Development Toolchain and Platform for Writing MCU-Based Embedded System Software

tomoaki kawada

Abstract

Microcontrollers (MCUs) are ubiquitous in today's embedded systems. Their use cases run the gamut from primary processing units in small appliances to internal controllers of application-specific ICs (such as storage controllers). Their increasing processing power allows for larger, more highly-integrated applications that were once only tractable by general-purpose systems. However, with that come all the problems general-purpose systems used to have (and are still having, to some extent), such as development costs and quality issues.

Embedded software uses a technique called partitioning to isolate software faults. Safety-critical system developers (e.g., automotive system developers) find it attractive because they need to integrate software provided by multiple sources, and it allows them to prevent software faults from affecting safety-critical software components. Unfortunately, partitioning has been shunned outside the safety-critical circles because it adds extra costs, increases a runtime overhead, and breaks code when misused. Memory protection is the essence of partitioning and has conventionally been implemented using a processor's ring protection mechanism. However, owing to the mechanism's highly generic nature, this approach has led to a significant software overhead.

A perpetual problem in engineering is the development cost. Component-based development (CBD) attempts to address this issue by dividing the system into separate components and enabling code reuse on a by-component basis. This allows for large and complex software to be constructed efficiently from reusable components, significantly reducing development costs and time. This benefit has led to an increasing interest in CBD by embedded system developers. A *component system* is a software framework that standardizes the component interface and the method of combining components to build a functioning system. The TOPPERS Embedded Component System (TECS) is an embedded-oriented component system designed to be integrated into μITRON 4.0-style RTOSes' configuration systems. However, the support for the TOPPERS third-generation kernels, including TOPPERS/ASP3 and TOPPERS/HRP3, was impeded by their updated API design to accommodate partitioning.

The embedded system landscape is ever-changing. The latest change is the emergence of connected devices or the so-called "Internet of Things", aiming to optimize embedded systems' activity by the seamless connection between embedded systems offered by the broad availability of an Ethernet connection. With this seamless connectiv-

ity, embedded system security becomes a concern since a vulnerability in embedded systems can be devastating. As such, it would be prudent to leverage the abundant security knowledge acquired through general-purpose systems in embedded systems.

This dissertation takes on three research topics. The first topic explores how to add memory protection support to an existing operating system by utilizing TrustZone for Armv8-M, a hardware-assisted security feature for microcontroller-based embedded systems. During the process, we identify and perform a qualitative comparison of three possible system configurations for TrustZone for Armv8-M. Based on one of such configurations, the SBI (single binary image) scheme, we develop ASP3+TZ, a memory-protection-enabled operating system, by modifying an existing operating system named TOPPERS/ASP3, which does not have memory protection. Finally, we show that the proposed method achieves memory protection at much lower overhead while offering the almost same level of memory isolation as existing operating systems.

The second topic proposes *TZmCFI*, a lightweight control-flow integrity (CFI) scheme for RTOS-based applications. CFI is a class of defensive techniques against control-flow attacks such as Return-Oriented Programming. Although it has been widely deployed in general-purpose systems, such existing implementations are inapplicable to small embedded systems because of architectural differences. TZmCFI embodies variants of several existing CFI techniques to provide a self-contained toolset for building an instrumented application. The toolset is comprised of a modified LLVM-based compiler and a runtime library called *Monitor*, which is isolated from untrusted code using TrustZone for Armv8-M. The modified LLVM code generator implements the traditional shadow stack technique by inserting calls to Monitor. Monitor wraps the application's interrupt handlers to protect them by *shadow exception stacks*, a variant of the traditional shadow stack technique. The performance evaluation indicates that our shadow exception stack implementation's runtime overhead is moderate if not significant, whereas our shadow stack implementation incurs an overhead lower than previous works.

The third topic presents a method to componentize the time event notifications from the TOPPERS third-generation kernel for the TOPPERS Embedded Component System (TECS). TECS supports the generation of static API statements, but it is limited insofar as it cannot generate certain complex statements, including those of the time event notifications. Therefore, we propose a TECS generator plugin that generates the complex static API statements required by the time event notifications. We evaluate the proposed time event notification component to demonstrate its runtime efficiency and establish the proposed plugin's utility.

Acknowledgements

I would like to express my sincere respect and gratitude to Professor Hiroaki Takada and Associate Professor Shinya Honda for giving invaluable suggestions and directions through the work.

Furthermore, I would like to thank everyone in Embedded and Real-Time Laboratory, in particular, Professor Yutaka Matsubara, Takuya Ishikawa, and Li Yixiao, as well as Associate Professor Takuya Azumi at Osaka University (presently at Saitama University); and Hiroshi Oyama at Okuma Corporation for their help and advice.

The work on TECS was made possible through a collaboration with the TOPPERS Project TECS working group. I would like to thank everyone in the working group for their helpful comments and suggestions.

Last but not least, I would like to thank my family, who provided me with their support and encouragement.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Microcontrollers (MCUs) are ubiquitous in today's embedded systems. Their use cases run the gamut from primary processing units in small appliances to internal controllers of application-specific ICs such as storage controllers. They started as 4-bit processors, but with constantly improving semiconductor technologies, 32-bit microcontrollers are becoming increasingly popular, and the applications of microcontrollers are still broadening. A microcontroller equipped with the most powerful MCU core, Cortex-M7 running at 1GHz is in development at the point of writing [78].

Their increasing processing power allows for larger, more highly-integrated applications that were once only tractable by general-purpose systems. However, with that come all the problems general-purpose systems used to have (and are still having, to some extent), such as development costs and quality issues. Embedded system developers devised their own solutions to suit the requirements specific to embedded systems and applied them to some success.

The first example of such efforts is partitioning, referring to the use of memory protection and access control techniques to isolate software faults. Safety-critical system developers (e.g., automotive system developers) find it attractive because they need to integrate software provided by multiple sources, and it allows them to prevent software faults from affecting safety-critical software components. The use of partitioning in safety-critical systems is also driven by functional safety standards, such as IEC 61508 [34], ISO 26262 [95], and DO-178C [86]. Under these standards, all software and hardware in a system must be developed to meet the highest safety level required by the system. Partitioning relaxes this requirement and allows for a significant reduction in the verification cost of untrusted third-party software components, provided that they are temporally and spatially isolated by an isolation mechanism and that the isolation mechanism itself is certified for safety. Unfortunately, outside the safety-critical circles, many developers have shunned partitioning because it adds extra costs, increases a runtime overhead, and breaks code when misused [112].

The second example is component-based development. A perpetual problem in engineering is the development cost. Component-based development attempts to address this issue by dividing the

system into separate components and enabling code reuse on a by-component basis. This allows for large and complex software to be constructed efficiently from reusable components, significantly reducing development costs and time [28]. A *component system* is a software framework that standardizes the component interface and the method of combining components to build a functioning system. Although many component systems are available, not all are suitable for embedded systems, owing to the tight design constraints (e.g., runtime/memory overhead) imposed by embedded systems [37]. There are several component systems amenable to embedded system development [41, 60, 67]. The TOPPERS Embedded Component System (TECS) [24] is one of such systems and is designed to be integrated into μITRON 4.0-style RTOSes' configuration systems. The RTOS integration has been successfully ported to TOPPERS/ASP [24] and TOPPERS/HRP2 [53] so far. However, the support for the TOPPERS third-generation kernels, including TOPPERS/ASP3 and TOPPERS/HRP3, was impeded by their updated API design to accommodate partitioning.

The embedded system landscape is ever-changing. The latest change is the emergence of connected devices or the so-called "Internet of Things" (IoT), aiming to optimize embedded systems' activity by the seamless connection between embedded systems offered by the broad availability of an Ethernet connection [29]. With this seamless connectivity, embedded systems' security becomes a concern. The impact of insecure embedded systems can be widespread and severe [23, 33, 48, 56, 61, 70, 91]. Embedded systems can continue operating for years after their vulnerabilities are disclosed [84] without possibility of applying security patches [58], suggesting the use of a defense-in-depth approach is necessary. Embedded systems deployed to remote locations must also resist any tampering attempts by their physical owners. Smart meters and biometric authentication mechanisms are good examples. There is a subtle difference between security and the traditional notion of safety; security refers to "the protection of a system against undesired access or usage," whereas safety means "[a system] behaves as specified with absence of unreasonable risk due to hazards caused by malfunctioning behavior" [31]. Nevertheless, safety and security go "hand-in-hand" [31] and can often be achieved through common tools. With this recent trend, traditional safety techniques like partitioning are increasingly important. Meanwhile, it would be prudent to leverage the abundant security knowledge acquired through general-purpose systems in embedded systems.

Unlike desktop and mobile systems, there are no central entities that control the development platforms for embedded systems. With so much diversity, fragmentation, potential risks caused by change,

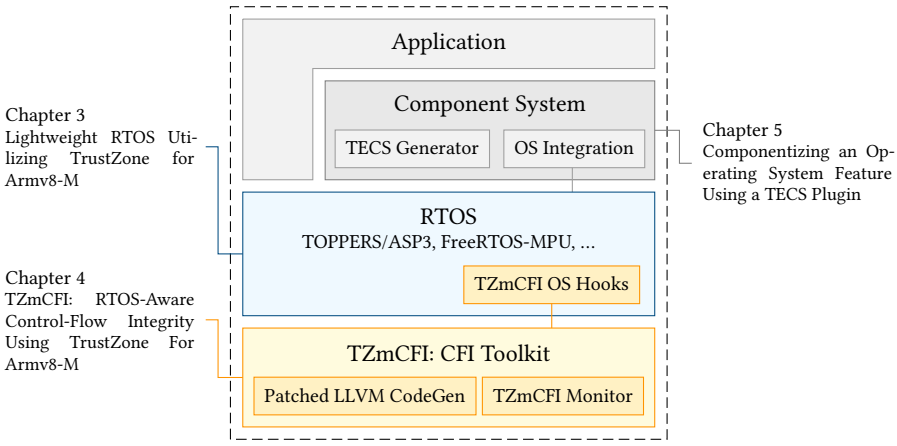


Figure 1.1: Each chapter's locus within a system.

and stringent non-functional requirements, and no central entities to force the evolution, embedded systems have lagged in many aspects, such as employed security techniques [17], choices of programming languages [52, 83], and software development methodologies [52]. While they are often taken as laziness or ignorance, some are genuine needs arising from embedded systems' nature [52]. Still, the ongoing trend is increasingly blurring the border between traditional embedded systems and connected systems. Embedded developers are vastly underequipped to deal with the cyber threats that the change brings about.

1.2 Contributions of the Dissertation

Our research takes on three research topics, each targeting different domains in a system, to improve the status quo (Fig. 1.1):

Lightweight RTOS Utilizing TrustZone for Armv8-M

Arm architecture is used in many embedded systems. Its prominent feature is TrustZone, which divides a whole system into Secure and Non-Secure "worlds." While it has been widely used for application processors, its full potential is just starting to be seen for microcontrollers because it was only 2018 when the first microcontroller supporting TrustZone was released, and microcontrollers use Arm M-profile, which uses a significantly different instruction set extension to support TrustZone. This microcontroller variant of TrustZone is called TrustZone for Armv8-M (Section 2.2).

Conventional embedded operating systems have implemented memory protection mechanisms with ring protection mechanisms, which, however, has led to a non-trivial overhead owing to the ring

protection mechanisms' highly generic and minimal nature. In this study, we explored a way to add the memory protection support to an existing operating system by utilizing TrustZone for Armv8-M. To achieve low-overhead memory protection, we propose a method that uses TrustZone in place of an MPU, which has been used in traditional operating systems, to significantly lower the inter-domain transition overhead involved in service calls and interrupt handling. Finally, we show that the proposed method achieves memory protection at much lower overhead while offering the almost same level of memory isolation as existing operating systems.

The contributions of this work are shown below:

- We identify and perform a qualitative comparison of three possible system configurations for TrustZone for Armv8-M, namely, Secure Library, Dual Operating System, and Single Binary Image architectures (Section 3.1.2).
- Based on one of such configurations, Single Binary Image architecture, we develop ASP3+TZ, a memory-protection-enabled operating system, and present its design (Sections 3.2 and 3.3).
- We conduct an experimental evaluation on ASP3+TZ and show a quantitative performance of a TrustZone for Armv8-M-based RTOS for the first time and confirm the superior performance characteristics of the SBI scheme compared to traditional memory protection schemes (Section 3.4). Furthermore, we provide a performance evaluation of the security mode transition of TrustZone for Armv8-M for the first time.
- We demonstrate that the SBI scheme accomplishes memory protection with only a few modifications to an existing operating system by comparing the number of lines of the code of ASP3+TZ to that of ASP3 (Section 3.4.3).

TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone For Armv8-M

In addition to the above, we propose the use of TrustZone for Armv8-M for control-flow integrity (CFI).

CFI is a class of defensive techniques against control-flow attacks such as Return-Oriented Programming. We propose a lightweight CFI scheme for RTOS-based applications, *TZmCFI*, which utilizes TrustZone for Armv8-M, a hardware-assisted security feature for embedded systems with tight resource constraints. TZmCFI embodies several existing CFI techniques to provide a self-contained toolset for building an instrumented application. The toolset is comprised of

a modified LLVM-based compiler and a runtime library called *Monitor*. The modified LLVM code generator implements the traditional shadow stack technique by inserting calls to Monitor. To protect exception handlers, Monitor replaces an application's exception vector table and wrap interrupt handlers with *exception trampolines*, which implement *shadow exception stack*, a variant of the traditional shadow stack technique.

The main contributions of this work are the following:

- We discuss the implementation techniques used in the build toolchain and the runtime library of TZmCFI in detail (Section 4.4).
- We propose *safe shadow exception stacks*, a variant of the traditional shadow stack technique that leverages TrustZone for Armv8-M and carefully avoids a caveat associated with Armv8-M's exception handling to protect exception handlers (Section 4.2). We implement this technique in TZmCFI.
- We evaluate the prototype system based on TZmCFI from a performance point of view using a mass-production commercial chip, NXP Semiconductors LPC55S69 (Section 4.5).

Componentizing an Operating System Feature Using a TECS Plugin

TECS enables integration into μITRON 4.0-style RTOSes' configuration systems by the "factory" feature. This feature allows the TECS generator to produce kernel configuration files according to template strings embedded in component definitions. Thus, it can be used to *componentize* kernel objects. However, this feature is limited in the complexity of the statements it can generate. Particularly, it cannot handle the time event notifications, which were introduced in the TOPPERS third-generation kernels to improve time event objects' usability in partitioned systems.

Therefore, we extend the TECS generator's functionality to componentize the time event notifications. We propose a TECS generator plugin that generates the complex static API statements required by the time event notifications. We evaluate the componentized time event notifications' overhead to demonstrate the proposed plugin's utility.

The contributions of this work are as follows:

- We design a TOPPERS/ASP3 time event notification component for TECS, considering usability and execution efficiency (Sections 5.2 and 5.4).

- We design and implement a TECS generator plugin to realize the component design (Sections 5.2 and 5.3).
- We evaluate the componentized time event notification, showing that the runtime overhead is considerably low (Section 5.5).

1.3 Outline of the Dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 contains background information on embedded software development, TECS, the Arm architecture, and control-flow security. Chapter 3 proposes the design of a memory-protection-enabled operating system that leverages TrustZone for Armv8-M (Section 2.2) and shows the experimental evaluation result of its prototype implementation, named *ASP3+TZ*. Chapter 4 presents TZmCFI, a holistic CFI solution for microcontroller-based embedded systems that utilizes TrustZone for Armv8-M. It describes TZmCFI's detailed design as well as design choices involved and evaluates its runtime overhead. Chapter 5 proposes an extension to TECS to fully accommodate the partitioning scheme supported by the TOPPERS Third-Generation Kernel Specification. Finally, Chapter 6 provides a summary and future direction, concluding this dissertation.

2

Background

2.1 Embeddded System Development

Embedded systems are computer systems designed to perform dedicated functionalities within larger systems. They are used everywhere, ranging from a coffee maker's pump controller to a smartphone's wireless chip to a pacemaker to an anti-air missile. Today's embedded systems fall into one or more of the following categories: safety-critical systems, for which malfunctions cause serious property damage, personal injury, or even death; cyber-physical systems, which interact with real-world entities; real-time systems with strict timing requirements; battery-powered systems; and connected systems, which interact with the outside world or other systems through the Internet, commonly dubbed as IoT devices.

In general, embedded systems are characterized by non-functional requirements, such as being able to respond to stimuli in a bounded time and consuming little power. In high-volume consumer products, the available computing resources are limited because of their cost sensitivity [57]. With their evolving connectivity, they are also expected to satisfy security requirements, which are often taken lightly [33] as they come into conflict with other, more immediate requirements.

Embedded hardware's diversity runs from 8-bit microcontrollers to full-fledged PC hardware. A significant portion of embedded systems run on 8-, 16-, or 32-bit microcontrollers and are sometimes referred to as *deeply embedded systems* [58]. It is this domain that IoT devices are centered around, yet where exploit mitigations are mostly absent [17].

2.1.1 *Real-time Operating Systems*

Operating systems streamline software development by providing hardware abstraction and time-sliced multitasking. RTOSes (real-time operating systems) are designed for real-time systems, which are often also resource-constrained at the same time. The notable examples include ChibiOS/RT, eCos, FreeRTOS, µC/OS, QMX, RTEMS, T-Kernel, TOPPERS, and VxWorks.

The µITRON specification was developed as an RTOS specification for small systems with 8- or 16-bit microcontrollers. It has enjoyed great popularity in the embedded market and was considered "[t]he

single, most successful RTOS in Japan" by some accounts [47]. It has spawned several successors, including T-Kernel and the TOPPERS kernels.

The TOPPERS kernels are a family of open-source RTOS kernels developed by the TOPPERS project [76]. Since the release of TOPPERS/JSP [75], a μITRON4.0-compliant real-time kernel, the TOPPERS project has striven to expand the scope by incorporating new features, such as partitioning, multiprocessing, and component-based development, and adapting to modern embedded system hardware and functional safety standards. The following list shows some of the kernels that have been developed based on the TOPPERS project's next-generation specifications:

- *TOPPERS/HRP2* [74] implements the TOPPERS New-Generation Kernel Specification [98]. It supports memory protection and time protection for partitioning (Section 2.1.2).
- *TOPPERS/ASP3* [73] is the smallest implementation of the TOPPERS Third-Generation Kernel Specification [100] and excludes partitioning.

FreeRTOS [20] is a popular open-source RTOS targeting microcontrollers. It emphasizes usability and portability; it supports hardware similar to what μITRON4.0 targets, but unlike μITRON4.0 and AUTOSAR, it lacks support for build-time kernel object creation (Section 2.1.4). As such, in some way, its design direction lies somewhere between μITRON4.0 and POSIX. FreeRTOS does not support partitioning, but its MPU ports [68] enable memory protection in a retrofitting fashion.

2.1.2 *Partitioning*

An operating system allows multiple software subsystems to co-exist in a single system, but because of software bugs, they can interfere with other subsystems' operation in undesired ways. There are two kinds of such interference: *spatial interference*, in which a misbehaving program illegally accesses another subsystem's memory or kernel objects, and *temporal interference*, in which a subsystem changes another subsystem's timing behavior, e.g., by spending too much CPU time in its interrupt handler. In these ways, a software fault in one subsystem can propagate and disrupt the entire system's correct operation.

Partitioning is an operating system feature included in many operating system specifications [15, 100, 114] to prevent or detect such interference between subsystems and contain the damage of software faults. Partitioning divides a software system into *protection*

domains (or simply *domains*) defined by the application developer. To ensure spatial isolation, the operating system enforces kernel object access policies through runtime checks and restricts each protection domain's memory access permission (*memory protection*). To ensure temporal isolation, the operating system strictly schedules each protection domain's execution according to a predetermined timing policy (*time protection*). The domain with unrestricted access to system resources is called a *system domain*, distinguishing itself from the other domains called *user domains*.

Memory protection is typically implemented using a hardware unit called an MMU (memory management unit), which virtualizes the memory space and assigns memory permission bits by page table-based address translation. A processor with an MMU also has a notion of *privilege modes*; a privileged mode allows full access to all processor features, hardware, and memory space, and an unprivileged mode only has limited access to them. These mechanisms to implement memory protection—an MMU and privilege modes—are collectively called a *ring protection mechanism*. An operating system utilizes a ring protection mechanism by executing system-domain code and user-domain code in the privileged mode and the unprivileged mode, respectively, and reconfigures the MMU for each user domain.

The page table walk done by an MMU is a time-consuming process; as such, an MMU is usually accompanied by a TLB (translation lookaside buffer) to cache the recent translation results, but this sacrifices timing determinism and makes it unsuitable for real-time systems. Furthermore, small embedded systems do not need memory virtualization, as they need only one address space. For this reason, processors designed for such systems have a more specialized hardware unit called an MPU (memory protection unit) in place of an MMU. An MPU does not perform address translation and uses a fixed number of memory regions represented by a set of hardware registers to assign memory permission bits. Thus, it "cuts out" a portion of the address space for each user domain. In the Armv8-M architecture [12], each MPU memory region is represented by a 32-byte-aligned continuous range of memory addresses, but other architectures may impose different requirements or use more complex representations.

To apply memory protection, protected memory regions containing variables and functions must be arranged following the MMU's or MPU's requirements. This gets complicated by the fact that each MMU and MPU has unique requirements, which are sometimes not expressible by a linker script; e.g., Armv7-M's MPU imposes an alignment requirement dependent on the region size. Some operating systems such as FreeRTOS-MPU require application developers to handle this manually. Others, such as TOPPERS/HRP2 [118], automate this process by a sophisticated kernel configuration system.

2.1.3 *System-Level Isolation*

The increasing complexity and connectivity of computer systems have impelled the need to protect subsystems involved in sensitive tasks, such as digital rights management, credential storage, cryptography, remote attestation, and biometric authentication. With ever-increasing software complexity and so many attack vectors present in today's complex processors like DMA attacks and debug ports, software-only techniques and existing ring protection schemes proved insufficient to protect such subsystems. This has given rise to tamper-resistant isolated processing environments, called *trusted execution environments* (TEEs) [90].

TEEs are implemented on top of hardware isolation mechanisms. Arm TrustZone [5] divides a whole system into Secure and Non-Secure domains or "worlds[1]." TrustZone support in Arm processors is implemented as a specialized sort of virtualization, where the two worlds are mapped to virtual CPU cores, each represented by a unique processor mode (Secure and Non-Secure modes) and executing in a time-sliced fashion. Arm TrustZone for Armv8-M, recently announced by Arm company, enabled system-level isolation and TEE implementation in widely-available microcontrollers.

There are other such mechanisms, though most of them either exclude microcontroller-based systems or require specialized hardware. MIPS32/64 Release 5 and later provides MIPS-VZ virtualization extension, which is conceptually similar to TrustZone. MIPS-VZ supports up to 255 guest domains and can distribute interrupts to guest domains without software intervention [102]. However, its memory protection scheme is built on an MMU and unsuitable for real-time systems and small embedded systems. Intel Software Guard Extensions (SGX) [69] allows applications to create protected regions called *enclaves* but targets servers and desktop systems. TrustLite [55] targets small microcontroller-based systems such as those based on MSP430 but requires specialized hardware.

2.1.4 *Kernel Configuration*

Embedded software is usually designed to provide a fixed set of functionalities throughout its lifetime. To optimize for specific use cases and minimize the overhead, many operating systems, such as Linux and FreeRTOS [20], support disabling unnecessary features at build time. Those designed for microcontrollers and real-time systems, such as AUTOSAR [15], OSEK/VDX, μITRON4.0 [51], the TOPPERS kernels [100], Drone [40], and RTFM (Real-Time for the Masses) [43,

1 In TrustZone's context, "domains" and "worlds" are often used interchangeably.

66], go further and define all kernel objects at build time. This "static operating system" design is beneficial in many ways:

- Because all kernel objects have their memory allocated at compile-time, there is no possibility of runtime allocation failures or memory leaks.
- This design reduces RAM consumption by moving constant data to ROM.
- This design reduces ROM consumption by not having a dynamic memory allocator.
- The kernel can optimize its internal structures based on the configuration information.
- The kernel can deduce the correct configuration or validate the given configuration based on global information. For example, RTFM automatically calculates each mutex's priority ceiling based on which task can access the mutex's protected resource.

On the other hand, this design complicates the application's build process by introducing additional build steps.

The syntax of a kernel configuration is specific to each operating system specification. For example, OSEK/VDX uses a configuration file written in OIL (OSEK Implementation Language). µITRON4.0 and its derivatives use their own language called *static API*, designed to be familiar to C programmers [116].

