Control-Flow Integrity for Real-Time Embedded Systems

Robert J. Walls
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
rjwalls@wpi.edu

Nicholas F. Brown
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
nfbrown@wpi.edu

Thomas Le Baron
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
tlebaron@wpi.edu

Craig A. Shue
Worcester Polytechnic Institute, Worcester, Massachusetts, USA
cshue@cs.wpi.edu

Hamed Okhravi
MIT Lincoln Laboratory, Lexington, Massachusetts, USA
hamed.okhravi@ll.mit.edu

Bryan C. Ward
MIT Lincoln Laboratory, Lexington, Massachusetts, USA
bryan.ward@ll.mit.edu

Abstract

Attacks on real-time embedded systems can endanger lives and critical infrastructure. Despite this, techniques for securing embedded systems software have not been widely studied. Many existing security techniques for general-purpose computers rely on assumptions that do not hold in the embedded case. This paper 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 work proposes RECFISH, a system for providing CFI guarantees 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. We empirically evaluate RECFISH and its performance implications for real-time systems. Our results suggest RECFISH can be directly applied to binaries without compromising real-time performance; in a test of over six million realistic task systems running FreeRTOS, 85% were still schedulable after adding RECFISH.

2012 ACM Subject Classification Security and privacy → Embedded systems security

Keywords and phrases Control-flow integrity

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.2

1 Introduction

Real-time and embedded systems (RTES) are predominantly developed in C because it offers high performance, low-level hardware control, and is often the only language supported by the manufacturer-provided toolchain for the target device. However, C also brings a host of potential memory errors, or vulnerabilities, that are both easy for developers to make, and easy for attackers to exploit. For example, memory-corruption vulnerabilities (e.g., buffer overflows) allow an attacker to overwrite portions of memory with attacker-provided values. Such vulnerabilities can be leveraged to hijack the control flow of a program by overwriting code pointers (e.g., function pointers or return addresses). Such attacks, commonly called *control-flow hijacking*, manipulate the execution of a program by redirecting control-flow transfers to either attacker-supplied code [33] or useful code sequences already in the program (e.g., return-oriented programming [ROP] [37]).

Several classes of defenses have been proposed for general-purpose systems to address control-flow hijacking. These include control-flow integrity (CFI) [5], which prevents such attacks by enforcing a precomputed control-flow graph (CFG) to runtime (indirect) control transfers in an application. Various other randomization-based [23, 26, 7, 24] and enforcement-based defenses [30, 31, 25, 32] have also been proposed in the literature.

However, there are a number of unique challenges that make existing implementations of these defenses ill-suited to RTES. First, embedded hardware is less capable and often lacks important hardware features that existing software defenses leverage. For example, the ARM Cortex-R architecture that we target in this work does not have a memory management unit (MMU) and, consequently, it does not support the abstraction of virtual memory nor does it provide isolation between kernel and application code. Second, in order to ensure the temporal correctness of the system, overheads associated with security defenses must be analyzed and factored into schedulability analyses. Third, embedded systems rely on toolchains tailored to each board and architecture, including custom versions of compilers (e.g., GCC) and proprietary IDEs (e.g., CodeComposerStudio). It is time-consuming (or impossible) to modify each of these toolchains to support new defenses.

Given these challenges, the security posture of many RTES lags behind that of general-purpose systems, despite being deployed in safety- or mission-critical applications. Given the proliferation of cyber-physical systems and Internet-of-things (IoT) devices, such systems are becoming ubiquitous in our society. Furthermore, such devices are increasingly Internet-connected, and therefore easily targeted by remote attackers. We must therefore develop security defenses for RTES that address the aforementioned challenges.

Towards that end, in this paper, we propose, implement, and evaluate a new defense for protecting RTES from control-flow hijacking attacks. Our defense, called Real-Time Embedded CFI for Secure Hardware (RECFISH), is inspired by past work on control-flow integrity but distinguishes itself from existing efforts in three key ways. First, RECFISH addresses the problem of custom toolchains by retrofitting binaries. This allows the developer to use existing toolchains without modification and even apply RECFISH protections to binaries without access to their source code. Second, we develop a new memory-isolation approach for ARM systems that does not rely on virtual memory. RECFISH provides the isolation between application and OS code needed to support secure context switching and enforce control-flow integrity. In particular, we modify a popular real-time operating system, FreeRTOS, to include RECFISH protections. Third, we provide a rigorous analysis of RECFISH's impact on real-time schedulablility and show that RECFISH can be applied to most systems without violating real-time requirements.

We evaluate the security and performance overhead of RECFISH using four broad classes of experiments. First, we perform the Basic Exploitation Test (BET) proposed by Carlini et al. [10] and demonstrate how RECFISH prevents various types of corruption used for control-hijacking, and how it secures the necessary CFI state from malicious modifications. Second, to evaluate the performance overhead, we run the CoreMark and BEEBS embedded-system benchmarks. Third, in order to better understand the sources of overhead, we conduct a series of microbenchmarks to quantify the CPU cycles necessary for each CFI operation.

Finally, based on the microbenchmark results, we empirically measure the effect on real-time *schedulability* [36], or the ability to analytically guarantee all deadlines will be satisfied, a fundamental metric in work on RTES. To our knowledge, we are the first to analyze the effect of a memory-corruption defense on analytical schedulability – this analysis provides a significant distinction from previous work in the embedded space (e.g., EPOXY [12]). Our contributions are summarized as follows:

- **Binary instrumentation for ARM:** We develop a CFI scheme, RECFISH, that protects both ARM-based bare-metal applications and those that run on FreeRTOS.
- **Protection mechanisms for CFI data structures:** We protect the instrumentation required for CFI as well as the shadow stack on low-resource ARM-based systems that lack native capabilities for such protections.
- **Process isolation 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 framework for ARM:** We create a binary-patching framework that rewrites precompiled ARM binaries to add CFI protection.
- **Evaluation:** We conduct both a security evaluation of RECFISH using BET and a performance evaluation using benchmarks, microbenchmarks, and schedulability.

2 Background and Related Work

2.1 Control Flow Integrity

CFI-based defenses check, at runtime, if program execution follows a legal control flow. Broadly, CFI schemes modify the target binary in three ways. First, at each indirect branch target, they insert a label to encode legal control-flow transfers. Second, at each indirect branch instruction, they insert instrumentation to verify the target has the expected label. Third, at each function return, they insert instrumentation to ensure control returns to the calling function.

Legal control flow is defined by the program's control-flow graph (CFG). Typically computed at compile-time, the CFG is a directed graph where the nodes represent *basic blocks* – i.e. sequences of program instructions ending in a branch – and the edges represent legal control-flow transfers between basic blocks. There are broadly two classes of branches: *direct branches* statically specify the target, while *indirect branches* depend on a register or memory value to specify the target at runtime. The latter are the target of control-flow hijacking attacks [37] and the focus of CFI. Note, checks are not needed for direct jumps when the code section of memory is read-only as the attacker cannot modify the target.

