obtaining the information necessary to enforce invariants S3 and S4 over the kernel prior to boot, and configuring enforcement of S1, S2 and S5 prior to running the first user-space process. Next, we describe our SPROBE placement strategy to continue enforcement of all these invariants throughout the system lifetime.

Enforcing S2. As the `WXN` is a bit in the *System Control Register* (`SCTLR`), the rootkit would have to write to this register in order to turn off DEP protection. Recall that we assume that the `WXN` bit is set at initialization time. Therefore, the idea is if we could insert an SPROBE at every kernel instruction that writes to the `SCTLR`, we can trigger the secure world whenever a rootkit may attempt to turn off the protection. When such an SPROBE is invoked, it simply must block values that attempt to unset `WXN` to achieve S2. Since ARM has fixed length instructions, finding such instructions in the kernel binary is straightforward, particularly compared to x86.

One issue is that there are multiple control bits in the `SCTLR`, such as `Alignment`

[3]The `PXN` bit is a permission bit in page table entries that determines whether a memory region is executable from privileged modes.

Check Enable bit. This causes false sharing as updating a non-WXN bit in the SCTLR will also hit the SPROBE and trigger the secure world thus bring unnecessary overheads.

Enforcing S5. Our solution does not insert additional SPROBES to prevent adversary from disabling the MMU. That is because the MMU Enable bit is in the same register, i.e., the SCTLR, as the WXN bit. All the SPROBES used to protect the WXN bit can also protect the MMU Enable bit. Therefore, S5 is satisfied. Given the fact that most operating systems do not disable MMU after it is turned on, it would be easy for the secure world to detect the presence of a rootkit if she tries to override the MMU Enable bit.

Enforcing S3. On ARM processors, the base address of page table is stored in a special register called Translation Table Base Register (TTBR). Similarly to above, by inserting an SPROBE at each instruction that writes to the TTBR, the secure world can fully mediate the operations that switch the page table, providing complete mediation of writes to this register.

Normally, to create separate address spaces for each process, the operating system allocates a different page table to each process. When a process is scheduled on processor, the operating system updates the TTBR with that process's page table base address. Thus, the secure world needs to be capable of ensuring that only valid page table bases are applied for each context switch. In order to do this, the secure world will have to maintain the integrity of the page tables. When a TTBR is asserted for the first time, the secure world must validate this new page table, ensuring that the addresses used are valid and checking compliance of the permission bits. For example, in Linux the kernel portion of the process's page table (i.e., addresses above 0xC0000000) must be the same for each process page table and the approved kernel code pages must not be writable. In addition, double mapping (i.e., two virtual pages are mapped to the same physical frame) must not exist in the page table [23], particularly between a code page and a data page, otherwise the attacker can modify kernel code by writing to that data page. By restricting updates of TTBR to only valid page tables, we enforce invariant S3. Note that we only need to validate a page table the first time that we see its TTBR value because of the way we control page table updates to enforce S4 below. Note further that unlike protecting the WXN bit, the SPROBES to protect the TTBR will be hit in a regular manner

Listing 3.1. Exception Vector Table

```
; we are at 0xffff0000 now
exception_vector_table:
reset:
    b    init            ; system boots
undefined:
    b    undefined
supervisor:
    b    syscall_start   ; system call
prefetch_abort:
    b    abort_handler   ; page faults
data_abort:
    b    abort_handler   ; page faults
unused:
    b    unused
irq:
    b    irq_handler     ; interrupts
fiq:
    b    fiq             ; not used
```

due to process context switch. As a result, some performance overhead is fundamental to this enforcement.

Enforcing S4. Placing SPROBES to prevent the page table from being modified illegally by rootkits is the most challenging part in our solution. Unlike the cases above, which focus on mediating access to a special register, page tables are just normal memory, so consequently there are many usable instructions that can write to them. Inserting SPROBES at all of those instructions (e.g., `store`) is not practical because of performance overhead. But why do we want to instrument all memory store instructions if what we really want is to monitor updates to page tables, which are only a small portion of the whole address space? Therefore, instead, we apply write-protection over page tables, so that any table updates will generate a page fault. Then, if we insert an SPROBE into the page fault handler, the secure world will be triggered upon every table update. This is essentially how shadow page tables are implemented in virtualization, but we need to implement this mechanism utilizing ARM hardware.

Unlike Intel x86, ARM requires operating system to place its exception vector table at a fixed location, either `0x00000000` or `0xFFFF0000`, determined by `Vectors` bit in the `SCTLR`. Normally operating systems use `0xFFFF0000` as the exception base address

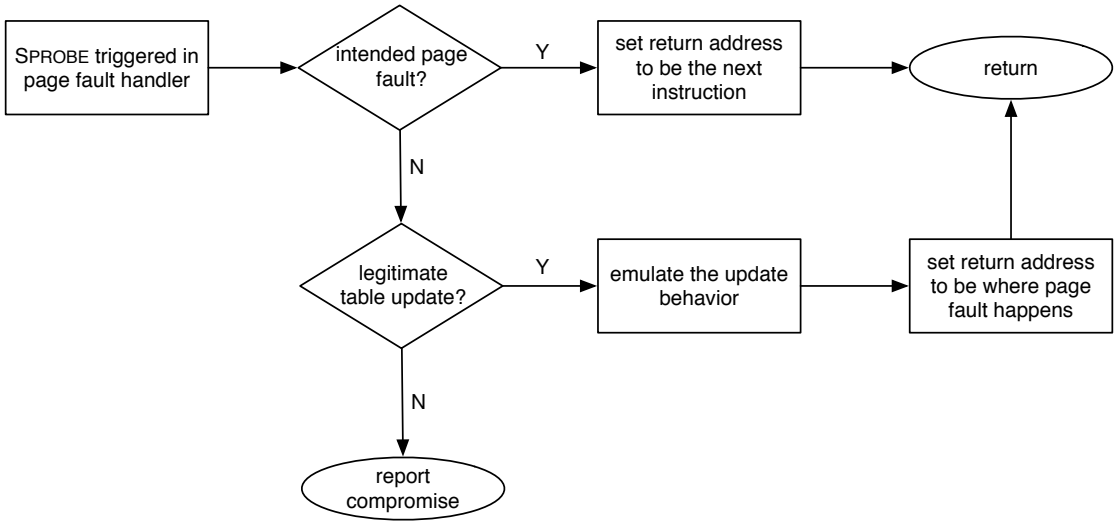

Figure 3.4. The handling of normal-world page faults in the secure world

because `0x00000000` is interpreted as NULL pointers. Because of this, the rootkit will not be able to re-define another exception vector table to bypass the SPROBE in the page fault handler. It is also worth noting that exception vectors are code rather than data, so they cannot be written by a rootkit without changing the code page permissions, which is what this SPROBE enforces. Each vector contains an instruction (normally a branch instruction), and the instruction is executed once a corresponding exception happens (e.g., page fault). To illustrate, we show an example of exception vector table in assembly code as Listing 3.1. We propose to insert an SPROBE at the start of the function `abort_handler`, which services page faults. Therefore, given that the adversary can neither re-define a new exception vector table, nor modify the current one, updates to the normal world's page tables are fully mediated by the secure world.

When the secure world is triggered due to a page fault in the normal world, there are three situations worth discussing, and the flow chart of SPROBE handling is shown as Figure 3.4. First, if the page fault is intended such as due to copy on write, we simply return to the exception handler to let the kernel deal with this exception. Second, if the page fault is caused by a legitimate table update like `mmap` system call, the secure world would first emulate the behavior of table update and then return to the faulted instruction as if nothing happens. By returning to the instruction that causes the page

fault instead of the next instruction in the exception handler, we make the whole procedure transparent to the operating system since it does not expect such an exception. Third, if the page fault is caused by either (1) making a page writable that maps to a physical frame containing kernel code, or (2) making a page executable that maps to any other physical frame, the secure world should block such page table modifications and trigger rootkit detection mechanisms. Therefore, S4 is satisfied.

Enforcing S1. Since all page table entries are protected from modification by enforcing S4, the `PXN` bits set prior to the installation of rootkit are safe from modification and the secure world can prevent the rootkit from creating new mappings in which the `PXN` bits are unset. In addition, S3 can reject a page table whose `PXN` bits are not properly set for new page tables, even those created by a compromised kernel. Thus, all the page tables must have the `PXN` bits set for all user-space pages, satisfying S1.

We enforce all five of these invariants by mediating a fixed set of instruction types. An implicit assumption behind our design is that the rootkit cannot jump into the middle of an instruction to discover unintended sequences. Fortunately, like most other RISC machines, ARM does not support misaligned instruction fetch and is thus not susceptible to this attack vector.

In summary, by enforcing invariants from S1 to S5, we mediate all the ways that a party, either the legitimate kernel or a malicious adversary, controls the system's virtual memory environment. S5 ensures the MMU is always on, which serves as a foundation for the rest of protections. S3 limits the usable page tables to only valid ones, preventing the adversary from setting the page table base to an arbitrary value. S4 protects the integrity of page table entries, thwarting any attempts to modify kernel code and thus protecting the SPROBES from unauthorized removal. S1 and S2 disallow execution over injected instructions and assure the enforced mediation is not bypassable. Overall, the five invariants restrict the adversaries to approved code pages, indirectly protecting any SPROBES inserted into these pages for enforcement purposes.

3.5 Implementation

We implement a prototype of SPROBES on top of a Cortex-A15 processor emulated by Fast Models 8.1 emulator [36] from ARM. Although SPROBES do not rely on the software (e.g., kernel) running in the normal world, to have a proof of concept of our solution, we run Linux 2.6.38 in the normal world as a case study. We build Linux from source code using the GNU ARM bare metal toolchain (e.g., arm-none-eabi). This toolchain is mainly for building applications for the ARM architecture without any operating system support. To make the SPROBES implementation simpler and more efficient, we extract all the necessary kernel information before hand. For example, we use *objdump* to inspect the address space layout of kernel code to identify the instructions to instrument.

One key step in the SPROBES implementation is to substitute the target normal world instructions with `smc` instructions. In most cases, the MMU is enabled in the normal world, so the secure world can only see its virtual addresses. To access a given virtual address in the normal world, the secure world needs first to translate it to the physical address and then create a corresponding mapping in its own page table. Note that the physical address space of the two worlds can be different, and the page table entry has an `NS` bit to indicate which world the physical address is from. To translate a normal world virtual address to the physical address, the Cortex-A15 architecture has an external coprocessor (CP15) that can perform such a translation, which avoids manually walking the page table in the normal world.

In order to enforce DEP protection, we disassemble the text section of kernel image and identify all instructions that write to the `SCTLR`. Then, in the secure world, we hard-code the addresses of those instructions and insert SPROBES after the Linux kernel is loaded in the normal world.

Similarly, to protect the `TTBR`, we scan the disassembled text section of kernel image and record all instructions that can write to the `TTBR`. However, ARM processors support more than one page table active (two in maximum) at a time, and one is determined by the `TTBR0` while the other is determined `TTBR1`. The `TTBRs` are used together

to determine addressing for the full address space. Which table is used for what address range is controlled via the Translation Table Base Control Register (TTBCR). In the actual implementation of Linux, by setting the TTBCR to a fixed value, it uses only one page table throughout its lifetime. However, to prevent an adversary from enabling the second page table, we still need to insert SPROBES to intercept writes to the TTBCR.

Protecting the page table requires the secure world to modify page permissions of the normal world. The secure world sets the page table memory to be read-only, so that any page table updates cause page faults. The implementation is similar to the mechanism to synchronize guest page tables and *shadow page tables* that are used by the VMM [37, 38]. Since we also insert an SPROBE to the page fault handler, the secure world would be triggered on every page fault. The connection between the page table updates to the secure world does not give the normal world software any opportunity to interfere with this function, as none of the normal world instructions can be executed in the middle. This requires us to insert an SPROBE at the very first instruction of the exception handler. There are two benefits by doing so: (1) it ensures no normal world instruction is executed before the secure world gains control and (2) in those cases where the operating system does not expect such a page fault (e.g., `mmap` system call), it minimizes the effects on the processor state of the normal world and thus makes state recovery (i.e., restore the register values before the page fault) easier.

3.6 Evaluation

In this section, we evaluate the security and performance of SPROBES when used to enforce kernel code integrity. For the security evaluation, we reason about how the placement strategy blocks every means by which an adversary can violate code integrity. For the performance evaluation, we run the Linux operating system in the normal world and measure how frequently the inserted SPROBES are hit.

3.6.1 Security Analysis

We evaluate the security of our solution by summarizing how our design has achieved the goal of protecting kernel code integrity for Linux 2.6.38. Specifically, we classify the 12 SPROBES [4] necessary to implement the placement strategy into 4 categories:

- **Type #1**: The 6 SPROBES that protect the `WXN` and `MMU Enable` bits in `SCTLR`.
- **Type #2**: The 4 SPROBES that protect the `TTBR` that defines the page table base.
- **Type #3**: An SPROBE that protects the `TTBCR` to enforce only one `TTBR`.
- **Type #4**: The SPROBE that is inserted at the beginning of the page fault handler.

At a high level, we demonstrate our design can effectively reduce the adversary to approved kernel code following these steps. We first show that the adversary has to change the memory environment, in order to execute injected or modified code in kernel space. Then, we illustrate how the combination of listed four types of SPROBES enables the secure world to detect all changes to the normal world memory environment. Finally, we claim that no memory environment changes that enable execution over unapproved code can bypass the checks in the secure world.

First, according to the Boot Configuration in Section 3.4.3, the only possible attack against the integrity of kernel code without tampering with the virtual memory environment is to modify the kernel image file, so that the system would be in a "compromised" state the next time it is loaded into memory. We foil such attacks by utilizing technologies like secure boot [14, 35] as assumed in the Trust Model in Section 3.4.1. Note that checking the integrity of kernel image file is sufficient to ensure load-time kernel integrity as kernel loading is at the very beginning of a boot sequence, before adversaries have access to the system.

Second, we claim that modifying the virtual memory environment will always be captured by the secure world regardless of its purpose. In essence, a virtual memory environment is uniquely defined by the active page tables as long as the MMU is on[5].

[4] We note that the actual number of required SPROBES depend on the number of instructions in each category. This can vary across operating system versions and compiler optimizations (e.g., inline function).

[5] Though the permission settings can be enhanced through bits like `SCTLR.WXN`.

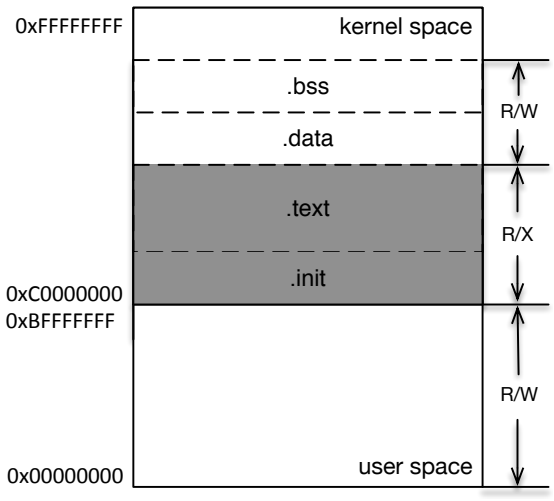

Figure 3.5. The virtual memory layout of the normal world

So, if not considering the cases where the MMU is disabled, modifying the virtual memory environment is just equivalent to modifying the active page tables. In specific, the attacker may either (a) switch to a different set of page tables that are under her control or (b) modify active page table entries in place. However, (a) would cause the hit of Type #2 and/or Type #3 SPROBES while (b) triggers Type #4 SPROBES since page tables are write-protected. Alternatively, the attacker can simply disable the MMU, accessing physical memory with no restrictions. Similarly, such operation will be trapped to the secure world as well because of Type #1 SPROBES.

Finally, we show that altering the virtual memory environment to enable execution over unapproved kernel code will not bypass the checks in the secure world. To achieve this, we need to draw a clear boundary between legitimate and malicious operations on the memory settings. To begin with, triggering Type #1 SPROBES, either by disabling the MMU or the DEP protection, is a clear indication of system compromise in the normal world because generally an operating system will not turn them off after booted.

Alternatively, an adversary can modify the page tables, triggering Type #2/#3 or Type#4 SPROBES, to execute unapproved kernel code. We discuss this case from two aspects. First, when modifying the page permissions, we enforce such modifications always comply with the permission settings as shown in Figure 3.5 – non-executable

SPROBE Type	#1	#2	#3	#4
Hit Frequency	N/A	313,836	N/A	85,982

Table 3.1. The hit frequency of different types of SPROBES

kernel data, unwritable kernel code, and non-executable user pages. Any attempt to violate this permission configuration will be regarded as a malicious operation. Second, an adversary can tamper with the page mappings. For example, by remapping a kernel code page to a different physical frame of her control, an adversary can virtually inject arbitrary code into the kernel space. On the other hand, by creating a double mapping between a data page and a code page, the adversary can overwrite existing code through the data page. To prevent malicious updates to page mappings, we ensure that the mapping between kernel code pages and the physical frames remain constant throughout the system lifetime, and no other virtual pages can be mapped to the same set of physical frames. By checking both page permissions and mappings upon page table updates, we ensure only a set of unmodified physical frames have been executed by the kernel, fulfilling the goal of kernel code integrity.

3.6.2 Performance Evaluation

In this section, we evaluate the performance impact of SPROBES on the normal world execution. The limitation of this performance evaluation is, our proof-of-concept was implemented on the Fast Models emulator, but Fast Models does not model accurate cycle counts but assumes every instruction takes one cycle. For example, arithmetic operations (e.g., `add`) and memory accesses (e.g., `load`) all take the same amount of time. This makes an accurate performance evaluation on Fast Models difficult, as the perceived overheads of software running on the emulator differs from those running on real hardwares.

To obtain a general understanding of how much overhead SPROBES incur, we count the number of instructions instead. An SPROBE hit causes 5611 more instructions (including the original `smc` instruction) to be executed in the secure world. Note that the number of instructions is a very coarse-grained measurement as it does not take microarchitectural events into account.

We run Linux 2.6.38 in the normal world with 28 startup processes including 4 daemon processes, 1 interactive process and 23 kernel threads. We run a shell script that invokes the write system call in a loop as the workload. We measure the individual cost for SPROBES of the four different types. To understand how frequently those SPROBES are hit in Linux 2.6.38, we use hit frequency, the average number of elapsed instructions between two consecutive hits, as the metric.

We list our results in Table 3.1. Both Type #1 and Type #3 SPROBES are not hit after boot, which means enforcing S2, S3 and S5 incurs negligible runtime overheads. Type #2 SPROBES are hit on each context switch. On average 313,836 instructions are executed between each Type #2 SPROBE hit, contributing to 2% of the instructions executed. We further measure the Type #4 SPROBES during boot stage when page updates are the most intensive. The result turns out Type #4 SPROBES are hit in every 22,424 instructions. After the kernel is setup, the hit frequency goes down to every 85,982 instructions. The runtime overheads incurred by SPROBES are acceptable for the protection offered, given that only 2 types of SPROBES are hit and incur fewer than 10% of the instructions executed. We note that current VM introspection methods trap all page faults as well, so the frequency of such traps would be similar.

3.7 Summary

In this chapter, we have presented the design and implementation of SPROBES, an instrumentation mechanism that enables the secure world to introspect the operating system running in the normal world on ARM TrustZone architecture. To enforce lifetime kernel code integrity, we identify a set of five invariants and present an informal proof of how 12 SPROBES enforce these invariants comprehensively. By enforcing all five of these invariants, we make a security guarantee that only approved kernel code is executed even if the kernel is fully compromised with modest performance slowdown.

Chapter 4

Enforcing Kernel Control-Flow Integrity

In this chapter, we show that fine-grained CFI enforcement for kernel software is possible, can be more efficient than coarse-grained enforcement, and can be applied comprehensively to kernel software, raising the bar for adversaries that wish to launch code-reuse attacks. We combine the proposed CFI enforcement with the kernel code integrity as detailed in Chapter 3 to setup the foundation for the whole-system execution integrity in the kernel space.

