Control-Flow Integrity for Real-Time Embedded Systems

A Major Qualifying Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

by

Bradford Bonanno

and

Kodey Converse

May 2018

APPROVED:

Professor Robert J. Walls, Major Project Advisor

Abstract

While security protections continue to be developed for general-purpose computers, real-time computing has remained unprotected against control-flow hijacking attacks. Existing solutions rely on hardware unavailable to embedded systems due to the cost, or impose excessive overhead, leaving real-time applications unable to operate within their time constraints. We propose RECFISH++, a Control-Flow Integrity implementation focused on protecting real-time embedded systems. By modifying LLVM and FreeRTOS, a popular compiler back-end and real-time operating system, we provide an end-to-end solution for protecting any real-time application on the ARM Cortex-M microprocessor against control-flow hijacking attacks.

Contents

Abstract i

1 Introduction **4**

2 Background and Related Work **6**

2.1 Control-Flow Hijacking 6

2.2 Control-Flow Integrity 7

2.3 Real-Time Embedded Systems 9

3 Design and Implementation **11**

3.1 Compile-time CFI Instrumentation 11

3.2 Forward-edge Protection 13

3.2.1 Model Enforcement Pass 13

3.2.2 Model Generation Pass 14

3.3 Backward-edge Protection 15

3.3.1 Memory Protection Unit (MPU) Permissions Setup 15

3.3.2 A Modified Interrupt Handler 16

3.3.3 The Shadow Stack 16

3.3.4 Task Initialization 17

3.3.5 Secure Context Switching 18

3.3.6 Secure-Spill Lowering Pass 18

3.4 Fine-Grained Protection 19

3.4.1 Instrumentation by Equivalence Class 20

3.4.2 Generating Globally Unique Labels 22

3.5 Unnecessary Instrumentation Analysis 24

3.5.1 Direct External Input 24

3.5.2 Sensitive Inter-task Communications 24

3.6 Limitations 26

4 Evaluation **27**

4.1 Performance Evaluation 27

4.2 Protection Evaluation 29

5 Conclusion **33**

5.1 Future Work 33

List of Figures

2.1 Trivial Control-Flow Graph 8
2.2 Label-based Control-Flow Integrity 9

3.1 System Component Overview 12
3.2 Modified Memory Regions 16
3.3 Indirect Branch Scenarios 20
3.4 Equivalence Class Scenarios 21

List of Listings

1 Indirect Branch Protection 14
2 Supervisor Exception Dispatching 17
3 Unprotected Register Spill 19
4 Secure Register Spill 19
5 Equivalence Class Instrumentation Algorithm 22
6 Type Extraction Within LLVM 25
7 Forward-edge evaluation 30
8 Backward-edge evaluation 31

Chapter 1

Introduction

As embedded systems are introduced into safety-critical roles such as flight-control systems and medical equipment, preventing their exploitation is paramount. These applications often use low-cost hardware with the ability to reliably meet tight real-time deadlines, rendering existing solutions for general purpose systems useless. Faulty devices released by uninformed manufacturers have the potential to cause physical damage due to the actions of an intelligent attacker [5,20].

Though attacks take many shapes, the most dangerous allow for arbitrary code execution on a target system. This class of exploit often leverages some form of control-flow hijacking. This technique redirects the execution of a program to instructions provided by the attacker or to pre-existing instruction sequences deemed useful [12,17]. Though full memory safety is the favorable defense for control-flow hijacking, high-performance overhead makes it impossible for real-time applications [10,11].

Control-Flow Integrity (CFI) is a subclass of memory safety which aims to lower overhead by protecting only a subset of memory, specifically pointers to executable addresses containing program code [3]. At a high level, CFI enforces a predetermined model of program execution at runtime. Should behavior stray from this model, an error is thrown. This enforcement is accomplished by monitoring control-flow transitions and detecting those which are not included in the execution model.

Few CFI implementations confront any of the challenges unique to real-time embedded systems. First and foremost, a majority of existing implementations exclusively target the x86 architecture used in general-purpose computers as opposed to the ARM instruction set commonplace in embedded computers. Secondly, even systems supporting ARM are not often designed with the strict scheduling requirements of real-time applications in mind [13]. Finally, the hardware used for real-time systems is often limited, meaning that the responsibility of task isolation often falls onto the real-time operating system (RTOS) and is often absent [1].

We propose a CFI scheme specifically targeting real-time embedded systems and their associated challenges. Within, we detail modifications to LLVM, a popular compiler back-end, to insert runtime checks into a program at compile-time that enforce its execution model. In addition, we detail modifications to FreeRTOS, a popular real-time operating system, to provide task isolation and as-necessary memory safety. Finally, we propose a novel method of securely removing runtime enforcement of CFI from safe tasks to increase the number of applications that will meet their real-time constraints with security guarantees.

The remainder of this work is structured as follows. Chapter 2 provides background information about control-flow hijacking, CFI, and real-time embedded systems necessary to understand our design decisions. Chapter 3 outlines the design and implementation of our CFI system, referred to as RECFISH++. Chapter 4 discusses the performance and memory overhead observed on a protected example application. Finally, Chapter 5 summarizes and concludes our work.

Chapter 2

Background and Related Work

Before we present our work, we will describe control-flow hijacking and systems that have been created to protect against this attack. We'll consider the popular real-time operating system FreeRTOS and the details of it. Lastly, we'll discuss the constraints of the hardware we will be using.

2.1 Control-Flow Hijacking

Control-flow hijacking is the most common major attack used to exploit programs [18]. This hijacking takes control of a program's *control-flow*, the order of code execution, to execute existing or injected code. Hijacking can often capture full control over the program. For example, hijacking can be used to make an FTP server transfer a password file or a web server execute arbitrary commands in a remote shell [19].

There are two methods used to subvert control-flow, both requiring memory corruption [18]. The first method exploits *indirect branches*, branches whose targets are dynamically determined. Where direct calls use a statically determined target, indirect branches use a *code pointer*, a pointer to executable code (e.g. function pointer). Direct calls don't need to be protected because their targets are statically encoded in the code and cannot be changed. Indirect branches are susceptible because their target may be modified by an attack if stored in writable memory. Attacks aim to redirect the control-flow by changing these pointers using memory exploits, such as buffer overflows. We consider this attack one to the *forward-edge*, as it targets the location of function calls.

Another common method used to subvert control-flow is manipulation of return addresses. The transition that occurs when a function returns closely resembles indirect calls, where its target is determined by an address stored on the stack. However, function returns are often considered separately from indirect branches when designing defenses due to the difference in how they are used and where they are located in memory. We consider this attack one to the *backward-edge*, as it targets functions transitioning to their return address.

There have been many methods for protecting against these two targets of attack, each of which attempts to balance performance with protection. Generally, these methods attempt to protect against a powerful adversary able to modify any data in writable memory. While this may seem extreme, it aims to accommodate new techniques for memory corruption that will inevitably be found. Additionally, implementations for embedded systems use a slightly more strict threat model where writable memory is also executable.[1]

One way to protect both types of attacks is to provide *full-memory safety* by protecting all of program memory. Full-memory safety is the only protection that can prevent all attacks; but, achieving it has proven very difficult and costly [18]. An implementation which tries to approximate this protection is AddressSanitizer [16]. With research showing that most vulnerabilities occur when arrays overflow, AddressSanitizer adds unallocated padding bytes between all arrays and then uses a bitmask system to check that before any memory write occurs, it is within allocated memory. It can be used to protect any general-purpose program during a modified compilation through *LLVM*, a widely-used compiler back-end. No similar approach has been taken for embedded-systems, presumably due to the high overhead needed for its strong guarantees.

While protecting all memory is the only way to guarantee a program's integrity, there are more practical measures that can be used which still provide strong measures of defense against control-flow hijacking.

The first of these measures is Code-Pointer Integrity (CPI), a *partial-memory safety* technique which prevents control-flow subversion by protecting code pointers [18]. Levee adds instrumentation to a source program during compilation to ensure that code pointers are always stored in a protected memory region [6]. Levee has been embedded in the commonly used Clang compiler, a C compiler front-end for LLVM, and is a widely-used solution for general-purpose systems. However, it is not usable by embedded architectures because it requires the advanced memory features of the Memory Management Unit (MMU).[2]

Another measure of defense against hijacking attacks is a *memory-randomization technique* called address space layout randomization (ASLR) [18]. ASLR doesn't prevent an attacker from subverting control but instead makes it difficult for it to be used effectively. It randomizes the location of the stack and the code section in a large address space to obfuscate the location of useful code segments (e.g. gadgets) [18] especially used in Return-Oriented Programming (ROP). This is not possible on embedded systems due to the requirement of virtual memory, again requiring the MMU.

