Control-flow Integrity for Real-time Embedded Systems

by

Nicholas Brown

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2017

APPROVED:

Professor Robert J. Walls, Major Thesis Advisor

Professor Craig A. Shue, Thesis Reader

Professor Craig E. Wills, Head of Department

Abstract

As embedded systems become more connected and more ubiquitous in mission- and safety-critical systems, embedded devices have become a high-value target for hackers and security researchers. Attacks on real-time embedded systems software can put lives in danger and put our critical infrastructure at risk. Despite this, security techniques for embedded systems have not been widely studied. Many existing software security techniques for general purpose computers rely on assumptions that do not hold in the embedded case. This thesis focuses on one such technique, control-flow integrity (CFI), that has been vetted as an effective countermeasure against control-flow hijacking attacks on general purpose computing systems. Without the process isolation and fine-grained memory protections provided by a general purpose computer with a rich operating system, CFI cannot provide any security guarantees. This thesis explores a way to use CFI on ARM Cortex-R devices running minimal real-time operating systems. We provide techniques for protecting runtime structures, isolating processes, and instrumenting compiled ARM binaries with CFI protection.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Modern real-time embedded systems have countless applications, varying in complexity. There are simple devices like thermostats or coffee makers, more complex systems like smartphone radios, and highly complex health and safety critical systems like jet engine controllers or automotive braking controllers. In safety critical systems especially, there is great risk when manufacturers release faulty devices. Failures in these systems can cause injury, and the designers of the system will be held liable [4]. Often these faults are in software and can be exploited by an intelligent adversary [5,6].

The most pernicious type of attack allows an attacker to execute arbitrary code on a device. Such attacks, commonly called control-flow hijacking, manipulate the execution of a program by redirecting control-flow transfers to either attacker-supplied code (e.g., stack smashing [7]) or useful code sequences already in the program (e.g., return-oriented programming [8]). State-of-the-art defenses against control-flow hijacking are largely based on the concept of control-flow integrity (CFI) [9]. Intuitively, CFI compares the behavior of a running program to a predefined model. If the program's behavior deviates from what is expected, an error is thrown. In particular, CFI monitors control-flow transfers and only allows a transfer if it is accepted by a precomputed control-flow graph (CFG) of the program. CFI is a subset of a larger category of security techniques called memory safety. While full memory safety is possible, the performance overhead (nearly 200% in some cases) makes it impractical for most applications [10,11]. CFI is an approach to memory safety on a small subset of memory, namely code pointers, which allows for significantly lower overhead as opposed to full memory safety.

While myriad CFI implementations exist for general-purpose systems (e.g., desktops and smartphones), real-time embedded systems present several unique challenges for CFI. First, many embedded systems do not have the hardware or software support for task isolation. Such isolation is common in general-purpose systems and relied upon by existing CFI solutions. Second, the scheduler in a real-time operating system (RTOS) can interrupt any instruction and can return to any arbitrary instruction. This makes the scheduler a high degree node in the CFG, which Carlini et al. have shown severely weakens the effectiveness of CFI [12]. Third, the majority of existing CFI solutions are for x86-based hardware, while embedded systems are commonly ARM-based, an architecture which has several challenges for CFI, such as multiple instruction sets and the lack of a dedicated function return instruction. Finally, real-time embedded systems generally have limited resources and strict timing requirements, limiting the amount of storage available for runtime structures required for CFI (e.g., shadow stacks). CFI instrumentation must adhere to the timing constraints in place by the real-time system.

We propose a CFI scheme for real-time embedded systems that addresses these challenges and prevents control-flow hijacking attacks. For our initial efforts, we focus on ARM-based systems running the FreeRTOS real-time operating system, but we anticipate that our system will be portable to any RTOS running on an ARM microcontroller, since the majority of the implementation is designed to be operating system agnostic. Existing approaches to CFI on embedded systems, such as C-FLAT [13] and TrackOS [14], move away from the traditional CFI approach but introduce new time-of-check to time-of-use vulnerabilities. Traditional approaches depend on process isolation in the presence of multiple threads, but most embedded systems do not have this isolation. Our work takes a more traditional approach to CFI, while adding the necessary protections to support multiple threads without true isolation using virtual memory. The contributions of this work are:

- **Protection mechanisms for runtime data structures used by CFI.** We protect the instrumentation required for CFI as well as the shadow stack, which is used to augment the CFG at runtime.
- **Binary instrumentation for ARM.** We create a reference implementation for ARM-based embedded systems running the FreeRTOS real-time operating system.
- **Technique for process isolation on ARM systems without virtual memory.** We devise a low-overhead method for isolating critical parts of a process on ARM systems where all processes run in the same address space.
- **A binary patching implementation for ARM.** We create a binary patching framework that allows scripting modification of precompiled ARM binaries to add features like CFI.

1.1 Outline

The remainder of this thesis is formatted as follows. Chapter 2 contains background information about ARM, CFI, and real-time operating systems. Chapter 3 describes our reference implementation for single-threaded, bare metal ARM systems. Chapter 4 describes the modifications made to FreeRTOS to support our bare metal implementation across multiple processes in the same address space. Chapter 5 evaluates the security and performance overhead of our CFI schemes. Finally, Chapter 6 summarizes this work and concludes the thesis.

Chapter 2

Background and Related Works

Before we can discuss the low-level details of control-flow integrity instrumentation on embedded ARM processors, we discuss the relevant background and related works which this thesis builds upon. We start with a brief overview of the embedded real-time ARM architecture, Cortex-R. Then, we discuss control-flow integrity, its variants, and its limitations. Finally, we explore some of the basic principles of real-time operating systems (RTOS), specifically FreeRTOS and the relevant differences between FreeRTOS and general purpose operating system kernels like Linux.

2.1 ARM Architecture

The ARM architecture is unique in how flexible it is. There are ARM chips designed for general purpose systems called the Application Profile (Cortex-A), high performance real-time systems called the Real-time Profile (Cortex-R), and low-power embedded systems called the Microcontroller Profile (Cortex-M). While all three of these profiles share a similar instruction set architecture (ISA), the underlying hardware is different to support different requirements. Cortex-A systems are often multi-core with high clock speeds, and they have all of the necessary hardware to efficiently run general purpose operating systems. Cortex-R processors are generally single core (although the more expensive models are multi-core), and they have special interrupt controllers and caching mechanisms to support the low-latency required by real-time systems. Cortex-M processors are single-core, with hardware designed for minimal power usage and security.

Table 2.1 highlights more of the differences between the different ARM lineups. In addition to belonging to ARM Cortex-A/R/M, ARM processors can be refined further based on the version of the architecture they support. At the time of this writing, modern ARM cores are either ARMv7 or ARMv8, with ARMv8 being widely adopted for Cortex-A models, while the ARMv8 versions of Cortex-R and Cortex-M processors have been announced. ARMv8 for Cortex-R (abbreviated ARMv8-R) introduces a new hypervisor mode that allows use of multiple real-time operating systems as well as efficient process isolation. While this feature could be highly beneficial for security purposes, these processors have not been made available yet, the remainder of this paper will be focusing on the ARMv7-R architecture. When ARMv8-R becomes more widely available, we believe

Architecture	Bit-width	Profile	Features
ARMv7-A	32	Application	Multiple cores, MMU, TrustZone
ARMv8-A	32/64	Application	Multiple cores, 64-bit support, MMU, TrustZone, hardware-accelerated cryptography
ARMv7-R	32	Real-time	Single- or multi-core, tightly coupled caching, MPU, real-time clock
ARMv8-R	32	Real-time	Single- or multi-core, tightly coupled caching, MPU, real-time clock, optional virtual memory, bare-metal hypervisor mode
ARMv7-M	32	Microcontroller	Single-core, low-power, Thumb2 ISA only, MPU
ARMv8-M	32	Microcontroller	Single-core, low-power, Thumb2 ISA only, MPU, TrustZone-M

Table 2.1: ARM processors and their features [2]

that the hypervisor mode will strengthen the security benefits this work provides (see Section 5.3.1 for more details).

Even after the release of ARMv8-R, the current ARMv7-R devices will still be in use. It would be impractical and expensive for manufacturers to replace all existing ARMv7-R hardware. Firmware updates, on devices that support them, are a much more practical way to add new features or fix bugs.

2.1.1 ARM/Thumb/Thumb2 Instruction Sets

Unlike other common architectures, ARM chips can operate on several different instruction sets. Most modern ARM cores support the ARM, Thumb, Thumb2, and ThumbEE ISAs. Some older ARM cores also support the Jazelle DBX (Direct Bytecode eXectution), which provides hardware support for executing Java bytecode. Additionally, ARM provides a standard interface for interacting with coprocessors such as hardware floating point units and the ARM Memory Protection Unit (MPU). Because our work exists mostly at the assembly language and machine code level, we need an understanding of the low level details of the ARM instruction set.

All ARM devices have 16 32-bit registers (R0-R15) and two status registers. Of the 16 32-bit registers, 4 have special purposes. R15 is reserved for use as the program counter (PC), R14 is the link register (LR), R13 is the stack pointer (SP), and R12 is the intra-procedure-call scratch register (IP). By convention, R0-R3 are used for function arguments, R4-R11 are used for temporary variables, and R0 is used for function return values. Registers R0-R7 are called the *Lo* registers, and R8-R15 are called the *Hi* registers. The Hi registers cannot be accessed by most 16-bit Thumb instructions, but they can be accessed by 32-bit Thumb instructions. All registers can be accessed from the ARM instruction set. The current program status register (CPSR) contains flags representing the current processing mode and status bits for conditional operations. The saved program status register (SPSR) is used during exception handling to restore the CPSR upon returning to normal processing.

Mode	Shared Registers	Banked Registers
System & User	R0-R15, CPSR	None
FIQ	R0-R7, CPSR	R8-R15, SPSR
Supervisor	R0-R12, CPSR	R13-R14, SPSR
Abort	R0-R12, CPSR	R13-R14, SPSR
IRQ	R0-R12, CPSR	R13-R14, SPSR
Undefined	R0-R12, CPSR	R13-R14, SPSR

Table 2.2: Register sets for ARM processing modes [3]

The ARM Procedure Call Standard [15] dictates that ARM subroutines must preserve the values of R4-R11 and LR. This means that if the subroutine calls any other subroutines using the branch-and-link instruction, it must push LR to the stack before the subroutine call.

2.1.2 Processing Modes

ARM processors can have up to nine different processing modes. On Cortex-R processors, there are only seven processing modes: User, System, Supervisor, Interrupt, Fast Interrupt, Abort, and Undefined. User and System modes are normal processing modes, while the other five modes are different types of exception states. In this section, we describe the main characteristics of these modes. Many of the registers are shared between modes, but there are some exceptions where the processing mode has its own banked registers that do not interfere with the registers in other modes. The banked registers for each mode are shown in Table 2.2. Our work depends on various ARM processing modes to perform operations at different privilege levels. Specifically, we use User, System, Supervisor, and IRQ mode.

User and System mode are similar, except System mode runs in a privileged state, allowing it to access regions of memory marked as privileged-only by the MPU. Unlike the other states, the majority of processing should be performed in either User or System mode.

When an ARM processor boots, the first code that executes is the `reset` routine. In the reset routine, the processor mode is set to Supervisor mode, which is a privileged mode that is designed for use as kernel mode for an operating system. Additionally, supervisor mode is used when handling a software interrupt (`svc`) instruction. This instruction can be used to implement system calls that need elevated privileges. Supervisor mode, along with all of the other exception modes, has a banked stack pointer register (SP_svc), link register (LR_svc), and saved program status register (SPSR_svc). These allow Supervisor mode to have its own stack, while also preventing it from modifying the previous mode's link register. The banked registers cannot be accessed by other processing modes. The SPSR is used to restore the previous program status upon exiting the exception state.