Previous research on CFI mainly focused on protecting user-level applications, while the proposed methods have significant limitations when applied to kernel software. Modular CFI [21] and forward-edge CFI [22] both predict call targets by matching function pointers with function signatures to reduce the size of target sets. However, such signatures are not always available given the presence of variable-argument functions and assembly functions in the kernel code. In addition, signature-based approaches may still result in both false negatives, when a non-target function happens to have the same signature, and false positives, when a function address is assigned to a function pointer of a different signature. Besides, these user-level CFI approaches do not capture system events such as interrupts, which may introduce non-trivial control flows to the kernel. Proposed applications of CFI enforcement to kernel software are too coarse-

grained to restrict the adversary effectively and either fail to enforce CFI comprehensively or are very expensive [9, 23].

To develop an effective CFI enforcement for the kernel software, we first compute a fine-grained CFG for kernel code. The insight for producing a fine-grained CFG for kernels is that we find that kernel code uses function pointers in a limited way, rarely creating data pointers to memory locations (variables, array elements, or fields) assigned function pointers. This enables us to design a static taint analysis that covers large kernel code bases with few exceptions requiring manual intervention. To enforce the computed CFG, we choose restricted pointer indexing, which was originally proposed by HyperSafe, as the default enforcement instrumentation, but reduce overhead by specializing the instrumentation based on the number of legal targets, and reuse checks that are already available in the kernel code. Finally, we also develop a design that enables comprehensive and efficient CFI enforcement in kernel software by reasoning about how system events are processed. As a result, we have eliminated over 70% of targets on both FreeBSD and MINIX relative to the current fine-grained CFI, while our implementation incurs 1.82% performance overhead on FreeBSD and 0.76% overhead on MINIX on macrobenchmarks, and 11.91% and 2.02% overhead on microbenchmarks.

4.1 Security Model

We base our work on the following security model. We trust that the data execution protection (DEP) is deployed in kernels by setting the code section as read-only and the data sections as non-executable. We also assume the kernel is benign but may contain vulnerabilities (e.g., memory corruption bugs) that enable an adversary to overwrite control data. However, we make the assumption that the kernel is free of attacks prior to the execution of the first user-space process. That is, we leave the detection of attacks at kernel boot, such as corruption of kernel images or loading of malicious code, to secure boot techniques [39] or authenticated boot techniques [40, 41]. Finally, we assume adversaries have no physical access to the machine, excluding hardware attacks from the picture.

Thus, we assume that all threats originate from inputs that the kernel receives after booting, such as malicious inputs from user-space processes and malicious network packets. We focus on attacks that alter the control flow of the kernel software illegally at some indirect control transfer. Thus, data-only attacks are out of the scope of this chapter, including those data-only attacks that change the values of variables used in conditionals to redirect control flow within a procedure [42, 43, 20].

We assume the MMU configurations, such as mappings and permissions configured by system page tables, can be protected by system defenses such as SPROBES as detailed in Chapter 3 or other hypervisor-based techniques [44, 6]. This is because, although the CFI-protected kernel restricts its execution to only authorized control flows even when compromised, the adversary may alter the system page table settings through *data-only* attacks. Given our system requires the kernel software to be statically linked (Section 4.2) to allow that all kernel code pages are identified offline, we believe these approaches apply to our situation as well. However, it is worth noting that, whatever MMU protection mechanism is used, its implementation and performance impact is orthogonal to the techniques discussed in this chapter.

4.2 Solution Overview

Our goal is to retrofit kernel software to enforce fine-grained CFI efficiently and comprehensively. We focus on kernel software because its integrity is fundamental to the integrity of the system at large and it is constrained in ways that enable computation of fine-grained CFGs.

In the first phase of our proposed solution, described in Section 4.3, we develop a method to produce accurate, fine-grained CFGs for kernel software. We hypothesize that we can collect an accurate set of indirect call targets for kernel code because we find that kernel code handles function pointers in restricted ways. In Section 4.3.1, we identify two simple constraints on the use of function pointer variables that we find that kernel code broadly obeys. For example, these constraints still allow kernel developers to use function pointers stored in arrays and structures, which is common.

With these constraints, we develop a static taint analysis to identify the indirect call targets in kernel code. In Section 4.3.2, we describe how to compute return targets for kernel code. While computing return targets for source code is straightforward, assembly code does present some challenges for detecting legal return targets (e.g., due to tail-call optimization and fall-through functions), so we design a CFG analysis that can detect return targets accurately and comprehensively.

We find kernel software amenable to the application of CFI enforcement because it is often statically linked. Secure software deployments often demand static-linking to prevent adversaries from replacing critical software components at runtime. For example, the Linux Security Modules were originally loadable kernel modules, but such "modules" must now be statically linked into the kernel. This is because, kernel modules introduce a violation of the DEP policy in such a way that the kernel must first load them into writable memory pages and then mark them as executable. By corrupting the loaded modules in the vulnerable time window, an adversary can inject her own code and hence perform arbitrary computation without needing to reuse existing code. As a result, the approved code must be determined at load-time, enabling techniques like SPROBES (see Chapter 3) to protect kernel code integrity by authorizing modifications of memory protections set by the kernel at boot time.

In the second phase of our proposed solution, we modify the kernel code to enforce the computed CFG. We find that two kinds of modifications are necessary. First, assuming the protection of kernel code integrity as discussed above, kernels still require further modifications to handle event processing (e.g., system calls and interrupts) in a manner compliant with the CFI enforcement at runtime. These issues have been investigated in the past in the HyperSafe [23] and KCoFI [9] projects. However, HyperSafe does not address attacks on kernel exit nor how kernel code must be modified to ensure comprehensive enforcement. KCoFI, on the other hand, describes comprehensive method for controlling event processing, but at significant expense. In Section 4.4, we specify invariants that must be enforced to ensure that the kernel code is always invoked from legitimate entries and returns safely. Our aim is for comprehensive CFI enforcement given system event handling at low cost, which we achieve by removing

all means for the kernel to modify event handling configurations, lightweight checking of exception handling, and non-preemptive kernel configurations.

In addition, given a fine-grained CFG computed according to the methods above, we develop a method to instrument the software to enforce the CFG at runtime as discussed in Section 4.5. While we use a standard form of instrumentation, called *restricted pointer indexing* [23], as the default, we identify two opportunities for optimization. We find that kernel software often uses function pointers to express flexibility, so in many cases kernel software only uses one target for indirect control transfers. As a result, once we know that an indirect control transfer has only one target, we convert it to a direct control transfer, which requires no additional instrumentation. In addition, based on the type of addressing mode used, we find that we can reuse code already produced by the compiler or by manual assembly, again enabling the removal of unnecessary instrumentation.

4.3 Computing Control-Flow Graphs

In the first phase, we develop methods that compute a fine-grained CFG from kernel source code. We propose an algorithm for computing control transfer targets for indirect calls in Section 4.3.1, and discuss and solve the challenges in mapping calls to returns to compute the return targets in Section 4.3.2. We note that the use of indirect jumps in source code is limited to *switch* statements, whose targets are stored in jump tables and can be found trivially, and *tail-call optimization*, which are treated as calls.

4.3.1 Computing Indirect Call Targets

We compute indirect call targets for kernel source code under constraints on the use of function pointer variables in the program code. Given these constraints, we design a static taint analysis to compute a set of allowed targets for each indirect call. The intuition behind that analysis is to track how function pointers are propagated in the kernel source. If an address-taken function can reach a particular call site, the function is a valid target for the call site. Based on our experience examining kernel source code,

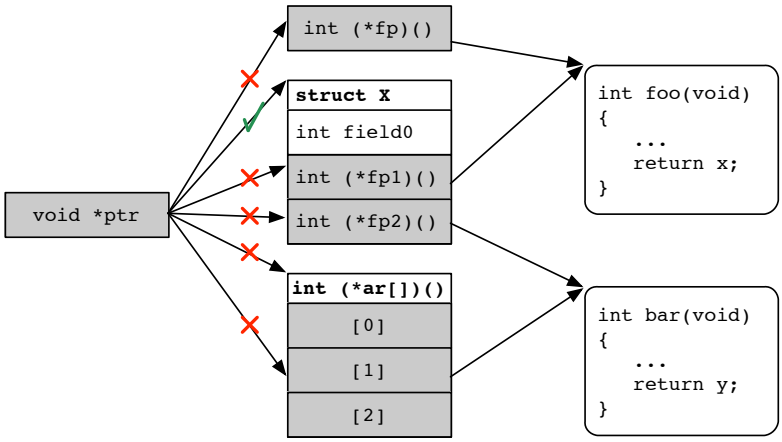

Figure 4.1. The assumption A2 of absent data pointers to function pointers

these constraints are followed broadly, enabling automated detection of targets. We evaluate the applicability of this method to kernel source code in Section 4.7.

In kernel source code, operations on function pointers are often limited to assignment and dereference. Thus, we propose two assumptions about operations on function pointers:

- (**A1**) The only allowed operation on a function pointer is assignment[1].
- (**A2**) There exists no data pointer to a function pointer.

There are a few important implications resulting from these assumptions. A1 limits the operations on function pointers, preventing them from being modified once assigned. In particular, this assumption precludes pointer arithmetic on function pointers and enables tainting by only tracking function pointer assignments. We believe arbitrary computations on function pointers are unlikely due to considerations such as readability, maintainability, and portability. A2 assumes the absence of data pointers to function pointers as illustrated in Figure 4.1. Note that A2 does not prohibit the presence of pointers to a structure that has function pointer fields. In fact, this is common in practice. For an array of function pointers, A2 only allows the array elements to be accessed by index from the array variable. Creating a pointer to the array or any of its

[1]Other than dereferencing for calls, of course.

elements will violate A2. We describe how we detect violations of these assumptions at the end of this section.

Our approach takes an address-taken function *f* as input and returns a set of function pointers for which *f* is a valid target as the output. To begin with, the approach taints all function pointers that are initialized with *f*. Note that a function pointer can be initialized either dynamically (e.g., assignment) or statically (e.g., global variables). Then we keep tainting function pointers to which a tainted function pointer is assigned. Because of assumption A1, we will not miss any function pointers tainted by function *f* by only tracking propagation via assignments.

Function pointers may either be referenced as a *variable*, an *array element*, or a *structure field*, and how we propagate taint differs in these three cases. If a function is assigned to a function pointer variable, we simply add the variable into the tainted list for that function. Note that type casts on function pointer variables are allowed because our taint tracking approach simply propagates taint across assignments regardless of the data types of the variables.

Alternatively, if a function pointer is assigned to any element in an array of function pointers, we taint the entire array. This is because programs normally access arrays using runtime indices. If our analysis cannot determine the location of the array element statically, we conservatively assume *f* can be retrieved from any index of the array.

Otherwise, if a function is assigned to a field in a structured type, we taint the field for all instances of that structure's type. That is, our *field-sensitive* taint analysis infers that if a function pointer can be assigned to a field of one instance of a structure type, it can be accessed at any indirect call site that references any instance of that structure type. However, in theory, a program can access the function pointer field through a different structure type (e.g., unions or type cast). To recognize this case, we actively check for any alternative definition for a structured type, and taint the aliased function pointer field as well.

We detect violations of both assumptions while performing the taint tracking. Instead of resorting to an over-approximation, we report detected violations to the user and stop the analysis. To detect violations of A1, we actively check if any of the tainted

function pointers are processed using an arithmetic operation. To detect violations of A2, upon every function pointer assignment, we check whether the function pointer, either on LHS or RHS, is accessed by directly dereferencing a data pointer. One concern of this approach is that type casts that cause violations that cannot be detected statically. For example, the kernel can perform pointer arithmetic on an integer pointer (`int*`) and make it point to a function pointer. This introduces an untracked data flow between function pointers and makes the result of the taint analysis an under-approximation, resulting in false CFI violations. In practice, we do not encounter this case in the CFI-protected kernel. Note that such implicit type casts are a common problem for all pointer-based analysis [45, 46]. Moreover, it is a bad programming practice and may cause undefined behaviors under standard compiler optimizations [47].

4.3.2 Computing Return Targets

Return instructions are used in conjunction with call instructions. Thus, the key task in computing return targets are to map the call sites to their corresponding return instructions statically. This solution is straightforward for source code once the targets of all indirect call sites have been resolved. This is because source code has well-defined functions, and we can easily identify the functions' return instructions. However, problems occur in assembly functions. First, programmers may apply certain compiler optimizations manually to assembly functions, which may hide the true return targets. Second, assembly functions may not adhere to the restrictions of functions in kernel source code, such as lacking well-defined boundaries, lacking return instructions altogether, and nesting functions inside of functions.

Tail-call optimization. Generally, a tail call is a function call that is performed as the final action of a procedure. One common optimization on tail calls is to reuse the current stack frame by deallocating local variables and *jump* to, as opposed to *call*, the target function, pretending it is being called by the caller of the current function. We show an example in Figure 4.2. In this case, `cstart` is invoked by `main` and further calls `arch_init` as the last operation. However, due to the optimization applied, `arch_init` will directly return to `main`.

```
main:                cstart:                  arch_init:
 ...                  ...                      ...
 call  cstart         jmp  arch_init           ret
```

Figure 4.2. An example of tail-call optimization in the kernel code

Fall-through functions. Assembly programmers may nest one function definition inside another, which enables the execution in one function fall through to the other. In the example shown in Figure 4.3, `memset_fault_in_kernel` and `memset_fault` share the same function body except the former has additional handling. Therefore, the return instruction in the example may return to call sites that either invoke `memset_fault_in_kernel` or `memset_fault`.

```
memset_fault_in_kernel:
  ...
memset_fault:
  ...
  ret
```

Figure 4.3. An example of the nested function in the kernel code

In general, mapping returns to call sites is equivalent to finding the return instructions that are immediately dominated by the targets of every given call site. This problem can be solved using an intra-procedural control-flow analysis. Given a call site and its call targets, we analyze the CFG of the target to identify the immediately dominated return instructions, adding the instruction following the call site to the set of return targets for each return instruction.

Intuitively, given a call site, our approach emulates the execution of each target function and records the encountered return instructions. Starting from the target function, we sequentially "execute" the instructions one by one just as a processor does until reaching a branch instruction. If it is an unconditional jump, we follow its control flow, e.g., the tail call optimization case; otherwise (i.e., a conditional jump), we follow both edges. The search along any path stops "executing" under three conditions. First, the current instruction has been "executed". This is because any return instruction encountered later should have already been discovered. Second, the current instruction is a

Instruction	Action

`iret`
`hlt`
`sysexit`
`...`
	stop

`jmp`	**follow**
`jcc`	**continue** and **follow**
`call`	**continue** or **stop**
`ret`	**stop**
other	**continue**

Table 4.1. Different cases in the emulated "execution" in finding assembly returns

stop instruction such as `IRET`, `HLT` or a call instruction to a non-return function. This indicates the current function may not have a return. Third, the current instruction is a return instruction. If this is the case, the call site is one of its targets. We summarize all the cases in Table 4.1.

The key challenge here is to identify non-return functions, which determines whether the "execution" should continue or stop on a call instruction. We solve this problem in a recursive way, that is, if the execution meets a call instruction, we look for its corresponding return instructions using the same procedure. As long as one return instruction is found, we continue the "execution"; otherwise, we stop. Note that recursive calls in programs can make this approach run into an infinite loop. Though we have not seen such case in assembly code, we propose to conservatively assume any recursive call will return eventually and let the "execution" continue. We run this approach on each call site to assembly functions, and combine the results from source code to produce a complete set of targets of function returns.

4.4 CFI Enforcement

In the second phase, we describe our approach to retrofitting the kernel software to enforce the CFG computed from the first phase comprehensively. We break this retrofitting effort into two tasks presented in this section and the next. In Section 4.4.1, we first outline an overview of the proposed approach for CFI enforcement over kernel software. Next, in Section 4.4.2 we detail the specific problems and solutions necessary to develop

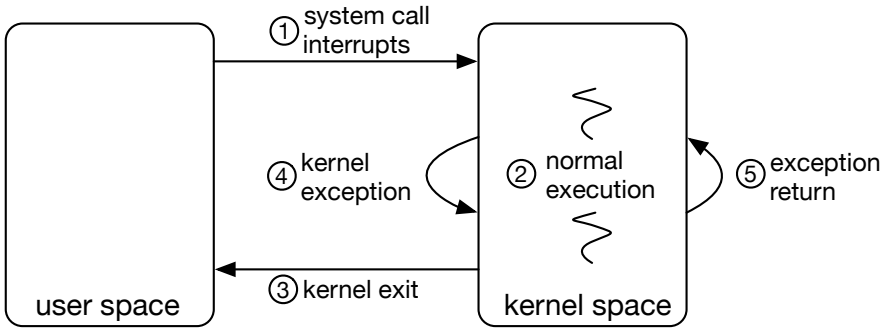

Figure 4.4. The execution model for non-preemptive kernels

the proposed approach.

4.4.1 Approach Overview

Comprehensive CFI enforcement for kernel software is challenging because kernels must be capable of processing system events and have the power to configure how they execute such events and code in general. First, kernel software not only may perform indirect control transfers during their execution, but also may be entered by system events out of the kernels' control and exit either to run kernel or user-space code. Kernel entry can occur due to three types of events, system calls, interrupts, and exceptions, which may be triggered by user-space code or asynchronously via hardware. Second, unlike user-space programs, kernel software has much more control over how code is executed, including defining how memory segmentation is configured, which determines the effective address used for each instruction fetch, and configuring the code used for its event handling.

An overview of an approach for comprehensive CFI enforcement in kernel software is shown in Figure 4.4. In step one, transitions from user-space to the kernel must only select known, approved entry points, such as event handlers for the three types of system events. To prevent an adversary from controlling kernel entry, we protect the integrity of various hardware-defined data structures that define those entry points. In step two, the kernel executes within the fine-grained CFG computed in the previous section. Then, two possible scenarios can occur. In one case shown in step three, the

kernel completes an event handler and exits back to user space. Here, we must restrict the exit to run only at user-space privilege and restrict the target instruction run at the exit. These steps cover the execution of all system call events. Alternatively, in step four, a system event may occur during kernel execution. In this case, we must ensure that the event can be taken without disrupting CFI enforcement and that the exit from these events pick up where the kernel execution was. Unfortunately, we cannot guarantee such properties for interrupts that preempt kernel execution since they may occur at any time, so we instead integrate CFI enforcement into kernels built to run non-preemptively. We discuss the implications of this choice in Section 4.8. Kernel exceptions, on the other hand, either cause a kernel panic or occur at discrete times. The only type of exception we have found that needs special support are page fault exceptions. These only occur for specific kernel code (e.g., `copy_from_user`) and continue executing either from the instruction that caused the page fault or from a predefined location that handles invalid user pointers. Thus, step four only includes page fault exceptions, and in step five the exception's exit returns deterministically from the page fault handling. See Section 4.8 for further discussion of CFI enforcement over kernel exception handling.

Compared with existing CFI approaches for system software (i.e., HyperSafe and KCoFI), we restrict all the system events found in commodity operating system kernel while making enforcement more lightweight and efficient. For example, to ensure comprehensive control, we not only prevent memory writes to critical tables as HyperSafe does, but also remove instructions that would write to the registers that define these tables. As an example of improving performance, we relax the constraints on context switching: rather than ensure the kernel returns to the exact same user-space instruction, we only guarantee the privilege is lowered.

4.4.2 Enforcing CFI in Kernel Software

Specifically, to enforce the approach described above, we propose four system invariants.

- (**S1**) Segmentation configurations must be protected from adversaries.

- (**S2**) Execution must always start from a legitimate entry point.
- (**S3**) All indirect control transfers must adhere to the CFG within the execution.
- (**S4**) Exits must be mediated and authorized.

Enforcing S1. Kernel software poses additional risks to control-flow hijacking attacks when compared to user-space applications because of the use of architecture-defined control data for various system configurations. Among these, memory segmentation (i.e., the Global Descriptor Table, or GDT) is fundamental in the sense that the code segment determines the base address used by both direct and indirect control transfers. We protect the GDT based on the observation that most operating system kernels never modify it once set. Therefore, we (1) zero out the `LGDT` instruction to prevent the adversary from setting up her own GDT via code reuse, and (2) write-protect the GDT throughout the system lifetime to prevent any in-place modification. To eliminate the side effects of the changed memory permissions, we separate the GDT from other kernel data by moving it into a distinct page.