It isn't possible to implement any of these widely-used protections on embedded systems due to the overhead and hardware capabilities required. We need some method for protecting program integrity without all of the additional runtime overhead.

2.2 Control-Flow Integrity

Control-Flow Integrity (CFI) is a technique that ensures a program only branches to valid locations at runtime, therefore ensuring that a program's control-flow has not been corrupted. CFI aims to catch the program when it behaves incorrectly, such as

[1]Due to the default memory permissions on these systems.
[2]The MMU is not available on most embedded systems.

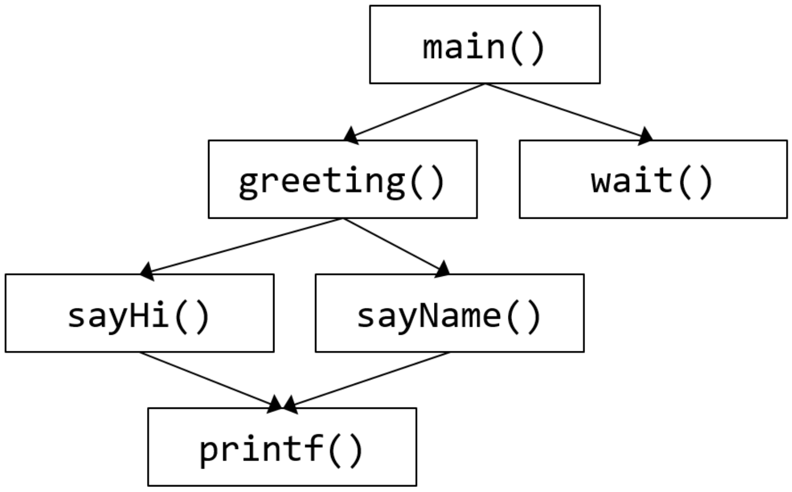

Figure 2.1: The execution model, or *CFG*, for a trivial program.

branching to the middle of an instruction or calling a function which wouldn't otherwise be possible, to verify its integrity.

The three steps that a CFI implementation must take to protect a program are as follows.

1. Understand the normal execution behavior of the program
2. Enforce this behavior using runtime checks
3. Ensure that these checks cannot be removed

One technique for CFI uses unique labels and checks placed in a pre-compiled program, as demonstrated by Abadi [3]. This approach first generates a *control-flow graph (CFG)*, a model outlining the normal execution behavior of a program, using a complex static analysis, exemplified by Figure 2.1. The labels and checks are then added to each indirect call in the non-writable code section to prevent corruption by an attacker, shown in Figure 2.2. At runtime, these added checks compare labels to ensure that each transition is valid according to the CFG.

One of the design decisions that must be made for a CFI implementation is its precision. A precise *fine-grained* solution encodes information about all possible transitions, requiring a label for each of those transitions and multiple checks. A less precise *course-grained* approach will sacrifice precision for reduced overhead, often only using one label to try to generalize multiple transitions. Where only transitions outlined in the CFG are allowed in fine-grained protection, coarse-grained protections will often not be able to detect every violation of the CFG and instead will greatly reduce the number of vulnerable transitions at a lower performance cost.

Another decision required is the method for protecting against return-address corruption. The label-based approach used for indirect branches isn't as powerful for returns,

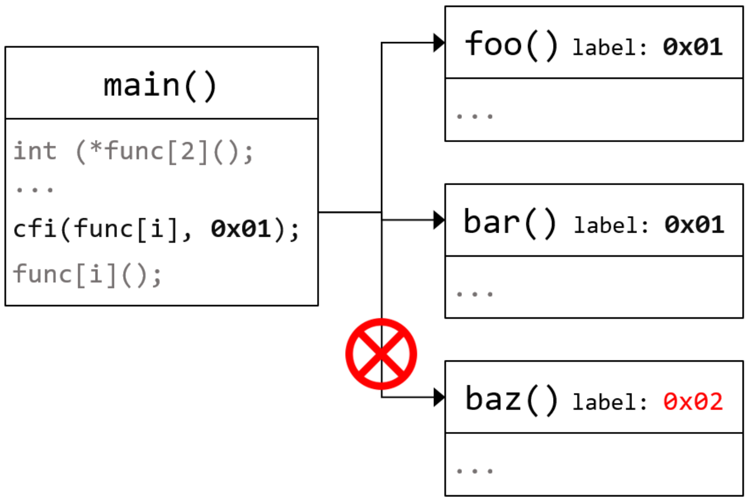

Figure 2.2: A psudeo CFI check using labels on a trivial indirect branch.

and while it can be used, its imprecision allows more possibilities for exploitation [3]. A more powerful approach is to protect return addresses, much like CPI protects code pointers, using a secured area of memory called the *shadow stack*. This approach can be used to very precisely protect backward-edges at a higher cost [18]. Epoxy demonstrates that protecting a region of memory for a shadow stack is possible using only the Memory Protection Unit (MPU) available on most ARM processors [4]. Specifically, it uses the two privilege modes supported by the MPU on ARM devices. Epoxy also uses custom LLVM compiler passes, one to add static code to configure the MPU and another to add instrumentation that changes the privilege mode when sensitive memory is accessed.

2.3 Real-Time Embedded Systems

Real-time embedded systems are computers dedicated to specific time-sensitive programs, such as the gyroscope on a plane's frame or the imaging device on an MRI. These computers are typically far less powerful than any general-purpose computer and are specialized to their purpose. As such, time and space efficiency of programs is a major concern.

The time-sensitive nature of the programs carried out on these systems must be specified through a set of real-time constraints, commonly referred to as *deadlines*. Deadlines typically outline explicit error-handling behavior that would run if a section of a program does not finish executing within a specific time frame.

A program's ability to meet its real-time constraints is referred to as its *schedulability*.

In a real-time context, any overhead added by external tools to guarantee security must not introduce so much overhead so that the program is no longer schedulable.

Real-time embedded systems require direct access to data as it comes in (without the buffer delays common in general-purpose operating systems) to meet deadlines. There are real-time operating systems capable of scheduling multiple *tasks* on one system, developed with these deadlines in mind. We'll focus on a widely used event-driven real-time operating system called FreeRTOS [1].

In FreeRTOS, programmer-defined tasks run pseudo-parallel, scheduled based on their individual priority levels. Each task is typically designed to respond to some type of input, whether from external sensors or network connections. Tasks can communicate with one another via time-sensitive or buffer-enabled functions provided by the operating system. Tasks are not typically isolated from each other and share the same address space due to the lack of hardware support for virtual memory.

We'll focus on the ARM Cortex-M processor, often used in these real-time embedded applications. There are three constraints that are important to consider when work working with this series of processor. First, performance overhead must be low enough to maintain the real-time schedulability of tasks. Second, the overhead of any instrumentation must be low enough that a program can still operate normally in the small memory space. Third, any region-specific memory permissions must be configured manually using the MPU due to the lack of a MMU.

Chapter 3

Design and Implementation

RECFISH is a Control-Flow Integrity (CFI) proof of concept for embedded systems which instruments pre-compiled programs. It uses performance-heavy tricks to overcome the limitations of working with a pre-built program binary. While able to successfully enforce protections, its focus was never on real-world usefulness. We want to take the ideas that are demonstrated by RECFISH and implement them in a way that will allow us to reduce its performance cost, improve its protections and make it easier to use; we'll call this next generation implementation *RECFISH++*. An overview of the design of our final system can be found in Figure 3.1.

3.1 Compile-time CFI Instrumentation

The design for the original RECFISH system targeted pre-built binaries, meaning that it could be applied to a program without its source code. There are a few limitations to this approach. Adding to or removing bytes from a binary is simple in concept, but difficult to implement due to hard-coded addresses being shifted by additional instructions [7]. This complexity results in less flexibility, making it difficult to improve the performance or security of RECFISH.