CFI implementations vary primarily in the choice of labeling scheme and the approach to protecting function returns. For performance reasons, some CFI approaches ignore function returns and only protect the other indirect branches. Other CFI-based defenses – including the original implementation by Abadi et al. [5] and the system proposed in this paper – rely on a runtime data structure, called a *shadow stack*, to securely store return addresses. This structure increases the precision of CFI and, by extension, the security of the instrumented program [17]; the tradeoff is higher overhead. See the survey by Burow et al. for a more comprehensive treatment of prior work on control-flow integrity [9].

Compared to earlier control-flow defenses (e.g., StackGuard [14], RAD [11], and DISE [13]), CFI implementations often consider a stronger threat model and provide stronger security guarantees. Specifically, CFI-based defenses must ensure that control-flow integrity is enforced even against adversaries that have full control of the data memory. In contrast, these earlier defenses do not protect all code pointers (only return address) and the shadow stack is either left unprotected from attackers with the ability to arbitrarily write to memory or the defense adds significant overhead by interposing on all (or a large subset of) memory writes.

2.2 Real-Time Embedded Systems

To facilitate writing real-time software, embedded-system designers often use a *real-time operating system* (*RTOS*). In a real-time OS, *tasks* are the rough equivalent of a process in a general purpose system. A *scheduler* is used to switch between executions of each task to meet pre-defined timing constraints.

RTOSes vary greatly in their complexity. On more powerful hardware, RTES can leverage versions of Linux compiled with `SCHED_DEADLINE` or `SCHED_RT`, which replace Linux's default scheduler with a real-time scheduler. On processors designed for embedded use – like those targeted for this work – the hardware typically does not meet the minimum requirements for Linux. For reference, in 2014, a minimally configured Linux kernel required at least 8 MB of program flash and 1.6 MB of RAM [40], whereas the test device for this work has only 1.25 MB of flash and 192 KB of RAM. The alternative to real-time Linux is using an embedded RTOS such as FreeRTOS 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. One of the most common RTOSes is FreeRTOS. Designed to be as small as possible, this free and open source RTOS fits in as little as 5 KB of program flash and under 1 KB RAM, depending on the features used [4]. FreeRTOS is highly portable, with ports for most major architectures. FreeRTOS, while minimal in nature, provides a few rich features such as mutexes, semaphores, shared queues, and software timers.

2.3 Real-time Security

There has been some prior work on providing increased security to real-time systems. However, most of this work has focused on different attack classes or adopt weaker threat models than considered here. For example, Hasan et al. [22] considered how to schedule security monitoring into real-time scheduling while respecting legacy real-time constraints. Others have considered information-leakage attacks via cache-based and other side channels [29, 35], and how schedule randomization can be applied to defend against such threats [39]. In this work, we consider a much stronger, more pernicious threat model, that of memory corruption and control-flow hijacking.

EPOXY [12] targets the same class of attacks as RECFISH, but the underlying approach is significantly different. First, EPOXY does not guarantee control flow integrity, i.e., EPOXY does not check the target of indirect branches. Second, EPOXY is compiler-based whereas RECFISH retrofits existing binaries. It is unlikely that EPOXY could be re-engineered to work on existing binaries, e.g., EPOXY's code diversification presents significant challenges if implemented outside of a compiler. Third, EPOXY only targets bare-metal applications whereas RECFISH is implemented for both bare-metal and FreeRTOS. As we explain later sections, context switching introduces additional security challenges, which EPOXY does not address; notably, EPOXY does nothing to protect the stack in a multi-task environment.

2.4 ARM Architecture

Our work focuses on ARM's Cortex-R architecture for high performance real-time systems. Most Cortex-R processors are single core and they have special interrupt controllers and caching mechanisms to support the low latency required by real-time systems. Unlike x86-based hardware, ARM Cortex-R does not support virtual memory. Consequently, all realtime tasks share the address space. The lack of a memory-management unit and high-quality entropy sources [19], coupled with a small address space, mean it is especially challenging to implement randomization-based defenses (e.g., ASLR). Further, these challenges also complicate the implementation of secure runtime data structures (e.g., the shadow stack).

Another important complication is that Cortex-R chips operate on several different instruction sets, such as the ARM and Thumb instruction sets. It is common for a single ARM binary to include instructions from multiple sets and switch among them during execution. Broadly, the Thumb instruction set and its variations are used to reduce code size with minimal reduction in performance. We further describe on the details of the Cortex-R architecture and its implications for the design of RECFISH in Section 3.

3 Design of RECFISH

RECFISH is a software defense for embedded ARM architectures. RECFISH takes a control flow graph and program binary as input, adds security instrumentation, and produces a protected binary. We divide the discussion of RECFISH into four components: *(i)* basic memory protections, *(ii)* forward-edge CFI, *(iii)* shadow stack operations, and *(iv)* secure context switching. The first three are presented in the context of bare-metal execution and the last in the context of FreeRTOS.

3.1 Threat Model

We assume a powerful adversary able to modify anything in writeable memory at any time, including all data on the stack or heap. The attacker cannot, however, modify read-only memory such as program code. Unlike other software defenses, we also assume that writeable memory is, by default, executable. Consequently, we must implement basic memory protections as part of RECFISH.

The attacker's goal is to subvert the control-flow of a program by modifying the target of an indirect branch. In ARM, an indirect branch is either *(i)* a branch instruction with a register operand, or *(ii)* any operation with the program counter register as the destination. These instructions are enumerated in Appendix A. RECFISH is charged with thwarting such attacks. As with previous work, RECFISH does not prevent memory corruption, but it does prevent corrupted code pointers from hijacking control-flow.

In a system executing without RECFISH modifications all of RAM is configured, by default, to be readable, writeable, and executable. The code is stored in ROM which is only readable and executable. Discussed in detail below, RECFISH uses binary instrumentation to check the targets of indirect branches and leverages the MPU to disable the execute permissions for RAM and to create a region of protected memory for the shadow stack (and other security-critical structures in FreeRTOS). Most code executes in an unprivileged mode and this protected memory region is only accessible from privileged modes.

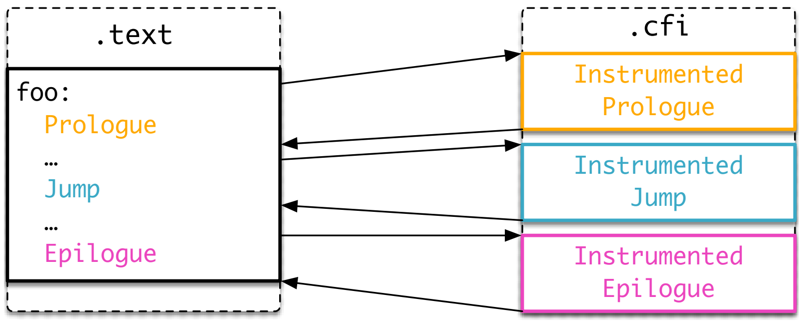


Figure 1 RECFISH uses trampolines to add CFI instrumentation to binaries without access to the source code.