We will use TOPPERS/ASP to illustrate the configuration process from a ten-thousand-foot view. In this operating system, a program called a *kernel configurator* is responsible for processing a given kernel configuration. An application developer writes a kernel configuration file in the µITRON4.0 configuration syntax. Fig. 2.1 shows an example of this. The kernel configurator analyzes the kernel configuration and generates a compiled configuration file (Fig. 2.2), which is a C source file including the global variable definitions of the defined kernel objects. The kernel code refers to these variables but leaves their definitions undefined. Linking to the compiled configuration file fills these holes, and the kernel becomes fully functional. The kernel configurator also outputs the mapping between object names and assigned object IDs in the form of a C header file (Fig. 2.3), which application code can use to refer to the defined objects, as in `sus_tsk(LOGTASK)`.

2.1.5 *Compilers*

C is still the lingua franca in embedded system development, and therefore a C compiler exists for almost every processor architecture.

```
INCLUDE("syssvc/logtask.cfg");
#include "sample1.h"
CRE_TSK(MAIN_TASK, { TA_ACT, 0, main_task, LOW_PRIORITY,
    STACK_SIZE, NULL });
```

Figure 2.1: An example of a TOPPERS/ASP kernel configuration.

```
static STK_T _kernel_stack_LOGTASK[COUNT_STK_T(
    LOGTASK_STACK_SIZE)];
static STK_T _kernel_stack_MAIN_TASK[COUNT_STK_T(STACK_SIZE)];

const TINIB _kernel_tinib_table[TNUM_TSKID] = {
    { (TA_ACT), (intptr_t)(LOGTASK_PORTID), ((TASK)(logtask_main
        )), INT_PRIORITY(LOGTASK_PRIORITY), ROUND_STK_T(
        LOGTASK_STACK_SIZE), _kernel_stack_LOGTASK, (TA_NULL), (
        NULL) },
    { (TA_ACT), (intptr_t)(0), ((TASK)(main_task)), INT_PRIORITY
        (LOW_PRIORITY), ROUND_STK_T(STACK_SIZE),
        _kernel_stack_MAIN_TASK, (TA_NULL), (NULL) }
};
```

Figure 2.2: The compiled configuration file `kernel_cfg.c` generated from Fig. 2.1.

```
#define LOGTASK     1
#define MAIN_TASK   2
```

Figure 2.3: The header file `kernel_cfg.h` generated from Fig. 2.1.

Unlike general-purpose systems, the choices of processor architectures used in embedded systems are not narrowed down to two or three because binary compatibility is not a big concern. Some processor architectures offer a very specialized programming model to the extent that they must be programmed with dedicated programming languages; e.g., XC [110] is a concurrent programming language based on C, designed for use with the XMOS architecture.

Well-recognized processor architectures, such as Arm, AVR, MIPS, PIC, and x86, provide the widest choices of compiler suites, at least of them being an open-source compiler. GCC (GNU Compiler Collection) is the most popular choice of an open-source compiler, and many processor vendors maintain forks of GCC for their processor architectures.

LLVM [62] and Clang, which are another open-source compiler toolchain, are gaining traction. They are licensed under the lenient MIT license, which has led companies to incorporate a part or whole of the LLVM infrastructure into their build toolchains. Clang has been notable for outstanding standard compliance with the latest C++ standards. Since Apple's involvement in their development in 2005, they (especially, Clang) have been used in many places, including Apple's own official development tools for macOS and iOS, Arm Compiler 6, AMD Optimizing C/C++ Compiler, AMD GPUs' graphics drivers, the PlayStation 4 console's official SDK, and Intel C++ Compiler. What characterizes LLVM and Clang is their excellent modularity and clear code structures; for this reason, they have been a popular platform for experimenting with new research ideas. [19, 35, 80, 106, 111] are examples of works based on LLVM.

There are emergent programming languages that are promising candidates for embedded system developments, such as Rust [104] and Zig [105]. However, their uses in safety-critical systems are impeded by their lack of standardized language specifications, code quality standards such as MISRA-C, and a reliable compiler implementation such as CompCert [63]. Many such languages use LLVM for code generation, but LLVM only supports a handful of processor architectures, which is another limiting factor of their use in embedded systems.

2.2 The Armv8-M Architecture

The Arm architecture is a family of instruction set architectures (ISAs) used in a wide variety of computers, such as smartphones, servers (AWS Graviton), supercomputers (Fujitsu Fugaku), game consoles (3DO, Nintendo Switch), and personal computers (the Acorn Archimedes, Raspberry Pi, Apple personal computers since 2020) as well as all sorts of embedded systems. The latest version Armv8 defines three architecture profiles: A-profile (application), R-profile (real-time), and M-profile (microcontrollers).

Arm M-profile (we will refer to it as Arm-M) is the smallest profile defined by the Arm architecture. Arm-M is highly specialized for use in microcontrollers; it only supports the Thumb instruction set (16-/32-bit compressed instructions), and an interrupt controller called an NVIC (nested vectored interrupt controller) is tightly integrated into the ISA. Arm-M defines some extensions, such as hardware floating-point arithmetics and a DSP instruction set, which is essentially a SIMD instruction set that uses existing 32-bit GPRs.

The Armv8-M architecture [12] is the latest architecture based on Arm M-profile. One of the prominent features of Armv8-M is the addition of support for TrustZone, a technology that allows hardware-enforced system-level isolation between trusted and untrusted components. Armv8-M incorporates itself into a TrustZone-based system through an extension to the instruction set, called *Cortex-M Security Extensions*[2] (CMSE).

TrustZone support was originally exclusive to Arm A-profile, which has optionally supported the Security Extensions since Armv6-A. Since Arm A-profile's Security Extensions has been around for quite a while, it found many use cases [46, 115]. In contrast, CMSE is still early in its days and leaves its full potential on the table. CMSE is strikingly different from the Arm A-profile Security Extensions, so most existing techniques are inapplicable to CMSE or require significantly different implementations.

2.2.1 *Execution Modes*

Arm-M provides two PE modes: *Handler mode* and *Thread mode*. The processor runs in Thread mode after reset, and normal code executes in this mode. Interrupt handlers execute in Handler mode. The processor automatically transitions to Handler mode upon taking an exception and transitions back to Thread mode when all active excep-

2 TrustZone for Armv8-M is a system feature that divides a whole microcontroller into two domains, whereas CMSE is a processor feature that assigns an execution environment to each domain. In the context of software running under CMSE, their distinction is insubstantial, and therefore they are often used interchangeably.

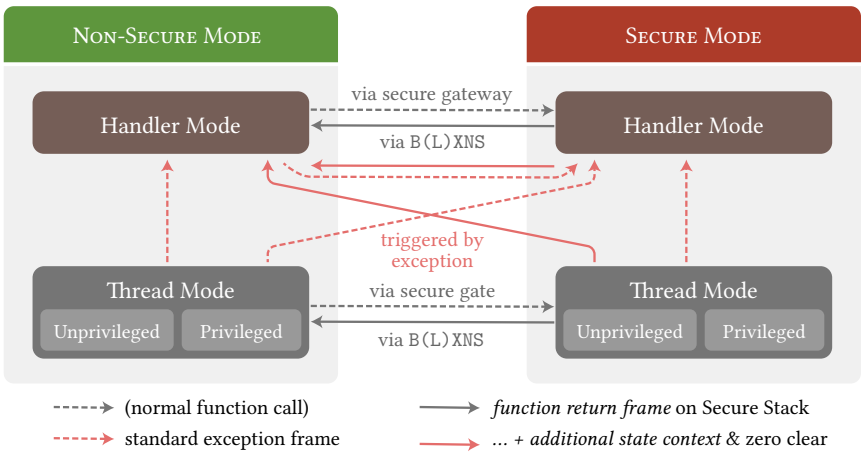


Figure 2.4: Possible transitions between Armv8-M processor states.

tion handlers[3] complete execution and return to the interrupted code. Arm-M provides two privilege modes: *Privileged mode* and *Unprivileged mode*. In Thread mode, the `CONTROL.nPriv` register determines the current privilege mode. In Handler mode, the current privilege mode is unaffected by this register, and the processor always runs in Privileged mode.

CMSE introduces the concept of security modes, which correspond to the Secure and Non-Secure "worlds" of TrustZone and are accordingly called *Secure mode* and *Non-Secure mode*. The security modes are orthogonal to the existing execution modes (Fig. 2.4). The processor determines the current security mode based on the security attribute of the memory where the currently-running code is located. Mode transitions are rigorously checked by the hardware and only permitted in a controlled manner such as through *secure gateways*, which are white-listed Secure entry points that can be called by Non-Secure code (Section 2.2.3), and through exceptions[4], for which the processor automatically transitions to the taken exception's associated security mode and performs extra steps to guarantee security (Section 2.2.4). The checks are mostly transparent to software and incur a minimal overhead, making TrustZone an excellent platform for implementing security mechanisms.

The concept of Secure and Non-Secure worlds extends not only to a processor core but also to the entire system. Memory transactions are tagged with security modes at a hardware level. Peripheral access controllers enforce security checks on incoming memory transactions and reject illegal memory accesses to devices that are marked as Secure, thus protecting Secure peripherals and memory against DMA attacks.

3 There can be more than one because of nesting.

4 In Arm-M, exceptions refer to both of software-generated traps and hardware-generated interrupts.

2.2.2 *Memory Protection*

Arm-M supports an MPU but not an MMU. In addition to an MPU, CMSE defines an SAU (security attribution unit) and IDAU (implementation-defined attribution unit) to assign security attributes to memory. The SAU is optionally integrated into the processor core and assigns security attributes in a similar manner to the MPU. It can define an implementation-defined number of memory regions and assign one of the following security attributes to each region:

- A *Secure* region can only be accessed in Secure mode. Memory transactions in this region are tagged as Secure.
- A *Non-Secure Callable* region has the same properties as a Secure region except that Non-Secure code can branch into this region. The admissible branch targets must be marked as *Secure gateways* by `sg` instructions (Section 2.2.3).
- A *Non-Secure* region can be accessed by both modes. Memory transactions in this region are tagged as Non-Secure.

The IDAU is an external hardware unit connected to the processor core and assigns security attributes in an implementation-defined manner.

Note that the SAU and the IDAU only affect access permission checks done *within* the processor and memory transactions generated by the processor. Peripherals in a TrustZone-based system are usually equipped with peripherals protection controllers [10] or implementation-defined hardware units to filter out memory transactions with invalid security attributes. In some designs, the IDAU and peripheral-side protection controllers use bit 28 of memory addresses to define Secure and Non-Secure aliases of each peripheral (e.g., in an LPC55S69 microcontroller [77], `0x20000000` and `0x30000000` are respectively designated as the Non-Secure and Secure base addresses of SRAM0).

CMSE adds the `tt` (test target) instruction, which evaluates the security and MPU attributes of a given memory address. This instruction also returns region numbers unique for each continuous MPU and security region. A program can validate a range of memory addresses by executing this instruction for each endpoint and checking if the returned region numbers are identical.

2.2.3 *Inter-world Function Calls*

CMSE assists with inter-world function calls by hardware (Fig. 2.4). Such calls can be done by standard function call instructions such

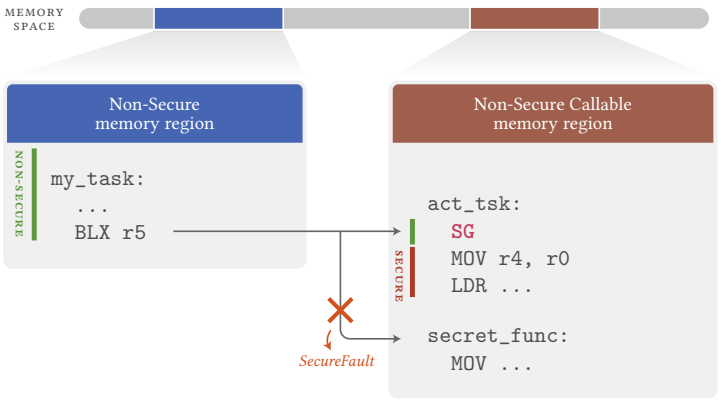


Figure 2.5: Calling into a Non-Secure Callable region.

as `blx` (branch with link and exchange) or the new Non-Secure call instruction `blxns`.

Branching into a Non-Secure Callable region triggers a transition to Secure mode. The valid branch targets must be predesignated by `sg` (Secure gateway) instructions (Fig. 2.5). Branching to other instructions in a Non-Secure Callable region or anywhere in a Secure region generates SecureFault. Returning to the caller uses a `bxns lr` instruction instead of the standard `bx lr` instruction.

Calling a Non-Secure function from Secure mode is done by a `blxns` instruction. Non-Secure function calls receive special treatment to ensure the corresponding function returns can be done safely. Usually, a function call stores the return target to the `lr` register, which could be corrupted by Non-Secure code to divert the control flow to an arbitrary location in Secure code. Therefore, instead, the processor pushes the return target and `RETPSR` to the Secure stack and stores a token called `FNC_RETURN` to the `lr` register. When the Non-Secure function finally returns, the branch to `FNC_RETURN` triggers a function return sequence in which the processor validates `RETPSR` and branches to the return target stored in the Secure stack.