To improve RECFISH, we consider moving its implementation to a modified compiler pass in LLVM. Being able to embed functionality into a program during compilation could give us a few advantages over the original approach. First and foremost, with the flexibility provided by a compile-time implementation, we can improve the performance by removing the costly *trampolines*, branches to appended code sections, without needing to rework hard-coded addresses. Along with this, we think the high-level information provided by a compiler would allow RECFISH++ room for future optimizations. Finally, RECFISH made no attempt to tackle the generation of a control-flow graph (CFG), a problem that may be solvable at compile-time.

We identify a few major challenges that must be considered to provide the forward-edge and backward-edge protections of RECFISH at compile-time. To provide more concrete descriptions of our challenges, we'll mention that we decided to use the LLVM compiler infrastructure to implement RECFISH++. While both LLVM and GCC would

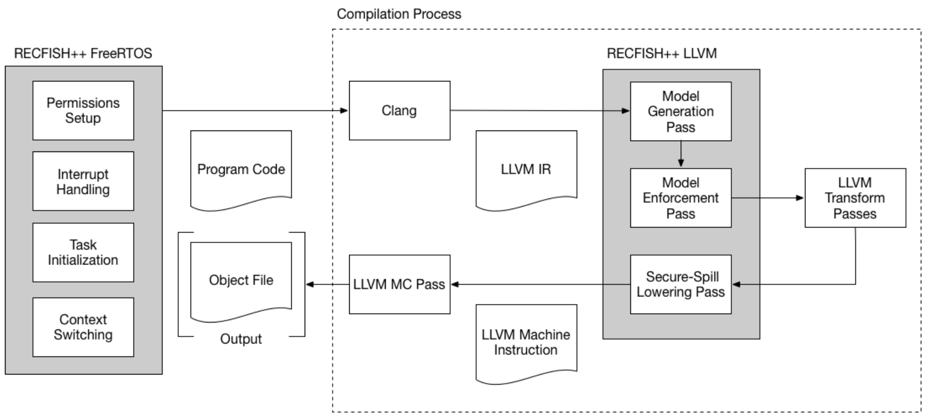

Figure 3.1: An overview of all RECFISH++ system components.

provide the necessary features for RECFISH++, we chose to use LLVM. LLVM's plugin structure is more mature than the equivalent in GCC; furthermore, the API appears stable and better documented. Additionally, LLVM with the Clang front-end is able to cross-compile C code using an existing toolchain, making it possible to compile to our target architecture. A final benefit lies within the research community; many existing memory safety projects have implementations within LLVM compiler passes that can be used for reference throughout our own. The basic challenges are similar when considering either GCC or LLVM, but we believe the details are easier to explain when only considering the one that we chose. Accordingly, we'll discuss our challenges with respect to LLVM.

One of the first and most evident challenges that must be faced when moving from modification of a pre-built program to modification at compile-time is the complete change in how the program is read and how instrumentation is added. Instead of using a disassembler to read the program and change the code directly, we would need to conform to the specifics of LLVM to perform these two tasks. One accepted method for extending LLVM is by adding a custom compiler pass to LLVM's Intermediate Representation (IR) stage of compilation. During this stage RECFISH++ would have access to an IR representation of a target program that could be traversed and modified [15]. This IR is accessible through C++ data structures and is both a target and language agnostic representation of a program. Learning how to work with this IR and then replicating the functionality of RECFISH is the first challenge to overcome.

While modification of a program that operates on LLVM's IR would be the most accepted approach, one challenge we face is its target-agnostic quality. RECFISH had access to the exact program, giving it the ability to search for particular ARM instructions and embed functionality that would not be modified again before execution. Using IR provides the benefit of being able to traverse the program more easily, but the instructions that it exposes are not ARM instructions and often don't even map one-to-one with ARM instructions. In order to implement RECFISH++ at compile-time and embed labels in the program, we need to understand how the code in the IR pass will be modified by optimizations and how it will be translated to ARM specific code. With this understanding, we will be able to decide where to place the different components (forward-edge and backward-edge) of RECFISH++ in the compilation process to modify a target program as needed.

Another challenge is compiling to the specialized ARM processor, particularly the Cortex M4. This consideration was a significant one in the implementation of RECFISH, and a major reason for why it focused on pre-built programs. The ARM processor is often used for proprietary applications, meaning that it is common for companies to use a proprietary toolchain for compilation. Without access, we will need to find an open-sourced toolchain in order to test demo applications.

One feature that RECFISH didn't have was the ability to instrument a program without any additional information about it, specifically its CFG. We don't think that a compile-time instrumentation should require a pre-generated CFG. Instead, we will aim to generate a full CFG during compile-time, and ensure that this CFG properly discovers indirect branches.

We were able to overcome these challenges in order to move the functionality of RECFISH to compile-time, giving RECFISH++ additional flexibility and better performance. We used a combination of libraries, documentation and trial and error to arrive at a fully-featured solution. We'll dive into both forward-edge and backward-edge protection to explain how we faced the above challenges in both of these areas.

3.2 Forward-edge Protection

The first protection that RECFISH++ replicates at compile-time is protection against forward-edge hijacking. Indirect branches within a program are transitions whose target is determined dynamically while it is running. In a C++ application, a common application of indirect branches is `virtual` functions, where the actual function that is run is determined by the object that it is being called on. While this programming construct is useful, it can be used to hijack an application by changing the address of the dynamic target.

3.2.1 Model Enforcement Pass

RECFISH++ is able to accomplish the forward-edge protections demonstrated by RECFISH at compile-time. We have implemented the protection which ensures that all forward-edge branches, i.e. indirect calls and jumps, may only go to possible indirect targets (i.e. the beginning of functions). Much like the original RECFISH implementation, we have added labels to a program that are placed at the source and destination of every indirect branch to allow for runtime checks of their validity. We place this functionality in a custom IR pass that is run during compilation.

With access to a target program's IR, the RECFISH++ forward edge pass is able to traverse and modify the program without restriction. The pass inspects each branch instruction in the program to determine whether it is direct or indirect. When an indirect branch is found, called a *call site* in LLVM IR, we place a label just before the branch and at the beginning of every function that the call site would be able to target in normal

```
func:
  ...
  bic.w lr, r0, 1 ; mask the last bit of the address
  ldr.w lr, [lr, -4] ; load the label of the target function
  movw r1, 0xdefd ; load half of the label
  movt r1, 0xe7f2 ; load second half of the label
  cmp lr, r1 ; compare the labels
  str r0, [sp, 12]
  beq.n normal ; continue to indirect call, normal case
  b.n violation ; handle as a violation
violation:
  mov.w r0, -1 ; pass -1 as argument to exit()
  bl exit ; exit the program
normal:
  ldr r0, [sp, 12]
  blx r0 ; make the indirect call
  ...


.word 0xe7f2defd ; matching label on target
foo: ; indirectly targeted function
  ...
```

Listing 1: The assembly added to an indirect branch which validates the label of the dynamic target.

execution, as determined by the control flow graph. The function labels are placed in the bytes directly preceding each function using what LLVM calls the function's *prefix*, a storage primarily used for embedding metadata during compilation, as shown on `foo()` in listing 1. We place the label in a comparison branch instruction right before the call site. This comparison ensures that the runtime target of the call is preceded by the correct label, aborting the program if a mismatch occurs. With these checks on every indirect branch in place, we have verified that all indirect control-flow changes will be valid according to a fine-grained CFG.

3.2.2 Model Generation Pass

One of the difficulties in this forward-edge protection is generating the control-flow graph. Without one, we would not be able to determine where an indirect branch may target in normal execution. LLVM's plugin infrastructure does provide a control-flow graph data structure that is available while traversing the target program. Unfortunately, this graph only includes direct transitions and does not perform the analysis that is needed to determine valid indirect transitions. In order to capture this additional information, RECFISH++ uses LLVM's DSA (data-structure analysis) library, which is part of the poolalloc project [8]. This project is no longer supported in the most recent versions of LLVM, and so RECFISH++ uses LLVM 3.8, the last version compatible with the DSA library. While this is not optimal, we have decided that it was the best option to avoid diving into the complexities of data-structure analysis. Using this library, we are able to fill in the gaps left by the default CFG and properly determine valid indirect branch targets.