Enforcing S2. Kernels are triggered by system events such as interrupts, exceptions and system calls. Depending on the specific event, the kernel will invoke different routines to handle the event. Upon interrupts, the control will be transferred to the routine specified in the Interrupt Descriptor Table (IDT) defined by the kernel. Alternatively, a system call will switch the current execution to the kernel at the address defined by two Model Specific Registers (MSRs) `IA32_SYSENTER_CS`:`IA32_SYSENTER_EIP`[2]. To satisfy S2, we must guarantee all these system events will trigger the kernel execution from legitimate entry points defined by the kernel. We achieve this goal in two steps. First, similar to GDT protection, we require both removal of `LIDT` instruction and write-protection for the IDT throughout the system lifetime. This prevents the adversary from manipulating the entries to interrupt handling. Second, we require that MSRs related to system calls are never modified once set. We achieve the latter by zeroing out all `WRMSR` instructions in the kernel after the system is booted. Note that some kernels do have the

[2] Traditionally, system calls are implemented using software interrupts (i.e., `INT`). Later, in order to enable faster transition, Intel introduced a pair of dedicated instructions `SYSENTER` and `SYSEXIT` for this purpose. MINIX supports this mechanism while FreeBSD still uses software interrupts for 32-bit kernels.

need to modify MSRs after boot. For these cases, we propose to duplicate the `WRMSR` instructions and hardcode the register identifier for them to guarantee that it can only write to MSRs other than the two related to the system call entry. The protection of both interrupt and system call entries satisfies S2.

Enforcing S3. To hijack the control flow during kernel execution, the adversary has to either control the return of a kernel exception or an instrumented indirect control transfer. To protect the return of page fault handling from adversaries, we only allow the kernel to return either to the predefined location that handles invalid user pointers or to the exact instruction that took the page fault. To achieve the latter, we store the return address to an unused debug register (or anywhere that is free of memory corruption) at the very beginning of exception handling, and check the subsequent exception return against the address we saved before. To prevent the adversary from controlling an instrumented indirect control transfer, we write-protect the target tables used by restricted pointer indexing. To eliminate the *time-of-check to time-of-use* (TOCTTOU) attacks on the index, in addition to disabling interrupts, we load the index into registers before executing the check. This approach not only guarantees no exception could ever happen in between, but is also safe from inter-processor races. Thus, S3 is satisfied.

Enforcing S4. The kernel will return the control to the user space eventually. Conceptually, the kernel returns the control back to the user space upon the completion of an interrupt (e.g., context switch) or a system call. In both cases, since the target instructions are outside of the kernel's CFG on purpose, we do not apply the same CFI enforcement to them. Instead, we ensure that these indirect control transfers would switch the subsequent execution to the user privilege (i.e., ring 3 on x86), preventing the adversary from launching control-flow hijacking attacks back to the kernel space. However, for system call return, we apply a stricter enforcement based on the fact that, for portability reasons, most kernels map the routine for invoking system calls to the user-level address space so that programs do not have to concern themselves with whether a system call is implemented using the software interrupt, `SYSENTER` or even its equivalent instruction `SYSCALL` on AMD processors. This implies that programs that are linked with standard library should obey this convention and make system calls through the

kernel-mapped routine. Therefore, we add checks on the system call return against the address of the mapped routine before returning to the user space, and panic the system if they do not match[3]. Note that this will potentially break the programs that make system calls through hand-written software interrupts. However, we did not encounter any such case in practice.

4.5 CFI Instrumentation

In this section, we describe our proposed approach for instrumenting kernel software to enforce the fine-grained CFG computed in Section 4.3. The idea is that we start with an effective, general instrumentation approach and when we detect a possible optimization opportunity we replace the default instrumentation with optimized instrumentation. In Section 4.5.1, we first briefly introduce the concept of restricted pointer indexing. Second, in Section 4.5.2 we describe two opportunities for optimizing CFI instrumentation. Specifically, we leverage the fine-grained CFG to replace instrumentation with direct control transfers when there is only one legal target, and we also make use of checks already in the kernel code to further optimize the instrumentation.

4.5.1 Restricted Pointer Indexing

Restricted pointer indexing is a CFI enforcement technique proposed by Wang *et al.* [23]. We choose this technique as our default for CFI instrumentation because it avoids the destination equivalence issue, which limits the granularity a CFI defense could achieve, by explicitly separating target sets for each indirect control transfer. Its main idea is to organize all the control data, e.g., return addresses and function addresses, into target tables and replace all the addresses used in indirect control transfers, either statically initialized (e.g., function pointers) or dynamically generated (e.g., return addresses), with corresponding indices.

To use restricted pointer indexing, one must transform all the indirect control transfers in such a way that they all use indices for transferring control by fetching the target

[3] `rt_sigreturn` is an exceptional case because it instructs the kernel to return to the instruction where the signal is taken. We treat these special system calls as interrupts.

address from the target tables. Note that for control transfer targets that may appear in multiple tables, its index has to be unique among all target tables. This is because when an index is introduced, e.g., by a call instruction, it can be used at multiple destinations, e.g., return instructions. Hence, if an index is mapped to different targets when used by different indirect control transfers, it would make the transformed program behave inconsistently. To ensure the same target must have the same index in all tables, we use padding entries when necessary. The padding entries point to the *panic* function in case they are misused by adversaries.

4.5.2 Optimizing CFI Enforcement

One key obstacle that prevents CFI from receiving wide adoption is the performance cost. We identify two opportunities to reduce the amount of instrumentation required for CFI enforcement.

Our first observation is that compilers may already insert code to restrict the targets of indirect control transfers to safe values, so we can simply leverage this available instrumentation. For instance, all indirect jumps produced when compiling *switch* statements use a read-only jump table and an index value for choosing the *case* statement to be executed. The compiler adds code to check that the index value is within jump table, so this code satisfies CFI by default. Similarly, in kernel software, programmers store references to all the system call service routines in a table and invoke the requested routine based on the system call number (i.e., index). The system call number (index) is restricted to be within the table. Given the high frequency of system call requests from user space, adding code to enforce this table index redundantly would lead to an unnecessary performance impact. If this code complies with instrumentation of *base plus index* cases shown in Table 4.2, then we do not need to add bounds checks[4]. Finally, kernel programmers may leverage absolute addressing for making direct control transfers in assembly code. In this case, they hardcode a jump target into a register, and then immediately make a control transfer. We claim that no enforcement is required for

[4]However, we add code to store the index for call instructions that use *base plus index* addressing to identify the return target (caller).

these indirect control transfers because they have a deterministic target at runtime.

Our second observation is that many kernel indirect control transfers have fixed targets that can be identified at link time. To enable flexible kernel configurations, kernel programmers often use function pointers that can be bound to the particular modules' functions specified in the kernel configuration. In many cases, the configuration specifies only one target for these function pointers. For example, in the MINIX microkernel, the kernel uses the ELF library to load server binaries. The library is designed to be generic so it allows callers to specify different functions to perform core tasks, e.g., memory allocation. However, the microkernel is assigned a fixed set of these functions at link time, so those function pointers have fixed values. Surprisingly, we found that the percentage of indirect calls which have fixed targets ranges from 31% to 66% across the FreeBSD kernel, MINIX microkernel and its user-space servers (see Section 4.7 for details). Therefore, we simply rewrite these indirect control transfers to semantically-equivalent direct control transfers to minimize both memory and runtime overheads.

4.6 Implementation

We implement our techniques on both FreeBSD 10.0 and MINIX 3.2.1 for Intel x86 platforms in four parts. The first part is to change the build process of kernel software so that kernel object files are linked against recompiled libraries rather than the system libraries because our technique requires whole-program analysis and instrumentation. We achieve this part by instrumenting the static linker to identify all relocatable object files that make up the final executable.

The second part is to compute a fine-grained CFG using the techniques presented in Section 4.3. We implemented this part as an LLVM pass. It contains 1,391 lines of C++ source code and runs on LLVM 3.5 [48]. The pass takes LLVM bitcode as input and returns the target functions of each indirect call site. Throughout its analysis, the pass will actively evaluate if any assumption made in Section 4.3.1 has been violated and raise an exception upon detection. For indirect calls and jumps written in assembly code, we detect them, but have to analyze them manually. However, we find that most

indirect calls and jumps in the assembly code comply with the code pattern described in Section 4.5.2 thus are free of control-flow hijacking attacks.

The third part is to modify the kernel to enforce system invariants presented in Section 4.4.2 in addition to S3 handled by typical CFI enforcement. We set up the protection, including removing various instructions and write-protecting configuration data structures at the end of system booting but before the kernel initiates the first user-space process (S1, S2). This allows the booting procedure to freely configure them without causing false positives. To eliminate the side effects of the changed page permission, we also relocate both GDT and IDT so that they are page-aligned and not co-located with other kernel data. To protect the kernel exiting to user space, we check the code segment selector that is about to be restored, and ensure that it points to the user code segment defined in the GDT (S4).

The last part is CFI instrumentation. This part has two steps. The first step is to instrument each relocatable object file individually, including replacing control data with indices, rewriting indirect calls and returns to equivalent index-aware instructions as designed by restricted pointer indexing. We summarize our instrumentations[5] in Table 4.2. It is worth mentioning that, for function calls, in addition to specifying the return address for the callee, our instrumentation pushes the return index before checking the target to inform the *panic* function where things went wrong. We check indices against bounds of target tables using unsigned comparison, preventing an adversary from bypassing the limit checks via integer overflow. At this step, indices, base addresses of target tables, and their contents are not determined yet because actual addresses of targets are only known after linking. To help the next step recognize the original indirect control transfers (e.g., indirect call sites and return instructions) we use a magic number `0xdeadbeef` for these unknown values. The second step is to fill each target table and update indices that are left blank in the first step. This step identifies the original call sites and return instructions via the magic number and updates them to the correct value. We implement both steps of instrumentation in a Python script containing 526 lines of source code.

[5]We use `ECX` to hold the index of return address and `EAX` to hold the call target if the original call site

Addressing mode	Original code	Basic instrumentation	Fixed-target instrumentation

Register Indirect	`call eax`	`push index`
		`cmp eax,bound`
		`ja panic`
		`jmp [table+eax*4]`
			`push index`
			`jmp foo`

Memory Indirect	`call [ebp+offset]`	`push index`
		`mov eax,[ebp+offset]`
		`ja panic`
		`jmp [table+eax*4]`
			`push index`
			`jmp foo`

Memory Indirect	`ret`	`pop ecx`
		`cmp ecx,bound`
		`ja panic`
		`jmp [table+ecx*4]`
			`pop ecx`
			`jmp target`

Base plus Index	`cmp eax,bound`
	`ja some_label`
	`call [table+eax*4]`
		`push index`
		`cmp eax,bound`
		`ja some_label`
		`jmp [table+eax*4]`
			`N/A`

Base plus Index	`cmp eax,bound`
	`ja some_label`
	`jmp [table+eax*4]`
		`N/A`	`N/A`

Table 4.2. CFI Instrumentation based on original code and addressing mode

There are a few cases we experienced where automatically applying the CFI instrumentations can break a system. For instance, in the context switch routine, the FreeBSD kernel uses a return instruction to resume the execution of the scheduled thread. However, newly created threads do not have a valid index for the instrumented return to function properly. Therefore, we manually handle the return by adding the corresponding index into its target table. Also, the FreeBSD boot process reloads the code segment register after paging is enabled via intersegmental return[6]. Since the return address is substituted with an index, it generates a fault. To make this instrumentation work, we manually rewrite the routine so that it uses `LJMP` to reload the code segment register, and uses a normal `RET` to return to the caller. Similarly, when retrofitting MINIX servers, we found an assembly library function `getcontext()` that attempts to directly read the return address pushed by a call site, which crashes the servers. We handled this issue by manually rewriting the procedure to be index-aware. In addition, in the MINIX microkernel, the system call invocation routine that is mapped to user space must not be instrumented because (CFI-unaware) user-space programs expect an address rather

references memory because both `EAX` and `ECX` are caller-saved registers in most calling conventions.

[6]In addition to changing the instruction pointer like a normal return, `LRET` instruction also updates the code segment register with a 16-bit segment selector from the stack.

than an index. Since this routine is only executed by user-space programs, we can leave it as is but mark it as non-executable from the kernel space via Intel's Supervisor Mode Execution Prevention (SMEP) feature [49] to prevent attacks.

4.7 Evaluation

In this section, we evaluate our techniques from three perspectives: technique utility, security improvement, and performance overhead. We first show the results of our analysis proposed in Section 4.3.1 on a variety of kernel software, including the FreeBSD kernel, the entire MINIX microkernel system (including all the user-space servers) and the BitVisor hypervisor. Next, we quantitatively evaluate the improvement on CFI protection achieved by this work. Finally, we evaluate the performance cost of our fine-grained CFI enforcement on FreeBSD 10.0 and the MINIX 3.2.1 systems.

4.7.1 Technique Utility

We evaluate the utility of the proposed analysis on three different kernel systems (e.g., FreeBSD, MINIX, and BitVisor) by counting the source lines of code (SLoC) using the open-source tool `SLOCCount` [50] against the number of constraint violations encountered. We include all the user-space servers of the MINIX microkernel because they implement core operating system functions (e.g., `vfs` implements a file system). It is worth noting that when counting the SLoC of MINIX, we do not include library code that is statically linked to these servers, but list them separately.

We show our results in Table 4.3. We have seen only one violation in BitVisor, where some code uses a constant pointer to a global function pointer for saving a replaced handler. We manually rewrite the code so that it directly references the global function pointer. In FreeBSD, six of the seven violations are in device drivers. Surprisingly, we found that one violation is because of a driver purposely overflowing structure fields in assignment operations. For the ease of analysis, we opt to exclude these drivers from the kernel image, as they are not required by our computer's configuration, rather than to manually fix them. However, the other violation is in the core part of the FreeBSD

	SLoC	A1	A2
FreeBSD	9,280,785	0	7
MINIX	16,276	0	0
vfs	9,783	0	0
vm	6,633	0	0
pm	2,856	0	0
rs	3,221	0	0
ds	587	0	0
pfs	2,625	0	0
sched	393	0	0
libc	153,264	0	0
libsys	4,336	0	0
libmthread	1,532	0	0
BitVisor[7]	34,987	0	1
total	**9,517,278**	**0**	**8**

Table 4.3. The amount of code being analyzed *vs.* the number of violations detected

kernel, which manages a special group of function pointers on the heap. Fortunately, the used data pointers neither point to a structure field nor to any function pointer variables, therefore, we manually identify the targets and add them to the CFG.

4.7.2 Security Evaluation

In the security evaluation, we answer: (1) how much stricter our enforcement of CFI is relative to existing CFI implementations and (2) how many gadgets remain. To quantitatively measure the improvement, we proposed a new metric, called *Average Indirect targets Allowed* (AIA). In addition, we also show the distribution of number of allowed targets per indirect control transfer (calls and returns) in the FreeBSD kernel, MINIX microkernel, and two of its crucial user-space servers. To answer the second question, we analyze the ROP gadgets remaining in the fine-grained CFI-enforced FreeBSD and MINIX using the open-source tool ROPgadget [51].

Researchers previously proposed a metric to evaluate the effectiveness of CFI enforcement, called *Average Indirect target Reduction* (AIR), proposed by Zhang *et al.* [25]. The problem with AIR is that it does not provide an intuition about how much a CFI implementation over-approximates the CFG, especially when the code base of the protected program is large. Imagine an x86 program with 1MB code. Given that every

	Fine-grained CFI		Signature-based CFI		Coarse-grained CFI	
	`call/jmp`	`ret`	`call/jmp`	`ret`	`call/jmp`	`ret`
FreeBSD	6.64	10.41	35.61	11.38	40,166	175,400
MINIX	4.67	5.14	16.56	9.29	688	2,008
vfs	2.89	5.11	4.63	5.38	578	2,289
vm	2.14	4.01	3.43	4.39	402	1,149
pm	2.55	2.93	4.88	5.38	281	608
rs	1.68	4.61	3.13	4.94	354	1,100
ds	1.67	3.85	2.89	4.13	201	561
pfs	2.54	4.95	4.42	5.43	912	2,785
sched	1.23	2.62	2.45	2.91	197	333
BitVisor	3.84	1.89	64.27	6.55	911	3,864

Table 4.4. Average Indirect Targets (AIA) for indirect branches in kernel softwares

byte could potentially lead an instruction, a CFI implementation achieving AIR of 99% still allows 10K targets for each indirect control transfer on average. However, in practice, an indirect control transfer often has many fewer targets (e.g., <10). Moreover, recent attacks [11, 13, 12] on coarse-grained CFI implementations show that an adversary often needs only a small subset of unintended control flows to realize her purpose. Fine-grained CFI can also be attacked if the target sets allow an attacker to fulfill her needs [20].

In order to better illustrate the protection quality, we propose the *Average Indirect targets Allowed* (AIA) metric, as defined in Definition 1. It is worth noting that this metric can only be used to compare CFI techniques on the same program as different programs can be constructed differently, yielding vastly different results.

Definition 1 (**Average Indirect targets Allowed (AIA)).** *Let* I_1, I_2, ..., I_n *be indirect control transfer instructions in the program, and* T_1, T_2, ..., T_n *be the set of allowed targets for them respectively. We define*

$$\text{AIA} = \frac{1}{n}\sum_{i=1}^{n} |T_i|.$$

To evaluate, we assume a coarse-grained CFI policy that uses two target sets as adopted by some recent work [25, 9] and a signature-based CFI policy as used elsewhere [21, 22]. We show the results in Table 4.4. From this comparison, our techniques enforce a much tighter policy than a traditional CFI implementation as it further re-

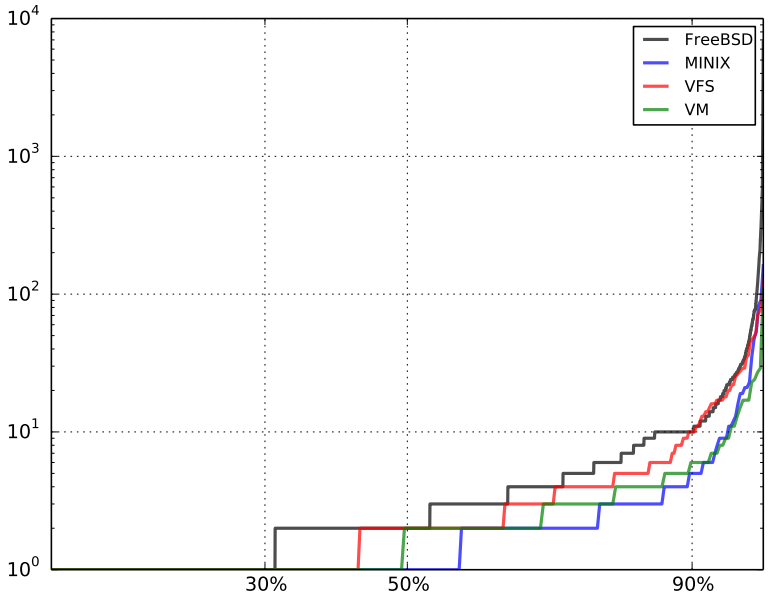

Figure 4.5. The accumulative distribution of indirect branch targets in FreeBSD and MINIX

duces over 99.6% return targets and over 99.1% call/jump targets that are otherwise allowed. Our approach also has higher precision than signature-based CFI as it reduces between 71.80% and 94.03% of call/jump targets across the three different kernels. The signature-based approach performs slightly better on MINIX servers because they have fewer address-taken functions and less assembly code, but still is between 37.6% and 49.8% worse than our proposed approach. In addition, we also find signature-based CFI will break both the FreeBSD and MINIX system due to missed targets, such as the one caused by signature cast, e.g., `int (*)(void)` to `int (*)(struct irq_hook *)`.

Next, we show the distribution of the number of allowed targets per indirect control transfer for FreeBSD, MINIX microkernel and its two biggest user-space servers in Figure 4.5. In summary, over 30% of indirect control transfers have fixed targets and over 90% have fewer than ten targets. However, in both FreeBSD and MINIX, there are a few indirect control transfers that have a large number of allowed targets, e.g., the return of `printf` function in FreeBSD has over 5K allowed targets. Note that, although every allowed target corresponds to a valid edge in the CFG, such flexibility could make the indirect control transfer an attractive target to adversaries and enable adversaries

to find potentially useful code for exploitation. Though the original CFI proposal [2] has suggested the use of a shadow stack for more restrictive protection, how to efficiently implement a shadow stack in system software remains an open problem. Safe stack [52, 45] has better performance than shadow stack, but it changes the stack layout and hence has compatibility issues when applied to kernel software. We will explore optimizations to these approaches as future work.