3.2 RECFISH for Bare-Metal Execution

RECFISH patches pre-compiled binaries to add security instrumentation. Binary patching promotes broad adoption of the defense as it allows developers to employ RECFISH without modifying existing toolchains. This capability is important for retrofitting security to existing devices that may otherwise never receive updates.

However, binary patching is more complicated than simply inserting additional instructions into the code section. Namely, the inserted instructions can break the relative addressing common in the ARM Thumb instruction set. For example, the instruction `ldr r1, [pc, #32]` loads data from an address 32 bytes after the program counter. With in-line checks, we must update this instruction (and likely many others) to point to the new location of the data. To avoid this issue, we instead instrument instructions by replacing them with *trampolines*, i.e., direct branches to CFI code appended to an unused memory section. At a high level, the patched binary follows the format shown in Figure 1. The original program code is in the `.text` section, and the CFI instrumentation goes into a new `.cfi` section.

3.2.1 Basic Memory Protections

Like previous CFI implementations, RECFISH depends on two basic memory invariants. First, code regions must be read-only. Second, writeable regions must be non-executable. Unlike x86, Cortex-R does not offer virtual memory support, so these protections must be implemented using the limited functionality of the memory protection unit (MPU) and privileged processing modes.

The MPU, included by most Cortex-R processors, supports developer-defined permissions for up to 12 memory regions. For each region, there are three sets of permissions to be set: user mode, privileged mode, and execute permissions. For example, a memory region can be configured to be read-only for user mode, read and write for privileged mode, and non-executable (in any mode). In addition to the basic memory protections described above, we also leverage the MPU to create a protected region for shadow stack operations (see Section 3.2.3). MPU violations result in a data abort and any attempts by an adversary to modify program code or execute from writeable memory will be prevented.

To enforce basic memory protections, RECFISH requires the device to have an MPU and two available MPU regions. This includes many Cortex-M, Cortex-R, and RISC-V devices. The primary difference between Cortex-M and Cortex-R, in the context of this work, is that the former uses memory-mapped registers that must also be protected – though this can be done in the same manner that RECFISH protects the shadow stack.

Listing 1 Example function that uses function pointers.

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

Listing 2 Disassembly of the `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
```

Cortex-R processors have seven processing modes: User, System, Supervisor, Interrupt, Fast Interrupt, Abort, and Undefined. We leverage these different modes to perform operations at different privilege levels. Specifically, we use the *unprivileged* User mode for normal code execution and forward-edge CFI checks and the *privileged* System, Supervisor, and Interrupt modes for other CFI functionality such as shadow stack operations and context switching. The most relevant distinction between privileged and unprivileged modes is that the former can access memory marked as privileged-only.

3.2.2 Forward-Edge Checks

We use a simple example to illustrate how RECFISH handles forward-edge checks with binary patching. Consider the function `foo()` given in Listing 1 and its disassembly shown in Listing 2. RECFISH instruments three components of this function: the prologue, the indirect call resulting from the function pointer usage on line 4, and the epilogue. Listing 3 shows the resulting instructions after RECFISH is applied; namely, each component is overwritten with trampolines to RECFISH instrumentation.

Listing 3 Instrumented version of the `foo()` function.

```
0x192:    b.w 0x13f60        # Branch to new prologue

0x196:    <label>            # Insert label
...
0x1e6:    bl 0x13f80         # Replace indirect call
                             # with CFI check
...
0x1f0:    b.w 0x13f98        # Branch to new epilogue
...
```

Listing 4 Function prologue instrumentation.

```
0x13f60:  push {r4, r7}      # Copy displaced instructions
0x13f62:  sub sp, 20         #   from the orig. prologue
0x13f64:  add r7, sp, 0      #   with modifications
0x13f66:  svc 0              # Call ss_push
0x13f68:  b.w 0x198          # Branch back, skipping label
```

Listing 5 Indirect call instrumentation.

```
0x13f80:  add r3, r3, r4     # Copied instruction
0x13f82:  push {r0, r1}      # Save registers
0x13f84:  ldrh r0, [r3, 3]   # Load target's CFI label
0x13f86:  movw r1, <label>   # Load expected label
0x13f88:  cmp r0, r1         # Compare the labels
          error:
0x13f8a:  bne error          # Error if mismatch
0x13f8c:  pop {r0, r1}       # Restore registers
0x13d8e:  bx r3              # Perform indirect jump
```

3.2.2.1 Function Prologue

RECFISH instruments the function prologue to embed the appropriate CFI label for `foo()`. This allows any calling function to verify that `foo()` is a legal target. As mentioned previously, RECFISH cannot simply insert this label without breaking relative addressing. Instead, RECFISH replaces 6 bytes of the original function prologue with a 4-byte branch to the CFI section (i.e., the trampoline) and a 2-byte label. The CFI section for the function prologue, shown in Listing 4, includes the instructions replaced in the original prologue, adds some shadow stack operations (discussed later), and returns to the instruction following the original function prologue.

3.2.2.2 Indirect Branches

RECFISH ensures that the target of the indirect branch is legal by checking the value of the target's label against the expected value. As with the prologue, RECFISH inserts these checks into a separate CFI code region (Listing 5) and uses a trampoline to jump to the check. Note that the target's label is stored in a function prologue – similar to what was discussed above for `foo()` – and the expected label is hard-coded into the instruction at `0x13f86`. In the event of a label mismatch, the instruction at `0x13f8a` will branch to error handling code, which in the current implementation will result in an infinite loop.

RECFISH must replace the 16-bit indirect branch instruction with a 32-bit direct branch to the CFI check. To make space, RECFISH replaces both the indirect branch and the preceding `add` instruction. The displaced `add` is moved to the start of the appropriate CFI code region. We use a branch-and-link operation as the direct branch to copy the return address into the link register for use by the called function.

3.2.2.3 Function Epilogue

The modified function epilogue reverses the operations performed during the new prologue. Namely, RECFISH restores the registers previously pushed to the normal stack and pops the return address from the shadow stack. To do this, RECFISH again replaces two 16-bit

Listing 6 Function epilogue instrumentation.