3.3 Backward-edge Protection

Return addresses are used during normal program execution to jump from a returning function to its caller. These transitions closely resemble the indirect branches considered in forward-edge protections, as the target is dynamically determined and is stored in unprotected memory. A label-based approach is effective but coarser. As such, we use the privilege-based shadow stack utilized by RECFISH, an approach originally outlined by Abadi et. al [3] with additional consideration for the address-space sharing within FreeRTOS.

Moving this backward-edge instrumentation to compile-time is not as straightforward as the forward-edge due to the hardware-specific nature of configuring the Memory Protection Unit (MPU) and using the supervisor mode. Supervisor Call instructions are not available in LLVM's IR, as IR is target agnostic. Determining when the return address will need to be saved requires hardware-specific knowledge. Custom hardware-specific context switching is required in order to securely store and restore context between pseudo-parallel tasks running on a single core processor. We'll break up the steps for moving the shadow stack to compile-time into six steps: configuring the MPU for proper protection, adding supervisor interrupt handlers to properly dispatch shadow stack operations (push and pop), implementing the shadow stack, altering task initialization to support the shadow stack, secure context switching, and instrumenting functions that spill a return address.

3.3.1 MPU Permissions Setup

The MPU provides a coarser set of memory permissions than those available with a full Memory Management Unit (MMU). With the MPU available on our development board, we are able to maintain permissions on 8 regions of memory simultaneously. We may split these permissions between two possible operating modes: user mode and supervisor mode.

In the ARM Cortex M4 architecture, user and supervisor mode operate entirely separately, even maintaining their own stacks and stack pointers. User mode is designed for the execution of application-level code, whereas supervisor mode is privileged and typically performs hardware configurations. Though the default mode of execution is user mode, supervisor mode may be entered via the Cortex M4 supervisor call, or `svc` instruction. Upon completion of the privileged supervisor call handler, the program execution re-enters user mode via a simple function return.

We leverage supervisor mode as a means of restricting access to our shadow stack. The MPU is configured with a region protecting the shadow stack, ensuring no read or

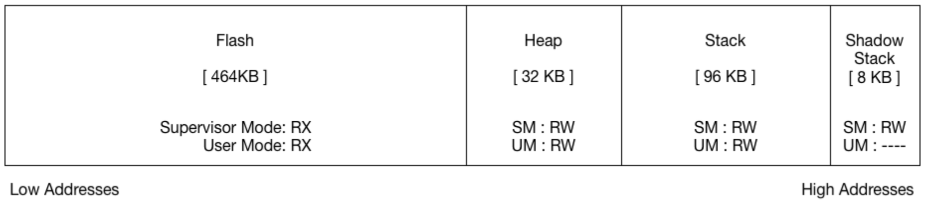

Figure 3.2: The modified memory region layout and permissions that RECFISH++ uses, enforced by the MPU.

write access for user mode while allowing full access to supervisor mode. This means data may only be pushed or popped from the shadow stack via supervisor mode, which will only be accessible via predetermined locations in the binary - the locations of the `svc` instructions.

In order to enforce the key CFI assumption that writeable memory is non-executable (and vice versa), we configure three additional MPU regions, shown in Figure 3.2. These regions are flash, heap, and stack such that the flash section may hold program code, and the heap and stack may hold runtime data.

3.3.2 A Modified Interrupt Handler

In order to trigger the correct shadow stack operation indicated by the argument passed to the supervisor call instruction, we must modify the program's supervisor interrupt handler. RECFISH++ is aimed at protecting programs running in FreeRTOS, therefore we make these changes to the FreeRTOS source code rather than at compile-time. We considered overwriting the FreeRTOS implementation with a custom handler during compilation but decided against it as it as this behavior might confuse users that attempt to edit the source code of FreeRTOS. We provide a modified `port.c` file within FreeRTOS containing a custom handler. This handler uses the argument passed by a supervisor instruction to choose the operation that should be performed, acting as a jump table to the shadow stack operations. A portion of this handler may be seen in Listing 2.

3.3.3 The Shadow Stack

The shadow stack is implemented by RECFISH++ in a similar way to RECFISH. This shadow stack mirrors the normal stack but is dedicated to saving return addresses when needed and retrieve them before return instructions. This area of memory is separate from the normal stack, and unwriteable during normal program execution in order to ensure that the return addresses are not tampered with. In addition to storing return addresses, this secure storage method is also used to protect context switches that, by default, store, sensitive context information in unprotected memory. As with RECFISH, this means that the context switching functionality in FreeRTOS must be able to access and use the shadow stack.

```
SVC_Handler:
  ... ; load the SVC argument into r8
  ldr.w sl, table ; load the table address
  ldr.w r9, [sl, r8, lsl 2] ; load table entry
  orr.w r9, r9, 1 ; set to thumb execution
  bx r9 ; branch to selected table entry

table:
  .word jump_table
  ...

jump_table:
  .word shadow_stack_setup
  .word ss_push
  .word ss_pop
  .word original_handler

ss_push:
  ...

ss_pop:
  ...
```

Listing 2: The jump table used to dispatch shadow stack operations from the Supervisor Exception handler.

The shadow stack has similar traits to the traditional stack data structure: a stack pointer and bounds to its available space. The memory space that the stack is able to span must both be large enough that it accommodates the stored return addresses through any normal program execution path, but small enough to avoid wasted memory space. RECFISH++ uses the same limits that RECFISH does, 126 bytes per task. The shadow stack pointer lies in a predefined location in memory. The push and pop operations on this shadow stack operate much like any traditional stack, where the push checks for bounds, stores the return address and decrements the stack pointer and the pop does the opposite.

3.3.4 Task Initialization

In order to isolate the return addresses between tasks, RECFISH++ sets up an independent shadow stack for each. We take this pattern from within the operating system; FreeRTOS maintains a Task Control Block (TCB) for each task, storing the pointer to the standard, unprotected stack within alongside other task metadata. Upon task creation, we add the task's independent shadow stack pointer within the TCB. The pointer to the currently active shadow stack is updated during each FreeRTOS context switch, just as the stack pointer is updated. This ensures operations occur on the shadow stack corresponding to the task calling the function.

3.3.5 Secure Context Switching

The context of tasks is the second piece of critical information stored on the shadow stack. During a standard FreeRTOS context switch, the task's state is saved to and loaded from an unprotected stack in order to allow for pseudo-parallel execution on single-core processors. This saving and loading are typically very lightweight thanks to hardware support, only requiring a few instructions to push return address and registers, and pop the return address and registers from the corresponding task's stack.

In our backward-edge implementation of RECFISH, we must instrument FreeRTOS to save and load this same state to a task's shadow stack, rather than the unprotected one. To do this, we use the shadow stack pointer as though it were the unprotected stack pointer in the FreeRTOS context switch routine. For the save routine, we first save the halting process's status register and return address to the shadow stack, then all of the user/system registers, and finally save extra context such as floating point registers if applicable to a given task. For the load routine, we apply the same steps but in reverse, first loading the extra context, then the user/system registers, and then the return address and saved process status register.

We made the same considerations for the shadow stack operations as we did with the supervisor handler. While it would be possible to embed this functionality at compile-time, we chose to again modify our custom `port.c` file with the `ss_push` and `ss_pop` operations. Additionally, we modify the FreeRTOS `task.c` file in order to allocate a shadow stack for each newly created task and to switch the current shadow stack pointer during a context switch.

3.3.6 Secure-Spill Lowering Pass

At the start of a function, the return address is stored in the Link Register (`LR` register), a section of memory integrated into the Cortex M4 processor. Though the return address is safe within, the `LR` register is often needed for other purposes during function execution, such as entering new function calls. In this scenario, the return address is spilled onto the unprotected stack for later use. It is this case that our backward-edge protection is primarily concerned with. In order to protect return addresses when they must be moved out of the `LR` register, RECFISH++ instruments affected functions with extra instructions that will store the address to the protected shadow stack. By doing this, we can ensure that the return addresses are not tampered with, and therefore protect the control-flow from hijacking.