As a rule of thumb, branching to Non-Secure code uses the new Non-Secure branch instructions (`bxns`, `blxns`) in place of the standard branch instructions. This design prevents faulty Secure code from transitioning to Non-Secure mode inadvertently and leaking confidential information via general-purpose registers.

2.2.4 *Exceptions*

This section mainly focuses on the security mode transition aspect of the Armv8-M exception handling model and saves some intricate details for Section 4.1.1, where they are more relevant.

When an exceptional event occurs, the processor interrupts the current execution, fetches the corresponding exception handler ad-

dress from an exception vector table, and transfers the control to that handler. CMSE introduces two separate exception vector tables for Secure and Non-Secure exception handlers. The handler address is fetched from the vector table corresponding to the exception's associated security mode. Software exceptions, such as BusFault and SVCall, are associated with the faulting security mode on their occurrence. Interrupts are assigned to a security mode according to NVIC's configuration registers. There are two exceptions to this rule: Firstly, an Armv8-M processor can have up to two system timers (also known as SysTick), which are statically assigned to security modes. Secondly, SecureFault and Reset are always taken in Secure mode.

An exception can cause a security mode transition in any direction (Fig. 2.4). Secure-to-Non-Secure transitions must be implemented carefully not to compromise security. For example, Arm-M's standard exception entry sequence leaves general-purpose registers intact, which could allow a Non-Secure handler to observe the interrupted Secure program's internal state. Also, Arm-M's standard exception return sequence interprets whatever is sitting on the stack specified by `EXC_RETURN` as an exception frame, which could allow Non-Secure code to "fake" an exception return. Armv8-M uses extended exception entry and return sequences to address this issue.

The standard entry sequence pushes an exception frame containing the values of `r0`–`r3`, `r12` to the interrupted mode's stack. The extended entry sequence extends the exception frame to include all unbanked general-purpose registers. Furthermore, it clears all unbanked general-purpose registers to hide their contents from the Non-Secure handler. Finally, it annotates the exception frame with an integrity signature. When interpreted as a memory address, the integrity signature refers to a non-executable location, so it is guaranteed to be distinct from the return address pushed by a Secure-to-Non-Secure function call (Section 2.2.3).

The entry sequence stores a special value called `EXC_RETURN` to the `lr` register. This value indicates the interrupted PE mode, security mode, and which of the main stack and the process stack contains the exception frame. The return sequence uses this value to locate the exception frame and return to the appropriate modes. The extended return sequence kicks in when this value indicates the current Non-Secure exception was taken in Secure mode. The extended return sequence validates the integrity signature and finally uses the extended exception frame to restore the original state.

```
0x08012d6c : ldmdavs fp!, {r0, r1, r4, r5, sp, lr} ;
    ldmdahi r1, {r0, r1, r7, r8, sb, ip, sp, lr, pc} ;
    pop {r3, ip, sp, pc}
0x080117d4 : andhs r3, r0, r8, asr #11 ;
    andhs r3, r0, r4, asr r6 ;
    pop {r4, r5, r6, r8, sl, ip, sp, pc}
0x08017cb8 : pop {r0, r1, r2, r3, r5, r6, r8, sb, sl, fp, pc}
0x0801006c : pop {r0, r1, r5, r7, fp, ip, sp, lr, pc}
```

Figure 2.6: Gadgets extracted by `ROPgadget.py`.

2.3 Control-Flow Security

Embedded software is often written in memory-unsafe languages such as C and C++. Although their low-level nature allows efficient use of computing resources, they put the onus of memory management on developers and are prone to memory errors, such as use-after-free and buffer overflows.

Memory errors can result in serious vulnerabilities such as arbitrary code execution. It is accomplished through the following two steps: the first step, *code injection*, loads malicious code onto the target's memory, followed by the second step, *control-flow hijacking*, which transfers the control to the injected code.

W⊕X (also known as DEP) is a memory access policy that mitigates such attacks by disallowing memory regions from being writable and executable at the same time. It can be efficiently implemented on standard hardware using a ubiquitously-available MMU or MPU. However, there is a class of attack techniques known as Return-Oriented Programming [93].

2.3.1 *Return-Oriented Programming*

ROP achieves a similar effect by reusing existing code fragments present in memory (called *gadgets*) to carry out useful operations. Gadgets are small instruction sequences ending with return instructions (Fig. 2.6), which can be abundantly found in an unprotected application. Using stack smashing techniques such as buffer overflow, the attacker carefully arranges the gadgets' addresses and input values on the stack. The arranged gadgets execute in a chain by the return instruction on each gadget, allowing the attacker to perform arbitrary computation. Since ROP does not inject any new code, existing mitigation techniques for code injection are ineffective against ROP.

ROP was first demonstrated on x86 [93]. Since then, many architectures have been shown to be vulnerable, including x86 [93], SPARC [30], RISC-V [54], Atmel AVR [44], and Arm [59].

Address Space Layout Randomization (ASLR) is a common defense against ROP, which randomizes the loaded modules' locations in memory to prevent attackers from reliably predict the gadgets' locations. However, an information disclosure vulnerability can defeat ASLR [94]. Also, for ASLR to be effective, every module must be randomized [32]. Finally, ASLR's applicability to microcontrollers is limited by the lack of virtual memory.

Stack canaries detect stack smashing by placing a random value at the end of an activation frame on function entry and checking if it is intact on function return. However, this technique is vulnerable to information disclosure and direct overwrite attacks because it assumes that the attackers do not know the value and that stack smashing always overwrites the canary [96].

2.3.2 *Control-Flow Integrity*

Control-flow integrity [16] is a class of defensive techniques against ROP attacks and other types of control hijacking. In essence, CFI is a twofold system: one part being a model of valid control paths, while the other one being runtime checking code that compares the model against the actual control path. The goal of CFI is to detect invalid indirect control transfer at runtime. A CFI scheme's ability to detect such control transfers is called *precision*.

The simplest, non-trivial model is a static control-flow graph generated by static code analysis [16]. The precision of static models is limited by the conservatism of the generated control-flow graphs as well as the limitation of and the difficulty in practical pointer analysis. For example, consider a function having multiple possible callers. The caller in a single function call is just one of them, but a static model cannot express this restriction and allows the control to return to the incorrect caller.

One possible way to improve the precision is to duplicate functions for each of, or more practically, each category of function calls. [71] proposed *call graph detaching*, which enhanced the precision of static models by making a copy of functions for indirect calls, thus detaching the direct call graph from the indirect call graph. Another possible way is to incorporate a dynamic element into the model to capture the program's runtime behavior. A *shadow stack* [16] is a data structure akin to a call stack that records a path of valid callers, created and maintained separately from the real call stack in a protected location. Other dynamic solutions include branch tracing [113], which utilizes branch recording features available in modern x86 processors, and πCFI [80], which activates edges in the CFG lazily as code pointers are generated at runtime.

Some CFI schemes only protect function calls or function returns, respectively called *forward-edge* and *backward-edge* CFI schemes. Shadow stacks only protect function returns. As such, they are said to be a backward-edge scheme. Due to orthogonality, shadow stacks can be complemented by forward-edge ones such as [106] to encompass all indirect branches.

The dynamic check is often accomplished by patching indirect branch instructions with runtime checking code. This is usually done at build time by a modified compiler (as in the case with [106]) or as a separate build process. It is also possible to do this entirely by rewriting the compiled binary machine code through the use of a binary instrumentation technique [16, 81], although it limits access to useful high-level information such as function prototypes. Memory-layout-preserving instrumentation such as [81] alleviates various difficulties arising from memory layout modification but imposes severe restrictions on the patched code, often requiring the use of expensive software trap instructions.

2.3.3 *Protecting CFI States*

CFI is a control-only protection technique, so it requires a separate mechanism to protect its own internal state data (provided that it has one). The choice of a mechanism significantly affects the overhead of dynamic CFI schemes. A well-known technique is to use special load/store instructions to access the protected data located in a different address space (e.g., x86 load/store with a segment selector [16]). Silhouette [111] took the opposite approach and replaced all load/-store instructions but those for shadow stack operations with Arm-M's unprivileged load/store instructions. [81] leverages the hardware-assisted security mode transition provided by TrustZone for Armv8-M by utilizing the fact that function calls triggering mode transition are protected by CFI themselves.

2.3.4 *System-Level CFI*

Most of the existing CFI solutions, including [16, 39] were designed for applications running in the user mode, assuming the operating system is trustworthy. Unlike them, we aim to protect a broader portion of software systems, including the operating system and exception handlers. We call this approach *system-level CFI* to contrast with the former one, which we call *user-space CFI*. The examples of system-level CFI include [19, 35, 81]. [108] targets embedded systems but is not considered as system-level CFI because it depends on the operating system's trustworthiness. System-level CFI allows the trusted computing base to be smaller. However, it requires the handling of ad-

ditional events, such as asynchronous exceptions and context switching.

2.3.5 *Previous Works on CFI*

CFI CaRE [81] is an earlier implementation of system-level backward-edge CFI employing shadow stacks for ensuring the CFI of normal functions as well as asynchronous exceptions. It does not address the quirk with Armv8-M's exception handling (described in Section 4.1.2) and does not support multitasking operating systems.

SVA [36] is an LLVM-based virtual machine for running operating systems, allowing the enforcement of object-granularity memory-safety and CFI. It virtualizes the low-level architecture (e.g., loads/stores, context switching, page table modification) of the target system and the operating system accesses them through newly introduced LLVM instructions. KCoFI [35] is an SVA-based CFI implementation for FreeBSD, which achieves a lower overhead by relaxing SVA's security policy and using lightweight instrumentation on store instructions to protect critical data structures.

µRAI [19] reserves a general-purpose register as a *state register* and stores a highly compressed representation of return targets in the state register. The presented implementation repurposes the return address register for this purpose, thus it breaks the standard ABI and disrupts debugging tools. Although the runtime performance and the RAM overhead is promising, the flash overhead is considerable. µRAI applies a whole-program analysis to calculate the optimal encoding of the state register. If the encoding does not fit in the state register, it occasionally needs to evict the contents of the state register to a structure similar to a shadow stack but for the state register. µRAI "occupies" Privileged mode to protect this structure, meaning Privileged user code should be non-existent, isolated through Software Fault Isolation [107], or assumed trustworthy (this issue can be mitigated by using CMSE, though). We note that relying on privilege levels to protect critical structures leaves the technique vulnerable to DMA attacks. Finally, the encoding of the state register being based on whole program analysis makes the performance characteristics of an instrumented program highly unpredictable under changes during development, which is undesirable in real-time systems.

Silhouette [111] protects a shadow stack by replacing all untrusted store instructions with Unprivileged store instructions, which are specific to the Arm-M architecture, and interleaving shadow stack operations with original code. The process of replacing store instructions is called *store hardening*. The Unprivileged store instructions only cover the most basic form of store instructions. Store hardening needs to emulate other forms of store instructions (e.g., indexing, FP

store, store-multiple), increasing the runtime and code overhead, and this can be especially punishing with the forthcoming Armv8.1-M architecture that includes 128-bit vector store instructions. Similarly to μRAI, Silhouette "occupies" Privileged mode, albeit in a different manner and suffers from all the issues with defensive mechanisms based on privilege levels such as vulnerability against DMA attacks and incompatibility with memory protection-enabled operating systems.

2.4 TOPPERS Embedded Component System

The TOPPERS Embedded Component System (TECS) is a component system specifically designed for embedded systems [24]. In this section, we describe the standard application development process, component model, and implementation model defined by the TECS specification [97]. After that, we describe how previous works "componentize" kernel features using TECS.

2.4.1 *Development Process*

Fig. 2.7 shows the standard development process using TECS [97].

Component designers create *celltypes*, which define the types of components. They also create signatures, which are sets of function prototypes with specific semantics used to define a component's interface. They define celltypes and signatures in a domain-specific language called TECS CDL. Component developers describe the celltypes' behavior by writing *celltype code*, which are in source files each associated with a celltype.

Application developers define *cells* (instantiations of celltypes) and define connections between them. This process is done similarly using TECS CDL.

At this point, the TECS CDL code includes all information needed to describe the application's overall structure. Application developers process the TECS CDL code using a program called *the TECS generator*. The TECS generator produces *the interface code*, which statically specifies how each cell calls other cells; header files; and a makefile. The TECS generator also generates celltype template code, which component developers may use to write celltype code. If the TECS generator's standard functionality turned out to be insufficient for a specific use case, it could be extended by TECS generator plugins (Section 2.4.2.6).