When external hardware generates an interrupt request, the processor will handle the interrupt in either Interrupt or Fast Interrupt Mode. Like Supervisor mode, both interrupt modes have banked SP, LR, and SPSR registers. The main difference between FIQ and IRQ modes is that FIQ

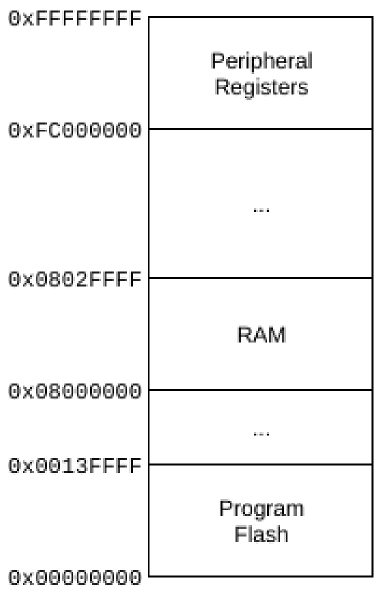

Figure 2.1: Simplified memory map for RM46L852 processor [1]

mode also has its own banked R8-R12, which means the FIQ service routine can use these registers for processing without saving them to the stack first. This allows FIQ requests to be handled much faster, which has various uses for fast data transfers or other tasks that need low latency responses. FIQ mode is also the only mode that can interrupt IRQ mode by default.

There are two kinds of aborts in ARM, prefetch aborts and data aborts. Prefetch aborts occur when the processor attempts to execute an instruction after a failed instruction fetch. This will occur when the processor tries to fetch an instruction from memory that is not executable, such as peripheral registers or regions marked non-executable by the MPU. Data aborts occur when the processor attempts to read or write data to memory that the memory system does not allow. In particular, this would include an attempt to write to flash memory from the CPU or an attempt to write to privileged memory from User Mode. In a general purpose operating system, an abort would likely result in a segmentation fault. In an embedded context, the default action is to enter an infinite loop or reset the CPU. Abort mode can be used in debugging, since the default handler will allow the developer to view the register state, while LR_abt will indicate which instruction caused the abort.

The final mode that ARMv7-R processors can use is Undefined mode. This mode is entered when the processor attempts to execute an opcode that it does not recognize. The intuition behind this exception state is to allow developers to implement instructions entirely in software, albeit at a high performance cost. For example, if a compiler does not support a certain coprocessor, the Undefined exception could allow the programmer to still use that coprocessor in a generic way.

2.1.3 Memory

All modern ARMv7 processors have a 32-bit address space. The layout of this memory is manufacturer and processor dependent, but in general, there are three important sections: non-volatile flash memory, volatile RAM, and memory-mapped peripheral registers. Flash memory stores program code and constants, and it may be used for non-volatile storage of data between reboots, although special procedures are required to write to flash memory. RAM is used for all volatile storage and may be used for executable code, if allowed by the Memory Protection Unit (MPU). The peripheral registers are used to configure various hardware on the chip, such as timers, analog-to-digital converters, and direct memory access controllers. All of these regions of memory exist in the same address space. Without protections, these regions could be exploited by an attacker to perform arbitrary read and write operations.

The reference system used for this work is a Texas Instruments Hercules RM46L852, an ARM Cortex-R4F processor. The full memory map for this processor is shown in Figure 2.1. We can see that this processor has 1.25 MB of non-volatile program flash, 192 KB of RAM, and an additional 64 KB of flash for emulated EEPROM storage. These three sections of actual memory do not fill up the entire 32-bit address space. Some of the additional sections include memory-mapped peripherals, mirrored images of flash, and reserved sections that may be used internally by the processor or not used at all.

Each of the sections of this address space has hardware limitations as to whether or not they can be written, read, or executed. Some sections will cause aborts, while others may just have no effect when trying to read or write. For more fine-grained memory permissions, most ARM Cortex-M and Cortex-R processors come with a Memory Protection Unit (MPU). The MPU enables setting up permissions for up to 12 regions of memory. For each region, there are three sets of permissions to be set: user mode access, privileged mode access, and execute permissions. For example, a region can be configured to be read-only for user mode, read and write for privileged mode, and non-executable (in any mode). Our work depends heavily on the MPU to create some isolation between privileged and non-privileged modes. MPU regions can be split into 8 subregions, where each subregion can be selectively disabled or enabled. Additionally, the main regions can overlap; the overlapped region will take on the permissions of the highest priority region. MPU violations result in a data abort, allowing the processor to reset or perform some kind of error correction. While not as powerful as a full memory management unit, the MPU can be used to set up some separation between user mode and kernel mode, although it was designed more as a safety feature than a security feature.

2.2 Control-Flow Integrity

Control-flow integrity (CFI) defends against control-flow hijacking attacks by ensuring that a given program follows a precomputed model of execution, called a control-flow graph (CFG). The intuition behind CFI is that if we are able to protect indirect control flow transfers, that is, any change in control flow computed at runtime, we prevent arbitrary code execution. Unlike other memory safety techniques [10,11], CFI is fast and each CFI check runs in constant time. In the remainder of this section, we discuss the intricacies of CFI, including the general implementation details and existing variants of CFI.

2.2.1 Threat Model

CFI schemes attempt to prevent exploitation under a powerful attacker model. The attacker is able to modify anything in writeable memory, including any data on the stack, heap, or in global memory. The attacker cannot, however, modify any read-only memory. Specifically, the attacker cannot modify program code, which is a common assumption, since most operating systems mark the program code section as being only readable and executable. Additionally, for CFI to be secure, an attacker must not be able to execute from any writeable memory.

The attacker's goal is to subvert the expected control flow of a program by modifying *code pointers* stored in memory or registers. A code pointer is any pointer that points to executable code. A function pointer is a type of code pointer. By modifying a code pointer, the attacker is modifying the target of an *indirect branch* instruction, which is any branch instruction where the target is calculated at runtime, rather than being statically inserted by the compiler. In ARM, an indirect branch is one of two kinds of instructions: (1) a branch instruction with a register operand, or (2) any operation with the program counter (PC) register as the destination. These instructions are enumerated in Table 2.3.

2.2.2 How CFI Works

All CFI schemes depend on two components: a control-flow graph (CFG) and binary instrumentation. Additionally, secure CFI schemes require a protected shadow call stack [12]. CFI implementations must accomplish three tasks:

1. From the CFG, extract a labeling scheme for all indirect jump targets.
2. At each indirect jump target in the binary, insert a label.
3. At each indirect jump instruction, insert instrumentation that ensures the jump target contains the expected label.

Assume for now that we have a CFG (we will discuss generating the CFG later). The CFG is a directed graph where the nodes represent basic blocks in a program, and the edges represent legal control-flow transfers from one basic block to another. Most of the nodes will be the source of one or two edges, since the majority of control-flow modifying instructions are direct calls or conditional branches, which have one or two possible targets. However, when a basic block ends in an indirect call or branch, there are often many possible targets for that control-flow transfer. The CFG generation algorithm will hopefully have reduced the total number of possible targets for that instruction. Each of the targets for that indirect branch are considered equal in the labeling scheme.

Mnemonic	Instruction	Description
`bx` *`Rm`*	Branch and exchange	Branch to target address *Rm*, and exchange instruction set based on least significant bit (LSB) of *Rm*. If LSB is set, switch to Thumb mode, else switch to ARM mode
`blx` *`Rm`*	Branch, link, and exchange	Branch to target address *Rm*, set link register, and exchange instruction set based on LSB of *Rm*
`ldm`{*`mode`*} *`Rm`*{*`!`*}, *`reglist`*	Load multiple	Load into registers in *reglist*, starting at address in *Rm*. If *Rm!* is specified, write back the final address into *Rm*. Mode specifies the addressing order: *ia* (increment after), *ib* (increment before), *da* (decrement after), *db* (decrement before). The pseudo instruction `ldmfd` is for loading from a full-descending stack. It is the same as `ldmia`.
`pop` *`reglist`*	Pop from stack	Same as `ldmfd sp!,` *`reglist`*
`rfe` *`Rn`*{*`!`*}	Return from exception	Pop PC and CPSR off of the stack pointer specified by *Rn* to return from an exception state. If *Rn!* is specified, write back new stack top to *Rn*.

Table 2.3: Different indirect jump operations in ARM

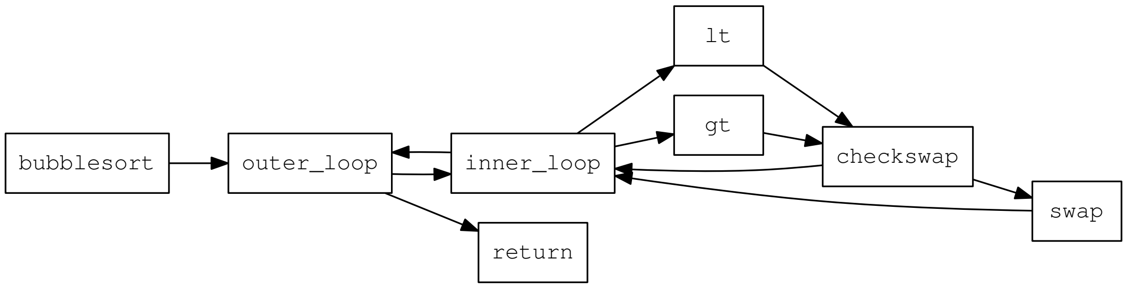

Figure 2.2: Example CFG of a sorting function

In general, a unique label is selected for each set of targets that has the same set of sources. This effectively results in a more permissive enforced CFG, but many CFI schemes take this approach because it offers improved performance, allowing label checks to be performed in constant time.

For example, imagine a sorting function that takes as an argument a pointer to a function that will perform the comparison between two elements. A possible CFG for this function is shown in Figure 2.2. The inner loop will call either the `lt` or `gt` function. Since these two blocks have the same source block, they would be given the same label for CFI instrumentation.

There are two goals to CFI instrumentation: (1) inserting label checks at all indirect control-flow transfers, and (2) labeling all possible indirect jump targets. To do this, every indirect jump is replaced with a few assembly instructions that attempt to read a label from some constant offset from the branch target address. The instrumentation then compares the label read from memory with a hardcoded label. If the labels match, the instrumentation continues with the original branch. If they do not match, then the code either jumps to some error handling code or exits, depending on the use case. In the case of an embedded system, the crash would be a good opportunity to safely shut down the system, preventing a control-flow violation from resulting in damage to an entity in the physical world.

The labels for CFI instrumentation must be chosen carefully. If the label appears coincidentally anywhere else in memory (except for a valid jump target), an attacker could circumvent CFI by overwriting a code pointer with an address that is the correct offset from the location where the label appears. Since an indirect jump may have multiple legal targets, this means that even a correct stateless CFI implementation may allow jumps that were not intended by the programmer. We will discuss this limitation further in Section 2.2.5. The other condition that each label must satisfy is that it should be side-effect free. Since the label is stored in executable code, the instrumentation should either ensure that the label is never executed, or it should be a side-effect free instruction. In the original CFI implementation, the x86 `prefetch` instruction was used to encode the label [9].

Another indirect jump that CFI instrumentation needs to handle differently is function returns. Unlike indirect calls, where there may be multiple targets, most functions should always return to their callsite. With a few exceptions, return addresses are stored on the stack, and can be overwritten by buffer overflows or other memory corruption techniques. To combat this, we instead store return addresses in a separate shadow stack in protected memory. The shadow stack should not be writeable except during the function prologue and epilogue. Unlike the other components of CFI, the shadow stack is not static. It is a dynamic component that enhances the precision of the CFG at runtime.

This section described the basic concepts behind implementing CFI. For a more concrete implementation example for ARM processors, see Chapter 3.

2.2.3 CFI Variations

Since the original CFI publication in 2005, several variations of CFI have emerged. Some of these variations trade security for speed, while others may forgo exploit prevention to increase the probability of exploit detection.