Finally, to measure the impact on code-reuse attacks from the perspective of reusable code, we analyze the ROP gadgets remaining in the instrumented systems using the open-source tool ROPgadget version 5.2. ROPgadget finds 61,565 gadgets in the original FreeBSD kernel and 2,164 gadgets in the original MINIX. The numbers reduced to 388/30 after our instrumentation. We verified all control transfers (both *direct* and *indirect*) allowed by the CFG and confirmed that none of these gadgets could be reached through a control flow we authorize. However, recent work [11, 13, 12] has shown that the adversary can use more complexed code sequences than traditional gadgets for attacks, thus we provide this analysis as a lower bound of reusable code.

4.7.3 Performance Evaluation

In this section, we evaluate the performance overhead of our fine-grained CFI implementation on FreeBSD 10.0 and MINIX 3.2.1 including all of its user-space servers. Our test machine consists of a dual-core processor running at 2.8GHz with 1GB memory. We use the unmodified systems as the baseline and evaluate the modified versions that are protected by both the traditional coarse-grained CFI and our fine-grained CFI techniques. We implement the coarse-grained CFI using two unified target tables, e.g., one for indirect calls/jumps and the other for returns.

Before presenting the actual results on performance overhead, we first show the distribution of different instrumentations in Table 4.5. These statistics indicate that each of the three types of indirect control transfers are well represented in real world programs, so we need to apply specialized instrumentation to them to save both runtime and memory overheads. For example, indexing the system call table is an example of a base-plus-index indirect control transfer that requires no additional enforcement.

	Basic	Fixed-target	Unchanged
FreeBSD	13,143	6,019	1,207
MINIX	138	199	11
vfs	269	184	22
vm	168	154	3
pm	89	165	5
rs	128	161	2
ds	75	101	3
pfs	360	307	20
sched	53	104	1

Table 4.5. The distribution of different instrumentations applied

Adding extra instrumentation to unify the implementation of restricted pointer indexing or even inserting a label at system call handlers as proposed by original proposal is unnecessary, given the frequency that this instruction is executed.

To evaluate the performance, we run the UnixBench 5.1.2 as the microbenchmark and kernel building as the macrobenchmark. We chose UnixBench because it runs on both FreeBSD and MINIX. We show the runtime performance overhead in Figure 4.6. In MINIX, the fine-grained CFI instrumentation incurs only 2.02%/5.64% (average/-maximum) overhead while the coarse-grained CFI instrumentation incurs 4.14%/7.55% overhead. In FreeBSD, the overhead is higher but the fine-grained CFI still performs better than the coarse-grained CFI (i.e., 11.91%/42.03% vs. 13.60%/50.69%). We attribute much of the performance difference between FreeBSD and MINIX to the cache performance: a monolithic kernel like FreeBSD has one large set of target tables while in a microkernel, each component has its own set of target tables, leading to better locality. We will explore how to colocate these tables in FreeBSD to improve the cache performance as a future optimization. For the macrobenchmark, the fine-grained CFI uses 1.82%/0.76% more time while coarse-grained CFI uses 2.01%/2.29% more time to build the FreeBSD/MINIX system. We believe that enforcing fine-grained CFI can be as efficient as existing coarse-grained CFI implementations.

Enforcing CFI uses more memory. On one hand, the instrumentation will require more instructions for the authorization of indirect control transfers. On the other hand, restricted pointer indexing requires saving all control data into target tables, which are

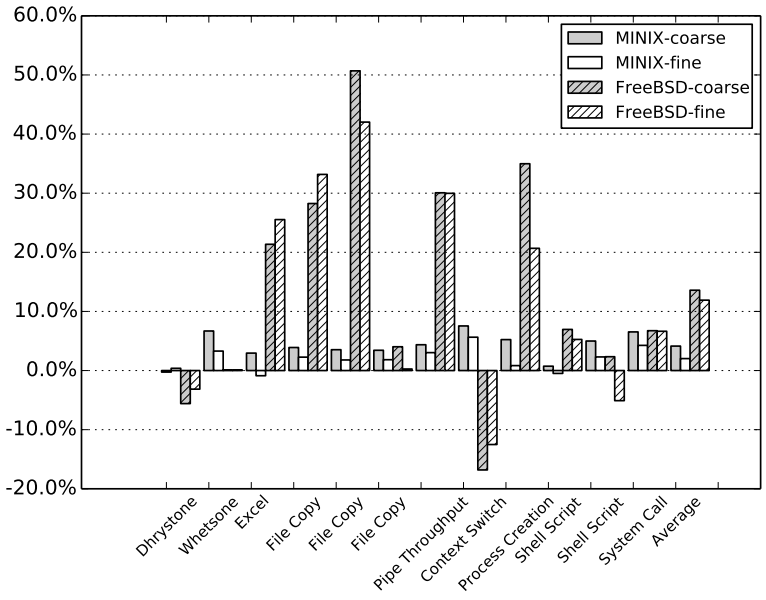

Figure 4.6. The performance overheads of instrumented kernels under both coarse-grained and fine-grained CFI

absent from the original system. On average, the typical coarse-grained CFI uses 25.3% more code memory. However, the fine-grained CFI uses 13.48% less static data memory and 9.01% less code memory than the coarse-grained CFI. There are two reasons. First, the fixed-target instrumentation saves both code memory, i.e., it has fewer instructions, and data memory, i.e., it does not need a target table. Second, unlike coarse-grained CFI, not all call sites necessarily have an entry in the target table because some call sites are not mapped to any return instruction. For instance, the function to restore user context has an `IRET` instruction instead. Therefore, even though fine-grained CFI may have redundant targets in the tables, it has better memory use than the coarse-grained version in our experiment.

4.8 Discussion

Kernel preemption is a technique to increase the responsiveness of the overall system. Basically, it allows interrupting kernel execution (e.g., system call handling) to schedule a higher-priority task onto the processor. This affects the kernel control flow in an

unpredictable way, as the kernel execution can be interrupted and returned to any instruction boundary. In this work, we build FreeBSD as non-preemptive[8] to avoid this issue. Although a preemptive kernel may increase the system responsiveness, it could degrade the system performance because of increased context switching. However, a kernel exception could only happen at certain places, and its handling and return are deterministic. FreeBSD uses kernel exceptions to emulate the virtual 8086 execution, which purposely modifies the exception return address. To avoid increasing the complexity of our implementation, we simply disable this feature in our kernel.

Kernel modules are a technique to extend kernel functionalities (e.g., supporting new devices, etc.) without recompiling the whole kernel. They are dynamically loaded into the kernel space, and invoked upon selected system events. This brings up challenges to the execution integrity enforcement for operating system kernels. First, kernel modules introduce a violation of DEP policy by initially mapping their code pages as writable for loading, and remapping them as executable afterwards. To address this issue, we envision kernel monitoring mechanisms such as SPROBES should only allow authorized kernel modules to be loaded and their code pages are checked against the recorded hash values before put into use. Second, loading kernel modules extends the kernel's control-flow graph, thus our CFI mechanism has to adapt to such changes. MCFI [21] is capable of combining CFGs from different modules and enforcing CFI for dynamically-linked user-space programs, and we envision similar techniques can be applied to kernel software as well. In this dissertation, we simply disable kernel modules, and we leave the better support of kernel modules to future work.

Control-Flow Bending [20] combines data attacks with control-flow hijacking attacks to enable malicious computation and showed the requirement of a stack integrity to protect user-space applications. As mentioned in Section 4.1, we do not protect the kernel from data attacks. However, our fine-grained CFI policy makes the control-flow bending attacks less likely to succeed because of the smaller size of targets allowed at runtime. Unfortunately, we found that kernel software includes some cases with large target sets, i.e., the return of functions that are commonly called, so these may also be

[8] MINIX cannot be built as preemptive by design. In fact, many operating systems do not enable kernel preemption by default (e.g., Linux).

prone to control-flow bending attacks. The authors of that work recommend applying the shadow stack approaches to prevent such attacks, but no kernel implementation of that technique is available yet. Efficient kernel implementations of stack integrity is challenging for three reasons. First, as x86-64 has largely dropped the support for segmentation, researchers propose to protect the shadow stack by hiding its location, while a kernel attacker can simply scan the page tables to uncover the location. Second, typical shadow stack implementations maintain a per-thread shadow stack pointer and swap it between memory and register when pushing/popping return addresses. This incurs non-trivial performance overheads. Third, exception handling and other instructions (e.g., `iret`) assume a specific, rigid stack layout that cannot easily be changed. We will explore an efficient implementation of such mechanisms in kernel software as future work.

4.9 Summary

In this chapter, we have presented an automated approach for enforcing fine-grained control-flow integrity for kernel software. We leverage and enforce constraints kernel software adheres to for function pointers to compute a fine-grained control-flow graph and authorize the indirect control transfers in the resulting graph comprehensively and efficiently. We also address system challenges that arise only in kernel software to protect its entries and exits. The result shows that our approach is both effective, i.e., eliminating over 70% of the indirect call targets in the FreeBSD and MINIX that are otherwise allowed by current fine-grained CFI implementations and 99% of the indirect control targets in a typical coarse-grained CFI implementation, and efficient, i.e., incurring less overhead than a comparable coarse-grained CFI implementation.

Chapter 5

Guarding Control Flows using Intel Processor Trace

In this chapter, we explore the effectiveness of using Intel Processor Trace (a hardware mechanism to log complete control-flow traces) to construct an online CFI system that is capable of enforcing various types of CFI policies over unmodified binaries, including the Firefox browser and its jitted code. We evaluate its performance costs and examine how they can be further reduced with possible hardware enhancements.

To date, researchers have focused on software-based implementations of CFI. Software-based approaches can be divided into three categories: compile-time instrumentation, static binary instrumentation, and runtime instrumentation. The main advantage of compile-time instrumentation [22, 21, 9, 23, 53, 54, 55, 56, 57, 58] is that it has better performance than other approaches. However, it has two main limitations. First, such techniques can only instrument programs supported by that compiler. Second, they "fix" the CFI defense for resultant software, preventing systems from customizing their CFI defenses at runtime (e.g., to tighten security or balance performance). Static binary instrumentation [2, 26, 27, 59] could instrument programs written in a variety of languages and legacy binaries, but as yet cannot enforce as accurate CFGs as compile-time, cannot apply some compile-time optimizations, and also fixes the instrumentation. Runtime instrumentation [60, 28] has the flexibility of turning the protection on

and off, but the overhead is far higher than for fixed instrumentation.

Hardware-based CFI enforcement avoids the limitations above by using hardware-generated logs to enforce CFGs, but current methods either fail to provide a complete CFI defense or incur unacceptable overheads when applied completely. For example, researchers have applied the Last Branch Record (LBR) feature to enforce CFI defenses using its ability to store 8-16 control transfers [61, 62, 63] to improve performance. Despite the performance benefits, defenses built on LBR are fundamentally limited by the capacity of the LBR stack, which misses most indirect control transfers. While researchers have shown that LBR traces may be augmented by heuristics [61, 62], others have shown that adversaries can evade these heuristic defenses [12]. To achieve complete enforcement, researchers have explored using the Branch Trace Store (BTS) to record control transfers [64] or leveraged the Performance Monitoring Unit (PMU) to trigger interrupts when the LBR fills [65]. Some have estimated that the BTS incurs a significant performance slowdown (20x-40x) [63], and the PMU will trigger interrupts after every 16 indirect control transfers.

We explore the effectiveness of a CFI enforcement built using a new, commercially available hardware feature, called *Processor Trace* (PT) on Intel CPUs [15]. Intel PT was designed to aid in debugging and failure diagnosis by recording the minimal information necessary to reconstruct complete control-flow traces. For instance, researchers have previously developed offline techniques that are able to diagnose complex failures using Intel PT [66]. Given Intel PT's ability to record control flow traces, we investigate using Intel PT for *online* enforcement of CFI policies by designing and implementing GRIFFIN, an operating system mechanism that leverages the Intel PT feature to enforce CFI policies over unmodified binaries. When a binary is run on GRIFFIN, GRIFFIN restricts the binary's execution by comparing each indirect control transfer in the binary's execution trace captured using Intel PT to a CFI policy for the binary. GRIFFIN may enforce CFI policies produced by executing the binary program (e.g., shadow stack) or leverage CFI policies produced elsewhere. In addition, GRIFFIN may change the CFI policies being enforced dynamically to manage performance and/or customize enforcement.

Our goal is to build a high performance CFI enforcement mechanism that is capable of enforcing a variety of CFI policies over unmodified binaries. Implementing online CFI enforcement using Intel PT to meet this goal presents several challenges. Intel PT logs the information necessary to reconstruct a complete control-flow trace, but leaves the reconstruction of the trace to post-processing. To enforce some CFI policies, we need to know the call sites executed at runtime, but Intel PT does not record call sites, as they can be inferred from other information. Thus, we must disassemble the binary and interpret the logged trace buffers to recover the program's control-flow trace necessary for CFI enforcement. We explore design choices that optimize the efficiency of trace processing and CFI enforcement. First, we design a data structure for fast lookup of basic block information for control-flow reconstruction that works efficiently for current programs of all sizes. Second, GRIFFIN leverages idle cores to run disassembly, trace processing, and CFI enforcement in parallel. The GRIFFIN design carefully prevents the need for synchronization delays among these parallel activities. While sequential processing may be necessary for enforcing some CFI policies, we retain parallel processing of trace buffers for the most time and only perform necessary CFI checks in a sequential manner.

We have implemented a prototype of GRIFFIN in the Linux 4.2 kernel that controls the execution of unmodified, user-space binaries and dynamically generated (i.e., jitted) code. We show that GRIFFIN is capable of enforcing three types of CFI policies, coarse-grained CFI, fine-grained CFI, and stateful CFI, on both forward edges (i.e., calls and jumps) and backward edges (i.e., returns), enabling GRIFFIN to enforce restrictive CFI policies recommended by researchers [2, 20] and to balance security with performance. GRIFFIN marks a significant improvement in hardware-based CFI enforcement mechanisms, by providing complete and flexible CFI enforcement that performs comparably to software-based instrumentation methods. When enforcing a fine-grained CFG on forward edges and a shadow stack [2] on backward edges, GRIFFIN incurs an overhead of 11.9% on the SPECint benchmarks and 6.2% on SPECfp benchmarks, as compared to 11.6% and 6.0% overhead, respectively, for software-based shadow stack enforcement alone [67]. We utilize memory and idle core resources to achieve this performance, so

we evaluate the impact of such resource usage, finding that GRIFFIN's physical memory allocation depends mainly the program's size and processing backlog and that performance degrades gracefully with fewer cores.

Despite a highly-optimized GRIFFIN implementation that achieves performance comparable to software-based instrumentation techniques, such performance is generally considered insufficient for widespread adoption. We propose two modifications to Intel PT logging that have the potential to improve performance and reduce memory usage for CFI enforcement on forward edges. First, by logging the indirect call sites as well as indirect call targets, GRIFFIN can enforce fine-grained CFI policies without performing control-flow reconstruction, resulting in a 89.50% improvement in trace processing on average for SPEC benchmarks. We no longer need to log the conditional branches for this approach, resulting in a 59.86% reduction in trace size on average (although some traces may increase in size slightly). However, this first approach undermines our ability to enforce stateful CFI policies because we lack information to reconstruct control flows to determine states. We propose a second approach that applies selective logging of conditional branches to collect the control-flow information necessary to evaluate stateful policies. For one stateful CFI policy [63], we find that the trace size for the worst case program increases from 0.88% of its original trace size after the first modification to 1.43% after enabling selective logging. Finally, we propose how to integrate our findings with proposed hardware features, in particular Intel's Control-flow Enforcement Technology (CET) [68], to enforce restrictive CFI comprehensively with the potential for improved performance and flexibility over software instrumentation.

5.1 Intel Processor Trace

In this section, we provide the background on Intel Processor Trace (PT) [15]. Intel PT is a recent hardware feature that enables the recording of complete control flows with low overhead. When properly configured, Intel PT hardware generates a variety of *data packets* that encode program flow information such as branch targets and branch taken

Packet	Size	Usage
PGE	$\leqslant$8	Packet Generation Enable packets provide the IP at which the tracing begins
PGD	$\leqslant$8	Packet Generation Disable packets mark the end of tracing
TNT	1	Taken/Not-Taken packets indicate the direction of conditional branches
TIP	$\leqslant$8	Target IP packets provide the target for some control-flow transfers
FUP	$\leqslant$8	Flow Update packets provide the source address for asynchronous events
PSB	16	Packet Stream Boundary packets are unique patterns in the output log to serve as sync points for software decoders

Table 5.1. Trace packets and their usage in Intel PT

indications. A software decoder can reconstruct the exact control flow when combining the recorded packets with program binaries. These packets may also include contextual, timing and bookkeeping information to enable richer analyses such as performance profiling.

To understand how Intel PT enables control-flow tracing, we show the main types of its generated data packets in Table 5.1. In the simplest form, Intel PT informs the beginning and the end of tracing through `PGE` and `PGD` packets, respectively. Throughout the program execution, Intel PT generates `TNT` packets to log if conditional branches are taken (i.e., `jcc`) and `TIP` packets to log targets of indirect branches (i.e., `call*` and `ret`). With such information, a software decoder can reconstruct the control flows based on program binaries. It is worth mentioning that direct branches (i.e., `jmp` and `call`) do not trigger any packets because of their deterministic effects on control flows.

Intel PT employs various techniques to minimize the size of generated packets. For instance, `TNT` uses one bit to indicate the direction of a conditional branch, and a one-byte `TNT` packet can log up to six executed branches. Additionally, Intel PT compresses a return into a bit in `TNT` if the return target can be determined from a previous matching call. To reduce the size of `TIP` packets, Intel PT compresses the target address based on the previous address by suppressing the same upper address bytes. All these features require a stateful processing of the logged trace packets, which presents a challenge for

parallel processing.

Intel PT outputs packets directly to physical memory to avoid the cost of address translation. For flexibility, it can be configured to use multiple buffers that are not contiguous in the physical address space through a table-like data structure. An interrupt can be triggered when a buffer becomes full, but the interrupt is not precise. Writes to the next buffer may have occurred when the interrupt is signaled, so one must ensure that all writes are accounted for.

Intel PT supports both user-level and kernel-level tracing. It also supports tracing selected processes and threads. In GRIFFIN, we focus on monitoring user-level executions of interested processes.

5.2 Threat Model

When designing GRIFFIN, we focus on defending against user-space code-reuse attacks based on the following threat model. We run GRIFFIN in the kernel. We assume the adversary has no control over the operating system kernel. This makes sure that the adversary cannot directly tamper with GRIFFIN. We assume the protected programs are benign but may contain memory safety errors (e.g., buffer overflow and/or use-after-free bugs) that enable the adversary to write to arbitrary memory locations within the address space. In particular, the adversary can corrupt control data (e.g., return addresses on the stack or function pointers on the heap) to subvert the control flows to launch code-reuse attacks. We assume the protected programs are compliant with data execution prevention (DEP). That is, the program should not modify its own code or map a code page as both writable and executable under legitimate execution.

5.3 Design Overview

In this section, we provide an overview of GRIFFIN's design. GRIFFIN is a hardware-assisted CFI enforcement system for defending against user-space code-reuse attacks. GRIFFIN leverages Intel PT to record the complete user-level execution of a monitored program and performs online control-flow checks based on the recorded execution

trace. GRIFFIN is capable of enforcing such checks on indirect control transfers both for *forward edges* (i.e., indirect calls and jumps) and *backward edges* (i.e., returns). The GRIFFIN system design focuses on enforcement, assuming that CFI policies are given.