Application developers, finally, compile and link the interface call and the celltype code to obtain the final application executable.

2.4.2 *TECS Primer*

This section introduces the fundamental concepts of TECS defined by the TECS specification [97].

2.4.2.1 *Cells and Celltypes*

Cells are instantiations of components. Cells can have *entry ports* and *call ports*, and their pairs can be *joined* such that one cell can access

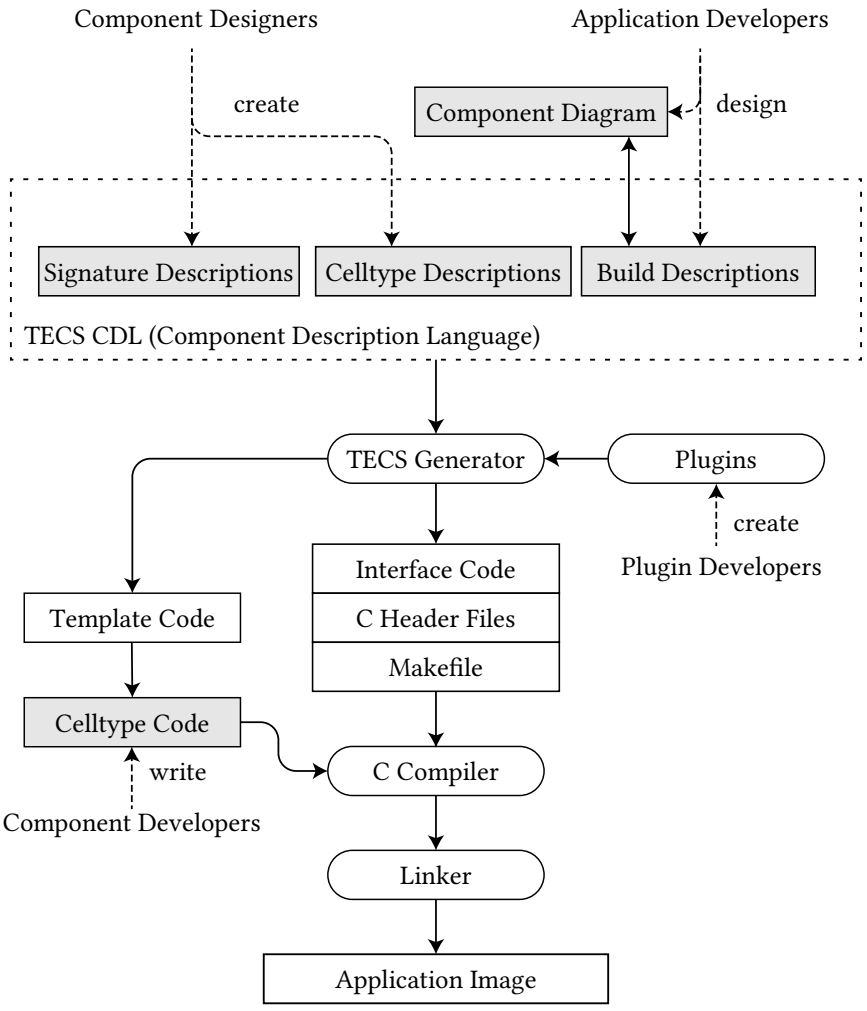


Figure 2.7: The standard TECS-based application development process.

the features provided by the other. Each call port must be joined to exactly one entry port.

A port can be specified as an array so that individual elements can be joined. When making a call through a call port array, celltype code passes an element index. Conversely, when handling a call made through an entry port array, celltype code receives the element index.

The types of ports are called *signatures*. When an entry port and a call port are joined, their signatures must match. Every call port or element of a call port array must be joined to exactly one entry port or element of an entry port, unless the call port is specified as an *optional call port*.

Entry ports declared `[inline]` are called *inline entry ports*. Inline entry ports, when used carefully, can reduce the function call overhead.

Cells can also contain zero or more *attributes* and *variables*. Attributes are constants associated with each cell, and they can be read by the application, factories, and plugins. Factories and plugins are discussed in detail below. Variables are similar to attributes, but their value can be updated at runtime.

Celltypes are a concept for representing a type of component, and for defining the properties shared by cells of the same celltype. The entry/call ports, attributes, and variables that a cell has are defined in its celltype, and each cell can have unique values for these attributes and variables. Moreover, they can have connections to the ports defined in celltypes of other cells. The behavior of a cell is defined by the celltype code associated with its celltype. A celltype code can have entry-port functions that describe the behavior of the cell when a certain entry port is called. Fig. 2.8 shows an example of TECS CDL code that defines a celltype and cell.

The standard TECS component diagram denotes cells by a rectangle with a cell name and a celltype name. Joins are denoted as lines connecting cells with a signature name in proximity, a port name at each end, and a solid triangle at the receiving end (Fig. 2.8).

The standard TECS component diagram does not specify the notations for attributes and variables. Nevertheless, we will use an ad-hoc notation for convenience.

2.4.2.2 *Signatures*

Signatures define the calling interface between cells. Each signature can contain one or more function-header definitions, which are similar to those in C language except that the direction of data flow and the array element count passed via a pointer are more explicit. For example, in the C language, a parameter of type `int*` can be used for passing an array of `int`s. However, it could also be used for returning it or passing and returning at the same time. Furthermore, the

```
// cell type definitions
celltype tCellType1 {
    call sSignature cCallPort;    // call port
};
celltype tCellType2 {
    entry sSignature eEntryPort;  // entry port
    attr { int someAttr; };       // attribute
    var { int someVar; };         // internal variable
};

// cell definitions
cell tCellType2 Cell2 {
    someAttr = 42; // specifying the attribute's value
};
cell tCellType1 Cell1 {
    cCallPort = Cell2.eEntryPort; // join
};
```

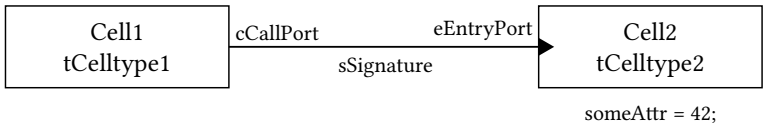


Figure 2.8: CDL code describing celltypes and cells and a corresponding diagram.

number of elements does not appear in C's function signature but is important for remote procedure calls. The signatures in TECS extend C's syntax by marking each parameter with `[in]`, `[out]`, `[inout]`, and `[size_of]` to specify their precise semantics. A signature might not have any function-header definition at all, in which case the signature is used as a tag to represent a certain kind of relation between cells. Fig. 2.9 shows an example of a signature definition.

2.4.2.3 *Composite Celltypes*

Composite celltypes are groups of logically-related cells and their connections. Similarly to normal celltypes, composite celltypes can be entry ports and call ports, and the internal cells' ports can be linked to them (such links are called `external joins`). Likewise, composite celltypes can have attributes, which are assigned to the internal cells' attributes.

Composite celltypes are expanded to individual cells during the generation process. Fig. 2.10 shows an example of a composite celltype, the corresponding diagram, and its expansion.

```
signature sSignature {
    // Receives a value by inArg, and returns a value by outArg
    void func1([in] int16_t inArg, [out] int16_t *outArg);

    // str contains "size" elements
    void func2([in, size_is(size)] char *str, [in] int16_t size)
        ;
};

// A signature with no functions
signature sEmptySignature {};
```

Figure 2.9: An example of a signature definition.

```
composite tComposite {
    entry sSignature eEntryPort;
    call  cSignature cCallPort:
    attr { int attribute; };
    cell tCelltype2 Cell2 {
        attribute = composite.attribute;
        cCallPort => composite.cCallPort;
    };
    cell tCelltype1 Cell1 {
        attribute = composite.attribute;
        cCallPort = Cell2.eEntryPort;
    };
    eEntryPort => Cell1.eEntryPort;
};

cell tComposite Composite {
    attribute = 42;
    cCallPort = SomeOtherCell.eEntryPort;
};
```

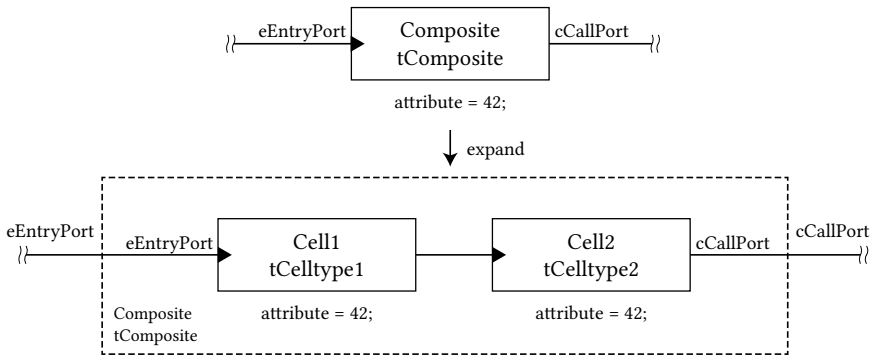


Figure 2.10: An example of a composite celltype, the corresponding diagram, and its expansion.

```
celltype tInterruptHandler {
  attr { INHNO inhNum; };
  factory { // cell "factory"
    write("tecsgen.cfg", "DEF_INH(%d, ...);", inhNum);
  };
};
cell tInterruptHandler INH1 { inhNum = 1; };
```

```
DEF_INH(1, ...);
```

Figure 2.11: A factory description and the generated kernel configuration.

2.4.2.4 *Factory Blocks*

By using *factory blocks* (Fig. 2.11), user-specified, formatted strings can be written to external files during the generation process. This feature is intended to be used to generate the kernel configuration file of an ITRON-compliant operating system.

There are two types of factory blocks: `FACTORY`, which is outputted for each used celltype, and `factory`, which is outputted for each cell. A factory block includes factory function calls. The current version of TECS only provides the `write` factory function. The `write` factory function takes the first argument as an output filename, the second argument as a `printf`-like format string, and any remaining arguments as formatting parameters. The `write` factory function recognizes and replaces special tokens in the format string, e.g., `$id$` → `tCelltypeName_CellName`.

2.4.2.5 *Implementation Model*

The TECS implementation model defines how the TECS generator realizes component definitions and behaviors as program code and data structures.

Information associated with each cell is stored in Control Blocks (CBs) on RAM and Initialize Blocks (INIBs) on ROM. CBs and INIBs can be omitted when they are not required (i.e., for *CB/INIB optimization*), reducing the memory footprint. Variable values are stored in CBs, while attribute values and call port descriptors are stored in INIBs. Each CB also includes the pointer to the corresponding INIB.

A *call port descriptor* is used to perform an indirect call when a call port is called. Fig. 2.12 shows the data structures involved in inter-cell calls in the most generic cases. To reduce the overhead in terms of execution time and memory consumption, indirect calls and call port descriptors can be eliminated whenever possible as a part of *call port optimization*.