The original CFI design can be described as *forward-edge CFI* with a shadow stack. Forward-edge CFI refers to the instrumentation of indirect calls and branches to constrain the number of possible targets. A purely forward-edge CFI approach would either ignore function returns (the backward-edge), or it would treat each return like another indirect branch. In the former case, all return address corruption attacks would be possible. In the latter case, an attacker would be able to use a function called from multiple locations as a way to make the control-flow jump between the various callsites of the function. The shadow stack ensures that functions return to their callsite, although this can degrade performance by adding a dynamic element to CFI.

Apart from choosing how to protect the backward-edge, another way that CFI implementations can vary is in their control-flow graph. CFG generation in the presence of indirect branches is an unsolved problem, since resolving indirect branches is believed to be as hard as pointer analysis. As a result, CFGs used to create the labeling scheme for CFI are imprecise. Because of this, and for performance reasons, some CFI approaches use a *coarse-grained* CFG. Instead of attempting to resolve all indirect branches targets, a coarse-grained CFG may adopt a simple policy for branch targets. For example, a coarse-grained policy for return instructions would be that the instruction preceding the return address is a call instruction. For function calls, a coarse-grained policy could be that the function at the target address matches the return type and argument types of the function pointer. While coarse-grained CFGs are simpler to generate and can make instrumentation faster, is has been shown that coarse-grained policies are not secure [12].

Fine-grained CFGs, by contrast, attempt to resolve indirect jumps to the smallest set of legal jump targets. Due to the difficulty of pointer analysis, fine-grained approaches, we cannot construct a *fully-precise* CFG. A fully-precise CFG is both complete and sound. For an individual node in the CFG, a complete analysis will not include any branch targets that are not actually taken by the program, and a sound analysis will always accept a legal branch target. In other words, completeness refers to eliminating false positives, while soundness refers to eliminating false negatives. False positives reduce security, while false negatives could break program functionality.

The CFG is a static component of CFI. Because it is precomputed and stateless, it makes CFI instrumentation faster and more deterministic with good cache locality. Unfortunately, even if we could create a fully-precise CFG, the overall precision of CFI could still be increased with knowledge of program state. This is where dynamic elements like shadow stacks can be used to enhance the security guarantees of CFI, at the cost of performance. While shadow stacks are a simple solution to protect function returns, normal indirect branches with a large target set could be exploited to branch to an unexpected location given the current program state. These kinds of branches are called *dispatchers* and they could allow attackers to perform return-oriented programming (ROP) attacks, where useful sequences of existing instructions are chained together by return instructions and a maliciously crafted stack. Each sequence of instructions is called a gadget, and gadgets are linked together in ROP chains to perform more complex computation. Some of the proposed CFI schemes reduce the effectiveness of dispatchers by using more program context. For example, some schemes record multiple control flow transfers, and periodically compare the resulting paths against the CFG. While this can result in a more precise measurement of control flow, it can allow an exploit to occur before detection, which may be undesirable depending on the application.

With all of these variations in mind, the theoretical golden standard of CFI is fully-precise CFI with a shadow stack. If we were able to create a sound and complete CFG, this variation would provide the best tradeoff between security and performance, since control flow cannot be exploited, label checks and the shadow stack have low overhead compared to more dynamic approaches.

	Original CFI [9]	binCFI [16]	Modular CFI [17]	MoCFI [18]	KCoFI [19]	C-FLAT [13]	TrackOS [14]
Prevent Exploit	Y	Y	Y	Y	Y	N	N
Detect Exploit	Y	Y	Y	Y	Y	Y	Y
Forward-edge	Inline	Inline	Inline	Inline	Inline	Record	Record
Backward-edge	Shadow stack	Inline	Inline	Inline	Inline	Record	Record
Granularity	Fine	Fine	Fine	Fine	Coarse	Fine	Fine
Static/Dynamic	Both	Static	Static	Both	Static	Dynamic	Dynamic
Architecture	x86	x86	x86	ARM	x86	ARM	AVR
AIR	99.13% [16]	98.86% [16]	99.99% [17]	Unknown	98.18% [19]	Unknown	Unknown

Table 2.4: Different CFI implementations and their features

2.2.4 Existing CFI Implementations

All of the variations discussed in Section 2.2.3 have been used in one or more CFI implementations. Table 2.4 highlights some of the CFI schemes relevant to this work and shows some of the defining features.

The original CFI proposal is a fine-grained approach with a shadow stack. Much of the theory behind CFI has not changed, and the original proposal remains one of the most secure and efficient variants today. The reference implementation actually took more of a coarse-grained approach, but the researchers identified that a more restrictive CFG would be more secure. This scheme was originally developed for 32-bit Windows XP, and has since been adopted by Microsoft as a compiler option called Control Flow Guard, which was introduced in Windows 8.1.

More recent CFI schemes are designed to work on precompiled binaries in the presence of dynamic linking. One of these implementations, binCFI [16], achieves similar security to the original CFI implementation without relying on relocation information from the compiler. One of the important outcomes of that work was a new metric for describing the precision of the control-flow graph: average indirect target reduction (AIR). The intuition behind the metric is that without CFI, an indirect branch can target any instruction, and the CFG reduces the size of the target set for each indirect branch. Equation 2.1 shows how AIR is calculated, where n is the number of indirect branches, T_j is the set of calculated targets for indirect branch j, and S is the number of possible targets in an unprotected program.

$$\frac{1}{n}\sum_{j=1}^{n}\left(1-\frac{|T_j|}{S}\right) \tag{2.1}$$

While AIR is not a perfect metric for evaluating CFI, it provides us with a quantitative measurement that gives us a way to sanity check our implementation. Acceptable AIR values are usually around 99% for a fine-grained practical implementation, while an extremely coarse-grained implementation (such as limiting indirect branches to any valid instruction boundary), could see an AIR of about 80%, depending on the architecture.

The majority of CFI schemes were developed for the x86 architecture. With mobile devices becoming more ubiquitous, some security research efforts, including CFI research, moved towards targeting the ARM architecture. Two of the first CFI schemes for ARM, MoCFI [18] and Control-flow Restrictor [20] were developed for Apple iOS devices. These CFI schemes identified and overcame some of the challenges associated with developing a CFI scheme on an ARM-based system. The main architectural challenges for CFI on ARM include:

- No dedicated return instruction
- Instructions can operate on the program counter
- Multiple instruction sets in a single binary

The original ARM CFI implementations only targeted iOS smartphones, but with the rise of the Internet of Things, ARM devices with no operating systems became more relevant. These embedded systems have qualities that make CFI a good potential hardening technique. For example, programs running without an operating system have no dynamic linking, which simplifies CFG generation. Additionally, some of the newer ARM Cortex-A and Cortex-M devices are shipping with a trusted execution environment called TrustZone, which could be used for measurement and attestation. Control Flow Attestation (C-FLAT) does exactly this; it leverages ARM TrustZone to implement a CFI scheme designed for embedded systems without operating systems [13]. Unlike traditional CFI, C-FLAT's instrumentation does not perform jump target checking. Rather, it records control-flow events, and periodically verifies the validity of each path from within a secure environment. Unfortunately, this dynamic approach introduces time-of-use to time-of-check vulnerabilities. C-FLAT will detect a control-flow violation, but it may not detect it until after malicious code has executed.

As we move on to discuss CFI on real-time ARM systems in Chapters 3 and 4, we are most interested in providing a better solution than C-FLAT and TrackOS, while taking a more traditional CFI approach. These two systems sacrifice some important safety-oriented security qualities of traditional CFI to avoid interfering with real-time functionality.

2.2.5 Limitations of CFI

As we have alluded to in previous sections, CFI is not a perfect defense against control-flow hijacking attacks. There are some vulnerabilities that depend on the implementation details. Additionally, recent research has shown that even against fully-precise CFI with a shadow stack, certain types of gadgets are available and can be used to completely subvert CFI.

In coarse-grained CFI, few unique labels are used in the instrumentation, and a specific indirect branch may have many incorrect targets in its target set. It is possible that the branch may only have one or two intended targets, but the coarse-grained policy gave it ten targets. This means that the majority of targets should not be legal, but an attacker would be able to execute the eight or nine unintended targets in the target set. In fact, while coarse-grained policies do decrease the number of possible ROP gadgets, a control-flow hijacking attack is still trivial.

Dynamic CFI implementations that rely on recording control-flow events suffer from a *time-of-use to time-of-check* vulnerability. Because these systems only periodically attest that the control-flow events followed the CFG, they can allow an exploit to occur before detecting it. If an attacker exploits a buffer overflow, the system will record the return to the corrupted address, and then continue executing. Only when it runs its verification task will it detect that an illegal control-flow event occurred. By that point, however, the damage will have been done, since the exploit already executed.

Even with fully-precise CFI with a shadow stack, programs can be vulnerable to *control-flow bending*, a type of attack that includes a new exploitation technique called *`printf`-oriented programming* [12]. In this kind of attack, the attacker must be able to corrupt the arguments to a function like `printf`. It turns out that by controlling the arguments to a format string function, an attacker can gain Turing-complete computation without ever violating the CFG. This kind of vulnerability can be generalized as an *argument-corruptible indirect call site (ACICS)* gadget [21]. By being able to control the arguments to the function being called indirectly, it is possible for an attacker to gain arbitrary execution, especially if the ACICS gadget allows a call to `system` or the `exec` family of functions.

2.3 Real-Time Operating Systems

Often the main purpose of an embedded system is to monitor or control physical systems. For example, an embedded system may be controlling an electric motor in an automobile or it could be working alongside an application processor to provide some low-level communication protocol. In these cases, certain procedures must be performed in *real-time*, meaning that after an event occurs, the processor must perform the appropriate computation by a certain deadline. There are two kinds of real-time systems, *soft real-time* and *hard real-time*, where the former can tolerate missing some deadlines, while the latter cannot. In hard real-time systems, a missed deadline puts the entire system into an unknown—and often unsafe— state. To facilitate writing software with real-time requirements, embedded system designers often use a *real-time operating system (RTOS)*.

Depending on the processor being used, RTOSs can vary greatly in their complexity. On more powerful hardware, like ARM Cortex-A processors, real-time processing can be facilitated by using Linux compiled with `SCHED_DEADLINE` or `SCHED_RT`, which replace Linux's Completely Fair Scheduler (CFS) with real-time schedulers. On processors designed for embedded use, like the Cortex-R4F used for this work, the hardware often fails to meet the minimum requirements for Linux. For reference, in 2014, a minimally configured Linux kernel still required at least 8 MB of program flash and 1.6*MB* of RAM [22], which clearly will not work on our 1.25 MB of flash and 192 KB of RAM. The alternative to real-time Linux is using an embedded RTOS such as FreeRTOS, VxWorks, or μC/OS, which are designed to run on devices with storage space and memory on the scale of kilobytes, rather than megabytes or gigabytes.

In an embedded RTOS, processes are referred to as *tasks*. Tasks are most similar to a system service or daemon in a general purpose operating system; they generally do not exit and they execute periodically, often sleeping until they are signalled to do some work. Depending on the RTOS, there are different ways that the OS can schedule tasks. Often there is some flexibility in choosing the type of scheduling; the most significant option being choosing between preemptive and nonpreemptive scheduling. In preemptive scheduling, the scheduler will run periodically during a timer interrupt, allowing tasks to be switched out at any time. In nonpreemptive scheduling, a task must willingly yield to the scheduler. The benefit of nonpreemptive scheduling is that it makes it much easier to reason about the schedulability properties of the system. A real-time system designer may use modeling software to prove that the system will not miss deadlines, and it is much simpler to model the system if the scheduler only runs at deterministic points in the application.

2.3.1 FreeRTOS

One of the most common real-time operating systems is FreeRTOS. Designed to be as minimal as possible, this free and open source RTOS can fit in as small as 5 KB of program flash and can fit in under 1 KB of RAM, depending on the features used [23]. FreeRTOS is highly portable, with architecture specific ports existing for most major architectures. FreeRTOS, while minimal in nature, provides a few rich features such as mutexes, semaphores, shared queues, and software timers.