```
0x13f98:  mov sp, r7         # Execute displaced operation
0x13f9a:  svc 1              # Call ss_pop
0x13f9c:  pop {r4, r7}       # Perform pop without PC
0x13f9e:  bx lr              # New return instruction
```

instructions with a 32-bit trampoline. In the instrumentation shown in Listing 6, the modified epilogue includes code to retrieve the return address from the shadow stack and move it into the link register. The original `pop` instruction is also modified such that the link register is no longer included in operands. This modification is necessary as the link register is not pushed to the normal stack in the new prologue. Finally, the code returns to the calling function using a branch-and-exchange to the link register.

3.2.3 Shadow Stack

The shadow stack is a region of memory, separate from the normal stack, used to securely store return addresses and increase the runtime precision of RECFISH checks. RECFISH makes the shadow stack inaccessible from the User processing mode using the MPU, but allows reading and writing from the Supervisor mode. Consequently, performing shadow stack operations (e.g., push and pop) requires RECFISH to jump into a privileged mode, modify the shadow stack, and jump back to an unprivilege mode. To implement this, we created a 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 determines the function number by reading the opcode of the software interrupt instruction. The ARM assembly code function to do this is shown in the Appendix in Listing 7. The short ARM assembly code functions for the shadow stack operations are shown in the Appendix in Listing 8.

Procedure calls are handled differently in ARM than x86. In x86, procedure calls are generally implemented with pairs of `call` and `ret` instructions. The `call` instruction is used to branch to the target procedure, saving the return address onto the stack. The associated `ret` instruction is later used to return to the caller, popping the return address off the stack. In ARM, procedure calls are implemented using a branch-link-exchange instruction [16]. Branch-link-exchange instructions atomically branch to the target location stored in the link register and then update the link register to store the return address. We leverage this behavior to reduce the number of writes to the shadow stack. Specifically, we only need to push `LR` to the shadow stack if the link register is spilled to the normal stack, e.g. when a procedure calls another procedure.

3.2.4 Implementation

Our prototype implementation uses the Capstone disassembly engine [1], the pyelftools [3] ELF file parser, and the Keystone assembler [2]. Capstone provides a powerful disassembly and instruction decomposition framework that makes it possible 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 procedures outlined earlier in this section to generate the instrumentation.

Finally, we use the Keystone assembler to write the patched code to a new binary. We follow the same procedure for function prologues, epilogues, and indirect branch targets until we have a fully instrumented binary.

3.2.4.1 Limitations

One limitation of our current implementation is that the size of the shadow stack must be manually configured. However, RECFISH uses additional instrumentation to ensure that the stack does not overflow.

Uncommon C features such as `setjmp` and `longjmp` pose additional challenges that our current implementation does not directly address. Though such functionality was not employed by any of the binaries we evaluated, we can extend RECFISH to support `setjmp/longjmp` without a substantial impact on the security or performance results presented. For example, we can adopt an approach similar to that used by DISE [13] and push the current stack pointer along with the return address to the shadow stack.

3.3 RECFISH for FreeRTOS

The primary challenge of extending RECFISH to the FreeRTOS real-time operating system is supporting context switching and multithreading. As we discuss below and in the following section, task preemption and the lack of memory isolation introduces the possibility of a memory error in one task being used to corrupt the memory of another.

Each task has its own stack. This stack is also used by the scheduler to save and restore state when switching from one task to another. Under our threat model, RECFISH must assume any information stored on the stack during a context switch could be modified by the attacker. Consequently, RECFISH cannot consider CFI checks as atomic operations as any context-switches that preempt a CFI check could introduce a time-of-check to time-of-use vulnerability. Specifically, CFI labels are loaded from read-only program code into registers. In the presence of context switching, however, registers with CFI-critical information – i.e., the two registers with labels and the register storing the branch target – could be saved to the task stack at any point during the CFI check. With careful timing, the attacker could overwrite these saved register values. Defenses on general-purpose systems do not have to address this challenge (for process threads) because register values are saved to kernel memory during a context switch.

To avoid this vulnerability, RECFISH saves task state in the task's shadow stack rather than on the task's unprotected regular stack. For FreeRTOS, this necessitates modification of the Task Control Block (TCB) structure, the task creation procedure, and the scheduler. The scheduler already runs in privileged mode with interrupts disabled, so we do not incur additional overhead from the Supervisor call instruction.

One alternative to using the shadow stack for context switching would be to disable interrupts during the CFI checks. However, we avoid this approach as it introduces additional scheduling challenges, i.e., it introduces a new source of latency for real-time tasks. Even in non-preemptive systems (where the scheduler only runs when a task yields), other real-time sensitive hardware interrupts could be negatively impacted by disabling interrupts.

3.3.1 Task Creation Modifications

We modified the FreeRTOS task creation procedure to assign a shadow stack to each task when it is created. Specifically, we extended the Task Control Block (TCB) structure to add a field for a shadow stack. We also modified the functions that initialize this structure, the FreeRTOS function `prvInitialiseNewTask` and the port-specific FreeRTOS function `pxPortInitialiseStack`. The `prvInitialiseNewTask` simply assigns the next available shadow stack to the TCB of the newly created task. The port-specific `pxPortInitialiseStack` function required more complicated changes. 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.

3.3.2 Scheduler Modifications

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. 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.

4 Evaluation

To evaluate RECFISH, we conducted three broad classes of experiments. First, we evaluated the security provided by RECFISH to show that the instrumentation will enforce the CFI policy, even in the presence of a powerful attacker. Second, we conducted a series of microbenchmarks to understand and quantify the overheads and costs associated with specific functionality within our instrumentation, and how commonly such functionality is invoked among many benchmark applications. Finally, the results of these microbenchmark experiments informed the design of a large scale *schedulability study*, or effectively a macrobenchmark study. This study demonstrates how the RECFISH overheads affect the guarantees that a given task system will meet all deadlines.

The reference system used for this work is a Texas Instruments Hercules RM46L852, an ARM Cortex-R4F processor. 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.

4.1 Security Evaluation

The security benefits of control-flow defenses are difficult to describe quantitatively. While measurements of ROP gadget reduction and Average Indirect Target Reduction (AIR) have been used in previous work, Carlini et al. have discussed how these measurements are misleading and reflect CFG precision more so than security [10]. Instead of using the aforementioned metrics, we adopt the standard qualitative analysis used in prior work on control-flow defenses for general-purpose systems; we show that RECFISH checks that all branches are legal with respect to the control-flow graph, those checks cannot be bypassed, and the shadow stack cannot be modified by an attacker. Because our focus is on the security of the proposed CFI instrumentation rather than the precision of the control flow graph, attacks that follow a legal control flow are out of the scope of this evaluation.