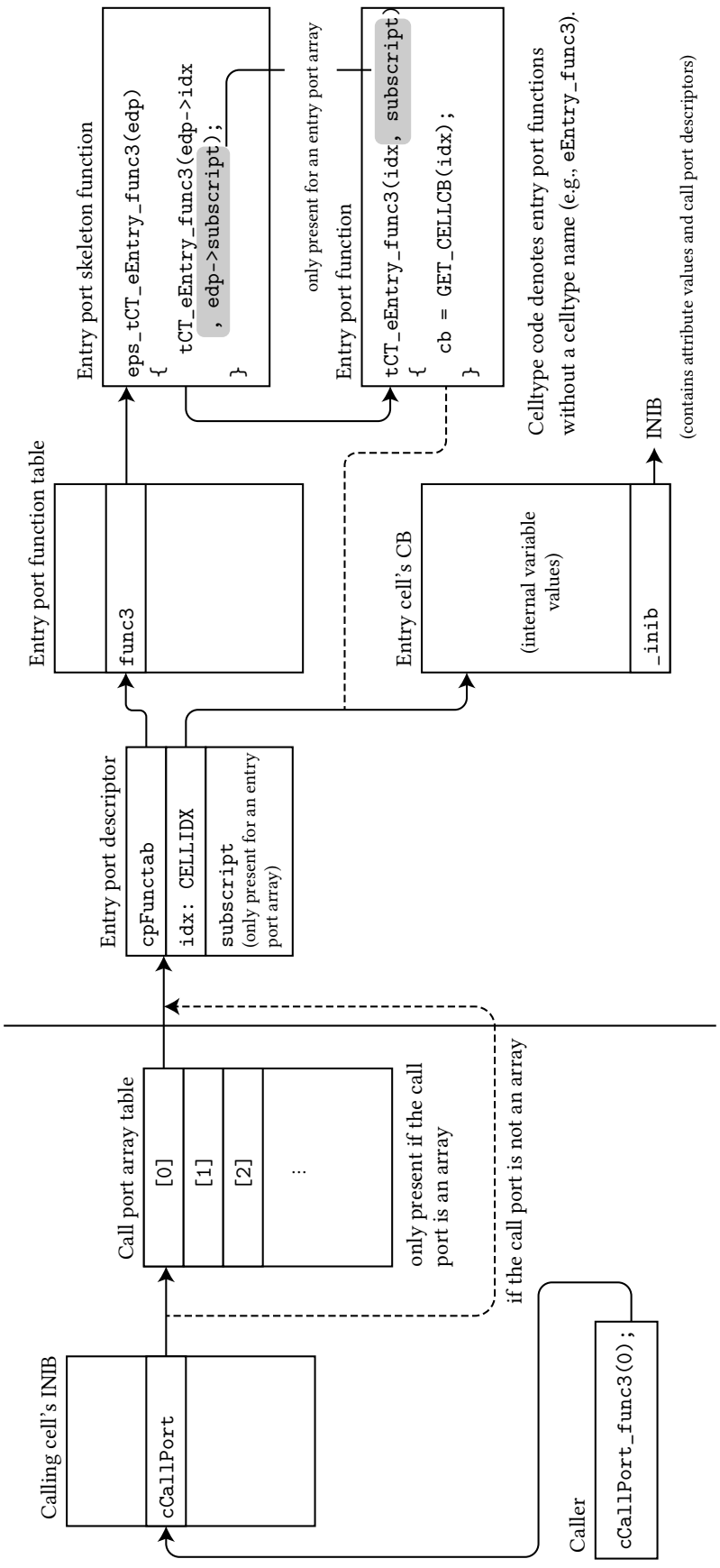


Figure 2.12: The data structures involved in inter-cell calls.

```
// During the generation, if there is at least one
// tExampleCelltype instance, then ExamplePlugin
// is loaded.
[generate(ExamplePlugin, "param1=1, param2=2")]
celltype tExampleCelltype { };
```

Figure 2.13: Using a celltype plugin

2.4.2.6 *TECS Plugin*

The functionality provided by the TECS generator can be extended by plugins written in Ruby. One of the plugin types supported by the TECS generator is the *celltype plugin*. Celltype plugins are attached to celltypes and can perform customized actions (e.g., outputting to an external file) when the cells are instantiated. Fig. 2.13 shows an example of this.

2.4.3 *RTOS Integration*

TECS is designed to be integrated into µITRON 4.0-style RTOSes' configuration systems. The integration is done by defining celltypes representing kernel objects and using factory blocks (Section 2.4.2.4) to generate the kernel configuration directives corresponding to their instances. These celltypes wrap the underlying kernel API and expose it as entry ports and call ports, allowing the kernel objects to be used under the unified component framework.

Fig. 2.14 shows a stripped-down version of ASP+TECS's `tTask`. When an application developer defines a cell of this celltype in their application, the TECS generator outputs a `CRE_TSK` (create a task) directive to a `tecsgen.cfg` file. The generated directive assigns the `id` attribute's value as the created task's identifier so that the celltype code can use it to refer to the created task. The application's top-level configuration must import the generated `tecsgen.cfg` file using an `INCLUDE` directive.

The `eTask` entry port provides access to task manipulation functions. Fig. 2.15 shows a simplified version of `tTask`'s celltype code. The `eTask_activate` entry port function[5] in this celltype code implements the behavior of the `activate` function of this entry port. The callee cell's `id` attribute value is available through the `ATTR_id` macro, which is defined in the generated `tTask_tecsgen.h` file. Using this identifier, this function delegates the rest of the processing to the `act_tsk` kernel function.

5 It is not the actual symbol name; it is a macro defined in `tTask_tecsgen.h` that expands to the full name `tTask_eTask_activate`.

```
signature sTask {
    ER      activate(void);
    ER      wakeup(void);
}

[active]
celltype tTask {
    entry    sTask   eTask;  /* task operations */
    call sTaskBody   cBody;  /* task body */
    attr{
        ID              id = C_EXP("TSKID_$id$");
        [omit] ATR      taskAttribute = C_EXP("TA_NULL");
        [omit] PRI      priority;
        [omit] SIZE     stackSize;
    };

    factory {
        write("tecsgen.cfg",
            "CRE_TSK(%s, { %s, $cbp$, tTask_start_task, %s, %s,
                NULL });",
            id, taskAttribute, priority, stackSize);
    };
};
```

Figure 2.14: An excerpt of the definition of `tTask` from ASP+TECS.

```
#include "tTask_tecsgen.h"

ER eTask_activate(CELLIDX idx) {
    CELLCB  *p_cellcb = GET_CELLCB(idx);
    return(act_tsk(ATTR_id));
}

void tTask_start_task(intptr_t exinf) {
    CELLCB  *p_cellcb = (CELLCB *) exinf;
    cBody_main();
}
```

Figure 2.15: An excerpt of the celltype code of `tTask` from ASP+TECS.

In the previous paragraph, we have seen how inter-cell calls are converted to kernel function calls. Now we will see how the opposite direction's calls are made. In Fig. 2.14, a parameter of the `CRE_TSK` directive registers the `tTask_start_task` function as the task's entry point. The preceding parameter is the value passed to the task entry point, which is, in this case, specified to be `$cbp$`, an opaque handle called a *CB pointer*. The `tTask_start_task` function places this value in a local variable named `p_cellcb`. The attribute macros such as `ATTR_id` and the call port macros use `p_cellcb` implicitly to access the current cell's information. Thus, despite not being a standard part of celltype code, the `tTask_start_task` function can now mimic the cell and call `cBody`, the call port representing the task body.

Unfortunately, this approach only works when (1) concatenating the outputs of all factory blocks is sufficient to give the desired result, and (2) the complexities of the required configuration directives are within the factory blocks' capability. TOPPERS/HRP2 [118] extended the µITRON 4.0-style kernel configuration syntax with protection domains to support partitioning, which violated the first assumption. HR-TECS [53] addressed this issue by a TECS plugin. The latest version of its specification, the TOPPERS Third-Generation Kernel Specification has met with a similar problem after it introduced a new feature called *time event notifications* to improve time event objects' usability in partitioned systems and violated the second assumption. We will investigate this matter and propose a solution in Chapter 5.

3

Lightweight RTOS Utilizing TrustZone for Armv8-M

The ongoing IoT trend has led to a burgeoning number of connected embedded systems and increasing demand for techniques to guarantee security and safety within embedded systems' stringent computing requirements. Partitioning, which divides a program into multiple protection domains and limits fault propagation between the domains, is a well-known technique in embedded systems. Partitioning is realized by combining one or more methods, including memory protection and time protection. Memory protection is the essence of partitioning as it provides spatial isolation necessary for partitioning mechanisms' integrity. A memory-protection-enabled operating system usually implements memory protection using the underlying processor's ring protection mechanism.

Embedded operating systems without memory protection implement system calls by standard function calls, while those with memory protection must perform a processor mode transition to enter the privileged mode. The mode transition is initiated by a software trap instruction, which raises a processor exception. When this happens, the processor enters the privileged mode and transfers the control to a trap handler inside the operating system kernel. The trap handler requires a non-trivial amount of processing before reaching the system call handler. In some cases, it nearly doubles a system call execution time compared to an operating system without memory protection [118], which is problematic in power-constrained embedded systems. Inlining system calls can somewhat reduce their execution times [38] but does not mitigate the mode transition's substantial overhead. Furthermore, the processor mode transition machinery required for system calls is tricky to implement correctly, and a poor, half-baked implementation easily results in vulnerabilities [112].

Because of the lack of native hardware support, most operating system specifications do not support interrupt handlers belonging to a user domain. As such, automotive systems needing such interrupt handlers [117] must emulate them by user tasks, incurring a substantial overhead.

A hardware mechanism supporting mode-crossing function calls will allow efficient implementation of a memory-protection-enabled operating system. Specifically, the mechanism needs to obviate the use of a trap handler in the system call implementation. It will also

close the gap between operating system implementations with and without memory protection, reducing the engineering cost.

Arm company announced Armv8-M, a new version of their processor architecture family for microcontrollers, in 2015. Armv8-M's new feature, TrustZone for Armv8-M, provides the aforementioned hardware mechanism through an instruction set extension called CMSE (Section 2.2). TrustZone for Armv8-M is still early in its days and leaves its potential on the table. The use cases that have been proposed so far include control-flow integrity [81] and remote attestation [18]. However, no attempts appear to have been made to use it to implement memory protection in an operating system.

The main contributions of this chapter's work are the following:

- We identify and perform a qualitative comparison of three possible system configurations for TrustZone for Armv8-M, namely, Secure Library, Dual Operating System, and Single Binary Image architectures (Section 3.1.2).
- Based on one of such configurations, Single Binary Image architecture, we develop ASP3+TZ, a memory-protection-enabled operating system, and present its design (Sections 3.2 and 3.3).
- We conduct an experimental evaluation on ASP3+TZ and show a quantitative performance of a TrustZone for Armv8-M-based RTOS for the first time and confirm the superior performance characteristics of the SBI scheme compared to traditional memory protection schemes (Section 3.4). Furthermore, we provide a performance evaluation of the security mode transition of TrustZone for Armv8-M for the first time.
- We demonstrate that the SBI scheme accomplishes memory protection with only a few modifications to an existing operating system by comparing the number of lines of the code of ASP3+TZ to that of ASP3 (Section 3.4.3).

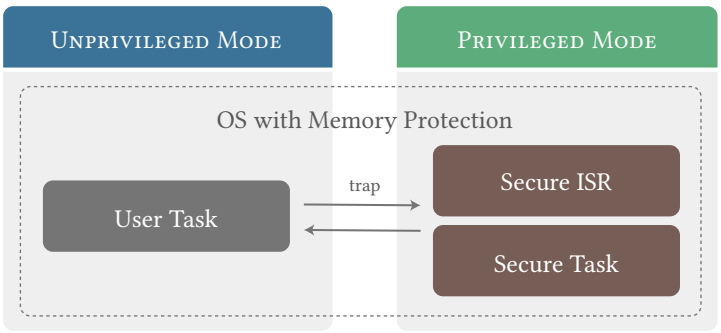


Figure 3.1: The architecture of a traditional operating system with memory protection.

3.1 Memory Protection by CMSE

An application can utilize CMSE (Section 2.2) in different ways depending on its specific design requirements. In this section, we first describe the traditional memory protection implementation for comparison. After that, we show three possible CMSE-based system architectures and compare them qualitatively.

3.1.1 *Traditional Approach: Ring Protection*

The traditional approach to implement memory protection in an RTOS is to leverage the processor's ring protection mechanism. An RTOS kernel taking this approach, such as FreeRTOS-MPU [68], TOPPERS/HRP2 [118], is usually linked together with user code, forming a single executable image (Fig. 3.1).

In this kind of system architecture, the kernel executes untrusted code in the unprivileged mode. The kernel configures the MPU to prevent untrusted code from corrupting the kernel's or other protection domains' mutable data. Some system calls can complete in the unprivileged mode; for example, the code running in the unprivileged mode could read the current system time without mode transition, assuming the global variable storing the value is exposed read-only, as done by vDSO [45] introduced in Linux 2.5. However, in most cases, system calls require read-write access to kernel structures, and therefore it would be unsafe to let them run in the unprivileged mode. One example is `dly_tsk` from the µITRON 4.0 specification [51], which puts the current task to sleep for a specified period of time. The function inserts the current task into a system-global timed event queue, pulls the next task to execute from the task ready queue, and then performs a context switch[1]. These steps require read-write access to

1 In this dissertation's context, context switch refers to the process of changing the executing task by saving the previous task's execution state and restoring that of the next task.

kernel structures, and it would be harmful to the kernel's integrity if such access were allowed freely in the unprivileged mode.

Trap handlers are a mechanism for unprivileged code to reenter the privileged mode to complete such system calls. To invoke a trap handler, unprivileged code raises a CPU exception by a predetermined method, such as Arm Thumb-2's `svc` (service call) instruction, any undefined instruction, and a branch to a non-executable memory region. Upon detecting an exception, the processor transfers the program counter to a trap handler registered by the kernel. The trap handler executes in the privileged mode and thus can complete the intended task.