Tasks and Memory Layout

Tasks in FreeRTOS can be viewed as lightweight threads sharing the same address space and heap, but separate stacks. The kernel views tasks as an array of circular doubly linked lists of Task Control Blocks (TCBs), which each list containing all tasks of a certain priority. The TCB for each task maintains information including the pointer to the top of its stack, priority, and task state (ready, blocked, suspended). Optionally, this structure can include information about synchronization objects (e.g. mutexes) in use by the task, or it could maintain tracing information, if enabled.

At boot time, FreeRTOS only allocates a few global variables and a heap. When tasks are created, the TCB and task stacks are allocated in the FreeRTOS heap, which is shared between all tasks by default. Since the `libc`-provided `alloc` family of functions is not appropriate for real-time use, FreeRTOS provides its own dynamic memory allocation scheme. While this dynamic allocation can be used, static allocation is preferred since it is deterministic and does not require a critical section to access the shared heap.

Porting

The majority of FreeRTOS is written in architecture independent C code, but each port to a new architecture requires a few special files to be implemented. Since we will be discussing modifying FreeRTOS in Chapter 4, we will briefly look at these files now. There are three files that are required: `portmacro.h`, `port.c`, and `portasm.S`.

The first file that the FreeRTOS core depends upon is a C header file called `portmacro.h`, which creates the type abstractions and a few macros for architecture-specific actions like disabling and enabling interrupts. The type abstractions include defining `StackType_t` and `BaseType_t`, which represent the stack element type and word size, respectively. Additionally, this file defines some constants like the stack growth direction and addresses of peripheral registers.

The second file is a C source file called `port.c`. This C file must define important procedures like the stack initialization for a task, the real-time clock configuration, and the real-time tick interrupt. Even within one type of processor like the Cortex-R4, different vendors may have different ways to configure hardware peripherals.

The last file is `portasm.S`, which contains port-specific assembly code. Two of the most important parts of this file (which we will be discussing modifying in Chapter 4) are the `portSAVE_CONTEXT` and `portRESTORE_CONTEXT` assembly macros. The scheduler uses both of these macros to switch context between tasks. Since this procedure requires modifying specific registers, it must be implemented in assembly language rather than C.

Chapter 3

Control-Flow Integrity on ARM Cortex-R

Before implementing a CFI scheme on an ARM-based real-time operating system, the logical first step is creating a working CFI system on a bare metal ARM program. Since the existing ARM implementations either require an operating system or are unable to handle both Thumb and ARM, we implement our own instrumentation and shadow stack. The rest of this chapter outlines the design goals, system model, threat model, and approach to our implementation. The evaluation of our implementation is detailed in Chapter 5.

3.1 Design Goals

The main goal of any CFI scheme is to make it difficult or impossible for an attacker to achieve arbitrary code execution. Our main design goal is to develop CFI instrumentation for the ARM and Thumb2 instruction sets that restricts control flow to prevent arbitrary code execution. As with previous work, we do not protect against arbitrary memory corruption vulnerabilities; rather we focus on preventing the exploitation of code pointer corruption. Our system will focus on the static label checks and runtime additions that CFI provides, rather than focusing on the CFG generation aspect of CFI. While it is shown that a CFI scheme is only as secure as its CFG is precise [12,16,21,24], we look to the literature for control flow graph generation [17,25,26], since CFG generation is beyond the scope of this project.

Performance and resource utilization are important aspects for any security software, but they are especially critical for embedded software. Performance overhead should be kept to a minimum, but most importantly, performance overhead should be deterministic. Embedded devices often have tight time constraints where guarantees about meeting these constraints are dependent on deterministic, bounded execution time. On general purpose computers, resource utilization is often overlooked; resources like disk space and RAM are abundant, so security software can often add significant resource usage without much penalty. On embedded devices, however, limited code storage and RAM means we need to take special care to minimize the additional code and runtime memory requirements for CFI.

There have been various theoretical and practical attacks against fine-grained CFI, including

Control Jujutsu [21] and Control-flow Bending [12]. While we recognize these attacks, we do not attempt to prevent them. Due to differences in programming paradigms for embedded systems, we believe that the likeliness of these attacks being possible on the majority of embedded devices is low. We discuss this more in Section 5.3.1.

3.2 System Model

The system that we are targeting with this work is the ARM Cortex-R processor, unlike previous work which targets the Cortex-A architecture [13,18,20]. The real-time features of the Cortex-R mean that it does not have virtual memory, and all memory protections are limited to the MPU hardware. Additionally, we assume that the application running on the processor can contain both ARM and Thumb instructions, and apart from any initialization code, any processing outside of an interrupt context is in User mode. Finally, as a requirement for CFI, at least two regions of the MPU will be used to disable writing to executable code while also preventing execution of writeable memory. The remaining regions can be used for any special memory protections required by CFI (e.g. shadow stack).

3.3 Threat Model

We assume the normal threat model for CFI implementations. We assume a powerful attacker model; without protections, the attacker can arbitrarily read and write to any memory at any time. This means that to meet the minimum requirements for CFI, we first need to apply memory protections. After memory protections are applied, the attacker is still able to read and write arbitrary memory at any time, but attempts to access to privileged-only memory from user mode will cause the device to crash. Since there are only two privilege levels, any code running in privileged mode must be critical to CFI and must not be vulnerable to memory corruption attacks. On general purpose systems this would be near-impossible to verify since privileged (i.e kernel) code contains millions of lines of code, but in embedded systems, these sections of code are generally small, manually verifiable, and do not accept arbitrary external inputs. These sections are primarily used for handling hardware interrupts. While we cannot entirely discount vulnerabilities in privileged sections of code, it is much more likely that the bug causing the vulnerability would be fixed due to faults occuring without any malicious intent.

3.4 Approach

As discussed in Section 2.2, CFI implementations have two major parts: CFG generation and binary instrumentation. Since sound and precise CFG generation is believed to be undecideable [21], we do not attempt to improve upon existing CFG generation algorithms such as those presented in [25, 26], and instead focus on the binary instrumentation aspect of CFI. Our binary instrumentation

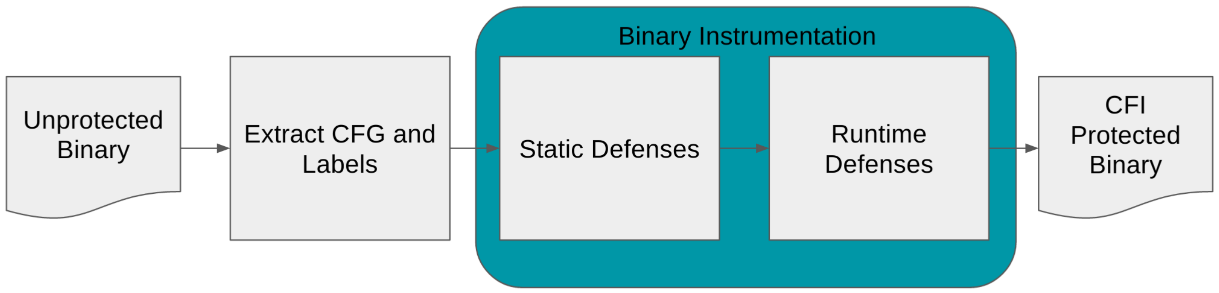

Figure 3.1: Overview of our CFI system

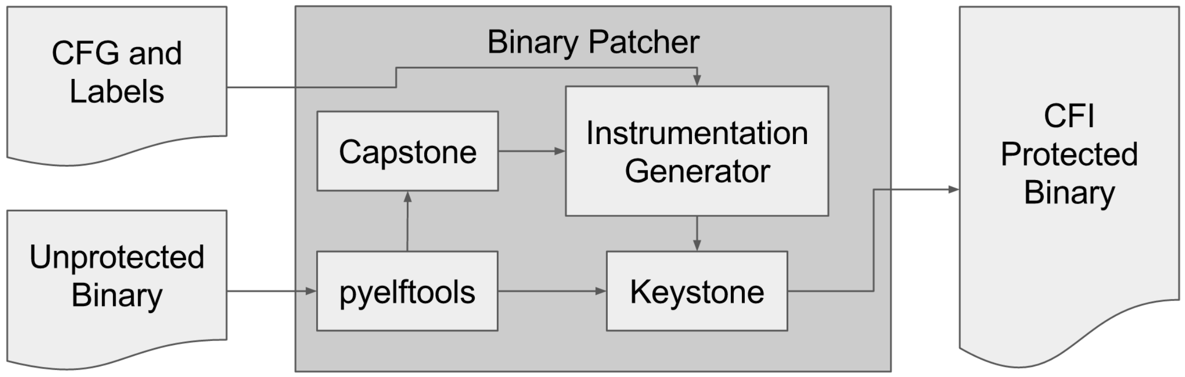

Figure 3.2: Overview of binary patching system

comprises two parts: static defenses and runtime defenses. The overview and workflow of this system is shown in Figure 3.1.

3.4.1 Binary Instrumentation

One of the major design choices for this CFI implementation is the instrumentation type. There are two options: modify the compiler and add instrumentation at compile time or patch the binary after compilation. While the compiler-based approach would likely have better performance, the patching approach is able to add CFI to a compiled binary without needing source code. One benefit of being able to instrument a compiled binary is that we can retrofit security to an existing device that may never receive updates. We chose the patching approach due to the vast numbers of proprietary embedded systems that ship with precompiled firmware without source code available for recompilation. An overview of our binary patcher is shown in Figure 3.2

To perform the binary instrumentation required for CFI, we use the binary patching approach. In this approach, the main idea is to instrument instructions by replacing them with trampolines to new code appended to the unused flash memory section. The naive version of this algorithm is shown in Algorithm 1.

Unfortunately, real ARM machine code has some properties that make this patching more difficult depending on the instructions being patched. There are two main cases where patching

Algorithm 1: Patching algorithm

```
input  : A binary $B$, Address to instrument $A_i$, target address $A_t$, instrumentation code $I$
output: Patched binary with added trampoline and instrumentation
1 $instruction \leftarrow B[A_i]$;          // Save binary instruction at instrumentation address
2 $I \leftarrow instruction + I$;           // Prepend saved instruction to instrumentation
3 $B[A_i] \leftarrow$ branch to $A_t$;      // Insert branch at instrumentation address
4 $B[A_t] \leftarrow I$;                    // Insert instrumentation at target address
```

becomes more complicated. The first is when the instruction being instrumented is a 16-bit Thumb instruction. The branch address required by the trampoline procedure has to be 32-bits, since ARM branches are PC-relative and we need to be able to encode a large offset to the unused flash section. The second case is when the instrumented instruction contains a PC-relative operation, since the new location of the instruction will have a different PC value.

To show how to handle these cases, we look to following toy example. Imagine we are trying to instrument the `foo()` shown in Listing 3.1. We want to instrument the function prologue, the indirect call, and the function epilogue. Looking at the disassembly of `foo()` when compiled in Thumb mode, we identify the instructions we need to instrument, shown in Listing 3.2.

Listing 3.1: C function to instrument

```c
int foo(int a, int b) {
    int (*func[2])(int, int) = {add, sub};
    static unsigned int i = 0;
    return func[i++ % 2](a, b);
}
```

Listing 3.2: Selected portions of disassembly for foo() function

```
0x192:    push {r4, r7, lr} #
0x194:    sub sp, 20        # Function Prologue
0x196:    add r7, sp, 0     #
...
0x1e6:    add r3, r3, r4    #
0x1e8:    blx r3            # Indirect Call
...
0x1f0:    mov sp, r7        #
0x1f2:    pop {r4, r7, pc}  # Function Epilogue / Return
```

In the function prologue, CFI requires that we push the return address to the shadow stack as well as embed a label according to the CFG. We need to replace 6 bytes of the function to encode both a 32-bit branch and a 16-bit label, as shown in Listing 3.3. We branch to the code shown in Listing 3.4, which modifies the original stack push operation to remove the link register from the

Listing 3.3: Rewritten version of foo() function

```
0x192:    b.w 0x13f60       # New prologue branches to CFI section

0x196:    <label>           # Insert label after branch
...
0x1e6:    bl 0x13f80        # Replace indirect call and previous instruction with
                            # branch to CFI section to do label checking
...
0x1f0:    b.w 0x13f98       # New epilogue branches to CFI section
```

register list, pushes the link register to the shadow stack, and returns to the instruction following the label placed in the original function prologue.