We design GRIFFIN to support multiple types of CFI policies to enable flexible trade-offs between security and performance. The simplest but least secure CFI policies GRIFFIN supports are the *coarse-grained* policies. Under this class of policies, GRIFFIN checks only if the destination of an indirect control transfer is legitimate (i.e., one of the allowed indirect branch targets in the enforced policy). GRIFFIN also supports *fine-grained* policies which are more secure than the coarse-grained policies because GRIFFIN checks whether both the source and destination of an indirect control transfer are a legitimate pair. To achieve the best security offered by CFI, GRIFFIN supports *stateful* policies. Under such policies, GRIFFIN additionally uses execution state to restrict forward and/or backward edges. For example, we show how GRIFFIN enforces a stateful *shadow stack* [2] policy on backward edges, which restricts return targets to their corresponding call sites. In Section 5.6, we evaluate the performance of a type of policy we call the *combination policy*, which enforces a fine-grained policy on forward edges and a shadow stack on backward edges, as recommended by researchers [2, 20]. Stateful, forward-edge policies are a relatively new area of research, but we design GRIFFIN to enforce a proposal to restrict indirect call sites to different sets of legal targets depending on the program's runtime control flow [63].

The goal of the GRIFFIN design is to optimize the performance of CFI checking when leveraging Intel PT traces. To do so, GRIFFIN performs both non-blocking and blocking CFI checks to achieve better performance without sacrificing security. Non-blocking checks are triggered when an Intel PT trace buffer becomes full. During such checks, the program continues to execute. Blocking checks are done when the monitored program makes a security-sensitive system call. Since such a call may impact the integrity of the system, GRIFFIN intercepts those system calls and performs checks of all the control transfers that have not yet been checked before passing the calls to the kernel.

Today's computers rarely run at 100% CPU utilization on all cores. GRIFFIN leverages idle cores on a multi-core system to perform security checks by having multi-

	Coarse-grained	Fine-grained	Stateful
Trace packets decoding	✓	✓	✓
Control-flow reconstruction	×	✓	✓
Sequential processing	×	×	✓

Table 5.2. Requirements for enforcing different types of policies to GRIFFIN: Trace packets decoding is mentioned in Section 5.4.1; control-flow reconstruction is elaborated in Section 5.4.2; and sequential processing is described in Section 5.4.3

ple worker threads perform control-flow checks simultaneously. The trace buffers of Intel PT may be dependent (see Section 5.1). For instance, both return compression and instruction address compression may rely on the state stored in a previous trace buffer. To enable worker threads process Intel PT trace buffers simultaneously, we must make the buffers independent. Fortunately, this is achievable on Intel PT by forcing a Packet Stream Boundary (PSB) packet at the beginning of each Intel PT trace buffer.

5.4 System Design

In this section, we present GRIFFIN's design in detail. The design is motivated by the types of policies GRIFFIN can enforce. We begin with the simplest policies, the coarse-grained policies, then describe the extensions necessary to support fine-grained and then stateful policies. We summarize the requirements for each type of policy in Table 5.2. In this section, we focus on the processing of a single user thread without jitted code. We will discuss the implementation details for multiple threads and jitted code in Section 5.5.

5.4.1 Coarse-Grained Policy

To enforce coarse-grained policies, we do not need to reconstruct the control flow because the destinations of indirect control transfers are given in TIP packets. To quickly check if an indirect control transfer is legitimate, GRIFFIN maps a page at a constant offset from each code page to store whether a code location is legitimate. We refer to it as a *coarse-grained policy page*. The distance between a code page and a policy page is a constant for fast lookup. We explain how this constant is chosen in Section 5.5.1. If a

code location is legitimate for indirect control transfers, then the corresponding location on the policy page is set to 1; otherwise, it is 0. To support dynamic libraries, GRIFFIN monitors library loading in each monitored process. When a library is loaded, GRIFFIN allocates a coarse-grained policy page for each code page in the library as well, simply adding legal destinations. Since the policy pages are set up at library loading time, GRIFFIN's worker threads can perform control-flow checks in parallel without worrying about synchronization. It is worth noting that we do not need to disable return compression since a compressed return is guaranteed to be legitimate as it matches with its corresponding call. The returns that are not compressed are recorded with `TIP` packets.

5.4.2 Fine-Grained Policy

Unlike the support for the coarse-grained policy, we encode the fine-grained policy in a bitwise matrix. We refer to it as the *policy matrix*. Each row of this matrix corresponds to a possible source in indirect control transfers, and each column corresponds to a possible destination in indirect control transfers. An entry in the matrix is set to 1 if the source-destination pair is legitimate; otherwise, it is set to 0. The matrix is stored at a constant virtual address for a monitored process. We grow this matrix dynamically by adding rows and columns when a library is loaded.

To enforce fine-grained CFI policies, we need to know the source and destination of each indirect control transfer. However, the source address is not directly available in the trace collected by Intel PT. To recover it, we need to reconstruct the control flow, so that we know the source address when an indirect control transfer occurs. Next, we first use an example to explain how to reconstruct the control flow based on Intel PT's trace packets and program binaries. Then, we describe how we make the control flow reconstruction run efficiently.

To reconstruct the control flow, we take as input Intel PT's trace packets and program binaries. The basic idea is to disassemble the binary and follow the execution by tracking the trace packets. We illustrate this process with an example shown in Figure 5.1. The initial `PGE` packet identifies that the execution begins at block `A`. Then, the direct `jmp` instruction at the end of block `A` leads the control to block `D`. Note that, di-

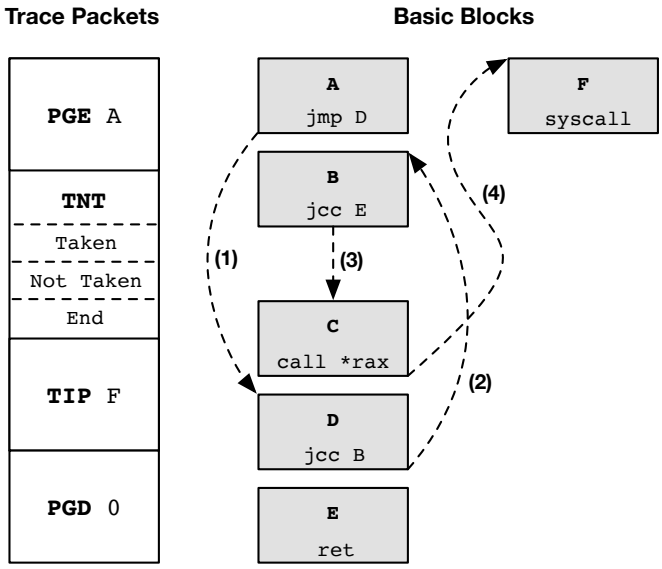

Figure 5.1. An example of reconstructing control flows based on trace packets

rect branches do not generate a trace packet due to their deterministic effect. The first "Taken" bit in the `TNT` packet indicates that the next conditional branch (i.e., the one at the end of block `D`) is taken, thus the control is transferred to block `B`. The conditional branch at the end of block `B` is not taken according to the next entry ("Not Taken") in the same `TNT` packet, and the control falls through to the next block `C`. It ends with an indirect call to block `F` as logged by the `TIP` packet. When a system call is invoked at the end of block `F`, the trace is marked as disabled by the `PGD` packet since we only trace user-level execution.

The process for reconstructing the control flow is straightforward, but the design challenge is how to make it efficient in GRIFFIN's online processing. Instructions, blocks, or functions tend to execute many times during a program's execution. Disassembling the same instructions over and over again is *not* efficient since we only need the information about the basic blocks for control-flow reconstruction. The key design question is how to store and look up such information in an efficient way.

Previous researchers have proposed several approaches to enable fast store and lookups. MCFI uses an array to store information for each code address [21]. However, in practice, program code can be scattered in the address space (e.g., dynamic libraries),

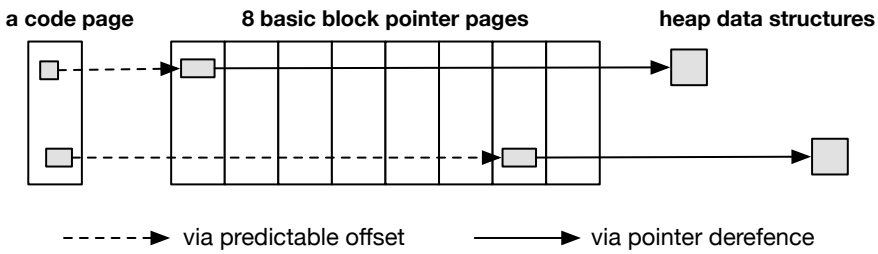

Figure 5.2. Relationship among basic blocks, pointers on the basic block pointer pages, and their heap data structures.

and MCFI leverages sandboxing techniques to restrict the program to use the first 4GB memory region. This requires modifying the system loader and potentially undermines other defenses, such as address space layout randomization, because of lower entropy. fastBT uses a simple hash table schema to map an original basic block to the translated block and reports a low conflict rate for SPEC benchmarks [69]. However, hash conflicts are inevitable when the program binaries are large. Furthermore, the hash table schema requires heavyweight locking for certain operations (e.g., code unloading), which slows both store and lookup operations that may happen in parallel.

To tackle these problems, we trade memory efficiency for lookup performance. Specifically, for each basic block, we allocate a heap data structure to store its information. Then, we allocate 8 pages at predictable offsets from each code page to store pointers to its heap data structures. We refer to such pages as the *basic block pointer pages*. We use 8 basic block pointer pages because every byte in the code page can lead a basic block and the pointer to its heap data structures takes 8 bytes. We show the conceptual layout in Figure 5.2 and explain how we pick the offset in Section 5.5.1.

When a worker thread looks up the information of a basic block, it reads the pointer from the basic block pointer page. If the pointer is `NULL` and thus the basic block is not disassembled yet, the worker thread disassembles the block, allocates a heap data structure to store the disassembled information, and updates the basic block pointer page with the new pointer. Since multiple worker threads may perform read and write to the same pointer on a basic block pointer page, we use the `compare-and-swap` primitive to make sure the pointer write is atomic and occurs only when the present pointer

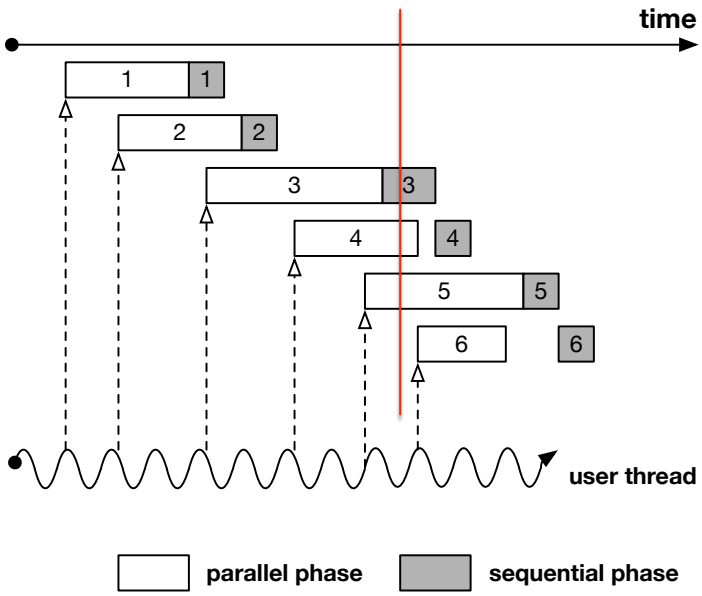

Figure 5.3. Two phases for processing of trace buffers

is `NULL`. It is worth noting that, although `compare-and-swap` is an expensive operation, it happens infrequently at runtime – only the first time a basic block is encountered and disassembled.

In a basic block's heap data structure, we also store a row and column index if the basic block may be the source and/or destination of an indirect control transfer. This allows GRIFFIN to quickly locate an entry in the policy matrix.

5.4.3 Stateful Policy

A stateful policy constrains the targets of indirect control transfers based on the program execution state, such as a call stack. Consequently, to enforce the stateful policy, we need to *sequentially* process the control-flow trace. This is seemingly in conflict with the goal of parallel processing. A key observation is that most computation in GRIFFIN is on parsing Intel PT trace buffers and reconstructing the control flow. The amount of time spent on checking CFI policies is actually small. This observation motivates us to split our parallel processing into two phases: the *parallel* phase and the *sequential* phase. In the parallel phase, multiple trace buffers of a single user thread can be processed by

multiple worker threads simultaneously. The output of each worker thread is a list of calls and returns encountered in a trace buffer. In the sequential phase, the list of calls and returns is processed in the order of execution. Note that, a trace buffer being processed in its sequential phase does not prevent other buffers from entering the parallel phase. For example, at the time denoted by the vertical line in Figure 5.3, buffer 4 and 5 are in their parallel phase while buffer 3 is processed in the sequential phase. This two-phase design provides the desired in-order processing while keeping GRIFFIN highly parallelized.

To enforce the shadow stack check on returns (i.e., backward edges), we simply check if a return matches the call on the top of the call stack. Exceptional cases in shadow stack enforcement are described in Section 5.5.5. To enforce stateful checks on indirect calls (i.e., forward edges), we adapt the design for fine-grained policies with two main changes. First, we extend the policy matrix to store a stateful policy. Specifically, we allocate additional rows in the matrix for sources that have multiple states to store acceptable destinations, associating each new row with the source and its state. Second, if an indirect call has a stateful policy, we store a row index for each acceptable state in the heap data structure of its basic block for fast lookup.

5.5 Implementation

We implemented a prototype of GRIFFIN on Debian 8 with a 64-bit 4.2 Linux kernel running on an Intel i7-6700K quad-core processor (a 6th-generation Skylake processor). To implement online disassembling, we ported an open-source disassembling library for x86 called distorm [70] (12,497 SLoC). We made a few changes in distorm such as adding support for Intel TSX instruction set. We also merged our code for TSX support into the mainstream version 3.3 of distorm. When disassembling a binary, we leverage `MODE` packets in an Intel PT trace to decide if it is x86 or x86-64. This allows GRIFFIN to support 32-bit programs running on a 64-bit kernel. Excluding the distorm code, our prototype consists of 1,625 SLoC in C (with a small amount of inline assembly).

In our prototype, we treat six system calls as security-sensitive: `mprotect`, `mmap`,

`mremap`, `remap_file_pages`, `execve`, `execveat`. This list is consistent with prior work [62, 63] and motivated by the observation that many exploits disable DEP to run injected code after hijacking the control. The list is configurable and can be extended to include other system calls. For example, in our experiments on real-world network applications (Section 5.6.2.2), we add four more security-sensitive system calls (`sendmsg`, `sendmmsg`, `sendto` and `write`) to prevent them from sending information over network after being compromised.

In the rest of this section, we first present how we manage memory. Then we describe how we handle context switch and fork. After describing our support for jitted code, we discuss the corner cases for checking the shadow stack. Finally, we show how we derive a stateful forward-edge policy.

5.5.1 Memory Management

GRIFFIN runs inside the Linux kernel. When it parses the trace buffers and performs control-flow checks for a monitored process, it runs in that process' context. We use both kernel and user memory pages in GRIFFIN. We use kernel pages for two purposes. First, the buffers given to Intel PT for tracing are kernel memory pages. Second, the heap data structures for storing basic block information are also allocated from kernel memory.

We store the coarse-grained policy pages, the basic block pointer pages, and the policy matrix in the user-level address space. The motivation here is that these pages are so frequently accessed that a fast lookup mechanism is key to GRIFFIN's runtime efficiency. To do so, we map these pages either at some constant offset from the corresponding code page (i.e., the coarse-grained policy pages and the basic block pointer pages), or at a fixed location (i.e., policy matrix). The user-level address space is private to each process, enabling GRIFFIN to use the same offset and the fixed location for different processes.

To store these pages in the user-level address space, we use the memory layout as shown in Figure 5.4. Note that the Linux kernel currently locates the main executable of a process at the bottom of the user-level address space, and locates other libraries

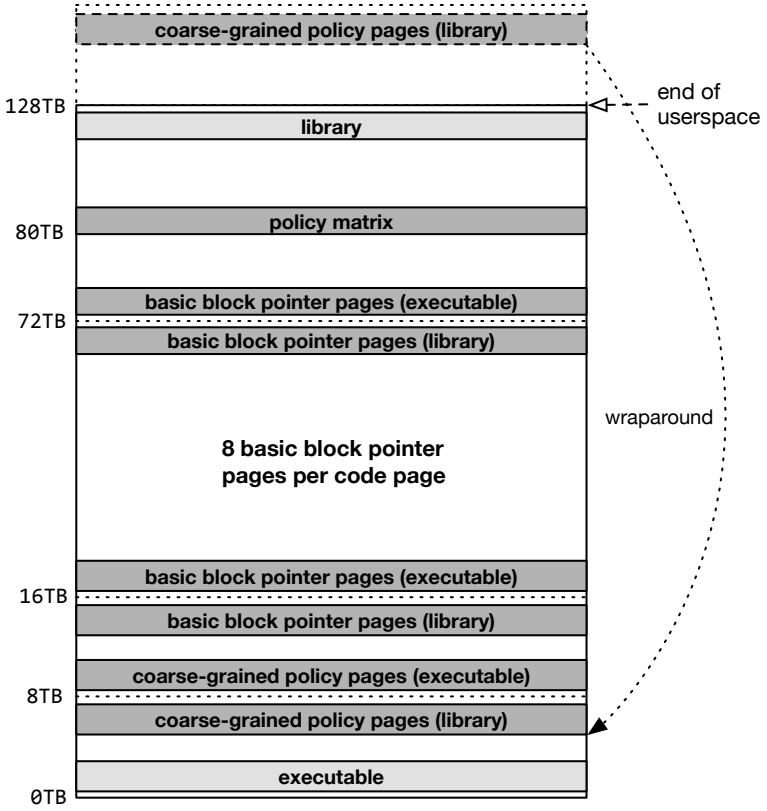

Figure 5.4. The user-level address space layout in GRIFFIN

at the top. The goal is then to find a way to map these pages so that no two pages would overlap each other. Conceptually we divide the 128TB user-level address space into 16 identical ranges of 8TB. We choose the constant offset to be 8TB. So, for each code page, we map its coarse-grained policy page and 8 basic block pointer pages in the subsequent 9 ranges. For libraries, we let the address of these pages wrap around the user-level address space as shown in Figure 5.4. Thus, for a basic block at `addr`, we store its basic block pointer at `((addr&~0x7)+((addr&0x7)+2)*8TB)&(128TB-1)`. One concern is that the coarse-grained policy page and the basic block pointer pages of the executable could overlap those of a library. Fortunately, the executable and the libraries are mapped at different portions of a range, hence those pages will not overlap. It is worth noting that the coarse-grained policy pages and the basic block pointer pages are position-independent. This allows us to share them between monitored processes. We

only need one physical copy for each binary, and they are lazily populated when a code page is accessed.

We store the policy matrix at 80TB. We map a $2^{16} \times 2^{16}$ bitwise matrix at this location. The matrix uses up to 512MB of the process's virtual address space but is lazily populated when dynamic libraries are loaded.

To prevent the monitored program from modifying these user-level memory pages, we map them as read-only user pages in our current prototype. To enable our kernel worker threads to write to these pages without triggering page faults, we set the `AC` bit in the `EFLAGS` register to override the SMAP protection, and clear the `WP` bit in the `CR0` register. This enables memory writes to read-only pages from the kernel. We acknowledge that a more principled approach is to map these pages as *supervisor* pages in the user address space. This requires modification to the memory manager and the page tables. We leave it to future work.

5.5.2 Context Switch

In GRIFFIN, we trace each thread separately. This requires changing the Intel PT configuration state during context switches. We follow the suggestion made in the Intel manual [15] to use `XSAVES` and `XRSTORS` instructions to save and restore the configuration state in context switch. We enabled these two instructions in Linux 4.2 kernel by adding 30 SLoC to its context switch code.

We encountered a limitation of Intel PT when implementing the support for the `XSAVES` instruction. This instruction saves the Intel PT tracing state of the departing thread into a memory buffer and disables tracing, which may trigger an interrupt if the buffer is full. Unfortunately, this interrupt traps *after* the Intel PT state is dumped into memory. This means any changes to the Intel PT tracing state will be lost and *not* be restored when the departing thread is switched back. This becomes a problem when we want to modify the tracing configuration to reset the buffer pointer from which subsequent trace will be logged and to force a `PSB` packet at the beginning of each trace buffer. Our current workaround is to directly modify the saved tracing configuration state in memory.