The exact implementation varies significantly from one processor architecture to another. Let us consider Arm-M architecture. This architecture has two execution modes: Handler mode, which is dedicated for exception handlers, and Thread mode, in which other code executes. A processor based on Arm-M contains an interrupt controller called NVIC, which manages the list of active (being handled) exceptions at any point. Such an interrupt handling design is immensely useful to lower the interrupt handling overhead in a bare-metal system or an RTOS-based system without memory protection. However, it requires special consideration for an RTOS with memory protection.

For example, the Arm-M ports of TOPPER/ASP3, TOPPERS/HRP2, and FreeRTOS use PendSV exception for context switching (see Section 3.3.3.1). Some system services trigger PendSV to perform a context switch. However, for this context switching technique to work correctly, PendSV must preempt the execution of Thread mode, not Handler mode, i.e., it must be the first-level, non-nested exception. This means that the trap handler cannot just handle system calls inside and instead must return to Thread mode before transferring the control to the code responsible for handling system calls.

Fig. 3.2 is a portion of TOPPERS/HRP2's SVC handler pertaining to transition to Thread mode and invocation of a service routine. First, `svc_entry` manipulates APSR to raise the privilege level associated with Thread mode. This change does not take effect immediately because the processor is running in Handler mode at this point. After that, `svc_entry_2` creates a fake exception frame. In normal circumstances, the processor generates an exception frame on exception entry and uses it to restore the original execution state on exception return. In this case, the processor is tricked into reading the execution state from the generated exception frame, thus branching to `svc_entry_4` instead of the actual original preemption point. The subsequent `bx lr` instruction triggers the hardware exception return sequence. Finally, `svc_entry_4` looks up and calls the specified system call handler.

```
ALABEL(svc_entry)
    /* ... */
    mov     r6, #0x00
    msr     control, r6             /* enter Privileged mode */
    /* ... */
ALABEL(svc_entry_2)
    /* ... */
    /*
     *  To exit Handler mode, we need to trigger an exception
     *  return. Construct a fake exception frame to "return"
     *  and resume execution from svc_entry_4.
     *  r10 = psp
     */
    ldr     r6, =EPSR_T    /* xPSR (T = 1) */
    ldr     r5, =svc_entry_4
    stmfd   r10!, {r4-r6}     /* push xPSR, pc, lr */
    stmfd   r10!, {r0-r3,r12} /* push r0-r3, r12 */
    msr     psp, r10
    bx      lr
    /* ... */
ALABEL(svc_entry_4)
    /*
     * The processor is now in Thread mode but we are still
     * in the SVC handler. Now, look up the SVC handler
     * table and call the specified inner handler.
     */
    rsb     r4, r4, #0              /* negate */
    ldr     r5, =svc_table
    ldr     r5, [r5, r4, lsl #0x02] /* read the handler table */
    blx     r5                      /* jump to a inner handler
        */
```

Figure 3.2: An annotated excerpt from TOPPERS/HRP2's SVC handler.

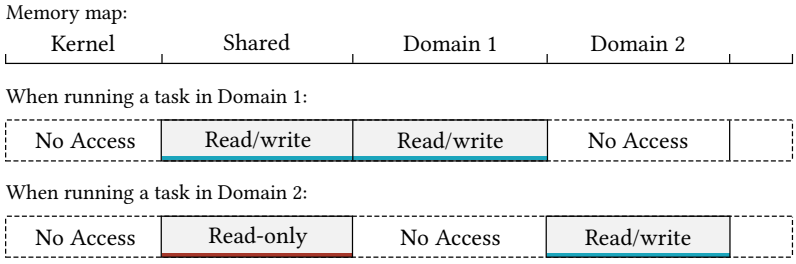


Figure 3.3: Configuring the MPU for each protection domain.

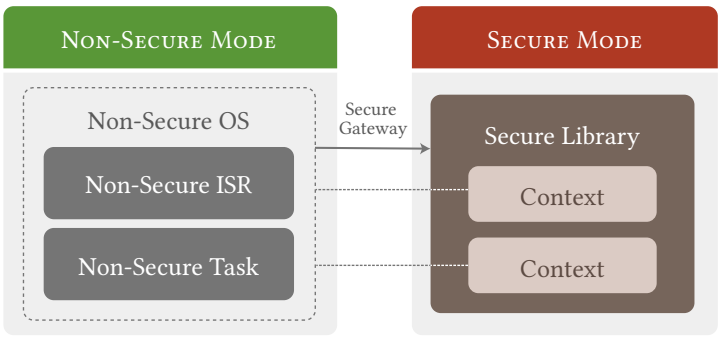


Figure 3.4: Secure Library architecture.

The kernel configures the MPU to impose a memory access policy on unprivileged code. The MPU configuration can be generated dynamically as is done by FreeRTOS-MPU or statically as is done by TOPPERS/HRP2, the latter requiring a sophisticated build-time system to meet the target MPU's requirements [118]. If there is more than one user (untrusted) protection domain, the MPU needs to be reconfigured when switching contexts (Fig. 3.3).

3.1.2 *System Architectures for CMSE*

We consider the following three system architectures for CMSE-based partitioning and memory protection.

Secure Library (SL) In *Secure Library architecture*, Secure code, compiled as a self-contained binary image, exposes functionalities in the form of a software library (Fig. 3.4). Secure code runs using a Secure stack[2]. Since Non-Secure interrupts can preempt Secure code just as usual and change the current task, the Non-Secure operating system must switch Secure contexts during a context switch. Arm CMSIS (Cortex Microcontroller Software Interface Standard) targets this architecture and standardizes the interface for a Non-Secure operating system to manage Secure contexts [1].

This architecture allows Secure code developers to minimize the chance of leaking secrets embedded in Secure code. For example, a Secure code developer could embed a cryptographic key in a Secure communication library. Once programming the code to a microcontroller, they could disable any external access (i.e., including debugger access) to the Secure domain, preventing other developers from tampering with the Secure code or stealing the confidential information.

This architecture results in a minimal trusted computing base (TCB) and thus is the standard approach for implementing a software-based

2 In Armv8-M, the stack pointer register is banked for each security state.

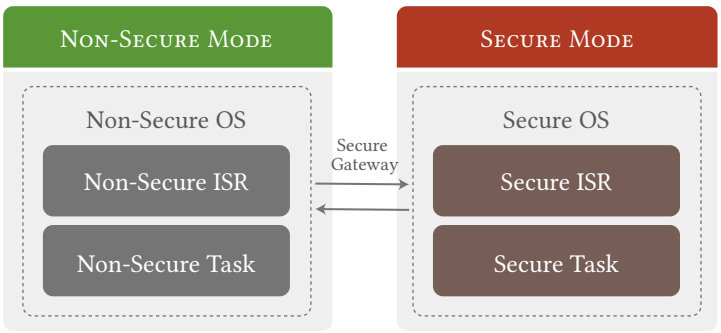


Figure 3.5: Dual Operating System architecture.

security mechanism, such as remote attestation [18] and control-flow integrity (Chapter 4).

DUAL OPERATING SYSTEM (D-OS) In *Dual Operating System architecture*, each of Non-Secure mode and Secure mode hosts an independent operating system kernel instance (Fig. 3.5). Each kernel instance is responsible for managing thread states belonging to the respective security mode.

Compared to Secure Library architecture, this architecture increases flexibility in what Secure code can do, but the operating system overhead is considerable on account of running two operating system instances simultaneously. Mixed scheduling across the domain boundary is not trivial and requires modification to the kernel code. Also, issuing service calls across the boundary requires an inter-kernel remote call, making this approach inapplicable if frequent inter-domain communication is required.

This architecture is conceptually similar to SafeG [115], which uses TrustZone for Arm A-Profile to allow an RTOS to run alongside a general-purpose operating system such as Linux. However, this solution is inadequate for a microcontroller-based system, which usually cannot afford the additional computational requirements to host a general-purpose operating system or the switch to a more expensive, more complex (requiring harder-to-verify software to control), and more power-hungry Arm A-Profile microprocessor.

SINGLE BINARY IMAGE (SBI) This work proposes *Single Binary Image architecture*, in which a single application binary image containing one kernel instance governs both Secure and Non-Secure modes (Fig. 3.6). In contrast to the first two architectures, this architecture is not designed to protect the secrets embedded in code from other developers working on the same system.

Single Binary Image architecture and the traditional system architecture based on ring protection (Section 3.1.1) are alike in that, in both architectures, the entire application is compiled as one binary

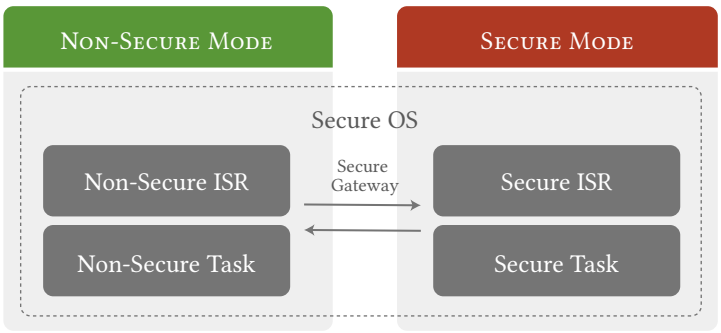


Figure 3.6: Single Binary Image architecture.

Table 3.1: Properties of system architectures for CMSE.

	SL	D-OS	SBI
Secure/Non-Secure isolated development	✓	✓	
Lower memory overhead	✓		✓
Lower runtime overhead		†	✓
Lower development overhead			✓

† Issuing service calls across the Secure/Non-Secure boundary requires an inter-kernel remote call, which results in a high runtime overhead.

image and runs on a single operating system instance. Being able to build the entire application with one linker run simplifies the development flow. Compared to Secure Library architecture, this architecture offers a lower context switching overhead because the kernel need not go through the Secure context management API. Compared to Dual Operating System architecture, this architecture does not incur the overhead of running two separate operating system instances.

Similarly to the traditional memory protection scheme, Single Binary Image architecture incurs a processor mode transition overhead on system calls made by untrusted code. However, CMSE provides a specialized hardware mechanism to make function calls across security modes with far lower latency than going through trap handlers. Therefore, we postulate that Single Binary Image architecture can approximate the traditional memory protection scheme while offering a significantly lower overhead.

Table 3.1 summarizes the properties of the system architectures presented in this section. As noted before, Single Binary Image architecture does not support the isolated development of each security mode. Other than that, we consider that Single Binary Image architecture is superior in all aspects.

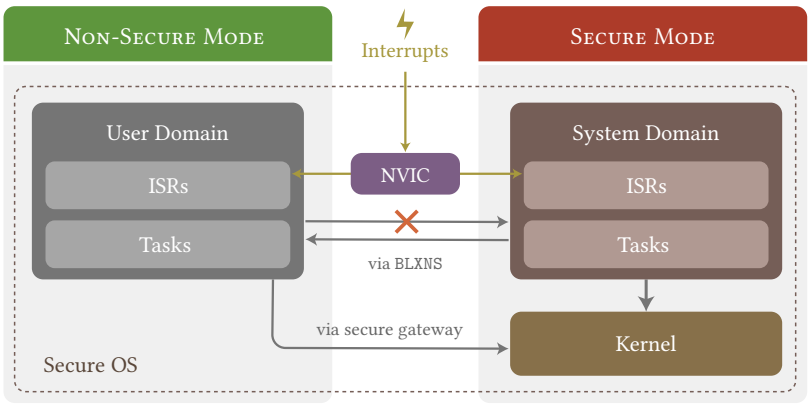


Figure 3.7: The concept of operating system based on the SBI architecture.

3.2 Design

This section discusses the proposed operating system design based on Single Binary Image architecture (Section 3.1.2).

We intend the proposed design to be implemented as a patch to an existing operating system kernel lacking memory protection. Thus the design is driven by the following goals: (1) Minimize the additional overhead introduced by the patch. (2) Keep the implementation cost small. To this end, we strove to ensure the proposed design utilizes CMSE's functionalities such as automatic register preservation and restoration to as much extent as possible.

3.2.1 *Domains*

In the vein of the standard design direction of a memory protection-enabled operating system, the proposed design partitions the system into a *system domain* and a *user domain*, each associated with Secure and Non-Secure modes, respectively. The code running in the user domain is constrained by a memory access policy enforced by SAU and IDAU and a kernel object access control enforced by the kernel.

It is often the case that an operating system supports more than one user domain, each having a unique set of memory access permission flags and accessible kernel objects. Partitioning aims to localize software faults in a faulting domain, so having multiple domains increases the system's capability to localize the damage caused by a software fault with the additional benefit of facilitating fault analysis. This principle does not change regardless of how memory protection is implemented, but the overhead tends to worsen as the number of domains increases. Since CMSE can perform mode transition at a much lower runtime cost, we anticipate that CMSE-based memory protection can support more user domains at the same runtime cost.