There are two challenges that must be solved in order to change how the `LR` register is spilled. First, RECFISH++ must be able to determine when it must to be spilled to make room for other uses. Second, we must add two calls to trigger shadows stack operations for both saving and loading the `LR` register to replace the original spilling functionality. Both of these challenges require target-specific information, meaning that LLVM's IR is not an option. Instead, RECFISH++ modifies a pass later in the compilation process that translates a program's IR to the target architecture, called

```
func:
  push r6, r7, lr ; spill LR with other needed registers
  ...
  pop r6, r7, pc ; restore return address directly to PC
```

Listing 3: An typical `LR` register spill onto the unprotected stack.

```
func:
  push r6, r7 ; spill other needed registers
  svc 1 ; call ss_push to spill LR securely
  ...
  svc 2 ; call ss_pop to return LR
  pop r6, r7 ; restore return address directly to PC
  bx lr ; branch to the return address
```

Listing 4: An instrumented function which uses `svc` calls to securely spill the `LR` register to the *shadow stack*.

the lowering phase. During lowering, ARM-specific features are available, including information on `LR` spills and the supervisor call instruction `svc`. RECFISH++ looks for `LR` spills and replaces them with corresponding calls to shadow stack operations via the `svc` instruction; any attempts by the compiler to save the `LR` register to the stack are replaced by a supervisor call with an argument of `1` to indicate `ss_push`, and similarly with `2` to indicate `ss_pop`. This way the `LR` register is spilled onto the shadow stack instead of the unprotected stack, with the determination of whether to spill the `LR` register still being made by the compiler. The effect of this instrumentation is shown by the difference between Listing 3 and Listing 4.

For a complete visual of the modified compilation process, refer back to Figure 3.1.

3.4 Fine-Grained Protection

With an implementation of RECFISH at compile-time, along with the performance benefits entailed, we now turn to consider how we can improve the protection offered by RECFISH++. Backward-edge protection seems to be sound, with no obvious vectors for hijacking being possible during normal execution. However, forward-edge does not provide this same guarantee due to the coarse-grained protection that it uses, which can be improved using a more fine-grained approach.

Though impossible guarantee only the correct indirect branch may occur at runtime without securing all code-pointers [6], we may reduce the number of potential attack vectors by generating unique labels for each equivalence class (discussed in Section 3.4.1). By using unique labels, we think that RECFISH++ could often reduce the subset of functions that a particular indirect branch may target, improving the forward-edge

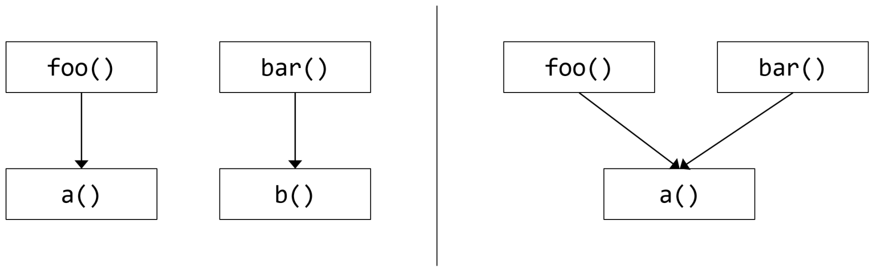

Figure 3.3: Left. *Scenario A.* Two isolated indirect callers. **Right.** *Scenario B.* Two indirect callers with the same target.

protection with no runtime performance decrease. This involves a complex analysis at compile-time to ensure that it does not inhibit normal program execution, but wouldn't involve any additional computation at runtime.

There are two major challenges that need to be addressed in order to create a more fine-grained protection. In order to ensure that the program is able to operate normally when it hasn't been hijacked, there must be a method for determining when it is safe to give an indirect jump a unique label [3]. Second, in order to start using additional labels, we would need to ensure that the new labels don't appear encoded anywhere in the binary other than where we place them; otherwise, these conflicting encodings would be a potential target for hijacking.

3.4.1 Instrumentation by Equivalence Class

Trying to limit where an indirect jump may target requires an understanding of how indirect jumps affect each other. Consider Figure 3.3, which shows two scenarios. In Scenario A, the two indirect jump instructions can potentially target two subsets of functions which have no target in common. A unique label could be assigned to `foo()` in Scenario A and all of its potential targets, which would ensure that `foo()` could never target one of `bar()`'s targets, and vice versa. Now consider Scenario B, where there is a single function that both `foo()` and `bar()` may target in normal execution. If we apply the same logic as in Scenario A and give `foo()` and its targets a unique label, then give `bar()` and its targets a unique label, `a()` would require two labels to fill these constraints. We require a method that is able to detect this scenario, and scenarios like it, in order to implement a more fine-grained approach.

We extend the functionality provided by the DSA library, previously discussed in Section 3.2.2, to organize indirect jumps into equivalence classes, allowing us to accurately determine when we can use a new label to separate the jump targets. These classes are used to divide the call graph into groups that cannot be further distinguished from each other with the label-based approach; functions with the same indirect caller must have the same label prefix, and functions with the same indirect target must have the same label check. Consider Figure 3.4, where each arrow represents an indirect call.

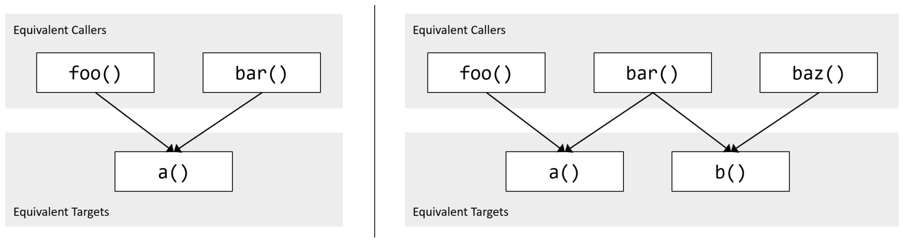

Figure 3.4: Left. *Scenario A.* A simple scenario requiring an equivalence class. **Right.** *Scenario B.* An equivalence class difficult to instrument with a special consideration.

The entirety of Scenario A is one equivalence class, and the entirety of Scenario B is one equivalence class due to overlapping targets.

RECFISH++ uses a spanning technique to find equivalence classes; it inspects each indirect jump, groups all of its potential targets, and then finds other jumps that share a target until no more similar jumps are found. This group of all jumps that share potential targets defines one equivalence class. For example, consider two indirect call instructions which may target `printf`; all of the possible targets of either of those calls would be combined and considered as one equivalence class, containing the same label within their prefix.

Equivalence classes are used in order to minimize runtime overhead. With a fully fine-grained CFI approach where each indirect jump is only able to jump to its own set of valid targets, each of these jumps would need to be instrumented with a loop that checked over multiple labels. This approach is unfavorable due to the increased performance overhead. Using equivalence classes, we can reduce the number of comparisons to one by losing some precision in what we consider a valid control flow. More precisely, control flows that may be considered invalid in a fine-grained approach will be allowed by RECFISH++ in order to reduce performance overhead.

The spanning approach is taken to avoid facing conflicting labels while discovering equivalence classes. Consider Scenario B in Figure 3.4. Imagine `foo()` was instrumented first, `bar()` instrumented second, and `bar()` last; without spanning, `qua()` would get a label and `thud()` would get a second unique label. This becomes a problem when `bar()` is instrumented, as one of the two labels would need to be changed to merge the equivalence classes. The spanning approach avoids this by visiting any of these functions and discovering the other two before any unique label or instrumentation is added. By spanning the entire equivalence class at once, we avoid ever facing a conflicting label at a call site we are visiting for the first time.

LLVM's call graph neither tracks indirect branches nor groups them into equivalence classes. In order to obtain this behavior, we extend our model enforcement pass by modifying our LLVM pass. This modified pass enumerates the indirect targets offered by DSA and stores indirect branches in a manner that we could later easily traverse both backward (from target to source) and forward (from source to targets). After a group of branches has been identified as an equivalence class, a new label is generated and all of the contained branches are instrumented with a CFI label check. The algorithm,

```
Instrumented = []
def VisitCallSite(CallSite)
   if Instrumented contains CallSite
       return
   ClassMembers = FindClassMembers(CallSite)
   for each Member in ClassMembers
       Instrument(Member)
       append Member to Instrumented

def FindClassMembers(CallSite, Members = [])
   append CallSite to Members
   Targets = GetIndirectTargets(CallSite)
   for each Target in Targets
       Callers = GetIndirectCallers(Target)
       for each Caller in Callers
           FindClassMembers(Caller, Members)
   return Members
```

Listing 5: The pseudo algorithm used to span equivalence classes and instrument indirect branches, called *CallSites*.

provided in Listing 5, will continue this process until all of the indirect branches have been explored and instrumented.