Listing 3.4: Function prologue instrumentation

```
0x13f60:  push {r4, r7}     # Copy instructions from original prologue, but
0x13f62:  sub sp, 20        # remove link register from the push register list
0x13f64:  add r7, sp, 0
0x13f66:  push {r0}         # Push r0 since we will need it
0x13f68:  mov r0, lr        # Copy link register value into r0
0x13f6a:  svc 0             # Call ss_push from supervisor mode (system call)
0x13f6c:  pop {r0}          # Restore r0 value
0x13f6e:  b.w 0x198         # Branch back to original function, skipping over label
```

Listing 3.5: Indirect call instrumentation

```
0x13f80:  add r3, r3, r4    # Copy instruction from before branch
0x13f82:  push {r0, r1}     # Push r0 and r1 since we will need them
0x13f84:  ldrh r0, [r3, 3]  # Load the label (halfword) from 3 bytes after branch target
0x13f86:  movw r1, <label>  # Load the expected label into r1
0x13f88:  cmp r0, r1        # Compare r0 and r1
          error:
0x13f8a:  bne error         # If not equal, enter infinite loop
0x13f8c:  pop {r0, r1}      # Restore r0 and r1
0x13d8e:  bx r3             # Perform indirect jump
```

The indirect jump at address `0x1e8` in Listing 3.2 is a branch-link-and-exchange instruction with a register operand. This is a 16-bit indirect call, so we replace it and the preceding instruction with a direct branch-and-link operation. By using the branch-and-link operation, we ensure that the return address is copied into the link register for use by the called function. In the instrumentation code shown in Listing 3.5, we perform a label checking operation based on the target of the indirect call. If the labels match, we continue with the indirect call, but if not, we enter an infinite loop.

The function epilogue needs to reverse the operations done by the function prologue. Namely, we need to restore registers pushed to the stack previously, and we need to restore the return address from the shadow stack. To do this, we again replace the two 16-bit instructions with a 32-bit branch instruction. In the instrumentation shown in Listing 3.6, we start by popping the return address off the shadow stack and moving it into the link register. We then modify the original pop instruction to remove copying of the link register into the program counter (since we removed pushing the link register in the prologue). Finally, we return using a branch-and-exchange to the link register.

Listing 3.6: Function epilogue instrumentation

```
0x13f98:  mov sp, r7          # Perform pre-return operation that we overwrote
0x13f9a:  push {r0}           # Push r0 so we can use it for return address
0x13f9c:  svc 1               # Call ss_pop from supervisor mode (system call)
0x13f9e:  mov lr, r0          # Copy return value from ss_pop to link register
0x13fa0:  pop {r0}            # Restore r0
0x13fa2:  pop {r4, r7}        # Perform original pop operation, without popping into pc
0x13fa4:  bx lr               # New return instruction
```

To implement this instrumentation, we use the Capstone[1] disassembly engine, the pyelftools[2] ELF file parser, and the Keystone[3] assembler. Capstone provides a powerful disassembly and instruction decomposition framework that makes it easy to identify the registers modified by any instruction. We search the executable for indirect branches and instructions that indirectly modify the program counter register (such as a load multiple operation where PC is a destination register). After enumerating the instructions that need instrumentation, we follow the simple procedure outlined in Algorithm 1 to generate the instrumentation. Finally, we use the Keystone assembler to compile the instrumentation code to machine code, and write the patched code to the binary file. We follow the same procedure for function prologues, epilogues, and indirect branch targets until we have a fully instrumented binary.

At a high level, the resulting binary follows the format shown in Figure 3.3. The code being instrumented is in the `.text` section, and the instrumentation goes into a new section called `.cfi`, as shown in Figure 3.4. Both of these sections are stored in flash memory, and are not modifiable at runtime.

3.4.2 Shadow Stack

To implement a shadow stack, there are only two main requirements: a block of free memory and a way to protect this memory from unwanted access. On ARM Cortex-R, the best way to protect this memory is to make it inaccessible from the User processing mode, but allow read and write from the Supervisor mode. The procedure to perform shadow stack essentially requires creating a

[1]http://www.capstone-engine.org/
[2]https://github.com/eliben/pyelftools
[3]http://www.keystone-engine.org/

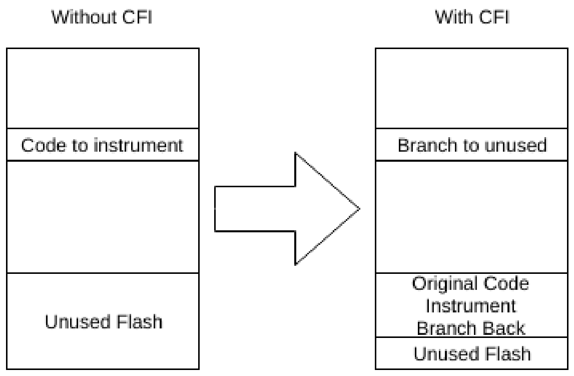

Figure 3.3: Input and output of the binary patching program

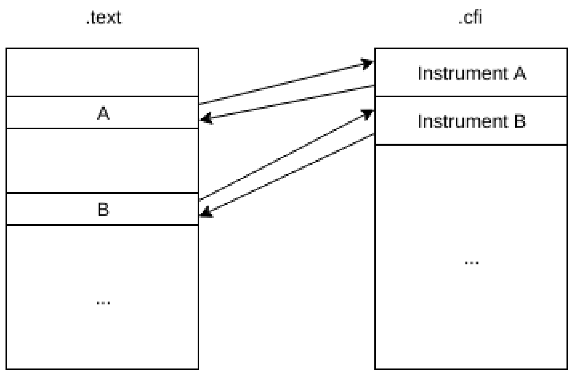

Figure 3.4: Connection between the new `.text` section and the `.cfi` section

system call interface using the ARM Supervisor call (`svc`) instruction.

The Supervisor call instruction takes one operand, an immediate value representing the function number. When it executes, the `svc` instruction triggers an interrupt on the processor. The handler for this interrupt must determine the function number by reading the opcode of the software interrupt instruction. The ARM assembly code function to do this is shown in Listing 3.7.

Listing 3.7: Supervisor call handler

```
do_syscall:
    stmfd   sp!, {r9,r10,r12,lr}     # Store registers
    mrs     r9, spsr                 # Move SPSR into general purpose register
    tst     r9, 0x20                 # Occurred in Thumb state
    ldrneh  r9, [lr, -2]             # Yes: load halfword and
    bicne   r9, r9, 0xFF00           #       extract function number field.
    ldreq   r9, [lr, -4]             # No:  load word and
    biceq   r9, r9, 0xFF000000       #       extract function number field.
                                     # r9 now contains SWI number
    cmp     r9, 2                    # Range check
    ldr r   10, table                # Load address of table
    ldrls   pc, [r10, r9, lsl 2]     # Jump to the appropriate routine.
    b       svc_out_of_range         # Branch to error condition if number out of range

table:
    .word jump_table

jump_table:
    .word ss_push
    .word ss_pop
```

The actual shadow stack implementation is simple. We declare an array of bytes, ensuring that it is aligned to the size of an MPU region. For our reference implementation, we use a 256 byte shadow stack. The C definition of the shadow stack is shown in Listing 3.8. Upon system startup, the MPU is configured to make the shadow stack readable and writeable by privileged code, and not accessible at all by User mode code. Additionally, the shadow stack is set non-executable, like the rest of RAM. Shadow stack operations require a system call to switch to Supervisor mode, which is the mode for handling a software interrupt exception. Depending on the function required, we either perform a simple stack push or stack pop operation, and return to user mode. The short ARM assembly code functions for the shadow stack operations are shown in Listing 3.9.

Listing 3.8: Shadow stack type definition

```
typedef struct shadow_stack {
    void *stack_top;
    void *stack_base;
    uint32_t size;
} shadow_stack;

volatile shadow_stack *current_ss;
```

Listing 3.9: Shadow stack operations

```
# Input: r0 containing value to push to shadow stack
# Returns: void
.type ss_push, %function
ss_push:
        ldr r9, current_ss_const    # Load current shadow stack pointer
        ldr r10, [r9]               # Load shadow stack top
        stmfd r10!, {r0}            # Push r0 to shadow stack
        str r10, [r9]               # Store new top of shadow stack
        exit_syscall                # Syscall exit macro

# Input: void
# Returns: value at top of shadow stack -> r0
.type ss_pop, %function
ss_pop:
        ldr r9, current_ss_const    # Load current shadow stack pointer
        ldr r10, [r9]               # Load shadow stack top
        ldmfd r10!, {r0}            # Pop from shadow stack to r0
        str r10, [r9]               # Store new shadow stack top
        exit_syscall                # Syscall exit macro

# Constant pointer reference to current_ss
current_ss_const    .word   current_ss
```

Chapter 4

CFI Integration into FreeRTOS

With a working CFI implementation for bare-metal ARM processors, we change our focus to integrating this instrumentation into an embedded application running FreeRTOS. With FreeRTOS, we now have an added complication: multiple threads running in the same address space with no isolation. The remainder of this chapter outlines the design goals, system model, threat model, and approach for integrating CFI into FreeRTOS. The evaluation for this system will be discussed in Chapter 5.

4.1 Design Goals

The design goals for CFI integration into the FreeRTOS operating system are essentially the same as those discussed for the bare metal implementation in Section 3.1. The main difference between the bare metal version and FreeRTOS is context switching and multithreading. This means that we can no longer assume that CFI checks are atomic operations, introducing the possibility of a time-of-check to time-of-use vulnerability. In the original bare metal implementation, CFI labels were loaded from memory into registers, and the label checking operation never stored any critical data in memory. Based on our threat model, this meant that an attacker could not tamper with the CFI check itself. In the presence of context switching, however, registers with CFI critical information (the two registers with labels and the register storing the branch target) could be stored in memory at any point during the CFI check.

Given this new vulnerability, the main goal of this CFI implementation is to prevent the time-of-check to time-of-use vulnerability by making CFI checks appear to be atomic. The easiest way to do this is to disable interrupts during CFI checks, but the additional latency for real-time tasks is undesirable. Instead, we leverage the limited available memory protections to ensure that saved context is safe from corruption.

4.2 System Model

Like the design goals for this system, the system model remains mostly unchanged from the bare metal model in Section 3.2. The hardware remains the same, but we now assume that the software contains a real-time operating system, adding multithreading with no address space isolation between threads. We are still working on an ARM Cortex-R processor with mixed ARM and Thumb2 instructions. In terms of privileged execution, we assume that tasks run in User mode, while the scheduler runs in IRQ mode, one of the privileged modes.

4.3 Threat Model

The threat model for the FreeRTOS CFI implementation is the same as discussed in Section 3.3. The attacker has arbitrary read and write access to the memory at any time. Now that context switching is within the system model, this gives the attacker a powerful attack against the unchanged bare metal implementation. As mentioned in Section 4.1, a time-of-check to time-of-use vulnerability in CFI checks is now exploitable under this attacker model.

4.4 Approach

The challenge with integrating our single-threaded, bare metal ARM implementation into an RTOS like FreeRTOS is that we now have multiple processes running in the same address space. Each process, or task as they are called in FreeRTOS, has its own stack, but there is no isolation preventing one task from writing to another task's stack at runtime. There is a FreeRTOS port that uses the MPU to prevent this, but it incurs a lot of overhead for context switching. The FreeRTOS-MPU port essentially makes it so that when a task is running, it can only read and write to its own stack. This makes it difficult to have shared resources between threads in common multithreaded programming paradigms, like shared queues with producer and consumer threads.