In addition to the qualitative analysis, we also tested RECFISH using the empirical methodology proposed by Carlini et al. – the Basic Exploitation Test (BET) – where a minimal, representative program is written with a known vulnerability (such as a buffer overflow) to show that a defense prevents an attacker from achieving their specific goal (i.e. arbitrary code execution) [10]. We elide further discussion of the BET results as RECFISH successfully prevented the attack.

4.1.1 Basic Memory Protections

RECFISH leverages the MPU to set memory as either writeable or executable, but not both. This prevents an attacker from inserting and executing their own code to bypass the CFI checks. Further, the attacker cannot modify program code to disable CFI checks. These two basic protections ensure the attacker's only attack vector is to modify writeable memory.

4.1.2 Label Assignment

The labels for CFI instrumentation must be chosen to satisfy the *global uniqueness assumption*. This assumption states that the byte sequence representing a label only appears in the code section as part of the CFI instrumentation. If this assumption does not hold and the label coincidentally appears somewhere else in code memory (e.g. as an instruction opcode), an attacker could circumvent CFI by overwriting a code pointer with an address that is the correct offset from the location where the erroneous label appears. Given that RECFISH patches pre-compiled binaries and that there is no dynamic linking in our target system, we can use static analysis to verify that all labels are globally unique.

Further, because 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 side-effect-free x86 `prefetch` instruction was used to encode the label [5]. In RECFISH, the forward-edge and shadow stack protections to prevent the label from being executed.

4.1.3 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. All critical operations of the CFI check are performed entirely in registers and thus are protected in the shadow stack if the check is pre-empted.

The source label is hardcoded in a `mov` instruction, so that cannot be modified. Consider the case where the target label matches the expected label. In this scenario, the target label either resides in read-only program code, or it has been inserted into writeable memory by the attacker. If the label is in program code and the labels are globally unique, the label must be valid and it precedes a legal branch target. If the label was maliciously inserted into writeable memory, the CFI check will allow the branch to be taken, but the MPU will prevent the processor from executing the code at the target. In summary, 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 code execution.

4.1.4 Backward-edge Instrumentation With Shadow Stack

Each backward-edge check has two parts: function prologue and function epilogue instrumentation. In ARM, we do not need to consider 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 the link register 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 to store the return address on the unprotected stack. All non-leaf functions are instrumented by RECFISH.

The instrumentation for non-leaf functions has a single goal – protect the return address by saving the link register on the shadow stack rather than on the unprotected regular stack. The general form for this instrumentation is shown in Listings 4 and 6. Importantly, memory accesses always occur at the current hardware privilege level (i.e., privileged or unprivileged). Most program code executes in unprivileged mode, but RECFISH places the shadow stack in a memory region accessible only during privileged mode execution. RECFISH uses the software interrupt (`svc`) instruction to change the processing mode from user to supervisor mode, allowing it to modify the shadow stack.

To manipulate the shadow stack, the attacker must exploit a memory corruption vulnerability while the processor is in privileged mode. Interrupt handlers execute in privileged mode, and thus are a potential avenue of attack; however, interrupt handlers in real-time systems are designed to be short and deterministic and can be designed without arbitrary memory writes. Finally, the shadow stack code itself is effectively atomic. If a context switch occurs during shadow stack operations, RECFISH pushes all of the context to the shadow stack, so there is no time-of-check to time-of-use vulnerability.

4.1.5 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 saved 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 previously, we combat this issue by storing all saved context in the shadow stack.

Since the scheduler runs with interrupts disabled, the scheduler operations are atomic from the perspective of program code. This means that there is no opportunity for an attacker to corrupt the context before it gets pushed to the protected shadow stack. Further, an attacker cannot change the pointer to a task's shadow stack because RECFISH protects the entire Task Control Block using the same privileged MPU region as the shadow stack.

4.2 Performance Impact

To measure the performance impact of RECFISH, we look at four 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. Third, we analyze the resource requirements for RECFISH via microbenchmarking. Finally, we perform a large-scale schedulability analysis to assess the suitability of RECFISH for real-time systems.

4.2.1 CPU Benchmarks

RECFISH is designed to work on embedded systems without a traditional operating system, thus benchmarks designed for general-purpose machines such as the SPEC CPU2006 benchmark are not appropriate for this evaluation. To measure the raw overhead associated with

CFI checks on a realistic workload, we used the CoreMark embedded system benchmark [18] and the BEEBS benchmark suite [34]. These easily portable applications run on a variety of embedded architectures. CoreMark performs various common embedded tasks, like matrix manipulation, linked list manipulation, state machine operations, and cyclic redundancy check (CRC) calculation. BEEBS combines benchmarks from MiBench [21], WCET [20], and DSPStone [42].

On our TI RM46L852 evaluation board, we measured the CoreMark score both with and without CFI using the default settings and 1000 iterations. 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. In this evaluation, all default settings were used, and 1000 iterations were run of the benchmark.

In the BEEBS benchmarks, over 70% of the applications saw less than 25% overhead. On benchmarks with few or no function calls and no indirect branches, we see no significant difference in execution time. However, in benchmarks like `recursion` and `fac` which both use recursive function calls, we see up six times slowdown. In practice, real-time embedded systems avoid using recursion because it introduces nondeterminism and can result in quickly running out of memory, so a slowdown of this magnitude is unlikely to occur in production systems. The `trio-snprintf` and `mergesort` benchmarks both have many indirect branches, but they only see about 0.5 times slowdown. The CFI checks are fast relative to other computation performed by these benchmarks.

4.2.2 Additional Resource Use

RECFISH requires an additional 10 bytes of storage per indirect branch and 8 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 required 964 bytes of instrumentation code for a 10 KB binary – just under a 10% increase in binary size.

RAM usage 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-byte 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.

4.3 Microbenchmarks

To better understand the overhead introduced by RECFISH, we measured each component of the instrumentation separately. In this section, we examine the number of CPU cycles RECFISH adds to indirect branches, function prologues, and non-leaf function epilogues.

4.3.1 Microbenchmark Design

We measured CPU cycles using the ARM Performance Monitoring Unit (PMU), configuring the PMU to count three events: CPU cycles, predictable branches, and incorrectly predicted branches. For each component, we took these measurements for a few different scenarios: no instrumentation, inline instrumentation, and the trampoline-based instrumentation used by RECFISH. We ran these microbenchmarks under different configurations of the branch predictor. By default, ARM uses a 256-entry, 2-bit history-based branch predictor with a hardware return stack [6]. The branch predictor can be configured to use static policies rather than the history-based policy and the return stack can be disabled. For the schedulability study, we also measured the context switch overhead in FreeRTOS, both with and without the shadow stack. For this measurement, we used the default branch predictor configuration.

4.3.2 Results