The `GetIndirectCallers` function is not provided by the DSA library. This is implemented by mapping each call site to a call target using a searchable data-structure before any equivalence classes are determined.

3.4.2 Generating Globally Unique Labels

Another challenge is ensuring that a particular label is unique throughout the executable portion of the binary. Without this, we foresee the possibility of a label conflicting with a commonly used instruction encoding and therefore providing potential targets for a hijack. While this challenge was trivially overcome in RECFISH by inspecting and searching the pre-built program, this becomes difficult when trying to solve at compile-time. At compile-time, it is difficult to predict how the program being compiled will be linked with other code that is not available to the compiler and therefore impossible to do a simple search through all existing encodings. We would need to find a method at compile-time for ensuring that the new labels introduced would be unique in order to ensure that protections are not lost.

In order for our approach for forward-edge protection to guarantee that a given call site may only call functions in its assigned equivalence class, the label chosen for the class must be globally unique. That is, the label should not show up anywhere else in the compiled program except within our instrumentation. If the label were to exist somewhere else in the binary, an attacker could possibly subvert the control-flow to the location that the conflicting label appears. RECFISH++ must make one key assumption about the typical Cortex M4 program in order to ensure global uniqueness at compile-time: only defined instruction encodings show up in the executable portion of the binary.

Choosing a four-byte label which contains no instruction encoding in the first or third byte decreases the possibility that the label exists somewhere else in the binary. The ARM Cortex M4 architecture supports the Thumb instruction set, consisting of both two and four-byte instructions. Choosing a label which is four bytes but which doesn't encode any instruction should ensure that the label doesn't exist anywhere in the program's normal operation, as long as this assumption holds. Within this assumption are two smaller ones that require knowledge of typical program construction.

First, it would be possible for an undefined instruction to be executed, as this is a method that is sometimes used to extend the hardware instruction set. Anytime an undefined instruction is executed, an Undefined Instruction exception is created and handled. This handler could be used to functionally add an intended side effect to an undefined instruction. We think that we can make the assumption that undefined instructions would not be used in this way because it would not be possible without modification of LLVM. RECFISH++ requires its own modifications to LLVM, and so we assume that any user that has custom modifications to LLVM will ensure that they do not conflict with this aspect.

Second, that all lines in the executable section could be executed normally. While this may seem obvious, it is possible for the compiler to embed information into the program that is never intended to be executed (which is precisely what we are doing with our labels). We assume that no such metadata exists outside of our own labels for similar reasons to the absence of undefined instructions; these would require modification to LLVM, and we don't intend to support versions of LLVM that have been modified out of our control.

These previous assumptions only consider the executable code sections of the binary. We assume that the MPU is configured to ensure that writable memory is not executable. Even if a label were to show up outside the code section, it would not be a useful tool because branching to it would result in a memory violation. We are able to ensure that this assumption is true by additional configuration of the MPU by RECFISH++.

In practice, it is difficult to ensure that these assumptions hold true for every application. We choose to rely on them to ensure global uniqueness for one additional reason: even if one of these assumptions failed, an attacker would have fewer options available to hijack the program. Not only is the chance of a label conflict low, the low probability that a useful gadget would be located directly after a conflict would make a vulnerability even less likely.

One last consideration must be made; the Cortex M4 architecture can operate with Thumb two or four-byte instructions. This means that the label must be interpreted as both a two byte and four-byte undefined instruction. There are 128 different encodings satisfying these requirements that RECFISH++ will choose from. In the case that more than 128 different equivalence classes are found, RECFISH++ will sacrifice precision by reusing encodings. In this way, unique labels can be systematically generated at compile-time. Note that no aspect of these protections relies on the secrecy of labels; instead, they rely on the difficulty of placing them in executable memory during execution.

3.5 Unnecessary Instrumentation Analysis

In an attempt to further improve the performance of RECFISH++ over its predecessor, we propose a system to filter out runtime checks that are unnecessary for providing security guarantees. Our intuition is that those tasks which do not receive external input are not vulnerable to control-flow hijacking so we may remove protections. In order to accomplish this, we rely on two assumptions: FreeRTOS tasks communicate solely through the operating system, utilizing one of several message-passing APIs [1], and the hardware running FreeRTOS may reconfigure the MPU from within its supervisor mode at any time.

We examine a program on a task by task level, searching for those which do not rely on user input to function. Any task which remains untouched by RECFISH++ must meet two criteria. Firstly, the task cannot directly receive external input via memory mapped I/O or a network connection. Secondly, the task cannot receive inter-task communications of sensitive types which may lead to exploitation. We will discuss each of these in turn.

For the duration of this section, we will consider a safe task one which does not receive external input, and therefore does not require runtime protection. Unsafe tasks, however, are those which cannot be verified to be safe. We consider tasks unsafe by default in order to over-approximate the vulnerability of a program.

3.5.1 Direct External Input

Marking tasks as unsafe due to the utilization of external input can be accomplished in a number of ways, however, many rely on coupling RECFISH++ to a specific micro-controller. A naive yet general approach would rely on a programmer-defined list of functions which receive external input. Due to the potential for human error in this approach, we propose a hardware specific alternative.

Embedded systems interface with the outside world via peripherals whose functionality have been mapped to specific addresses — a process referred to as memory mapped I/O. Extending the compile-time portion of RECFISH++, we may check each read and write instruction for its target address, flagging those which appear to be for the purpose of peripheral communication. This task may be non-trivial on certain embedded systems, however, the approach was designed with the STM32L4 Discovery Board in mind as the memory layout dedicates a specific range of addresses for peripheral communication [2].

In order to readily identify network connections, we further couple ourselves to FreeRTOS and act upon the assumption that applications will correctly leverage the tools provided by it. As such, we search tasks for calls to the provided `FreeRTOS_socket()` function and mark those tasks unsafe.

3.5.2 Sensitive Inter-task Communications

Throughout a typical FreeRTOS application, tasks communicate with one another to delegate responsibilities asynchronously. Because of this, we cannot consider each task

```
/* FreeRTOS APIs store the message
 * as the second argument
 */
Value* pointed = callInst.getArgOperand(1);
pointed = pointed->stripPointerCasts();
Type* pointedType = pointed->getType();
processOperandType(pointedType);
```

Listing 6: A few lines of code operating on LLVM IR which can capture the function type of an branch.

in a vacuum but must consider the potential vulnerabilities introduced via communications with other tasks. If one task receives malicious input and sends that to another, both must be instrumented by RECFISH++ in order to provide control-flow guarantees.

One potential compile-time approach would be to build a communication graph via static analysis. Each node in the graph would be a task, and each edge would be a directed flow of information. The benefit of this approach would be a very precise understanding of the target program, allowing for the maximum number of tasks filtered out for instrumentation when combined with the methods of identifying external input discussed in the previous section. Only those tasks which receive direct external input or from which there is a path along the communications graph to direct external input require runtime checks. The downside, however, is that an accurate communications graph would be difficult to determine at compile-time due to the potential for control-flow contingent communications.

In order to simplify the problem into a more manageable one, we take a coarser approach. Instead, we simply examine each task in isolation, determining whether it receives any sensitive communications from *any* task via one of the FreeRTOS provided message-passing APIs. Despite the fact that the source of the information may be benign, we treat it as though it could contain malicious information in order to maintain our control-flow guarantees.

Now that we have established an approach to identifying communications between tasks, we must classify communications as sensitive or insensitive. To do so, we draw inspiration from Kuznetsov et al [6]. In classifying pointers as sensitive or insensitive in order to isolate them, the code pointer integrity system *Levee* provides a list of potentially dangerous data types. These data types include character pointers, void pointers, code pointers, and pointers to complex data types containing any of the previous. We operate under the same model for consistency among similar works.

Utilizing the rich type system available at compile-time from within the LLVM Transform Pass, it is trivial to identify the type of the message being passed via any of the FreeRTOS APIs. A rough snippet can be seen below.

The simplicity of this extraction highlights one of the major advantages of RECFISH++ over RECFISH. Access to higher level semantic information allows for rich static analysis impossible on a precompiled binary.

3.6 Limitations

A few additional challenges that we have considered, but have not solved. The first challenge lays in the FreeRTOS initialization process, where we are not able to use a protected region for return addresses of operating-system code. We set up shadow stacks and configure the MPU just before tasks are created, but in the period between power-on and task creation, backward-edges are not being protected. We'll consider possible solutions for this in the Section 5.1, but for now, we don't prioritize this because we are focused on protecting the user-defined tasks, not FreeRTOS.