5.5.3 Fork

The `clone` system call is a UNIX primitive to create a new task (thread/process). We hook the `clone` system call to notify GRIFFIN on task creations. GRIFFIN allocates a trace buffer for each new task and initializes its Intel PT configuration state accordingly. To do so, we directly write to the new task's `XSAVES` area, so that the initialized state will be loaded into the processor's registers when its context is switched in.

To support stateful checking for a new process, GRIFFIN follows the semantics of `fork` and makes the child process inherit the call stack state from its parent process. Specifically, GRIFFIN flushes the current trace buffer of the parent process and informs the sequential phase to duplicate the call stack state after processing the buffer. Note that though the sequential phase of the child process is blocked until the duplication is done, the execution of the child process and its parallel-phase trace processing are not.

5.5.4 Just-In-Time Compilation

Managed languages such as JavaScript often implement a Just-In-Time (JIT) compiler to transform the byte code into native code at runtime for faster execution. In the browser environment, an adversary often controls the input program. Researchers have demonstrated a technique called JIT spray to coerce the JIT compiler to generate desired byte sequence with a carefully crafted input program [71]. This enables the adversary to deliver her shellcode onto an executable page and achieve arbitrary code execution after redirecting the control flow.

Protecting programs that have JIT engines such as a browser is non-trivial to GRIFFIN. The key challenge is that jitted code can change over time, so that GRIFFIN can use obsolete basic block information to reconstruct the control flows, causing undefined behaviors. A simple flush of the basic block information upon changes to jitted code does not work in practice because pending trace buffers may rely on the old basic block information. Keeping a history of code changes and precisely matching trace packets with the right code version can be both difficult and expensive.

To tackle this challenge, we propose to refactor the JIT engine so that it never modifies existing code in place. As a proof-of-concept, we refactored the Firefox's baseline JIT

engine to eliminate in-place code updates that alter the original control flows. Specifically, we made the following changes.

Code retirement. When the jitted code is no longer needed, Firefox will poison the code by converting it to no-ops, and unmap the pages via the `munmap` system call. We modified the JIT engine to skip poisoning. When Firefox unmaps an executable page, we mark the page and defer the actual page reclaim until there is no pending trace buffer that was generated before the `munmap` system call.

Incremental garbage collection. Firefox uses an incremental garbage collector that divides the mark-and-sweep process into time slices for better responsiveness. The incremental garbage collector has to account for new object allocations between slices of the mark-and-sweep process. Thus, the JIT engine dynamically inserts jump instructions to execute code that marks newly allocated objects. We made the jump instructions persistent and changed the code to mark newly allocated objects only if the mark-and-sweep process is under way.

5.5.5 Shadow Stack

When implementing the shadow stack, we have to handle the following corner cases.

Intel TSX. Intel Transactional Synchronization Extension (TSX) is a hardware mechanism that exposes and exploits hidden concurrency in multi-threaded applications. Intel TSX enables transactional execution over the programmer-specified critical region through newly-defined hardware interfaces. If a transaction aborts, the process state will be rolled back as if none of the instructions within the transaction has been executed. Intel PT logs TSX events when a transaction begins, commits or aborts. To support TSX, we store these events together with calls and returns. If a transaction aborts, the calls and returns in the aborted transaction are skipped.

Signals. Signals are asynchronous events to processes in Linux. To deliver a signal, the kernel pauses the normal process execution and transfers the control to a specified handler. Because the signal handler is "called" directly from the kernel, its return is handled by the kernel as well. Specifically, the kernel prepares a special return address to a trampoline that exits the signal processing. Consequently, the return from the sig-

nal handler to the trampoline does not have a matching call. We allow this return as long as the current execution is in signal handling and the target is the kernel-specified trampoline.

setjmp/longjmp. The standard `setjmp` and `longjmp` primitives allow a program to skip stack frames on function returns. When a `longjmp` happens, a subsequent return may not match the last call but instead matches an earlier call on the shadow stack. To handle this case, a principled approach is to detect `setjmp` and `longjmp` before allowing a return to match a call that is not on the top of the shadow stack. However, our current prototype simply allows such matching without checking if it is caused by a `longjmp`. We leave a better implementation to the future work.

Exceptions. C++ programs use a dedicated return instruction to transfer the control between a `try` statement and a `catch` statement when an exception is thrown. We uncover all exception handlers by parsing the `.eh_frame` section and the `.gcc_except_table` section. And we allow the return in the `_Unwind_RaiseException`, which handles the exception dispatching, to target a valid exception handler.

On-Stack Replacement. On-Stack Replacement (OSR) is a runtime technique commonly used by JIT engines to switch between different implementations of the same function [72]. It works by trapping the execution and replacing the stack frame with a new one as if the process was running in a different function. We modified the Firefox to inform GRIFFIN of the OSR entries, and forgive an unmatched return if it targets a valid OSR entry.

Inline Caching. Firefox uses inline caching to implement the JavaScript `switch` statement due to the dynamic nature of object comparisons. Therefore, the matching between the object in the `switch` statement and the `case` statement is performed in a different function. Once the function found a matching `case`, it modifies its own return address to complete the control transfer. We modified Firefox's JIT compiler to emit semantically-same indirect jump instructions to avoid causing shadow stack violations, and leave them to be handled by the forward-edge policy.

	function pointer	return address
stack	30	2
heap	20	0
data	10	0
bss	20	0
total	**80**	**2**

Table 5.3. Successful exploits in RIPE benchmark on Debian 8.2 (ASLR and stack protection disabled) without the protection of GRIFFIN

5.6 Evaluation

In this section, we evaluate the effectiveness and the performance of GRIFFIN. We run the RIPE benchmark [73], a collection of exploits, as a sanity check of the effectiveness of GRIFFIN for CFI enforcement. For the performance evaluation, we test GRIFFIN when enforcing coarse-grained and combination policies on the SPEC CPU2006 benchmarks used in most CFI research [21, 22, 63, 54, 25, 67, 28][1]. We also examine how GRIFFIN performs on real-world applications including a browser (Firefox 45), a web server (nginx 1.6.2), an FTP server (vsftpd 3.0.2), and an email server (exim 4.84). When evaluating coarse-grained policies, we disassembled the binaries based on their debug information to uncover all legitimate targets for indirect control transfers. We compute fine-grained policies by matching function signatures as previous works do [21, 27, 22]. Additionally, we evaluate a stateful forward-edge policy on SPEC benchmarks [63]. Finally, we explore how more targeted logging would impact the efficiency of control-flow checking to suggest future hardware designs.

5.6.1 Effectiveness Evaluation

We run the RIPE benchmark [73] as a sanity check of GRIFFIN's implementation as a working CFI enforcement mechanism rather than a comprehensive security evaluation. The RIPE benchmark consists of a vulnerable program and a set of 850 exploits using various techniques, which can be categorized by the type of code pointers (e.g., function pointer or return address), the location of memory corruption (e.g., stack or heap).

[1] We run all SPEC CPU 2006 benchmarks, except 447.dealII and 481.wrf, which are noted not to compile on current systems by other researchers as well [28]. The performance numbers in Table 5.4 do not include them either.

The RIPE benchmark was originally developed on Ubuntu 6.06. In our experiment, many exploits failed because of built-in system protection mechanisms such as DEP and ASLR, changes in the runtime layout, as well as compatibility issues due to the usage of newer-version libraries.

To make more exploits succeed on the vanilla Debian 8.2, we disabled the ASLR and compiled the vulnerable program without stack protection. We ended up with 82 working exploits as listed in Table 5.3. GRIFFIN can deterministically detect and prevent all 82 attacks under both coarse-grained and the combination policies. This experiment shows that our prototype of GRIFFIN works as expected. It does not mean that GRIFFIN can stop all control-flow attacks. The security of GRIFFIN is determined by the control-flow policies deployed, and we show GRIFFIN is capable of enforcing many, known CFI policies.

5.6.2 Performance Evaluation

In this section, we evaluate GRIFFIN's runtime performance and memory overheads by running SPEC CPU2006 benchmarks and a set of real-world applications. We allocate a 64KB buffer for Intel PT to trace each user thread, and an additional 4KB buffer to prevent trace packet loss from the interrupt skid issue (see Section 5.1). We discuss the performance impact of different buffer sizes using SPEC CPU2006 benchmarks. We use six worker threads to parallelize trace buffer processing. We evaluate the impact of applying different numbers of worker threads on the nginx web server.

5.6.2.1 SPEC CPU2006

We run SPEC CPU2006 benchmarks compiled with GCC 4.9 under -O2 optimization level to evaluate GRIFFIN's runtime performance on CPU-intensive benchmarks. The experimental results on SPEC CPU2006 benchmarks are shown in Figure 5.5. For each benchmark, we make four measurements: (1) the Intel PT hardware overhead measured by taking interrupts without processing trace buffer; (2) total overhead for enforcing coarse-grained policies; (3) control-flow reconstruction overhead for fine-grained and stateful policies; and (4) total overhead for enforcing combination policies. Cases

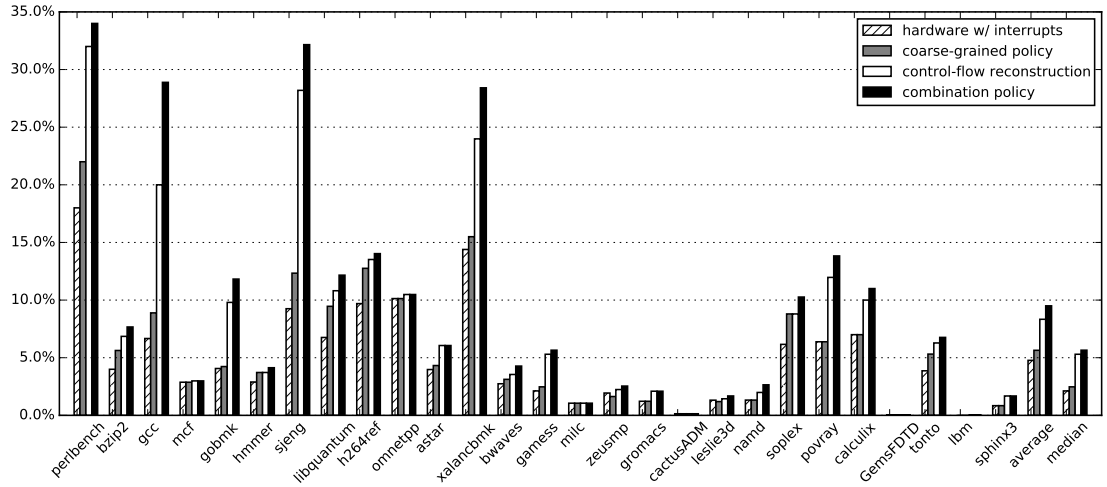

Figure 5.5. The performance overhead for running SPEC CPU2006 benchmarks with different configurations

(2-4) all include hardware overhead.

On average, Intel PT tracing introduces a 4.7% slowdown. The average slowdown under the coarse-grained policy is 5.6%. Control-flow reconstruction incurs a 8.3% slowdown on average. The average slowdown under the combination policy is 9.5%. The average overhead is sensitive to outliers like perlbench. For medians, the slowdown becomes 2.4% under the coarse-grained policy, and 5.6% under the combination policy. We also checked the performance impact of different Intel PT trace buffer sizes. We found that using 64KB buffers is slightly better than using either 4KB or 1MB buffers.

Finally, we examined the performance impact of stateful forward-edge policies. We did not observe noticeable performance differences because both the number of stateful indirect calls and the number of associated states are limited, as acknowledged by [63]. However, our lessons learned from enforcing stateful, forward-edge policies influence future hardware designs as discussed in Section 5.6.2.6.

We compared GRIFFIN's performance with prior solutions in Table 5.4. The prior solutions are from three categories: compiler-based, binary-instrumentation-based, or hardware-based. We chose these solutions because they are representative in each category and they measured SPEC CPU2006 benchmarks on Linux. To have direct comparisons, we further divided SPEC CPU2006 benchmark into SPECint and SPECfp. In

	Leg. Binary	No Instru.	Complete	Policy	SPEC CPU2006	
					SPECint	SPECfp
MCFI [21]	×	×	✓	fine-grained	4.0	1.6
πCFI [54]	×	×	✓	fine-grained (incremental CFG)	4.0	1.75
shadow stack [67]	×	×	✓	shadow stack	11.6*	6.0
	×	×	✓	para. shadow stack	4.7	3.0
binCFI [25]	✓	×	✓	coarse-grained	9.6	6.5
Lockdown [28]	✓	×	✓	combination	47.18	20.55
ROPecker [62]	✓	✓	×	heuristics	2.3	
PathArmor [63]	✓	×	×	combination (restrict constant callback)	3.3	
GRIFFIN	✓	✓	✓	coarse-grained	9.3	3.0
	✓	✓	✓	combination	16.0/11.9*	6.2

Table 5.4. The comparison between different CFI techniques. The numbers with an asterisk exclude perlbench and gcc in SPECint. We also exclude Fortran benchmarks evaluated in Lockdown and GRIFFIN from SPECfp.

Table 5.4, four prior solutions, MCFI [21], πCFI [54], ROPecker [62] and PathArmor [63], have lower performance overheads than GRIFFIN, however, MCFI and πCFI do not support the shadow stack check, and ROPecker and PathArmor are not complete (i.e., they do *not* check all indirect control transfers). binCFI [25] only supports a coarse-grained policy but its performance is not better than GRIFFIN mainly because it uses static binary instrumentation. Lockdown [28] supports a policy similar to the combination policy, but its performance is much worse than GRIFFIN because of its dynamic binary instrumentation. Finally, the shadow stack work [67] evaluated two compiler-based shadow stack schemes: traditional shadow stack and parallel shadow stack (i.e., maintain one stack pointer for both the normal stack and the shadow stack). The performance of the parallel shadow stack scheme is much better than the traditional shadow stack scheme, but it is vulnerable to a recent attack [74]. GRIFFIN implements both the traditional shadow stack scheme and the stateless fine-grained forward-edge check in the combination policy. GRIFFIN's performance under this more restricted policy is on par with the compiler-based implementation of the traditional shadow stack scheme. It is worth noting that the latter implementation is vulnerable to TOCTTOU attacks as acknowledged in [67]. A more secure implementation of the traditional shadow stack scheme has a worse performance as 18.4% on SPECint (excluding perlbench and gcc).

For certain benchmarks in SPEC CPU2006, we have high performance overhead.

	Direct	Indirect
Packet decoding	1,268,577,830	40,664,705,637
Following blocks	18,488,367,185	36,821,730,597
Total	**19,756,945,015**	**77,486,436,234**

Table 5.5. Measured CPU cycles for decoding packets and following basic blocks

We found that it is likely due to the high volume of indirect control transfers in these benchmarks. For example, perlbench generates over 8,000 `TIP` packets in a 64KB tracing buffer on average, which is 400 times more than the bzip2 benchmark.

```
/* Direct JMP */
for (i = 0; i < 0x90000000; i++) {
    asm volatile (
        "jmp 1f\n"
        "1:\n");
}

/* Indirect JMP */
for (i = 0; i < 0x90000000; i++) {
    asm volatile (
        "movq $1f,%%rax\n"
        "jmp  *%%rax\n"
        "1:\n"
        ::: "rax");
}
```

Figure 5.6. The test workload for profiling the impacts of indirect branches

To check our hypothesis, we ran two mostly-identical programs on GRIFFIN as shown in Figure 5.6 and measured the performance overhead caused by control-flow reconstruction. Both programs contain a busy loop, and the former has a direct jump in the loop while the latter has an indirect jump. The mere difference turns out to cause a big performance disparity. The direct-jump program has a 3% slowdown, but the indirect-jump program has a 25% slowdown. We further divide the control-flow reconstruction into two steps: (1) decoding packets, which takes a buffer as the input and returns a list of trace packets, and (2) following basic blocks, which takes in both the decoded packets and the disassembled basic blocks to reconstruct the control flow. Then we profile the execution of each step individually and show the results measured in CPU cycles in Table 5.5.

The CPU cycles spent in packet decoding for the indirect-jump program is 30 times more than the direct-jump program. This is because the indirect-jump program generates a trace that is roughly 23 times larger than that of the direct-jump program due to the additional `TIP` packets. Regarding the time spent on following basic blocks, the indirect-jump program takes twice as many CPU cycles as the direct-jump program. This is because an indirect jump makes GRIFFIN access the basic block pointer page to retrieve the pointer to the corresponding heap data structure when reconstructing the control flow, incurring an additional memory access compared to a direct jump whose target basic block is stored in the heap data structure.

5.6.2.2 Real-world Applications

We evaluate GRIFFIN's performance on real-world applications including three server programs and a client program under both the coarse-grained and combination policy. There are two main differences between these applications and the SPEC CPU2006 benchmarks. First, they are less CPU-intensive. The server programs are I/O-bound and the browser is user-oriented. Second, they all use multiple processes/threads. This could potentially impact GRIFFIN's performance because of the competition on CPU resources.

To benchmark the nginx web server, we use ApacheBench [75] to create 32 concurrent connections and send 10,000 HTTP requests for files of different sizes. To benchmark the vsftpd server, we use the pyftpbench [76] to request 10MB files in 10 concurrent connections. To benchmark the exim email server, we run the sendemail script [77] to repeatedly send 1KB emails. The chosen workloads are consistent with prior works [63, 65]. We evaluate their throughput reduction.

To evaluate GRIFFIN's performance on Firefox, we use the SunSpider 1.0.2 benchmark. In our current prototype, we do not enforce forward-edge policies for indirect control transfers within jitted code but execute the checks in a permissive way to measure the runtime overheads. We note that an actual policy may be derived from the JIT compiler with additional engineering efforts [78]. However, transitions between the interpreter and the jitted code in Firefox can be protected with minimal manual efforts

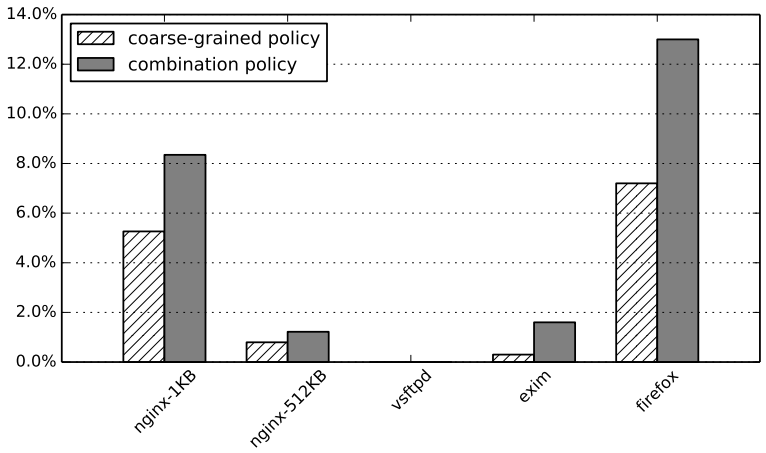

Figure 5.7. The performance overheads of real-world applications

for two reasons. First, there is a single transition point from the interpreter to the jitted code. Second, the jitted code can only invoke exported interpreter functions that are well marked with macro `JS_PUBLIC_API`. For non-jitted code in Firefox, we enforce checks on both forward and backward edges as in other experiments.

We show the results in Figure 5.7. On average, GRIFFIN incurs modest performance overheads for server programs – 1.8% under the coarse-grained policy and 2.7% under the combination policy. This is because server programs are often I/O-bound. When they wait on I/O, no Intel PT trace is generated, and GRIFFIN has more time to process the control-flow trace. For Firefox, GRIFFIN incurs 7.5% overheads under the coarse-grained policy and 13% under the combination policy.

5.6.2.3 Worker Threads

GRIFFIN leverages multiple worker threads to speed up the processing of trace buffers and CFI enforcement. To understand how different numbers of worker threads impact the performance on real-world applications, we run the nginx web server under the combination policy with the same workload as discussed in Section 5.6.2.2.