We feel that this isolation scheme places too many restrictions on the way that FreeRTOS tasks can behave, so we take a slightly more permissive approach to creating isolation. While our approach will allow tasks to write to another task's stack, it will not allow overwriting a task's saved context. The main intuition behind this is that forward-edge CFI and the shadow stack will prevent any control-flow exploit from executing in every case except when a task has been switched out during a CFI check. If this happens, the registers being used for the check will be saved into unprotected memory, allowing a well-timed exploit to overwrite the memory containing the CFI check registers. With this vulnerability, the attacker could overwrite the register containing the label or the register containing the indirect branch target. This time-of-check to time-of-use vulnerability is a common race condition in multithreaded programs, and the traditional CFI threat model does not take this into account.

One solution would be to disable interrupts during the CFI checks, but there are two reasons why this is not ideal. First, disabling interrupts and enabling interrupts both require an instruction.

Making CFI checks have to execute two more instructions adds overhead and increases the space required for instrumentation. The second reason is that any time interrupts are disabled, we are introducing a new source of latency for real-time tasks. Even in non-preemptive systems (where the scheduler only runs when a task willingly yields), other real-time sensitive hardware interrupts could be negatively affected by disabling interrupts frequently.

As an alternative to disabling interrupts during every CFI check, we propose storing the saved context in the task's shadow stack, rather than on the unprotected regular stack. The scheduler already runs in privileged mode with interrupts disabled, so we do not incur additional overhead from the Supervisor call instruction. To do this, FreeRTOS needs to be modified to support multiple shadow stacks, which means modifying the Task Control Block (TCB) structure, the task creation procedure, and the scheduler.

4.4.1 Shadow Stacks

Before modifying FreeRTOS to add a shadow stack to each task, we need to statically allocate a block of memory to use for the shadow stacks. The C definitions and declarations are shown in Listing 4.1. By statically allocating the memory and aligning it to the size of a shadow stack, we can more easily use the MPU to protect the region, since all regions must be aligned to the size of the region. The shadow stack operations have not changed from those shown in Listing 3.9.

Listing 4.1: C type definition and declaration for multiple shadow stacks

```c
static union {
    struct {
        StackType_t ss_data_arr[NUM_TASKS][SS_SIZE_WORDS];
        shadow_stack ss_arr[NUM_TASKS];
    } ss_data;
    uint32_t words[SS_SIZE_WORDS * NUM_TASKS] __attribute__ ((aligned (SS_SIZE_BYTES * NUM_TASKS)));
} shadow_stack_data;

// For use by the FreeRTOS task creation procedure
StackType_t next_ss = 0;
StackType_t (*ss_data_arr)[SS_SIZE_WORDS] = shadow_stack_data.ss_data.ss_data_arr;
shadow_stack *ss_arr = shadow_stack_data.ss_data.ss_arr;

// For MPU initialization
const void *ss_data_block = (void*)&(shadow_stack_data);
const StackType_t ss_data_size = sizeof(shadow_stack_data);

// To keep track of the current shadow stack
volatile shadow_stack *current_ss;
```

4.4.2 FreeRTOS Tasks

With a block of memory allocated for the shadow stacks, we need to modify the FreeRTOS task creation procedure to assign a shadow stack to each task when it is created. To do this, we have to modify the Task Control Block (TCB) structure to add a field for a shadow stack. The relevant portions of the TCB defintion are shown in Listing 4.2.

With the data structure in place, we modify the functions that intiialize this structure, the FreeRTOS function `prvInitialiseNewTask` and the port-specific FreeRTOS function `pxPortInitialiseStack`. All that `prvInitialiseNewTask` has to do is assign the next available shadow stack to the TCB of the newly created task. The port-specific `pxPortInitialiseStack` function is a bit more complicated. When FreeRTOS creates a task, it sets up the stack such that the task appears to have been switched out by the scheduler. This is an optimization that allows FreeRTOS to simply use its restore context routine to start a task, rather than needing a special procedure.

For ARM devices, the stack initialization routine needs to emulate having pushed the relevant registers to the stack, as well as a few variables that FreeRTOS depends upon for each task. We modify this procedure to put all of the registers in the shadow stack, since we do not want any saved context to be vulnerable to malicious modification.

Listing 4.2: FreeRTOS Task Control Block definition

```c
typedef struct tskTaskControlBlock
{
    volatile StackType_t        *pxTopOfStack;
    #if ( cfiUSE_SHADOW_STACK == 1)
        shadow_stack *ss;
    #endif
    ...
} tskTCB;

typedef tskTCB TCB_t;
```

4.4.3 Context Switching

With support for shadow stacks added to task creation, the only part left is to modify the scheduler to use the shadow stack. Normally, the scheduler saves context for each task on the task's unprotected stack. Instead, the scheduler needs to save context to the protected shadow stack.

Most of the scheduler is written in ARM assembly, and the instruction set makes it easy to save context to the unprotected stack. Normally, to save the register information to the stack from the scheduler, only two instructions are needed: `srs` and `push`. The `srs` mnemonic is the *store return state* instruction, which pushes the IRQ return address and the saved process state register to the system or user mode stack (see Section 2.1.2 for more information regarding processing modes). After saving the return state, the scheduler switches to system mode and pushes the rest of the registers to the stack. To modify this to use the shadow stack, we need to get the pointer to the top of the shadow stack, and use this like the stack pointer. The full code for the context saving routine is shown in Listing 4.3, and the full code for the context restoring routing is shown in Listing 4.4.

Listing 4.3: FreeRTOS save context routine with shadow stack

```
.macro portSAVE_CONTEXT:
    dsb                                 # Data synchronization barrier
    isb                                 # Instruction synchronization barrier
    # Store registers in the shadow stack
    push    {r0, r1, r2}                # Save temp registers
    ldr     r0, current_ss_const        # Load location of current shadow stack into r0
    ldr     r0, [r0]
    ldr     r1, [r0]                    # Load shadow stack top into r1
    mrs     r2, spsr                    # Load saved process status register (SPSR) into r2
    stmfd   r1!, {r2}                   # Push SPSR to shadow stack
    stmfd   r1!, {lr}                   # Push LR to shadow stack
    stmfd   r1, {r3-r12, lr}^           # Push user/system mode registers onto shadow stack
    sub     r1, r1, 44                  # Manually decrement shadow stack top
    mov     r3, r0                      # Now that r3-r12 are saved, we can use r3 and r4
    mov     r4, r1                      # to point to the shadow stack to finish saving context
    pop     {r0, r1, r2}                # Restore temp registers
    stmfd   r4!, {r0-r2}                # Push r0-r2 to the shadow stack
    str     r4, [r3]                    # Store the new top of shadow stack
    # The rest of the context information is stored on the system mode stack
    cps     SYS_MODE                    # Switch to system mode
    ldr     r2, ulCriticalNestingConst  # Load critical nesting count
    ldr     r1, [r2]
    push    {r1}                        # Store critical nesting count on system mode stack
    ldr     r2, ulPortTaskHasFPUContextConst
    ldr     r3, [r2]
    cmp     r3, 0                       # Check if task uses floating point
    fmrxne  r1, fpscr                   # Get floating point status/control register, if used
    vpushne {d0-d15}                    # Save floating point registers, if used
    pushne  {r1}                        # Save fpscr, if used
    push    {r3}                        # Save whether task has floating point context
    ldr     r0, pxCurrentTCBConst       # Get location of stack pointer in TCB
    ldr     r1, [r0]
    str     sp, [r1]                    # Save new top of stack
.endm
```

Listing 4.4: FreeRTOS restore context routine with shadow stack

```
.macro portRESTORE_CONTEXT:
    # Restore context information from system mode stack
    ldr     r0, pxCurrentTCBConst       # Get top of stack
    ldr     r1, [r0]
    ldr     sp, [r1]                    # Load task stack pointer
    ldr     r0, ulPortTaskHasFPUContextConst
    pop     {r1}
    str     r1, [r0]
    cmp     r1, 0                       # Check if task uses floating point
    popne   {r0}                        # Restore floating point, if needed
    vpopne  {d0-d15}                    # Restore floating point registers, if needed
    vmsrne  fpscr, r0                   # Restore fpscr, if needed
    ldr     r0, ulCriticalNestingConst
    pop     {r1}
    str     r1, [r0]                    # Restore the critical section nesting depth.
    # Restore registers from the shadow stack
    ldr     r11, current_ss_const       # Load current shadow stack
    ldr     r11, [r11]
    ldr     r12, [r11]
    ldmfd   r12!, {r0-r10}              # Pop r0-r10 off shadow stack
    push    {r0, r1, r2}                # Save r0-r2 for use as temporary values
    mov     r0, r11                     # Copy shadow stack top and address
    mov     r1, r12                     # into new temp registers
    ldmfd   r1!, {r11-r12, lr}          # Restore the rest of the registers
    add     r2, r1, 8                   # Get shadow stack top address +8
    str     r2, [r0]                    # Store new shadow stack top
    mov     r12, r1                     # Copy previous shadow stack top to IP
    pop     {r0, r1, r2}                # Restore temporary registers
    dsb                                 # Data synchronization barrier
    isb                                 # Instruction synchronization barrier
    rfefd   r12                         # Return from exception using shadow stack
.endm                                   # as stack pointer
```

Chapter 5

Evaluation

As with any security technique, we expect a trade-off between performance and security, where better security generally results in higher performance overhead. Our CFI implementation follows this model. In this chapter, we first discuss the security of our instrumentation, showing that the instrumentation will enforce CFI policy, even in the presence of a powerful attacker. Then, in two parts, we discuss the performance overhead our CFI instrumentation introduces. Specifically, we evaluate the raw overhead for bare metal ARM CFI using the CoreMark embedded CPU benchmark, and we explore the potential adverse effects of latency in FreeRTOS after introducing CFI. Finally, we end with a discussion of the limitations of our system, and we highlight some of the directions that future work could take to improve this system.

5.1 Security Evaluation

Unfortunately, measuring the security benefits of a proposed CFI scheme is difficult to describe quantitatively. Measurements that have been used in the past include ROP gadget reduction and average indirect target reduction (AIR) [16]. While Carlini et al. have discussed how these measurements are misleading, they provide no alternative quantitative measurement of the effectiveness of a CFI system [12]. Additionally, these measurements try to quantify precision of CFG generation alforithms, which was not a focus for our work. Instead of AIR, Carlini et al. propose a Basic Exploitation Test (BET), where a minimal, representative program is written with a common vulnerability (such as a buffer overflow) to show that a CFI scheme prevents an attacker from achieving a specific goal [12].

5.1.1 Basic Exploitation Test (BET)

When evaluating the security of CFI schemes, the quantitative measurements used in the literature do not quantify security—they quantify the precision of the control-flow graph. Often the actual instrumentation is overlooked; how do we know that the instrumentation itself is not exploitable? To find out, we abstract away the CFG, and assume we have a labeling scheme that meets the assumptions for CFI. Then, we perform a Basic Exploitation Test (BET). Given a set of small, vulnerable programs that cover the scenarios that our instrumentation is supposed to protect, we examine the program from the attacker's point of view. The threat model states that the attacker can read or write arbitrary memory at any time. Realistically, the attacker can read or write memory only when exploiting some memory corruption vulnerability, which means the memory access will occur at the current privilege level. Given this realistic attacker model, we examine the instrumentation to ensure two invariants:

1. At every forward-edge indirect branch, the only branch that can occur is a branch to a target with the matching label.
2. Every function call returns to the instruction following the calling instruction.

Before examining the actual instrumentation, we look at the attacker's capabilities with memory protections in place. The ARM MPU ensures that memory is writeable or executable, but not both. Additionally, the MPU makes the shadow stack accessible only to privileged mode execution. So, the attacker can read or write to any address in RAM, unless access is privileged-only, in which case the attacker needs to be able to exploit a memory corruption vulnerability while the processor is in privileged mode. Interrupt handlers execute in privileged mode, which unfortunately means that a vulnerable interrupt handler could subvert this CFI scheme. But, since interrupt handlers in real-time systems are designed to be short and deterministic, we will assume that they cannot read or write arbitrary memory during their execution. While this assumption may not hold true for every case, we depend on it for the protection of user mode tasks.