Another challenge is the inability to merge equivalence classes across separate source files. The compiler is only able to modify and traverse one source file at a time. While this didn't affect the demo programs we tested (due to the rarity of indirect branches), it is possible that a normal program wouldn't operate correctly due to mismatched labels across files. Again, we'll discuss potential solutions in the section 5.1 section.

Lastly, statically finding the targets of indirect branches is very difficult at compile-time, both due to the complexity and the last of existing implementations. We chose to use LLVM's DSA tool, which has not been maintained since LLVM 3.6 or since early 2015. While this implementation seems to be fully capable of static analysis, it is not being actively maintained and is therefore difficult to have a high confidence in. Our CFI protection is only as precise as our generated CFG, so further consideration of this library should be made.

Chapter 4

Evaluation

To evaluate RECFISH++, we'll perform an overhead and security evaluation. We'll both the memory and CPU overhead, as well as attempting to exhaust all possible scenarios of the system to understand potential vulnerabilities.

4.1 Performance Evaluation

For each major component of RECFISH++, we'll break it into small pieces of functionality to consider overhead. We'll start with inspecting forward-edge, then backward-edge, and finally secure context switching. A summary of our overhead is available in Table 4.1.

Protection	Component	Memory (bytes)	CPU (cycles)
Forward-edge	Label comparisons	24 per ind. branch	33 per ind. branch
	Labels	4 per ind. target	N/A
	Register space	0-8 per ind. branch	N/A
Backward-edge	Secure spills	0 or 8 per function	N/A
	Supervisor except.	N/A	86 per spill
	Shadow stack	132 per task	28 per spill
	Initialization	300 total	not measured
Context switch	Secure switching	160 total	88 per switch

Table 4.1: A summary of the overhead incurred by RECFISH++.

RECFISH++ adds checks to each branch in the program with a dynamic target. These indirect branches seem relatively rare, with only two-hundred seventy indirect branches of 6,361 total branches, or 4%, showing up in a demo program.[1] We'll inspect the amount of overhead that our forward-edge protections add per indirect

[1]We used the AWS FreeRTOS demo program to evaluate our approach. More information available at `https://aws.amazon.com/documentation/freertos/`.

branch, considering the instrumentation on the branch and the labels placed on the target functions.

There are three sources of memory overhead that our forward-edge instrumentation adds, two of which require space in the code section and one of which indirectly requires space on the stack. The first source of memory overhead that we'll consider is the runtime check that we add. Per indirect branch, RECFISH++ adds fifteen instructions or twenty-four total bytes to the code to check the dynamic target for a label. The second source of overhead are labels, which add four bytes to each function that can be indirectly targeted. The last source of memory overhead is are the two registers required for the compare, where it is possible that the function will need to spill the previous contents of the registers onto the stack to allow us to use them.

There is only one source of runtime overhead where no CFI violation is detected, which are the additional instructions added to load and compare labels.[2] Of the thirteen instructions added, only eight of them are executed in the normal case, typically taking only about thirty-three CPU cycles to perform. We think that this overhead added per indirect branch is satisfactory.

The backward-edge protections are a major factor in our total overhead. The instrumentation for protecting return addresses is complex and frequently required. We separate the backward-edge overhead into the modified secure `LR` spilling, the modified Supervisor Exception handler and the shadow stack.

There are a few sources of memory overhead added by our backward-edge instrumentation. These sources include the possible added instruction for spilling `LR` onto the shadow stack and the modifications to FreeRTOS (modified Supervisor Exception handler jump table, shadow stack operations, shadow stack setup, Memory Protection Unit (MPU) configuration, shadow stack pointer in each Task Control Block (TCB), and shadow stack region). For each function that must spill `LR`, RECFISH++ directly adds between zero and two instructions, between zero and eight bytes, depending on whether there are other registers that must be spilled as well. The constant overhead added by the FreeRTOS modifications, covering the exception handler, shadow stack setup and operations, and the MPU configuration, total to about three-hundred bytes. For each additional task, our modified operating system adds only one four-byte shadow stack pointer to it's TCB. We allocate 128 bytes for the shadow stack per task in addition to the stack, but it may be possible to mitigate this overhead.[3]

As with memory overhead, there are a few sources of runtime overhead by backward-edge protections. We'll inspect the total runtime overhead on an instrumented function and the total runtime overhead required to set up the task shadow stacks on startup. Each access to the shadow stack consists of a Supervisor call, the Supervisor Exception Handler, and an operation on the shadow stack, which in total adds about twenty-nine instructions and fifty-seven CPU cycles. We think that this operation will be the most significant overhead on a typical program because an access to the shadow stack is made twice per each instrumented function, once to push and once to pop. Of the

[2]A failed check results in a hard fault and is thus hard to quantify the performance of; we leave this analysis to future work in trying to recover from this violation.

[3]Return addresses are being stored on the task shadow stack *instead* of the normal task stack, so it may be possible to solely reallocate space from the stack to the shadow stack without changing its functionality at runtime.

fifty-seven cycles needed to operate on the shadow stack, forty-three cycles are required to transition in and out of the Supervisor exception and fourteen are needed for the actual stack operation.

Most of the overhead associated with context switching overlaps with backward-edge protection because the context-switching instrumentation adds to the shadow stack. We'll focus on the additional overhead required for our modified context switching to ensure that control-flow is isolated completely between tasks.

The only source of memory overhead required for context switching is the additional instructions added to the FreeRTOS context switch functionality. This functionality only redirects the TCB saving and restoring to the shadow stack, and doesn't require any additional space for storing task state.[4] In total there are forty-three instructions or 160 bytes added for this functionality, which is included in our modified FreeRTOS.

The runtime cost of RECFISH++ modified context switching is not very significant. While our context switching needs to access the shadow stack and therefore needs to run in supervisor mode, the original switching already ran in supervisor mode. In total, all of the forty-three instructions that were added are executed during each task switch, costing about eighty-eight additional CPU cycles per switch.

RECFISH++ is able to provide a reasonably precise protection against all methods for control-hijacking using memory vulnerabilities, with full memory protection for backward edges and a low-overhead label-based forward-edge protection. Combined, these protections are powerful, able to protect a program with no requirement for additional changes by developers to an embedded real-time software.

4.2 Protection Evaluation

There are many states that the added RECFISH++ instrumentation can be in during execution. We'll consider situations such as context switching interrupting any of our instructions, situations where labels are found in writable memory, and situations where critical registers are stored on the stack during elevated execution of the Supervisor Exception handler. We'll step through each of our protections individually and consider whether there are measures that could bypass them and hijack the program.

Each indirect branch in the program is instrumented with a RECFISH++ label and check. This check is simple, consisting of a read from memory, a comparison and a branch depending on the result. We consider this simple flow alone, and then consider each possible event that could interfere with it to attempt to exhaust all possible scenarios and validate its protection.

In the typical execution of a forward-edge check, it will execute lines 3 through 9 of Listing 7, and then branch to the existing indirect call. The label loaded in by lines 3 and 4 will always read the two-byte aligned label directly before the loaded indirect target, masking away the last bit of the address (which signifies instruction set of the target function). When the target's label matches the inner label loaded by lines 5 and 6, the comparison will properly continue to make the indirect branch. If the labels don't match, the branch at line 9 will not be taken and the branch at 10 will, causing `exit(-1)`

[4]The shadow stack pointer is stored in the TCB, but we consider that to be backward-edge overhead.

```
1  func:
2    ...
3    bic.w lr, r0, 1 ; mask the last bit of the address
4    ldr.w lr, [lr, -4] ; load the label of the target function
5    movw r1, 0xdefd ; load half of the label
6    movt r1, 0xe7f2 ; load second half of the label
7    cmp lr, r1 ; compare the labels
8    str r0, [sp, 12]
9    beq.n normal ; continue to indirect call, normal case
10   b.n violation ; handle as a violation
11 violation:
12   mov.w r0, -1 ; pass -1 as argument to exit()
13   bl exit ; exit the program
14 normal:
15   ldr r0, [sp, 12]
16   blx r0 ; make the indirect call
17   ...
18
19 .word 0xe7f2defd ; matching label on target
20 foo: ; indirectly targeted function
21   ...
```

Listing 7: Inspecting the flow of a forward-edge check to ensure it is safe

to be called. The protection is straightforward and operates properly in both possible states (violation and normal).