From Figure 5.8 we can see that using fewer worker threads affects GRIFFIN's performance because of the decreased parallelization. In general, the difference decreases as the file size increases, except for the 64KB case which may be due to the variance of

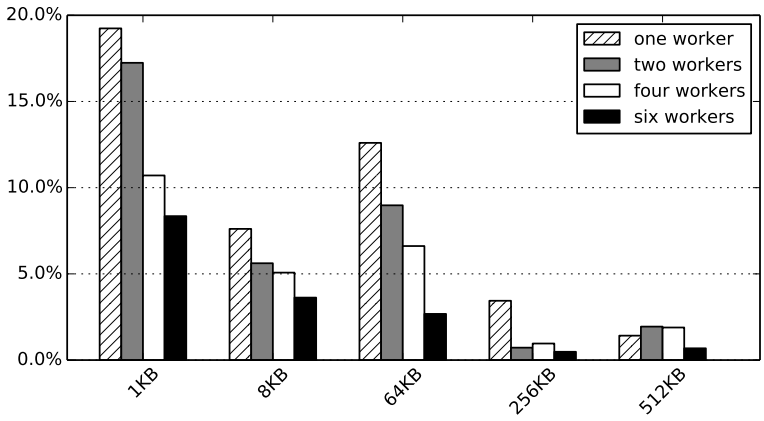

Figure 5.8. The impacts of different numbers of worker threads on GRIFFIN's performance for the nginx web server

file system buffering. This is aligned with our expectations on I/O bound programs, as the program spends more time on waiting for I/O the performance slowdown will decrease.

5.6.2.4 Memory Usage

We measured GRIFFIN's memory usage under the coarse-grained and combination policy for both SPEC CPU2006 benchmarks and real-world applications. GRIFFIN's memory usage has two parts: control pages for policies and basic blocks, and Intel PT trace buffers. Control pages are dynamically populated at runtime and can be shared between processes. In our experiments, we measured the actually populated control pages. The size of Intel PT trace buffers changes dynamically depending on the rates of Intel PT generating new trace packets and GRIFFIN processing existing trace packets. We measured the peak size of Intel PT trace buffers.

The results are shown in Table 5.6. The memory usage for control pages is determined by the enforced policy and program code size. The combination policy uses more control pages than the coarse-grained policy because of the need for control-flow reconstruction. Firefox has a much larger code base than other programs (>100MB) and incurs the highest use of control pages. The peak memory usage for Intel PT trace buffers is determined by two factors: the difference between trace generation and con-

	coarse-grained		combination	
	control pages	buffers	control pages	buffers
perlbench	1.2	5.2	7.3	356.5
bzip2	0.4	2.0	1.4	7.6
gcc	2.2	2.8	16.4	11.6
mcf	0.4	2.3	1.4	3.3
gobmk	1.0	1.0	6.0	445.2
hmmer	0.7	1.1	2.3	1.1
sjeng	0.5	1.8	1.7	113.4
libquantum	0.5	2.0	1.5	21.1
h264ref	0.8	2.5	4.2	3.9
omnetpp	1.2	2.3	4.9	2.7
astar	0.5	1.4	1.7	1.5
xalancbmk	3.3	4.7	13.7	202.6
bwaves	0.1	1.4	3.5	32.8
gamess	4.5	2.7	55.2	3.1
milc	0.5	1.0	2.2	0.8
zeusmp	0.1	1.0	5.3	1.2
gromacs	0.7	5.3	5.7	9.8
cactusADM	0.3	2.0	5.1	2.2
leslie3d	0.1	0.4	4.4	0.5
namd	0.7	2.6	2.6	7.3
soplex	1.1	2.5	4.6	2.2
povray	1.3	1.6	4.6	1.8
calculix	0.9	3.8	8.6	8.9
GemsFDTD	0.1	3.8	6.7	9.7
tonto	2.7	3.6	15.5	8.7
lbm	0.4	2.5	1.4	2.1
sphinx3	0.6	1.1	3.1	1.3
nginx	0.6	1.0	10.0	1.0
exim	0.1	0.8	8.6	0.8
vsftpd	0.1	0.5	6.0	0.6
firefox	103.7	9.8	377.4	84.1

Table 5.6. Peak memory usage (MB) for control pages and trace buffers under the coarse-grained and combination policy

sumption rate and the time a program executes. We discuss how GRIFFIN triggers countermeasures when the memory usage increases in Section 5.6.2.5.

5.6.2.5 Runtime Policy Change

By the design of separating the CFI enforcement from the policy, GRIFFIN naturally supports changing the enforced policy at runtime. We demonstrate one approach that strikes the balance between the performance and security using coarse-grained policies and combination policies. Specifically, GRIFFIN monitors the memory usage for

each program by the number of outstanding trace buffers to process. If the number of outstanding trace buffers surpasses a threshold, GRIFFIN switches to a coarse-grained policy. Once it catches up, GRIFFIN switches back to a fine-grained policy. Note that enforcement cannot be resumed for some stateful policies, such as the shadow stack. In this case, we fall back to a fine-grained policy instead.

Given perlbench incurs the highest runtime overheads in our evaluation for SPEC CPU2006 benchmarks, we use it to evaluate the effectiveness of dynamic policy change. We use three different thresholds – 256MB, 128MB and 64MB. Obviously, a small threshold will have better performance but weaker security because GRIFFIN will spend longer runtime to enforce the coarse-grained policy. In our experiment, we observed 1.0%, 1.7% and 3.0% of the trace is checked by the coarse-grained policy under the three thresholds, respectively. Compared to enforcing the combination policy for the whole execution of perlbench, the incurred overheads are reduced by 8.8%, 17.6% and 23.5%, respectively.

5.6.2.6 Hardware Enhancements

Current Intel PT hardware does not encode branch source address and instead requires GRIFFIN to reconstruct the control flow to infer such information. As shown in Figure 5.5, GRIFFIN spent a significant amount of time on control-flow reconstruction while CFI checks are enforced at little additional cost. Thus, a natural optimization is to augment the trace with addresses of indirect branches to eliminate the control-flow reconstruction when enforcing stateless policies.

We manufacture traces that include branch addresses before every indirect control transfer and evaluate the impact on the trace size and performance for fine-grained CFI enforcement. Specifically, we insert a `FUP` packet before every `TIP` packet and remove all `TNT` packets. We show the results in Table 5.7. Our experiment shows an average of 89.50% performance improvement on trace processing and 59.86% trace size reduction when enforcing a fine-grained stateless policy for SPEC CPU2006 benchmarks. In the worst case, the performance still improves by 61.79% and the trace size increases by no more than 18.54%.

Benchmark	Size Ratio	Proc. Time Ratio
perlbench	110.71%	35.63%
bzip2	0.46%	0.04%
gcc	73.26%	11.69%
mcf	2.32%	0.32%
gobmk	8.21%	0.50%
hmmer	11.03%	33.51%
sjeng	78.97%	0.55%
libquantum	0.87%	0.90%
h264ref	118.54%	38.21%
omnetpp	108.02%	30.63%
astar	1.71%	0.17%
xalancbmk	112.81%	35.67%
bwaves	8.86%	1.00%
gamess	41.58%	6.3%
milc	6.19%	0.88%
zeusmp	0.26%	0.97%
gromacs	5.31%	0.54%
cactusADM	48.80%	7.52%
leslie3d	0.35%	0.06%
namd	5.61%	1.16%
soplex	48.41%	8.23%
povray	101.96%	20.52%
calculix	43.00%	7.54%
GemsFDTD	53.29%	11.41%
tonto	79.37%	17.99%
lbm	1.40%	0.24%
sphinx3	12.58%	2.30%
average	**40.14%**	**10.50%**

Table 5.7. The trace size and process time (enforcing fine-grained policies) of the manufactured trace compared to the original Intel PT trace for SPEC CPU2006 benchmarks

However, the proposed enhancement undermines GRIFFIN's potential to enforce stateful policies because of the lack of detailed control-flow information, such as conditional and direct branches. We propose that future hardware designs should explore allowing GRIFFIN to selectively toggle the generation of the complete control-flow trace (i.e., `TNT` packets) for the minimal execution necessary to enforce stateful policies.

For forward edges, reconstructing only necessary control flows enables GRIFFIN to enforce stateful checks based on partial (but sufficient) execution state with little extra processing. For example, we modify the manufactured trace to include complete

control-flow information right before a constant callback is passed until the callee returns to support PathArmor's stateful policy. For the program with the most stateful forward edges (433.milc) in SPEC benchmarks, its trace size increases from 0.88% of its original trace size after removal of all `TNT` packets to 1.43% after enabling selective logging.

For backward edges, enforcing a shadow stack requires GRIFFIN to reconstruct the entire control flow to track all direct calls. To improve the efficiency of the shadow stack enforcement, we likely need a different approach. Fortunately, a recently introduced feature called Intel Control-flow Enforcement Technology (CET) [68] enables shadow stack enforcement for unmodified binaries directly from the hardware. We envision GRIFFIN can leverage CET to do efficient shadow stack checking in the future.

5.7 Discussion

Intel PT is a hardware extension designed to capture information about software execution with low overheads. However, the initial version of Intel PT only offers control-flow tracing [15]. Thus, the current implementation of GRIFFIN only supports stateful policies that rely on control-flow information such as a shadow stack. Enforcing stateful policies on forward edges is an on-going research topic [54, 63]. We note that the state-of-the-art stateful forward-edge policies can be derived solely from control-flow information [63]. Besides, we envision that in the future generation of Intel processors, Intel PT will be capable of capturing data flows, enabling GRIFFIN to enforce richer stateful policies and even to detect data-flow attacks.

We implement GRIFFIN in the kernel space. This increased the attack surface of the operating system kernel despite the defenses proposed in this dissertation. Particularly, the disassembler ported to the Linux kernel consists of 12,497 SLoC. However, we assume GRIFFIN protects benign programs, so the input to the disassembler (i.e., instructions in the program) is not from an adversary. This makes the latent vulnerabilities (if they exist) of the disassembler hard to exploit. In the future, we will explore techniques to lower the privilege of worker threads and reduce the amount of privileged code.

We examined the four SPEC CPU2006 programs (perlbench, gcc, sjeng, xalancbmk) that incur high performance overhead (see Figure 5.5) when GRIFFIN enforces the combination policy. We found that the legal CFGs of these programs offer tremendous flexibility for adversaries to launch attacks without violating even a fine-grained CFI policy. For example, Perl includes a series of three indirect branches within a loop that can execute a wide range of targets, enabling flexible interpretation of Perl scripts. Researchers have previously shown that CFI in general may provide limited protections for those programs with expressive CFGs [20, 79]. Using the same example of Perl, if an adversary corrupts the bytecode the interpreter is "executing" (i.e., control any input to the interpreter loop), she can perform Turing-complete computations while complying with the CFG. The other three programs also have similar loops with many indirect branches, which would enable similar control-flow bending attacks [20] that are not prevented by CFI defense. For these types of programs, we likely need a different approach other than CFI to provide the desired protections.

5.8 Summary

In this chapter, we present GRIFFIN, a hardware-assisted, operating system CFI enforcement mechanism that uses Intel Processor Trace (PT) to protect user-space processes, including the Firefox browswer and its jitted code, from control-flow hijacking attacks. GRIFFIN is capable of enforcing stateful CFI policies on both forward and backward edges (e.g., shadow stack), enabling the strong prevention for attacks on control flow, but additionally enables tradeoff between security and performance by supporting a variety of control-flow policies. GRIFFIN is designed to leverage Intel PT traces efficiently by enforcing coarse-grained CFI using the trace directly, reconstructing control flow information from Intel PT traces in parallel to enforce fine-grained CFI policies, and producing stateful information in parallel for checking sequentially for low overhead. As a result, GRIFFIN can achieve comparable performance with software-based instrumentation defenses on the SPEC CPU2006 benchmarks, showing that strong, hardware-based CFI enforcement can be a viable alternative.

Chapter 6

Related Work

In this chapter, we cover related work in two broad categories. First, we cover kernel-level monitoring techniques that are used to protect invariants such as code integrity for operating system kernels in Section 6.1. Second, we cover general techniques that mitigate exploitations over memory safety errors in the remaining sections.

6.1 Kernel-level Monitoring

SecVisor enforces lifetime kernel code integrity by implementing a tiny hypervisor [80] (e.g., fewer than 2,000 SLoC) and running the operating system on top of it. It virtualizes the physical memory and applies memory protection directly to guest physical pages, which is transparent to the guest operating system. If the guest system ever writes to a kernel code page, a page fault will be triggered and detected by SecVisor. Besides, SecVisor checks other system invariants regularly, e.g., when the kernel returns the control to the user space.

NICKLE is another VMM-based system that ensures only authenticated code can be run in the kernel. It is implemented as part of a VMM. It maintains a duplicate physical page (called *shadow memory*) for each kernel code page. Shadow memory serves for instruction fetch from the guest operating system but, other than that, it is transparent to the guest. Therefore, the guest can only execute approved code in the shadow memory, which is integrity-protected by the VMM.

Garfinkel *et al.* proposed the first VMM-based monitor called Livewire to protect system invariants [81]. The Livewire system introduced the introspection mechanism for virtual machine monitors and demonstrates its application by enforcing the kernel code integrity as well as constant data (e.g., system call table).

Zhang *et al.* proposed a coprocessor-based kernel monitor [82]. Though it may not have full access to the state of a host processor (e.g., registers), by using an additional piece of hardware like PCI cryptographic coprocessor, the solution improves the overall system performance when compared with VMM-based monitoring.

Petroni *et al.* further proposed a prototype of coprocessor-based kernel monitor called Copilot [83]. It works by periodically computing checksums of various regions in kernel memory, and sends the result to a remote administration station. The isolation of the coprocessor provides a similar autonomous environment to the TrustZone architecture (see Section 3.2), while the secure world could have more controls on the conventional system as demonstrated in Chapter 3.

Wang *et al.* built a hardware-assisted tampering detection framework called HyperCheck for VMMs [84]. HyperCheck utilized Intel's System Management Mode (SMM) to reliably check the state of the VMM and securely communicate to a remote administration server to identify compromises.

Azab *et al.* proposed a similar system to SPROBES (see Chapter 3) to protect the integrity of Android kernel based on TrustZone [85]. Like SPROBES, the proposed system instrumented the normal world on memory management operations and verified them in the secure world. The system has been implemented and evaluated on Samsung's TrustZone-enabled processors and reported similar performance overheads to SPROBES.

6.2 Control-Flow Integrity

In 2005, Abadi *et al.* introduced Control-Flow Integrity (CFI) [2], which restricts a program execution to its Control-Flow Graph (CFG). The initial implementation of CFI labels indirect branches and target instructions, and allows an indirect control transfer

at runtime if the source and destination have the same label. However, many early implementations focus on the runtime performance and enforce a loose CFG in practice (e.g., use one or two labels) [25, 26, 55, 56, 57, 58, 9, 23].

Attackers realized the deficiency and started launching CFI-compliant attacks [11, 13, 20, 18, 86, 12]. These advanced code-reuse attacks hijack the control flows in a way that is allowed by the enforced CFI policy. Therefore, researchers started developing fine-grained CFI [22, 21] and context-sensitive CFI implementations [54, 63] to mitigate these attacks. Fine-grained CFI allows each indirect branch to have its own set of targets to further restrict the adversary's options, while context-sensitive CFI takes program state (e.g., a call stack) into account and enforces stricter policies. Besides, opaque CFI proposes to hide the enforced CFI policy from the adversary [87]. However, Control-Flow Bending attacks demonstrate techniques to bypass existing CFI implementations without a shadow stack [20].

6.2.1 CFI for Privileged Software

Researchers have explored to apply CFI to privileged software. HyperSafe [23] enforces lifetime CFI for a hypervisor. We adopted their instrumentation approach (i.e., restricted pointer indexing) in Chapter 4. kGuard protects the kernel from *ret2usr* attacks by enforcing all indirect control transfers within the kernel must not target a user-space address [88]. Criswell *et al.* made the first attempt to enforce CFI on conventional operating systems [9]. They built their enforcement on top of the secure virtual architecture [10], a software layer between the operating system and hardware. However, their implementation suffers from high performance overheads.

6.2.2 Binary Rewriting

Researchers have also developed approaches to enforce CFI for legacy binaries without requiring the source code. For instance, binCFI and CCFIR derive the CFI policy directly from binaries and insert checks for enforcement [25, 26]. TypeArmor improves the precision of the computed CFI policy by taking high-level program semantics into account [27]. However, like compiler-based approaches, static binary rewriting "fixes"

CFI checks into the program and prevents from choosing the desired protection to balance security and performance at runtime.

Dynamic binary instrumentation has also been explored to protect legacy binaries from control-flow hijacking attacks. It has the flexibility to choose the protection at runtime as GRIFFIN does. ROPdefender [60] implements shadow stacks and Payer *et al.* [28] complements the protection by enforcing fine-grained CFI policies for forward edges as well. However, the performance overhead is inherently high due to the cost of dynamic binary instrumentation.

6.2.3 Hardware-based CFI

Researchers have proposed various hardware-based defenses to protect binaries without instrumentation. CFIMon [64] relies on Branch Trace Store (BTS) to record control transfers and execute CFI checks. However, BTS incurs significant performance slowdown (20x-40x) [63]. CFIMon has only been evaluated for I/O-bound applications.

Researchers have explored using the Last Branch Record (LBR) feature to build CFI defenses. LBR records a *small* number of the most recent control transfers with minimal overheads. However, despite the performance benefits, defenses built on LBR are fundamentally limited by the capacity of the LBR stack. For instance, both kBouncer [61] and ROPecker [62] rely on heuristics to detect attacks. Unfortunately, a recent proposal shows that an experienced adversary can break these heuristics by using long gadgets and/or launching history-flushing attacks [12]. PathArmor [63] attempts to enforce the strong shadow-stack policy on backward edges as GRIFFIN does. However, PathArmor can only check a small fraction of executed returns at runtime due to the LBR limitation. According to a trace we collected from nginx, it checks less than 0.1% of total returns.

To overcome the limitation of LBR, CFIGuard [65] proposes to combine the LBR feature with the Performance Monitoring Unit (PMU). The key idea is to program the PMU to trigger an interrupt when the LBR stack fills. By properly handling such interrupts, CFIGuard can check all executed indirect control transfers. Though CFIGuard proposes to improve the performance by only recording indirect control transfers, triggering an interrupt on every 16 of them can still slow down programs significantly, particularly

for CPU-intensive programs. Like CFIMon, CFIGuard has only been evaluated for I/O-bound server applications.

6.3 Address Space Layout Randomization

Address space layout randomization (ASLR) makes code-reuse attacks harder by varying the location of existing code at runtime. The basic form of ASLR has been deployed on modern operating systems. Researchers have demonstrated approaches to bypass ASLR through information leakage [11, 89] and brute force attacks [90]. Moreover, an attacker can analyze the address space layout of the victim process at runtime and launch just-in-time ROP attacks [91].

Researchers have proposed techniques to prevent both direct and indirect code disclosure. To prevent direct code disclosure, researchers propose to use execute-only memory to disable data accesses to executable pages [92], and demonstrate a practical implementation (called readactor) through the use of extended page tables [44]. To prevent indirect code disclosure, readactor uses position-independent trampolines to redirect indirect calls and returns to avoid revealing the actual code layout through code pointers. However, some form of code-reuse attack through these trampolines is still possible as acknowledged in [44].

6.4 Memory Safety

Researchers have attempted to bring memory safety to unsafe languages like C/C++. Memory safety prevents memory corruption in the first place to defeat both control-flow and data-flow attacks. CCured [93] and Cyclone [94] instruments programs to include base and bound information for each pointer and insert checks on pointer dereferences to detect out-of-bounds accesses. However, such approaches raise program compatibility issues by storing the bound information using a multi-word fat pointer, introducing programmer visible changes to the memory layout. On the other hand, SoftBound [46] stores the bound information as disjoint metadata and improves the compatibility.

However, enforcing complete memory safety is expensive and thwarts adoption in practice, thus researchers have explored lightweight approaches to trade security for performance. WIT [43] only protects against out-of-bounds write while allowing out-of-bounds read. This may allow sensitive information to be leaked through vulnerabilities such as heartbleed [95]. Code-pointer integrity [45] is based on SoftBound while focuses on protecting code pointers. It reports that only 6.5% of pointer accesses are instrumented in SPEC CPU2006 benchmarks, and improves the performance significantly compared to full memory safety.

Chapter 7

Conclusion

To conclude the dissertation, we first summarize our approaches to enforcing execution integrity for software systems, then discuss possible future directions on mitigating exploitations over memory corruption bugs.