Forward-edge Instrumentation Without Context Switching

Each forward-edge check has two parts: the source and destination instrumentation. The source instrumentation replaces the indirect branch and its preceding instruction with a direct branch to the correct location in the `.cfi` section. The general format of the indirect call instrumentation is shown in Listing 5.1. In the `.cfi` section, the instrumentation starts with the preceding instruction, fixing up addresses if necessary. From there, it saves the two registers required for the forward-edge label check, loads the label from the source and destination into those registers, and compares them. If the labels match, the two registers used for the check are restored and the indirect branch is taken. Otherwise, the execution enters an infinite loop.

By inspecting each of the instructions used here, we can see that the critical part of the CFI check is performed entirely in registers. The source label is hardcoded in a `mov` instruction, so that cannot be modified. The target label is either in program code, or it is in RAM. If it is in program code, it must be a legal target. If it is in RAM, the CFI check will allow the branch to be taken, but the MPU will prevent the processor from executing the code at the target. There are three possible outcomes from the label checking code: the branch is taken and execution continues, the branch is taken and the MPU prevents execution, or execution enters an infinite loop. In any of the three cases, the attacker cannot achieve arbitrary execution.

Listing 5.1: General format of indirect call instrumentation

```
    <previous_instruction>
    push {r0, r1}                          # Save temporary registers
    ldrh r0, [<target_reg>, <label_offset>] # Load label from target
    movw r1, <label>                       # Move expected label into register
    cmp r0, r1                             # Compare target vs expected
error:
    bne error                              # If label mismatch, infinite loop
    pop {r0, r1}                           # Restore temporary registers
    bx <target_reg>                        # Perform indirect branch
```

The destination is instrumented by replacing the first three instructions with a direct branch to another location in the `.cfi` section and a label. The direct branch simply points to a copy of the function prologue with the shadow stack code (which we will examine later in this section) and a branch back to the instruction following the label. As long as the label is stored in unmodifiable program code, there is no way to exploit the target instrumentation.

Forward-edge Instrumentation With Context Switching

The only time that the CFI checks can be tampered with is when context switching is possible. If the scheduler interrupts the CFI check and puts CFI-critical registers into memory, our threat model dictates that the attacker could use this as an opportunity to corrupt the CFI-critical registers. When context is restored, the corrupted values will be loaded into the registers, potentially allowing the attacker to bypass CFI. As stated in Section 4.4.3, we combat this by storing all saved context in the shadow stack.

Since the scheduler runs with interrupts disabled, the scheduler operations are essentially atomic operations. This means that there is no opportunity for an attacker to corrupt the context before it gets pushed to the protected shadow stack. The other possible attack vector would be to change the pointer to a task's shadow stack. Luckily, it is easy enough to simply protect the entire Task Control Block with another MPU region or just verify that the shadow stack pointer lies within the protected region of memory.

Backward-edge Instrumentation With Shadow Stack

Like each forward-edge check, each backward-edge check also has two parts: function prologue and function epilogue instrumentation. In ARM, we are not concerned with the backward-edge in leaf functions, that is, functions at the end of a call tree that do not call any other functions. Leaf functions do not push the return address to the stack; they keep the return address in LR and end the function with a `bx lr` instruction. In non-leaf functions, however, the compiler will generate a matching pair of `push {<reglist>, lr}` and `pop {<reglist>, pc}` instructions. These are the functions that we need to instrument.

Listing 5.2: Function prologue and epilogue instrumentation

```
# Prologue instrumentation
push {<reglist w/o lr>}     # original push without LR
<next_instruction>          # Instruction following the push, if overwritten by label
push {r0}                   # Push r0 since it will contain argument to ss_push
mov r0, lr                  # Copy link register value into r0
svc 0                       # Call ss_push from supervisor mode (system call)
pop {r0}                    # Restore r0 value
b.w <original_function + 6> # Branch back to original function, skipping over label

# Epilogue instrumentation
<previous_instruction>      # Perform pre-return operation that we overwrote
push {r0}                   # Push r0 so we can use it for return address
svc 1                       # Call ss_pop from supervisor mode (system call)
mov lr, r0                  # Copy return value from ss_pop to link register
pop {r0}                    # Restore r0
pop {<reglist w/o pc>}      # Perform original pop operation, without popping into pc
bx lr                       # New return instruction
```

The instrumentation for non-leaf functions has a single goal—protect the return address by saving LR in the shadow stack rather than the unprotected stack. The general form for the instrumentation is shown in Listing 5.2. The software interrupt (`svc`) instruction changes the processing mode from User to Supervisor mode, allowing us to modify the shadow stack. The shadow stack code runs with interrupts disabled, so the shadow stack operation itself is assumed to be atomic. All other calculation is performed in registers, so the attacker cannot modify LR without a context switch. If a context switch occurs, we push all of the context to the shadow stack anyways, so there is no time-of-check to time-of-use vulnerability.

5.2 Performance Impact

Like any CFI scheme, there is performance overhead associated with adding this instrumentation. To measure the performance impact, we look at three different measurements. First, we use an embedded system benchmark to determine the overhead associated with the bare metal instrumentation. Second, we look at the additional latency added to FreeRTOS context switching by adding the shadow stack. Finally, we analyze the resource requirements for this CFI scheme.

5.2.1 CPU Benchmarks

Since this CFI scheme is designed to work on embedded systems without a full operating system, we are unable to measure overhead using the industry standard SPEC CPU2006 benchmark. To measure the raw overhead associated with CFI checks on a realistic workload, we look to the CoreMark embedded system benchmark [27]. This easily portable application runs on a variety

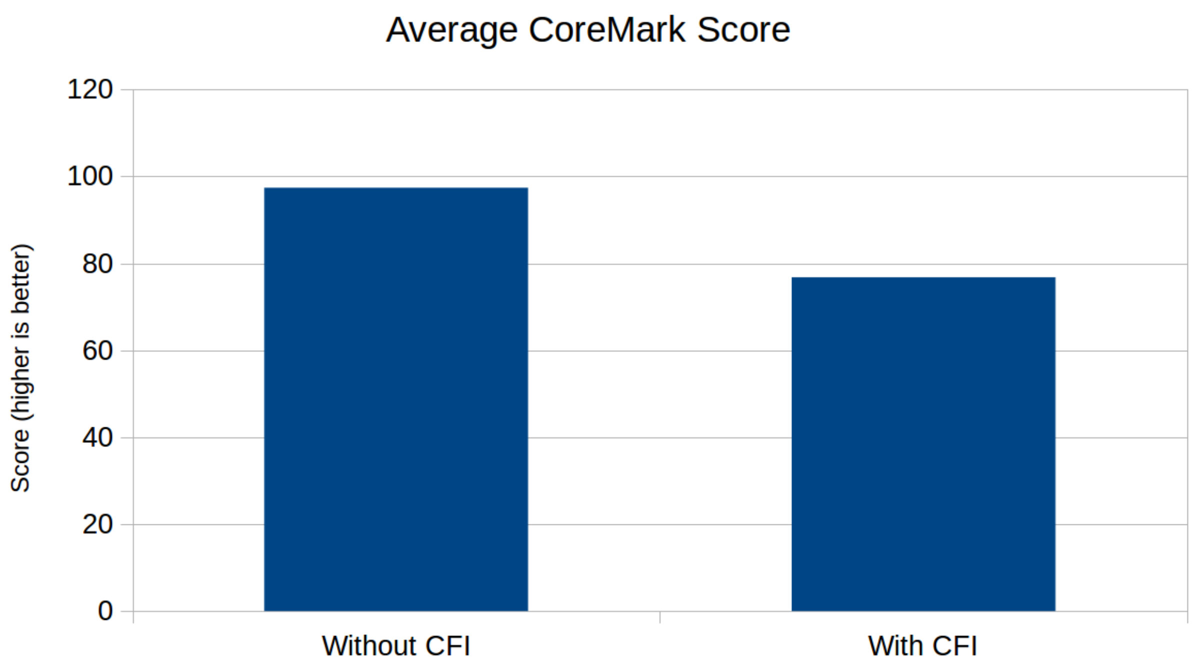

Figure 5.1: CoreMark results with and without CFI. Default settings, 1000 iterations

of embedded architectures without an operating system. It performs various common embedded tasks, like matrix manipulation, linked list manipulation, state machine operations, and cyclic redundancy check (CRC) calculation.

On our TI RM46L852 evaluation board, we measured the CoreMark score both with and without CFI. Without CFI, the recorded CoreMark score was 97.371, which is a reasonable score for that hardware running in Thumb mode. With CFI, we recorded a score of 76.767, a decrease of about 21% compared to the non-CFI score. Additionally, we recorded an approximately 30% increase in total execution time for the benchmark code. These results are shown graphically in Figure 5.1.

5.2.2 FreeRTOS Performance

When measuring the overhead due to CFI on FreeRTOS, using the embedded benchmark is not enough. While the benchmark may give us a general amount of overhead for single-threaded processing, we are more concerned with the multi-threaded performance. In particular, we need to know CFI's effect on latency. If we assume that we have a schedulable system where all deadlines are met without CFI, how much latency must the tasks be able to tolerate for CFI not to break the real-time guarantees?

First, we identify the sources of latency. CFI label checks and shadow stack operations increase the worst-case runtime of a task. The new worst-case runtime scales linearly with the number of indirect branches and non-leaf functions in the task. Shadow stack operations and the scheduler operate with interrupts disabled. Any time that interrupts are disabled, the latency for handling

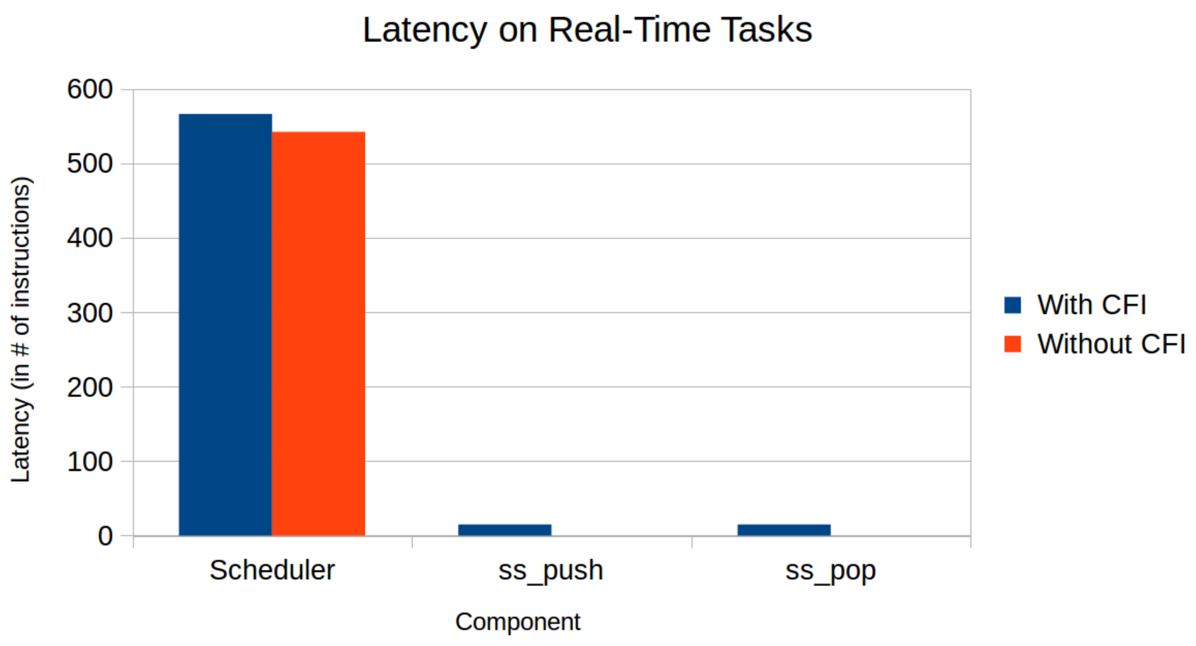

Figure 5.2: Latency (in number of instructions) of CFI critical sections in FreeRTOS with two running tasks

interrupts and context switching is increased. If adding forward-edge CFI checks to a system does not break the schedulability of the system, the only other sources of latency relate to the shadow stack. Particularly, the shadow stack instrumentation and the new context switching operation add latency of the system. The additional latency added by shadow stack operations is shown in Figure 5.2. The latency added by `ss_push` and `ss_pop` is exactly 15 instructions. The exact runtime for this depends on caching and instruction execution time, but this can vary from processor to processor. In the scheduler, we add 20 instructions, which is not an unreasonable amount of additional processing compared to the 567 instructions in the worst-case execution of the scheduler with two tasks in the system. In general, the worst-case execution of the scheduler without CFI is $(167+188n)$ instructions, where n is the number of tasks in the system, excluding the idle task.