Under default CPU settings, we found that unconditional indirect branches without RECFISH instrumentation take 11 CPU cycles to execute. When adding inline checks to these branches, we saw a varied number of cycles. In the worst-case, the forward-edge CFI check takes 41 cycles, although we only see this worst-case result in the first iteration of the experiment. In later iterations, the branch predictor determined that the conditional branch inside the CFI check was not likely to be taken, so the CFI check sped up to 26 cycles after 5 iterations. For the trampoline method used in RECFISH, we saw a worst-case forward-edge check of 61 cycles, which sped up after 5 iterations to 44 cycles. By examining the execution time under different branch predictor settings, we could account for the majority of variability in execution time. By disabling the branch predictor's dynamic history function and return stack, only the first iteration of each microbenchmark was slower than the rest. We were unable to determine the cause of this slowdown, but potential causes could include pipeline stalls, pipeline flushes, or a conflict that prevents the CPU from dual-issuing instructions.

The other component that RECFISH affects is non-leaf function calls, since these functions must save the return address at the start of the function and restore it at the end. RECFISH requires that the return address be stored in the shadow stack, rather than on the unprotected user stack. Since there are no conditional branches in the shadow stack operations, we saw a constant increase from 19 cycles for the combined function prologue and epilogue without CFI to 275 cycles with RECFISH. Most this overhead is associated with changing the execution mode from User mode to Supervisor mode. Additionally, during these 275 cycles, interrupts were disabled twice for 11 cycles during the handling of the `svc` instruction, which was used once in the prologue and once in the epilogue. Since the combined function prologue and epilogue in RECFISH requires significantly more CPU cycles than the unmodified binary, we expect more performance degradation in binaries with many calls to short non-leaf functions. By contrast, programs with many calls to longer functions (or leaf functions) will see less performance degradation from shadow stack operations. Indeed, this matches our previous observations of the macrobenchmarks.

The final microbenchmark that we measured was FreeRTOS context switch overhead, which is critical to the schedulability study in Section 4.4. Without RECFISH, FreeRTOS context switches take a total of 120 cycles, 57 for saving context and 63 for restoring it. With RECFISH, we saw a moderate increase of context switch time to 159 cycles, 80 for saving and 79 for restoring.

Table 1 Microbenchmark results for individual components with the default branch predictor. All units are CPU cycles.

Component	CFI Type	Worst Case	Best Case
Indirect Branch	No CFI	11	11
Indirect Branch	Inline CFI	41	26
Indirect Branch	RECFISH	61	44
Function Call	No CFI	19	19
Function Call	Inline CFI	237	237
Function Call	RECFISH	275	275

4.4 Schedulability Study

Next, we incorporate the microbenchmark results into a large-scale schedulability study, which demonstrates the effect RECFISH has on the ability to ensure that all deadlines will be satisfied.

4.4.1 Schedulability

We begin our schedulability discussion with the *periodic task model* [28], which is implemented in FreeRTOS. In this model, a *task system* τ is composed of a set of n *tasks*, denoted $\tau = \{T_1, \ldots, T_n\}$. Each task is mathematically modeled as a tuple, $T_i = (e_i, p_i)$, and is comprised of a (potentially infinite) sequence of *jobs*, which are common invocations of the same logic. Each job of T_i executes for at most e_i time units, or its *worst-case execution time* (*WCET*). Jobs of T_i are *released* or made ready for execution every p_i time units, and must complete by their *deadline*. We assume the deadline of each job is p_i time units after it is released, i.e., when the next job of the task is released. The utilization of T_i is given by $u_i = e_i/p_i$, and the utilization of the task system U, is the sum of all tasks' utilizations, $U = \sum_{T_i \in \tau} u_i$. We assume fixed-priority scheduling, and evaluated schedulability using standard fixed-priority time-demand analysis [27].

4.4.2 RECFISH Schedulability

RECFISH introduces sources of overhead that do not exist in an unprotected system. In the rest of this section, we demonstrate how these overheads affect schedulability. To do so, we must first consider how the RECFISH overheads should be incorporated into the time-demand analysis. In our overhead analysis, we only consider overheads incurred on normal control-flow paths. While detecting and triggering an exception on invalid control flow incurs overhead, the code that must execute to handle such an exception is highly application specific and we thus exclude them from the scope of this study.

RECFISH introduces several sources of overhead, which are described more completely and quantified in Section 4.3. These overheads fall into two distinct categories: runtime checks, which occur at indirect branches and function prologues/epilogues, and context-switch-related overheads. These two types of overheads are accounted for analytically using different techniques.

We can account for the time spent in CFI checks by inflating the execution time of the task.[1] We assume that each CFI check at an indirect branch (respectively, function epilogue and prologue) imposes an overhead of Δ^{b} (respectively, Δ^{f}), and that each job of T_i executes

[1] We note that for a mere 11 cycles, interrupts are disabled. This is handled as a *priority inversion* and is incorporated into our analysis.

at most C_i^b indirect branches and C_i^f functions. To incorporate the overhead of all of the CFI checks, we simply inflate the execution time of each task to account for the time spent in CFI checks, $e_i' = e_i + C_i^b\Delta^{\mathrm{b}} + C_i^f\Delta^{\mathrm{f}}$.

The second source of overhead in RECFISH is the additional context-switch overhead associated with handling the shadow stack. We denote this overhead by Δ^{c}. Specifically, Δ^{c} includes both the time to save the context of one process, as well as restore the context of the next. We leverage existing techniques for handling context-switch overhead [8]. Instead of charging the overhead of the context switch to the task whose context is being saved or restored, instead, analytically, we charge the overhead to the preempting, higher-priority task, as is commonly done for analyzing cache-related preemption delays [38]. Notably, each job can only preempt at most one other job. Therefore, we analytically inflate each task's WCET to account for this overhead as, $e_i' = e_i + \Delta^{\mathrm{c}}$.

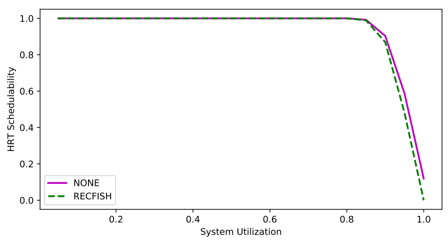
The other modifications to FreeRTOS required from RECFISH, such as augmenting the TCB and the task initialization procedures do not affect schedulability. It is quite common for a real-time system to startup and initialize the real-time tasks before entering a real-time mode, during which all deadlines must be satisfied. The performance implications of the remaining aspects of RECFISH fall within this initialization mode, and therefore do not affect schedulability. As such, in our schedulability experiments, we consider that when RECFISH is enabled, the execution time of each task is analytically treated as $e_i' = e_i + \Delta^{\mathrm{c}} + C_i^b\Delta^{\mathrm{b}} + C_i^f\Delta^{\mathrm{f}}$. In the context of this study, these are the only overheads considered so as to focus on the specific effects the RECFISH-specific overheads have on schedulability.

4.4.3 Experimental Design

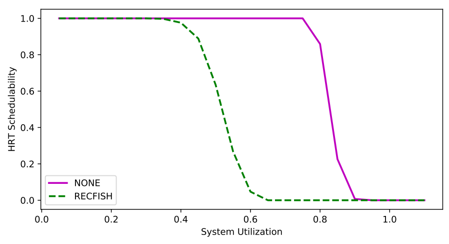
We conducted a large-scale schedulability study to evaluate the tradeoff between security and schedulability enabled by RECFISH. Practical real-time applications have varying task-system parameters, and the interplay among these parameters, analysis pessimism, and implementation overheads can have significant schedulability implications. Also, an overhead may be observed to be minor, but if it is unpredictable, difficult to incorporate into schedulability, or otherwise subject to analysis pessimism, it may significantly affect schedulability. Accordingly, we consider many classes of real-time task systems in our experimental design.

Overall, our schedulability study is conducted as follows. Using several different random distributions, we generate over six million analytical task systems with different parameter values. We then evaluate the schedulability of each task system both with and without RECFISH applied.