We've shown that the check runs correctly when executed sequentially. It is possible for a hardware interrupt to change this normal execution flow though, and any change to the target between the time it is loaded (line 3) and the time the branch is made (line 16) would pass the check. Most notably, the timer exception used by FreeRTOS to meet real-time deadlines may interrupt a task at any instruction to schedule another task. The two potential scenarios of a hardware interrupt are that the function resumes normally or that the execution resumes in another scheduled task. We don't consider these transitions to expose any vulnerability. If the function resumes, the only modifications to the target could be made by FreeRTOS exception handler, which we consider to be safe. If another task is scheduled, our secure context switch will protect all registers on the shadow stack, so the function can eventually resume normally with a guarantee that the target was not corrupted. RECFISH++ forward-edge checks are correct even in the presence of hardware interrupts.

The labels that forward-edge protections rely on are as precise as the control-flow graph (CFG). When indirect branches must be merged into an equivalence class to avoid using multiple labels, our protection becomes less precise (fig. 3.4). This imprecision means that the CFG is not protected perfectly, possibly creating a vector for exploitation. The rarity of indirect branches makes this merging very infrequent, but ultimately this is

```
1 func:
2   push r6, r7 ; spill other needed registers
3   svc 1 ; call ss_push to spill LR securely
4   ...
5   svc 2 ; call ss_pop to return LR
6   pop r6, r7 ; restore return address directly to PC
7   bx lr ; branch to the return address
```

Listing 8: Inspecting backward-edge secure spilling to ensure it is safe

a security trade-off for performance.

Along with the precision of forward-edge, we'll also consider the accuracy. RECFISH++ and Control-Flow Integrity (CFI), in general, are not a direct form of memory safety, meaning that they do not protect the code pointers but instead ensure the integrity of their use in indirect branches. In most cases, our protection cannot be exactly accurate because we are not protecting the memory directly. For example, where a CFG finds that an indirect branch has three valid targets, an attack could still subvert the program but would only be able to target these three targets. We don't consider this to be a vulnerability because RECFISH++ is able to significantly reduces the potential targets, from every addressable byte to three targets in this example. This is another example of a security trade-off for performance, as current techniques for memory safety are too expensive.

Our protection also relies on the uniqueness of labels, a guarantee difficult to make at compile-time. By our assumptions that LLVM has not been modified to embed data in the code or utilize the `UNDEFINED` for functionality, we can ensure that conflicting labels are not encoded anywhere in the code section of the program besides directly before instrumented indirect targets. The only other scenario in which a conflicting label may occur is if written to writable memory. Branching to any place in writable memory, even if there is a matching label, will result in a hard fault due to the non-executable flag set on the MPU. Therefore, we are able to ensure that the control flow cannot be hijacked anywhere besides instrumented indirect targets.

We'll evaluate the safety of backward-edges as well in both the typical scenario, and each possible edge case from there.

For each instrumented function, our secure register spilling will execute lines 3 and 5 of Listing 8 to push and pop the return address on the shadow stack. This modified spilling replicates the functionality of normal `LR` spilling, saving it from the register before the function body is executed and restoring it to the register just before the function returns. Return addresses are never stored in unprotected memory, and this typical scenario seems safe.

Return addresses are only ever modified using a single Supervisor Call instruction. Interruptions are not a concern for this single instruction, as the exception handler and shadow stack operation both occur within the Supervisor Call exception while interrupts are disabled.

The backward-edge protection is precise. It protects return addresses directly and therefore is not constrained by the precision of a CFG. With the shadow stack, we can guarantee that each backward-edge has not been corrupted in any manner.

When the Supervisor Exception is triggered, the hardware moves all current registers to the stack, which is not protected. This means that during the exception, the return address for the calling function is stored in vulnerable memory. This is the most vulnerable that a return address ever can be with RECFISH++; yet, it is still safe because the processor only runs our handler, no user-defined code.

The last aspect of our backward-edge protections that we'll consider is the uninstrumented FreeRTOS initialization functions. Shadow stacks are set up per task, and so any code that is executed outside of a task is not able to be protected. We currently cannot provide any backward-edge guarantees during initialization. We focus on the user-defined code instead and assume that the lack of any input and the well-tested code of FreeRTOS initialization will protect it.

The last component to evaluate is the modified context switching that RECFISH++ uses to isolate tasks. This context switching works similarly to backward-edge instrumentation, redirecting critical information to the shadow stack instead of the normal stack.

The goal of our secure context switching is to make sure that information stored in registers and the shadow stack pointer are not modifiable outside of their owning task. During a switch, this critical information is transferred to the shadow stack in Supervisor mode and is restored from the shadow stack before returning to a task. The critical information is briefly stored on the stack, as with backward-edge, but again this is only accessible from the handler.

Chapter 5

Conclusion

Throughout this work, we describe a CFI scheme specifically designed to meet the unique challenges of real-time embedded systems. We show the modifications to both the LLVM compiler back-end and the FreeRTOS real-time operating system that are required for protection from control-flow hijacking. Our system required us to implement multiple compiler passes at varying compilation stages. We insert labels and runtime label checks during an LLVM transform pass, and we insert supervisor calls during the frame lowering for the ARM Cortex-M architecture designed to protect spilled return addresses by storing and restoring them from the shadow stack rather than an unprotected one. To complement this, we alter the SVC handler within the FreeRTOS Cortex-M port to implement pushes and pops to the shadow stack. To prevent time-of-check to time-of-use vulnerabilities, we modified the FreeRTOS operating system to isolate saved context from running threads. These modifications allow us to protect real-time applications using the FreeRTOS real-time operating system at compile time.

To the best of our knowledge, the implementations described throughout this work are secure from exploitation given that no privileged-mode code within FreeRTOS contains memory vulnerabilities. The shadow stack implementation within our modified FreeRTOS provides a subset of the guarantees provided by isolation, and as such is valuable independent of the rest of RECFISH++.

The overhead of RECFISH++ is reasonable for real-time applications. With the reasonably low microbenchmarks observed for both memory and performance overhead in testing, we believe our system is viable for many currently insecure applications.

5.1 Future Work

To further improve the performance of RECFISH++, we believe an integration of the reference system for removing unnecessary instrumentation is paramount. Though our system design maintains security guarantees, we have no way of estimating what percentage of instrumentation would remain in an application. We remain hopeful that only a small subset of the program would require runtime checks, but concrete data is needed to confirm the usefulness of this approach.

The second area of future performance optimizations could lie in shifting the shadow stack operations to require fewer supervisor calls. The high overhead of simply entering and exiting this interrupt discussed in Section 4.1 makes it the single largest bottleneck of performance. We believe reducing the number of supervisor calls may be possible, however, we propose no possible directions to do so.

Finally, the robustness of RECFISH++ may be improved via adding support for unique label generation across files. This major limitation of the compile-time approach may be solved through modifications to the linking process, merging equivalence classes and sanity-checking global uniqueness prior to producing an executable binary. Though the LLVM Linker, known as LLD [14], may be a suitable place for these modifications, it is possible this tool is incompatible with the version of LLVM in which RECFISH++ is implemented. Alternatively, these modifications may be made directly to the linker within the GNU ARM embedded toolchain utilized throughout this project [9].

Bibliography

[1] Freertos kernel. `https://www.freertos.org`. Accessed: 2018-03-18.

[2] Stm32 mcu discovery kits. `http://www.st.com/en/evaluation-tools/stm32-mcu-discovery-kits.html?querycriteria=productId=LN1848`. Accessed: 2018-03-23.

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

[4] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting bare-metal embedded systems with privilege overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 289–303. IEEE, 2017.

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

[6] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity.

[7] J. R. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 291–300, New York, NY, USA, 1995. ACM.

[8] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. *SIGPLAN Not.*, 40(6):129–142, June 2005.

[9] A. Limited. *GNU Arm Embedded Toolchain*. ARM Limited.

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):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*, volume 45, pages 31–40. ACM, 2010.

[12] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

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

[14] L. Project. *LLD - The LLVM Linker*. LLVM.

[15] L. Project. *LLVM 3.6 Documentation*. LLVM.

[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[17] 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*, pages 552–561. ACM, 2007.

[18] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.

[19] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

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