5.2.3 Additional Resource Use

Since this CFI scheme is designed to run on embedded systems with constrained resources, additional resource use is important to measure. Specifically, we report code storage size requirements, RAM usage requirements, and hardware peripheral usage. In cases where our scheme may cause an existing binary to exceed the limits provided by the hardware, more expensive hardware may be necessary.

In its current state, our system requires an additional 10 bytes of storage per indirect branch and 12 bytes per non-leaf function prologue and epilogue. While we cannot generalize the number of non-leaf functions and indirect branches in any given program, the CoreMark benchmark (a 10

KB binary) required 964 bytes of instrumentation code. This is just under a 10% increase in binary size.

As for RAM usage, it depends on the system being instrumented. On bare metal systems, a single shadow stack is required, which on our evaluation system, we used a shadow stack size of 256 bytes plus 12 bytes for the shadow stack structure—a total of 268 additional bytes of RAM for the shadow stack. In FreeRTOS, however, we used a larger shadow stack, since context information is stored on the stack. So we required 528 bytes per task, which encompassed a 512 bytes shadow stack, 12 byte shadow stack structure, and 4 byte pointer to the shadow stack stored in each Task Control Block.

Finally, our implementation depends on some additional resources. We need at least two MPU regions to prevent execution from RAM and to protect the shadow stack. On our hardware, a maximum of 12 regions could be configured, so our utilization was minimal. Also, we require two supervisor calls out of a possible 256 available in Thumb mode.

5.3 Discussion

With these results, we can see that while this system appears to provide many security benefits at the expense of moderate performance and low resource overhead, there are still some limitations that should be addressed. The major limitation, inherent to CFI itself, is CFG generation. A limitation specific to our work is our dependence on privileged execution, where we assume that all privileged execution is secure.

5.3.1 Limitations

All security measures have limitations; this CFI scheme is no different. Some of the limitations are inherent to CFI, while others are specific to this implementation. Limitations inherent to CFI include CFG precision and the maximum security benefits of the static forward-edge checks. Our implementation has some additional limitations, especially when concerned with processing modes in ARM.

It is well-documented that CFI schemes can only be as secure as the CFG is precise [12,16,21,24]. There are two sources of imprecision: the difficulty of sound and complete CFG generation and the labeling scheme extracted from the CFG. Sound and complete CFG generation is believed to be undecidable [12,21], so to preserve functionality of programs, CFI schemes often use a more permissive CFG, potentially allowing some unintended indirect branch targets. On top of the inherent imprecision, the labeling scheme itself often introduces more imprecision for performance reasons. To reduce overhead, most CFI schemes, including this one, assigns one label to an indirect jump and its targets. Unfortunately, this means that if two sources share a target, they need to use the same label. For example, if Source 1 can target A or B, and Source 2 can target B or C, the labeling scheme would allow Source 1 to target C, even though that edge was clearly not in the CFG. This imprecision can allow an attacker to achieve Turing-complete computation in the presence of certain instruction sequences [12,21]. We do not attempt to remove this imprecision, but we do believe that due to smaller executable size and differences in programming techniques for embedded systems, this imprecision is less likely to be exploitable.

One of the major limitations in our work is the dependence on ARM processing modes. Since there are only two available privilege levels on ARM Cortex-R, we make a potentially unrealistic assumption about privileged mode execution. We assume that all privileged execution cannot be exploited. While good embedded software design will keep privileged sections short, deterministic, and easily verifiable, there is no guarantee that all existing applications were built this way. If there is a vulnerable privileged section of code, the attacker can modify the shadow stack, allowing corruption of return addresses or CFI context, essentially rendering CFI useless. One way to prevent this would be to prohibit all writes to the shadow stack except during privileged shadow stack operations. To do this, we have to use ARM MPU coprocessor instructions to change the shadow stack region permissions at the beginning and end of every shadow stack operation. This would limit the shadow stack corruption attack surface, requiring the attacker to be able to execute MPU instructions to disable the MPU before he can corrupt the shadow stack. However, this would result in significant overhead, nearly doubling the amount of instructions needed for each shadow stack operation. The new ARMv8-R architecture may have a better way to handle this vulnerability with its bare metal hypervisor mode, but these processors are not widely available yet, and existing systems with ARMv7-R processors will likely not be upgraded.

5.3.2 Future Work

Apart from addressing the limitations mentioned previously, there are some other directions that future work could take. One of the major areas that this work and other CFI work could benefit from is a better analysis of the security guarantees provided by CFI. Another possible topic for future work is a more in-depth analysis of the effects of security measures like CFI on real-time systems. Finally, the prototype system created for this work can likely be optimized for performance.

Previous work has suggested AIR [16], ROP gadget reduction [16], and the Basic Exploitation Test (BET) [12] as ways to evaluate the security of CFI-based systems. Unfortunately, AIR and ROP gadget reduction are misleading metrics, especially in the context of embedded systems where it may only take one malicious indirect jump to cause catastrophic failure. The BET, while a good sanity check for CFI instrumentation, is prone to human error and would be better if performed by some formal analysis. Future work could focus on modeling CFI systems and proving their correctness with formal methods, rather than human analysis.

In our evaluation, we briefly discussed CFI's effect on the latency of real-time tasks. While this analysis is helpful, it is limited in scope without real-life applications to test against. Unfortunately, very few embedded applications are open source and freely available, so it is difficult to make generalizations about our scheme's effect on schedulability in real systems. Future work could focus on modeling FreeRTOS with and without CFI, such that a more formal schedulability analysis would be possible with easily tunable parameters.

Our instrumentation required hand written ARM assembly code. While we focused on making the instrumentation as fast as possible given the binary patching approach, there are likely some optimizations that could be made within the scheduler, CFI checks, and shadow stack operations. One of the most useful optimizations would be reserving registers for use only by CFI, preventing the need to save registers needed for labels and the shadow stack. This approach may not be possible with binary patching, but it is possible with a compiler-based CFI scheme.

Finally, the new ARMv8-R architecture adds a bare-metal hypervisor mode with a new privilege level and faster MPU operations. Future ARM Cortex-R CFI implementations could take advantage of this functionality to perform faster CFI checks with better guarantees on security, since the hypervisor privilege level could be used exclusively for shadow stack operations, meaning we would no longer have to assume that all privileged execution is secure.

Chapter 6

Conclusion

In this thesis, we described a reference implementation for control-flow integrity on ARM systems. In this implementation, we show how we can use hardware features of embedded ARM processors to meet the various requirements for CFI, specifically the write-exclusive-or-execute assumption and the protected shadow stack. To instrument ARM binaries, we created a binary patching tool that can be used to retrofit CFI to precompiled binaries without needing source code. Finally, we extended this implementation to the FreeRTOS real-time operating system, which introduces multi-threading without process isolation. To prevent time-of-check to time-of-use vulnerabilities, we modified the FreeRTOS operating system to isolate saved context from running threads. With these modifications in place, we were able to apply the CFI technique to embedded applications using real-time operating systems.

To the best of our knowledge, our instrumentation itself is secure against exploitation as long as there is no privileged-mode code in the system that have memory corruption vulnerabilities. The FreeRTOS shadow stack implementation provides enough process isolation for CFI assumptions to met, and could even be used without CFI just to protect saved context from malicious corruption.

We found that our CFI scheme had moderate overhead on bare metal ARM benchmarks. In the CoreMark embedded benchmark, we saw a 21% decrease in score, which corresponded to a 30% increase in execution time. Although we saw a moderate increase in execution time, we had a minimal increase in program size, requiring just under 10% more code storage than the unprotected binary. For FreeRTOS, we measured the new worst-case latency introduced by CFI. We found that despite adding 20 instructions to the scheduler, the worst-case execution path in the scheduler was already over 500 instructions long in a system running only two tasks.

While this work makes a significant step towards hardening real-time embedded systems, there are many directions that future work could take. Beyond just optimizing our existing work, we would like to have a way to use formal methods to prove correctness of CFI instrumentation, which means we need a way to model CFI schemes and hardware architectures to show that the CFI scheme maintains the expected invariants. Additionally, in the embedded systems space, we would like to be able to model CFI in a way that allows us to perform a more formal schedulability analysis for real-time tasks. Finally, the new ARMv8-R architecture provides promising features that could provide stronger guarantees for this system as well as other embedded system hardening techniques.

Bibliography

[1] *RM46x 16/32-bit RISC Flash Microcontroller Technical Reference Manual.* Texas Instruments, Inc, 1 ed., 2015.

[2] "Processors—ARM." `https://www.arm.com/products/processors`, 2017. Accessed: 2017-03-15.

[3] *Architecture reference manual. ARMv7-A and ARMv7-R edition.* ARM Ltd.

[4] A. Greenberg, "Hackers remotely kill a jeep on the highway with me in it," *Wired, 21 July*, 2015.

[5] L. Hay Newman, "Medical devices are the next security nightmare," 2017.

[6] K. Zetter, "Medical devices that are vulnerable to life-threatening hacks," 2015.

[7] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[8] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552–561, ACM, 2007.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 340–353, ACM, 2005.

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[11] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, pp. 31–40, ACM, 2010.

[12] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 161–176, 2015.

[13] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: Control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 743–754, ACM, 2016.

[14] L. Pike, P. Hickey, T. Elliott, E. Mertens, and A. Tomb, "Trackos: A security-aware real-time operating system," in *International Conference on Runtime Verification*, pp. 302–317, Springer, 2016.

[15] R. Earnshaw, "Procedure call standard for the arm architecture," *ARM Limited, October*, 2003.

[16] M. Zhang and R. Sekar, "Control flow integrity for cots binaries.," in *Usenix Security*, vol. 13, 2013.

[17] B. Niu and G. Tan, "Modular control-flow integrity," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[18] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Mocfi: A framework to mitigate control-flow attacks on smartphones.," in *NDSS*, 2012.

[19] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy*, pp. 292–307, IEEE, 2014.

[20] J. Pewny and T. Holz, "Control-flow restrictor: Compiler-based cfi for ios," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 309–318, ACM, 2013.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 901–913, ACM, 2015.

[22] T. Zanussi, "microYocto and the internet of tiny." Embedded Linux Conference, 2015.

[23] "FreeRTOS FAQ relating to memory management and usage." `http://www.freertos.org/FAQMem.html`, 2017. Accessed: 2017-03-28.

[24] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 401–416, 2014.

[25] C. Lattner, A. Lenharth, and V. Adve, "Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World," in *Proceedings of the 2007 ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI'07), (San Diego, California), June 2007.

[26] B. Niu, *Practical Control-Flow Integrity.* PhD thesis, 2016.

[27] S. Gal-On and M. Levy, "Exploring coremark—a benchmark maximizing simplicity and efficacy," *The Embedded Microprocessor Benchmark Consortium*, 2012.