We randomly generated sporadic task systems using a similar experimental design as previous studies [8]. We generated task systems with a total system utilization in $U \in \{0.05, 0.1 \ldots, 1.0\}$. Per-task utilizations in each task system were chosen to be *light*, *medium*, or *heavy*, which correspond to uniformly distributed utilizations in the range $[0.001, 0.1]$, $[0.1, 0.4]$ and $[0.5, 0.9]$, respectively. Tasks were randomly generated using the chosen distribution until the desired system utilization was reached. The periods of all tasks were chosen uniformly from either $[3, 33]$ ms (*short*), $[10, 100]$ ms (*moderate*), or $[50, 250]$ ms (*long*). Based on the micro-benchmark experiments presented previously, we assume $\Delta^{\mathrm{c}} = 39$ cycles. We also considered two different values for $\Delta^{\mathrm{b}} \in \{33, 50\}$ cycles, which reflect the overhead at an indirect branch if branch correctly predicted or not. (In provisioning a hard real-time system, one may assume branches are always mispredicted, whereas in a soft real-time system, less analysis pessimism may be necessary.) We also measured $\Delta^{\mathrm{f}} = 256$ cycles.



(a) Bimodal indirect branches, few functions, short periods, heavy utilizations.



(b) Common indirect branches, frequent functions, moderate periods, light utilizations.

Figure 2 Example schedulability graphs.

Based on our microbenchmark results, we considered several distributions for the frequency of indirect branches and functions within each task. We considered that either no indirect branches were taken (*None*), or the number of indirect branches were uniformly chosen at a rate of one indirect branch among $[10^3, 10^5]$ cycles (*common*), $[10^6, 10^7]$ cycles (*rare*), or *bimodally* between the two distributions none (90%) and common (10%). Similarly, we considered the following distributions for the number of of functions: the total number of functions per task is chosen uniformly among $[1, 100]$ (*few*), or uniformly at a rate of one function among $[10^2, 10^3]$ cycles (*frequent*), $[10^3, 10^4]$ cycles (*moderate*), or bimodally between the two distributions moderate (90%) and few (10%).

We considered the cross product of these possible system parameters, resulting in 5,760 unique configurations. For each configuration, we generated and evaluated 1,000 task systems for schedulability, for a total of over six million task systems.

The chosen taskset parameters were pioneered by Brandenburg [8] and have been widely used in the community. We extended the taskset generation models to account for the frequency of indirect branches and functions, based upon data from the benchmarks we measured. To our knowledge, our work is the first to consider the effect of a control-flow hijacking defense on schedulability, and therefore we could not compare against other defenses from a schedulability perspective.

4.4.4 Schedulability Results and Observations

Based on these experiments, we generated 288 *schedulability graphs*, two of which, which demonstrate the performance extremes, are depicted in Figure 2. From these figures, we draw several observations.

Observation 1: RECFISH has a negligible impact on schedulability for some classes of task systems. This observation is supported by inset (a) of Figure 2. In this system configuration, there are relatively few tasks (because of the heavy utilizations) and relatively few CFI checks on indirect branches or functions. As a result, RECFISH has a negligible effect on schedulability, while improving security.

Observation 2: RECFISH has a significant impact on schedulability for some classes of task systems. This observation is supported by inset (b) of Figure 2. In this system configuration, there are many tasks given the light task utilizations, and each of those tasks has many CFI checks on both indirect branches as well as function calls. In such a system configuration, we would expect RECFISH to have a more significant effect on schedulability.

In this case, there is roughly 30% *utilization loss* due to RECFISH, i.e., CFI checks and their impact in overhead analysis cause 30% of the available system utilization to be sacrificed in order to meet all deadlines.

Observation 3: Across all generated task systems, 85% of those that were schedulable without RECFISH, were schedulable with RECFISH. This aggregate statistic demonstrates the practical applicability of RECFISH. In 85% of the generated task systems, RECFISH could be applied without compromising schedulability. Only those task systems that stress the available computing resources are unlikely to be schedulable in the presence of RECFISH. From these results, we believe that RECFISH can be deployed in most RTES with only a minimal increase in system size, weight, and power.

4.4.5 Optimization Opportunities

While a real-time system is composed of many tasks, only a subset of those tasks typically take external input, i.e. directly interact with the adversary. Any exploitation must involve one of those tasks. Specifically, a task is *unsafe* if it accepts external inputs from the user, and *safe* otherwise. We denote safe (resp. unsafe) tasks with the superscript T^S (resp. T^U). We can reduce the overhead of RECFISH and improve schedulability on many task systems by controlling the execution order of safe and unsafe tasks.

Consider the following example with two tasks T_i^S and T_j^U. Let us assume that T_j^U takes external input, contains a memory error, and the attacker can leverage that error to corrupt memory. Let us also assume that T_i^S does not contain any errors nor does it take external input. Without RECFISH, if T_j^U has a higher priority than T_i^S then T_i^S can be pre-empted by T_j^U. Thus, the attacker can leverage the bug in T_j^U to manipulate memory used by T_i^S. Therefore, RECFISH checks are needed in both T_i^S and T_j^U to provide control-flow integrity in this situation. If we prevent the preemption of T_i^S by T_j^U, then we need not conduct the CFI checks on T_i^S, only T_j^U. This can be realized either by marking such tasks as non-preemptive, or in fixed-priority scheduling which is supported in FreeRTOS and our RECFISH implementation, by increasing the priority of T_i^S over that of all unsafe tasks. Therefore, by carefully choosing priorities, we can eliminate the need for some CFI checks and reduce the security overhead, while still providing the same security guarantees.

This raises the question, how should tasks be prioritized to minimize the number of RECFISH-related CFI checks? We consider a technique we call *task pushing* in which the priority of otherwise safe tasks are increased above the unsafe tasks. While *task pushing* initially seems to have a negative impact on schedulability, we find that the resulting reduction in RECFISH overhead actually increases the number of schedulable task sets.

To test task pushing, we generated additional task sets using the same configurations as discussed earlier, and added an additional parameter for the probability of a task being labeled safe. We considered several distinct values for this probability, 0%, 25%, 50%, and 75%, which was constant for each generated task system. We then used a brute-force algorithm to test all possible combinations of pushed tasks. We find that with task pushing, the percentage of schedulable task sets with RECFISH increases from 85% to 88%, 91%, and 95%, respectively. We also measured through simulation[2], the total amount of overhead observed during a hyperperiod, or the point at which the schedule repeats (the least-common

[2] We assumed for this simulation that the execution time e_i was exact.

multiple of all periods). The average RECFISH overhead observed across all generated task systems was 12%, 8%, and 4%, for safe-task probability of 25%, 50%, and 75%, respectively. This is down from 16% overhead when RECFISH checks are applied to all tasks.

While these results are promising, there are still many open questions. For example, for now we assume the developer provides the safe/unsafe label, but can automated mechanisms (e.g., static program analysis) be leveraged to provide these labels? Can RECFISH-related overheads be further reduced under different schedulers (e.g., non-preemptive unsafe tasks, or other more dynamic scheduling policies)? How should information passing between safe and unsafe tasks be handled? Intuitively, we believe it is simpler and more efficient to secure a few well-defined interfaces between tasks rather than allowing an attacker unfettered access to memory. Further, our analysis assumes that state from one execution of a task does not carry over to subsequent executions. How do we use secure memory regions, e.g., the shadow stack, to safely and efficiently persist that state? Finally, there is potential for greater optimization through careful design by the application developer. For instance, if the developer designs the tasks such that external input is always handled in low priority tasks then the process of task pushing is greatly simplified.

5 Conclusions