7.1 Dissertation Summary

In the dissertation, we present three works that are combined to enforce both the code integrity and control-flow integrity (CFI) for the entire software stack including both the kernel and user space. In Chapter 3, we propose a set of five general invariants on mediating memory management operations to enforce lifetime kernel code integrity, and implement a proof-of-concept system based on the novel introspection mechanism called SPROBES for the ARM TrustZone architecture. Based on the enforcement of kernel code integrity, in Chapter 4, we propose a taint-based analysis to compute the fine-grained CFG for operating system kernels and develop a comprehensive fine-grained CFI enforcement by additionally handling system events such as interrupts that introduce non-trivial control flows. Conceptually, the two chapters combine to enable a kernel protected by the execution integrity. In Chapter 5, by leveraging a recent hardware capability called Intel Processor Trace (PT), we build an operating system mechanism that is capable of enforcing a variety of CFI policies over all running, unmodified user-space programs with deployment flexibility and runtime efficiency, and show how the

performance can be further improved when future Intel PT hardware generates more targeted logging traces.

7.2 Future Directions

However, this is not the end of the story, and we discuss interesting future directions for mitigating memory corruption vulnerabilities. We focus on the improvement of runtime efficiency and/or stronger security guarantees.

7.2.1 Heterogeneous Control-Flow Integrity Enforcement

In Chapter 5, we present the GRIFFIN system to enforce CFI system-wide. For better performance, we design GRIFFIN to utilize multiple processor cores to execute CFI checks in parallel on the collected program's traces. However, GRIFFIN is not limited to local CPU resources. In fact, it can further leverage any available computing resources such as GPUs, co-processors (e.g., Xeon Phi) and/or idle machines within a connected network to speedup the CFI enforcement. Given each parallel activity is mostly independent from each other, we believe the investment of additional computing resources can improve GRIFFIN's performance effectively.

7.2.2 Hardware-Enforced Control-Flow Integrity

Intel announced the control-flow enforcement technology [68] (CET) in June, 2016. Intel CET can be configured to enforce coarse-grained CFI on forward edges (i.e., `call*`,`jmp*`) and shadow stacks on backward edges (i.e., `ret`) for software running in both the kernel space and the user space. Specifically, for forward edges, Intel CET introduced the `ENDBRANCH` instruction that is used to mark a valid target. For backward edges, Intel CET enforces a shadow stack by maintaining a separate stack protected from ordinary memory writes and matching returns with the call at the top of the shadow stack.

When a violation is detected, Intel CET will raise a control-flow exception to trigger countermeasures. To avoid false alarms for benign program execution, an implementation needs to handle various corner cases when enforcing a shadow stack as shown

in Section 5.5. On the other hand, if an active attack is detected, the operating system can terminate the compromised process. We outline a design on how GRIFFIN should integrate CET to provide strong defenses with better efficiency in Section 5.6.2.6.

In addition, Intel CET motivates the exploration of the shadow stack enforcement for operating system kernels. In Section 4.8, we discussed the challenges on an efficient software implementation of kernel-level shadow stack. Intel CET overcomes these challenges by (1) introducing a new page protection bit that protects the shadow stack from ordinary memory writes and (2) maintaining the shadow stack pointer in a dedicated register for faster accesses.

Bibliography

[1] ANDERSEN, S. and V. ABELLA (2004), "Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies," .

[2] ABADI, M., M. BUDIU, U. ERLINGSSON, and J. LIGATTI (2005) "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, ACM, pp. 340–353.

[3] CHECKOWAY, S. and H. SHACHAM (2013) *Iago attacks: Why the system call api is a bad untrusted rpc interface*, vol. 41, ACM.

[4] TA-MIN, R., L. LITTY, and D. LIE (2006) "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*, USENIX Association, pp. 279–292.

[5] CHEN, X., T. GARFINKEL, E. C. LEWIS, P. SUBRAHMANYAM, C. A. WALDSPURGER, D. BONEH, J. DWOSKIN, and D. R. PORTS (2008) "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ACM SIGARCH Computer Architecture News*, vol. 36, ACM, pp. 2–13.

[6] RILEY, R., X. JIANG, and D. XU (2008) "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection*, Springer, pp. 1–20.

[7] SESHADRI, A., M. LUK, N. QU, and A. PERRIG (2007) "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operating Systems Review*, **41**(6), pp. 335–350.

[8] "Xen: Security vulnerabilities," `https://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html`.

[9] CRISWELL, J., N. DAUTENHAHN, and V. ADVE (2014) "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, pp. 292–307.

[10] CRISWELL, J., A. LENHARTH, D. DHURJATI, and V. ADVE (2007) "Secure virtual architecture: A safe execution environment for commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 41, ACM, pp. 351–366.

[11] GOKTAS, E., E. ATHANASOPOULOS, H. BOS, and G. PORTOKALIDIS (2014) "Out Of Control: Overcoming Control-Flow Integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[12] CARLINI, N. and D. WAGNER (2014) "ROP is Still Dangerous: Breaking Modern Defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, pp. 385–399.

[13] DAVI, L., A.-R. SADEGHI, D. LEHMANN, and F. MONROSE (2014) "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, pp. 401–416.

[14] ARM INC. (2009), "ARM Security Technology Building a Secure System using TrustZone Technology," `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html`.

[15] (2016) "Intel 64 and IA-32 Architectures Software Developer's Manual," *Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*.

[16] ROEMER, R., E. BUCHANAN, H. SHACHAM, and S. SAVAGE (2012) "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, **15**(1), p. 2.

[17] BLETSCH, T., X. JIANG, V. W. FREEH, and Z. LIANG (2011) "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ACM, pp. 30–40.

[18] SCHUSTER, F., T. TENDYCK, C. LIEBCHEN, L. DAVI, A.-R. SADEGHI, and T. HOLZ (2015) "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE, pp. 745–762.

[19] SOTIROV, A. (2007) "Heap feng shui in javascript," *Black Hat Europe*.

[20] CARLINI, N., A. BARRESI, M. PAYER, D. WAGNER, and T. R. GROSS (2015) "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *SEC'15: 24th Usenix Security Symposium*.

[21] NIU, B. and G. TAN (2014) "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, p. 58.

[22] TICE, C., T. ROEDER, P. COLLINGBOURNE, S. CHECKOWAY, Ú. ERLINGSSON, L. LOZANO, and G. PIKE (2014) "Enforcing forward-edge control-flow integrity in gcc & llvm," in *USENIX Security Symposium*.

[23] WANG, Z. and X. JIANG (2010) "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*, IEEE, pp. 380–395.

[24] SAILER, R., X. ZHANG, T. JAEGER, and L. VAN DOORN (2004) "Design and Implementation of a TCG-based Integrity Measurement Architecture." in *USENIX Security Symposium*, vol. 13, pp. 223–238.

[25] ZHANG, M. and R. SEKAR (2013) "Control Flow Integrity for COTS Binaries." in *USENIX Security*, pp. 337–352.

[26] ZHANG, C., T. WEI, Z. CHEN, L. DUAN, L. SZEKERES, S. MCCAMANT, D. SONG, and W. ZOU (2013) "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*, IEEE, pp. 559–573.

[27] VAN DER VEEN, V., E. GÖKTAS, M. CONTAG, A. PAWLOWSKI, X. CHEN, S. RAWAT, H. BOS, T. HOLZ, E. ATHANASOPOULOS, and C. GIUFFRIDA (2016) "A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level," in *IEEE Symposium on Security and Privacy (Oakland)*, IEEE.

[28] PAYER, M., A. BARRESI, and T. R. GROSS (2015) "Fine-grained control-flow integrity through binary hardening," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, pp. 144–164.

[29] HUND, R., T. HOLZ, and F. C. FREILING (2009) "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." in *USENIX Security Symposium*, pp. 383–398.

[30] VOGL, S., R. GAWLIK, B. GARMANY, T. KITTEL, J. PFOH, C. ECKERT, and T. HOLZ (2014) "Dynamic hooks: hiding control flow changes within non-control data," in *Proceedings of the 23rd USENIX conference on Security Symposium*, USENIX Association, pp. 813–828.

[31] VOGL, S., J. PFOH, T. KITTEL, and C. ECKERT (2014) "Persistent data-only malware: Function Hooks without Code," in *Symposium on Network and Distributed System Security (NDSS)*.

[32] ALVES, T. and D. FELTON, "TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems. White paper, ARM, July 2004," .

[33] GE, X., H. VIJAYAKUMAR, and T. JAEGER (2014) "SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture," in *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST 2014)*.

[34] ARM INC., "CoreLink System Memory Management Unit," `http://www.arm.com/products/system-ip/controllers/system-mmu.php`.

[35] WINTER, J. (2008) "Trusted computing building blocks for embedded linux-based ARM trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, ACM, pp. 21–30.

[36] FAST MODELS EMULATOR, http://www.arm.com/products/tools/models/fast-models/.

[37] BARHAM, P., B. DRAGOVIC, K. FRASER, S. HAND, T. HARRIS, A. HO, R. NEUGEBAUER, I. PRATT, and A. WARFIELD (2003) "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, **37**(5), pp. 164–177.

[38] KERNEL BASED VIRTUAL MACHINE, http://www.linux-kvm.org/page/Main_Page.

[39] SUH, G. E., D. CLARKE, B. GASSEND, M. VAN DIJK, and S. DEVADAS (2003) "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing*, ACM, pp. 160–171.

[40] SMITH, S. W. (2002) "Outbound Authentication for Programmable Secure Coprocessors," in *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002)*, pp. 72–89.

[41] GARFINKEL, T., B. PFAFF, J. CHOW, M. ROSENBLUM, and D. BONEH (2003) "Terra: a virtual machine-based platform for trusted computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pp. 193–206.

[42] CASTRO, M., M. COSTA, and T. L. HARRIS (2006) "Securing Software by Enforcing Data-flow Integrity," in *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pp. 147–160.

[43] AKRITIDIS, P., C. CADAR, C. RAICIU, M. COSTA, and M. CASTRO (2008) "Preventing Memory Error Exploits with WIT," in *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pp. 263–277.

[44] CRANE, S., C. LIEBCHEN, A. HOMESCU, L. DAVI, P. LARSEN, A.-R. SADEGHI, S. BRUNTHALER, and M. FRANZ (2015) "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *2015 IEEE Symposium on Security and Privacy (S&P 2015), 18-20 May 2015, San Jose, California, USA*.

[45] KUZNETSOV, V., L. SZEKERES, M. PAYER, G. CANDEA, R. SEKAR, and D. SONG (2014) "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[46] NAGARAKATTE, S., J. ZHAO, M. M. MARTIN, and S. ZDANCEWIC (2009) "SoftBound: Highly compatible and complete spatial memory safety for C," in *ACM Sigplan Notices*, vol. 44, ACM, pp. 245–258.

[47] WANG, X., H. CHEN, A. CHEUNG, Z. JIA, N. ZELDOVICH, and M. F. KAASHOEK (2012) "Undefined behavior: what happened to my code?" in *Proceedings of the Asia-Pacific Workshop on Systems*, ACM, p. 9.

[48] LATTNER, C. and V. ADVE (2004) "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, IEEE, pp. 75–86.

[49] "Supervisor mode execution protection," `http://vulnfactory.org/blog/smep/`.

[50] "SLOCCount," `http://www.dwheeler.com/sloccount/`.

[51] SALWAN, J., "ROPGadget," `http://www.shell-storm.org/project/ROPgadget/`.

[52] "LLVM SafeStack," `http://clang.llvm.org/docs/SafeStack.html`.

[53] LI, J., Z. WANG, X. JIANG, M. GRACE, and S. BAHRAM (2010) "Defeating return-oriented rootkits with return-less kernels," in *Proceedings of the 5th European conference on Computer systems*, ACM, pp. 195–208.

[54] NIU, B. and G. TAN (2015) "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 914–926.

[55] BLETSCH, T., X. JIANG, and V. FREEH (2011) "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACM, pp. 353–362.

[56] HUND, R., T. HOLZ, and F. C. FREILING (2009) "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." in *USENIX Security Symposium*, pp. 383–398.

[57] DAVI, L., A. DMITRIENKO, M. EGELE, T. FISCHER, T. HOLZ, R. HUND, S. NÜRNBERGER, and A.-R. SADEGHI (2012) "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." in *NDSS*.

[58] ZENG, B., G. TAN, and G. MORRISETT (2011) "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*, ACM, pp. 29–40.

[59] ZHANG, M., R. QIAO, N. HASABNIS, and R. SEKAR (2014) "A platform for secure static binary instrumentation," *ACM SIGPLAN Notices*, **49**(7), pp. 129–140.

[60] DAVI, L., A.-R. SADEGHI, and M. WINANDY (2011) "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ACM, pp. 40–51.

[61] PAPPAS, V. (2012) *kBouncer: Efficient and transparent ROP mitigation*, *Tech. rep.*, Citeseer.

[62] CHENG, Y., Z. ZHOU, M. YU, X. DING, and R. H. DENG (2014) "ROPecker: A generic and practical approach for defending against ROP attacks," in *Symposium on Network and Distributed System Security (NDSS)*.

[63] VAN DER VEEN, V., D. ANDRIESSE, E. GÖKTAŞ, B. GRAS, L. SAMBUC, A. SLOWINSKA, H. BOS, and C. GIUFFRIDA (2015) "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 927–940.

[64] XIA, Y., Y. LIU, H. CHEN, and B. ZANG (2012) "CFIMon: Detecting violation of control flow integrity using performance counters," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, IEEE, pp. 1–12.

[65] YUAN, P., Q. ZENG, and X. DING (2015) "Hardware-Assisted Fine-Grained Code-Reuse Attack Detection," in *Research in Attacks, Intrusions, and Defenses*, Springer, pp. 66–85.

[66] KASIKCI, B., B. SCHUBERT, C. PEREIRA, G. POKAM, and G. CANDEA (2015) "Failure sketching: a technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ACM, pp. 344–360.

[67] DANG, T. H., P. MANIATIS, and D. WAGNER (2015) "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ACM, pp. 555–566.

[68] "Intel Control-flow Enforcement Technology," `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

[69] PAYER, M. and T. R. GROSS (2010) "Generating low-overhead dynamic binary translators," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ACM, p. 22.

[70] DABAH, G., "diStorm - Powerful Disassembler Library For x86/AMD64," `https://github.com/gdabah/distorm`.

[71] BLAZAKIS, D. (2010) "Interpreter exploitation: Pointer inference and JIT spraying," *BlackHat DC*.

[72] HÖLZLE, U., C. CHAMBERS, and D. UNGAR (1992) "Debugging optimized code with dynamic deoptimization," in *ACM Sigplan Notices*, vol. 27, ACM, pp. 32–43.

[73] WILANDER, J., N. NIKIFORAKIS, Y. YOUNAN, M. KAMKAR, and W. JOOSEN (2011) "RIPE: Runtime Intrusion Prevention Evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*, ACM.

[74] CONTI, M., S. CRANE, L. DAVI, M. FRANZ, P. LARSEN, M. NEGRO, C. LIEBCHEN, M. QUNAIBIT, and A.-R. SADEGHI (2015) "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*, ACM, pp. 952–963.

[75] "ApacheBench: a complete benchmarking and regression testing suite," `https://httpd.apache.org/docs/2.2/programs/ab.html`.

[76] "pyftpdlib," `https://github.com/giampaolo/pyftpdlib`.

[77] "sendEmail," `http://caspian.dotconf.net/menu/Software/SendEmail`.

[78] NIU, B. and G. TAN (2014) "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 1317–1328.

[79] EVANS, I., F. LONG, U. OTGONBAATAR, H. SHROBE, M. RINARD, H. OKHRAVI, and S. SIDIROGLOU-DOUSKOS (2015) "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 901–913.

[80] SESHADRI, A., M. LUK, N. QU, and A. PERRIG (2007) "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operating Systems Review*, **41**(6), pp. 335–350.

[81] GARFINKEL, T., M. ROSENBLUM, ET AL. (2003) "A Virtual Machine Introspection Based Architecture for Intrusion Detection." in *NDSS*.

[82] ZHANG, X., L. VAN DOORN, T. JAEGER, R. PEREZ, and R. SAILER (2002) "Secure coprocessor-based intrusion detection," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ACM, pp. 239–242.

[83] PETRONI JR, N. L., T. FRASER, J. MOLINA, and W. A. ARBAUGH (2004) "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor." in *USENIX Security Symposium*, San Diego, USA, pp. 179–194.

[84] WANG, J., A. STAVROU, and A. GHOSH (2010) "HyperCheck: A hardware-assisted integrity monitor," in *Recent Advances in Intrusion Detection*, Springer, pp. 158–177.

[85] AZAB, A. M., P. NING, J. SHAH, Q. CHEN, R. BHUTKAR, G. GANESH, J. MA, and W. SHEN (2014) "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 90–102.

[86] GÖKTAŞ, E., E. ATHANASOPOULOS, M. POLYCHRONAKIS, H. BOS, and G. PORTOKALIDIS (2014) "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard," in *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, pp. 417–432.

[87] MOHAN, V., P. LARSEN, S. BRUNTHALER, K. W. HAMLEN, and M. FRANZ (2015) "Opaque Control-Flow Integrity." in *NDSS*.

[88] KEMERLIS, V. P., G. PORTOKALIDIS, and A. D. KEROMYTIS (2012) "kGuard: Lightweight Kernel Protection against Return-to-User Attacks." in *USENIX Security Symposium*, pp. 459–474.

[89] LEE, B., L. LU, T. WANG, T. KIM, and W. LEE (2014) "From zygote to morula: Fortifying weakened aslr on android," in *2014 IEEE Symposium on Security and Privacy*, IEEE, pp. 424–439.

[90] BITTAU, A., A. BELAY, A. MASHTIZADEH, D. MAZIÈRES, and D. BONEH (2014) "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*, IEEE, pp. 227–242.

[91] SNOW, K. Z., F. MONROSE, L. DAVI, A. DMITRIENKO, C. LIEBCHEN, and A.-R. SADEGHI (2013) "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, IEEE, pp. 574–588.

[92] BACKES, M., T. HOLZ, B. KOLLENDA, P. KOPPE, S. NÜRNBERGER, and J. PEWNY (2014) "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 1342–1353.

[93] NECULA, G. C., S. MCPEAK, and W. WEIMER (2002) "CCured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, ACM, pp. 128–139.

[94] JIM, T., J. G. MORRISETT, D. GROSSMAN, M. W. HICKS, J. CHENEY, and Y. WANG (2002) "Cyclone: A Safe Dialect of C," in *USENIX Annual Technical Conference, General Track*, pp. 275–288.

[95] "Heartbleed," `http://heartbleed.com`.

Vita

Xinyang Ge

EDUCATION

Ph.D., Computer Science and Engineering Dec 2016
The Pennsylvania State University, University Park
Advisor: Dr. Trent Jaeger

B.Eng., Software Engineering Jun 2012
Nanjing University

PROFESSIONAL EXPERIENCE

Penn State, University Park, PA Aug 2012 – Aug 2016
Research Assistant, Advisor: Trent Jaeger

Worked on system security researches. Protected the operating system kernel with both code integrity and control-flow integrity, and developed an operating system mechanism to protect user-space applications.

Microsoft Research, Redmond May 2015 – Aug 2015
Research Intern, Mentor: Weidong Cui

Developed a prototype system for supporting Intel Processor Trace on Windows, enabling efficiently tracing multithreaded applications and recovering the exact control flows afterwards.

Microsoft Research, Redmond May 2014 – Aug 2014
Research Intern, Mentor: David Molnar

Developed an Azure cloud testing service that runs SAGE, a whitebox fuzzer employing symbolic execution to find defects as fast as possible by maximizing the code coverage, for resource-efficient large-scale fuzz testing of Windows applications.

eBay Inc., Shanghai, China Aug 2011 – May 2012
Technical Intern, Mentor: Eddy Cai

Developed a specialized search engine for historical SQL queries to help new database administrators find reusable queries.

Nanjing University, Nanjing, China Feb 2011 – Jun 2011
Teaching Assistant, Instructor: Jidong Ge

fryy: a small operating system kernel designed from scratch for illustrating how OS functions (e.g., task management, file system, etc.) are implemented on real hardwares.

State Key Laboratory for Novel Software Tech., Nanjing, China Mar 2010 – Feb 2011
Research Assistant, Advisor: Zhenyu Chen

Implemented an experimental recommender system and proposed a prediction approach based on regression for improving the quality of recommendation.