CFI schemes are only as secure as the CFG is precise [41, 10, 17, 15]. 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 [10, 17], 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. For example, coarse-grained approaches assign a single label to all legal targets. This imprecision can allow an attacker to achieve Turing-complete computation in the presence of certain instruction sequences [10, 17]. RECFISH mitigates these attacks by using fine-grained labeling and a shadow stack to increase precision.

One potential limitation of RECFISH is application-specific uses of privileged mode execution in tasks, i.e., privileged code that is written by the developer and is not part of RECFISH. In practice, this issue is unlikely to become a barrier to adoption. First, it is uncommon for tasks themselves to have privileged sections (outside of handling hardware interrupts). In the evaluated benchmarks, privileged code was limited to the RECFISH and FreeRTOS code. Second, privileged code in tasks may not be an issue as long as that code omits MPU-sensitive instructions. Specifically, the MPU in Cortex-R can only be modified in privileged mode using the `mcr` and `mrc` instructions. As long as those two instructions only appear in RECFISH code – this is statically verifiable – then RECFISH can rely on its own CFI checks to prevent MPU instructions from being executed outside of normal control flow and, consequently, prevent unwanted modification of the MPU. The proposed ARMv8-R architecture provides another mechanism to address this issue 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.

In summary, RECFISH can be applied to both baremetal and FreeRTOS applications. The defense introduces a minimal amount of program storage and RAM overhead, requiring only 10 bytes of program storage per indirect branch and just 8 bytes per shadow stack operation, and a constant, configurable block of memory for the shadow stacks. Further, in the 85% of task systems where RECFISH can be applied without compromising schedulability, there is no impact on real-time performance.

While this work makes a significant step towards hardening real-time embedded systems, there are many directions for future work. Beyond optimization, future work would benefit from the use of formal methods to analyze the correctness of the CFI instrumentation. Finally, new features in the upcoming ARMv8-R architecture could be leveraged to provide stronger performance and security guarantees for RECFISH as well as other embedded system hardening techniques.

References

1 The Capstone Disassembly Engine. `http://www.capstone-engine.org/`.

2 The Keystone Assembler. `http://www.keystone-engine.org/`.

3 pyelftools. `https://github.com/eliben/pyelftools`.

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

5 Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

6 ARM Limited. Cortex-R4 and Cortex-R4F Technical Reference Manual, 2011.

7 Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *23rd USENIX Security Symposium*, 2014.

8 B. Brandenburg. *Scheduling and Locking Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

9 Nathan Burow, Scott A Carr, Stefan Brunthaler, Mathias Payer, Joseph Nash, Per Larsen, and Michael Franz. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1), 2017.

10 Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, 2015.

11 Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *21st International Conference on Distributed Computing Systems(ICDCS)*. IEEE, 2001.

12 Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-metal Embedded Systems With Privilege Overlays. In *IEEE Symposium on Security and Privacy*, 2017.

13 Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to Protect Return Addresses from Attack. *SIGARCH Computer Architecture News*, 2005.

14 Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Battie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptative Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, 1998.

15 Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014.

16 Richard Earnshaw. Procedure call standard for the ARM architecture. *ARM Limited, October*, 2003.

17 Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

18 Shay Gal-On and Markus Levy. Exploring CoreMark—A benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

19 Jacob Grycel and Robert J. Walls. A Random Number Generator Built from Repurposed Hardware in Embedded Systems. *CoRR*, abs/1903.09365, 2019. `arXiv:1903.09365`.

20 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

21 Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*. IEEE, 2001.

22 Monowar Hasan, Sibin Mohan, Rakesh Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security in legacy hard real-time systems. In *37th IEEE Real-Time Systems Symposium*, RTSS, 2016.

23 J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. ILR: Where'd my gadgets go. In *IEEE Symposium on Security and Privacy*, 2012.

24 T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-Generated Software Diversity. In *Moving Target Defense*, Advances in Information Security. Springer, 2011.

25 Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

26 Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

27 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm, Exact characterization and average case behavior. In *1989 IEEE Real-Time Systems Symposium (RTSS'89)*, December 1989.

28 C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

29 Sibin Mohan, Man-ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. Real-time systems security through scheduler constraints. In *26th Euromicro Conference on Real-Time Systems*, ECRTS, 2014.

30 Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, PLDI, 2009.

31 Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, 2010.

32 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42 (6), 2007.

33 Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

34 James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint*, 2013. `arXiv:1308.5174`.

35 Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh Bobba. A generalized model for preventing information leakage in hard real-time systems. In *21st Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2015.

36 Danbing Seto, John P Lehoczky, Lui Sha, and Kang G Shin. On task schedulability in real-time control systems. In *17th IEEE Real-Time Systems Symposium*, 1996.

37 Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM conference on Computer and communications security*. ACM, 2007.

38 Bryan Ward, Abhilash Thekkilakattil, and James Anderson. Optimizing Preemption-Overhead Accounting in Multiprocessor Real-Time Systems. In *22nd International Conference on Real-Time and Network Systems*, RTNS, 2014.

39 Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Liu Sha. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *22nd Real-Time embedded Technology and Applications Symposium*, RTAS, 2016.

40 Tom Zanussi. microYocto and the internet of tiny. Embedded Linux Conference, 2015.

41 Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*, volume 13, 2013.

42 Vojin Zivojnovic, Harald Schraut, M Willems, and R Schoenen. DSPs, GPPs, and multimedia applications-an evaluation using dspstone. In *International Conference on Signal Processing Applications and Technology*, 1995.

A ARM Indirect Jumps

The following table lists all indirect branch operations in ARM. All such branches much be instrumented to enforce forward-edge control flow integrity.

Table 2 Indirect jump operations in ARM.

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*.

B Selected Source Code

The following source code details the instrumentation used by RECFISH to handle shadow stack operations. Note, this code will jump into the higher privilege mode needed to access shadow stack memory.

Listing 7 Supervisor call handler.

```
do_syscall:
  cpsie   aif                    # Re-enable interrupts
  stmfd   sp!, {r9,r10,r12,lr}   # Store registers
  mrs     r9, spsr               # Working register
  tst     r9, 0x20               # Test if thumb state
  ldrneh  r9, [lr, -2]           # Yes: load halfword
  bicne   r9, r9, 0xFF00         #   and get func num
  ldreq   r9, [lr, -4]           # No:  load word and
  biceq   r9, r9, 0xFF000000     #   and get func num
  ldr     r10, table             # Load address of table
  ldr     pc, [r10, r9, lsl 2]   # Jump to routine

table:
  .word jump_table

jump_table:
  .word ss_push
  .word ss_pop
```

Listing 8 Shadow stack operations.

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

# Input: void
# Returns: value at top of shadow stack -> lr
.type ss_pop, ss_pop:
        ldr r9, current_ss_const  # Load stack pointer
        ldr r10, [r9]             # Load stack top
        ldmfd r10!, {lr}^         # Pop into user mode lr
        str r10, [r9]             # Store new stack top
        exit_syscall              # Syscall exit macro

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