Still, supporting only one user domain remains a plausible option in a trade-off between performance and security.

All kernel code runs in Secure mode, and Non-Secure code issues system calls through Secure gateways (explained in Section 3.2.2).

3.2.2 *Service Calls*

The standard approach of allowing a user domain to make service calls is to use trap handlers, but this approach incurs a significant runtime overhead. Instead, the proposed design exposes each kernel API function as a Secure gateway (Fig. 3.7). The front-end of a Secure gateway is a thin wrapper called *a veneer function*, placed in a Non-Secure-Callable region. Parameter and caller validation can be implemented in the veneer function or as a modification to the API implementation.

3.2.3 *User Interrupt Handlers*

Some operating system specifications such as AUTOSAR [15] support user interrupt handlers, viz., interrupt handlers belonging to a user domain. In a traditional operating system design, such interrupt handlers impose a larger overhead and exhibit a worse response time compared to normal interrupt handlers. The ring protection mechanism automatically switches the processor to a privileged mode upon taking an interrupt. Thus, to implement user interrupt handlers, the operating system must first capture interrupts by privileged interrupt handlers and then transition to an unprivileged mode before invoking the corresponding user interrupt handler. This process is exceedingly inefficient and becomes even more complicated in the presence of interrupt nesting and unique interrupt handling mechanisms like the one seen in Arm-M. For this reason, many operating systems such as TOPPERS/HRP2 elected not to support user interrupt handlers, in which case they have to be emulated by user tasks.

CMSE supports an independent exception vector table for each security mode and can take any security mode's interrupts regardless of the current security mode. The security mode transitions caused by interrupts are handled automatically by hardware and do not require the operating system's intervention. The proposed operating system leverages this feature to implement user interrupt handlers.

Both security modes share the same interrupt priority space; for example, an application developer can configure interrupt priorities in a way that certain user interrupt handlers have higher priorities than those of normal interrupt handlers.

3.2.4 *(Un)privileged Functions*

Privileged functions are a mechanism to allow application developers to define system-domain functions callable by a user domain. In a CMSE-based memory protection scheme, privileged functions are just application-defined Secure gateways. Furthermore, CMSE provides a lightweight method for Secure code to call back into a Non-Secure function. This method can be used to create *unprivileged functions*—the opposite of privilege functions. Implementing unprivileged functions is slightly more complicated because the kernel must be able to catch exceptions generated in a user domain and safely abort the faulting unprivileged function call.

3.3 Implementation

We developed ASP3+TZ (TOPPERS/ASP3 + TrustZone), a prototype implementation of the proposed operating system design (Section 3.2) to evaluate the overhead associated with Single Binary Image architecture (Section 3.1.2). We used TOPPERS/ASP3 as the starting point. The implementation only covers the fundamental features required for basic memory protection; it does not support multiple user domains, kernel object access control, or unprivileged functions.

This section walks through the changes we made to implement ASP3+TZ.

3.3.1 *Kernel Services*

Most system calls require transition to Secure mode to modify the kernel's internal state. The transition can be done automatically by defining kernel API functions as *Secure gateways*.

CMSE provides `sg` (Secure gateway) instruction, which serves as an entry point to Secure mode. This instruction transitions the processor into Secure mode when Non-Secure code jumps to one of such instructions. When Non-Secure code calls a Secure gateway, the processor clears the least significant bit[3] of the `lr` (link register) register's value, signifying the Secure gateway was called by Non-Secure code and must re-enter Non-Secure mode before returning to its caller. This form of a return address can only be recognized as valid by CMSE's `bxns` instruction, and hence Secure gateways must return to their caller by `bxns` instruction rather than the usual `bx` instruction. This design is a precaution against *return-to-Secure-region attacks* and prevents Non-Secure code from tricking a Secure gateway into returning to an arbitrary Secure location.

CMSE's `sg` instruction only takes effect in a Non-Secure Callable region so that an arbitrary bit pattern in Secure memory does not unintentionally create an entry point. The standard way to define Secure gateways is to create small wrapper functions called *veneer functions* and place them in a Non-Secure Callable region, separating them from the function bodies located in a Secure region. Veneer functions have special prologues and epilogues that use the `sg` and `bxns` instructions mentioned above. A linker adhering to the CMSE toolchain requirements [8] can generate veneer functions automatically.

The veneer function generation by a linker is specifically designed for what we call Secure Library architecture (Section 3.1.2) and has no

3 When Thumb instruction set was introduced to Arm architecture for the first time, the least significant bit of a program counter represented the processor mode to execute the code in. Arm M-Profile only supports Thumb and has dropped the classic Arm instruction set, so the bit is always set.

```
ns_act_tsk:
    sg
    push {lr}
    bl act_tsk
    pop {lr}
    mov r1, #0
    mov r2, #0
    mov r3, #0
    mov ip, #0
    msr apsr_nzcvq, r1
    bxns lr
```

```
ns_wai_flg:
    sg
    push {r4, lr}
    tst r3, #3
    tt r4, r3
    bne sg_perm_check2_fail
    tst r4, #TT_RESP_NSRW
    beq sg_perm_check2_fail
    bl wai_flg
    pop {r4, lr}
    mov r1, #0
    mov r2, #0
    mov r3, #0
    mov ip, #0
    msr apsr_nzcvq, r1
    bxns lr
```

Left: A secure gateway with no pointer parameters.
Right: The fourth parameter undergoes pointer access permission check by TT instruction.

Figure 3.8: Examples of ASP3+TZ's secure gateways.

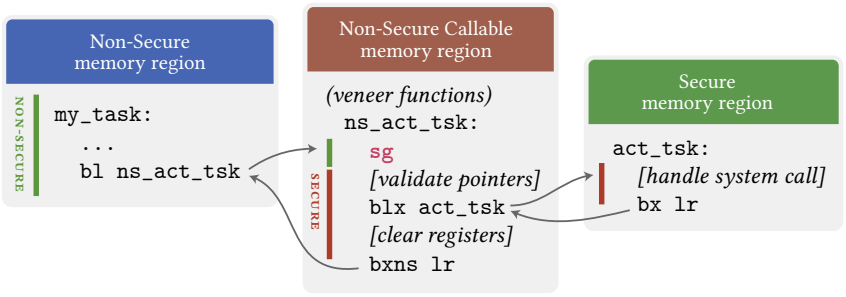


Figure 3.9: The overall call flow of a kernel API function.

use here. We instead wrote the veneer functions manually in assembler, one for each of the kernel API functions, prefixing their names by `ns_` (Fig. 3.8). Fig. 3.9 shows the overall call flow of a kernel API function when it is called through a veneer function.

In addition to calling the function body, veneer functions involve the following steps:

POINTER VALIDATION Some kernel API functions take pointer parameters and read or write their referents. They execute in Secure mode and thus must first make sure the passed pointers refer to Non-Secure memory not to allow the Non-Secure caller to illegally observe or modify Secure memory contents. CMSE provides TT instruction, which determines the security attributes of a specified address. We implemented the veneer functions in a way that they validate the passed pointers and return an error code on validation failure (Fig. 3.8, the right half).

```
 typedef struct task_context_block {
+    uint32_t    *sp_nonsecure;  /* PSP (Non-Secure) */
     intptr_t    pc;             /* PC */
-    uint32_t    *sp;            /* PSP */
+    uint32_t    *sp_secure;     /* PSP (Secure) */
 } TSKCTXB;

 typedef struct task_initialization_context_block {
-    size_t      stksz;           /* stack size */
-    void        *stk_bottom;     /* stack bottom ptr */
+    size_t      stksz_secure;           /* stack size */
+    void        *stk_bottom_secure;     /* stack bottom ptr */
+    size_t      stksz_nonsecure;        /* stack size */
+    void        *stk_bottom_nonsecure;  /* stack bottom ptr */
 } TSKINICTXB;
```

Figure 3.10: The modification to the task control block.

This check usually belongs to the function bodies, but in this case, we decided to implement it in veneer functions not to affect the functionalities of system calls in Secure code.

CLEARING REGISTERS Most of the general-purpose registers are shared between Secure and Non-Secure modes. This means that anything left in caller-saved registers is carried over when transitioning back to Non-Secure, potentially leaking confidential information. For this reason, `r0-r3` registers should be cleared before executing `bxns`, as shown in Fig. 3.8.

3.3.2 *Non-Secure Task Stacks*

In Arm M-Profile, the stack pointer is banked between Handler and Thread modes, and the corresponding stack pointers are called MSP (Main Stack Pointer) and PSP (Process Stack Pointer), respectively. Furthermore, CMSE banks the stack pointer between Secure and Non-Secure modes, so there are four stack pointers in total. For example, in user task code, the `sp` register points to Non-Secure PSP. If it makes a system call and enters Secure mode, the `sp` register now points to Secure PSP.

User tasks must have separate Non-Secure and Secure stack regions and stack pointers. Although managing two stacks can be tedious, it would be unreasonable for both security modes to share a single stack pointer from a security point of view. Secure code may store sensitive temporary values on the stack. Recall that Non-Secure interrupts can be taken even when the processor is running Secure code. If such sensitive values were stored on a Non-Secure-accessible stack, a Non-Secure interrupt handler would be able to corrupt those values. A

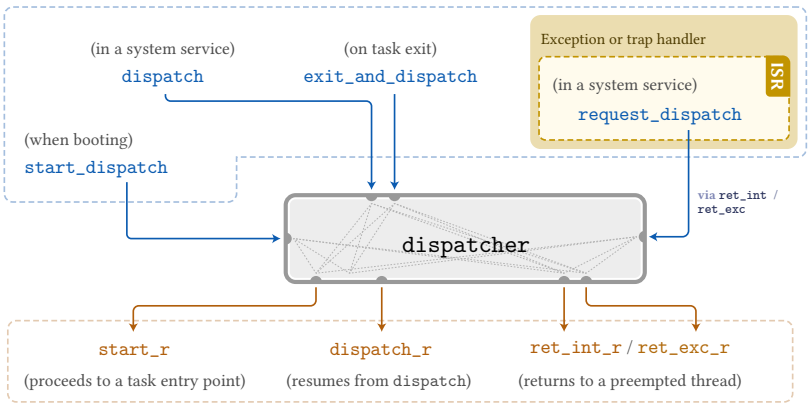


Figure 3.11: The entry and exit points of TOPPERS/ASP3's dispatcher.

naïve workaround is to reconfigure SAU to protect a portion of the stack from Non-Secure access during the duration of a system call. However, this workaround is flawed because of the unavoidable timing gap between entering a Secure gateway and reconfiguring SAU.

We extended the kernel structures related to tasks, specifically the task context block and task initialization context block, to hold two separate stack regions and pointers (Fig. 3.10). We modified the kernel configurator to allocate two separate stack regions to each task and store them in the task initialization context block. We also modified the task context initialization function to set up the stack pointer for each security mode.

3.3.3 *Dispatcher*

The dispatcher is the part of a kernel responsible for setting up the processor state for the execution of a scheduled task and comprises the last step of context switching. This section describes ASP3+TZ's dispatcher implementation. First, Section 3.3.3.1 describes the dispatcher interface used by the TOPPERS/ASP3 kernel. After that, Section 3.3.3.2 shows how the existing Arm-M port of TOPPERS/ASP3 implements it. Finally, Section 3.3.3.3 describes the changes we made to the Arm-M port's dispatcher to use it in ASP3+TZ.

3.3.3.1 *The Dispatcher Design of TOPPERS/ASP3*

The kernel's target-specific code defines the following entry points to the dispatcher (Fig. 3.11):

- The kernel calls *`start_dispatch`* during the boot process to dispatch the first task.

- When a task performs a blocking operation or wakes up a higher-priority task, the kernel calls *dispatch* to save the current task's state and dispatch the next runnable task.
- When a task exits, i.e., when there is no need to preserve the execution state, the kernel calls *exit_and_dispatch* to dispatch the next runnable task.
- When an interrupt handler performs an operation that changes the current task, the kernel calls *request_dispatch*. This function does not instantly start dispatching; it pends dispatching until all of the currently-active interrupt handlers (there can be more than one because of nesting) complete execution.

TOPPERS/ASP3's porting manual specifies how the target-specific code should implement the above entry points by pseudocode. In the pseudocode, the dispatcher exits through one of the following exit points, chosen based on how the destination task was interrupted last time.

- If the task has never run since its activation, the dispatcher jumps to *start_r*, which invokes the task's entry point.
- If the task voluntarily yielded the processor by calling `dispatch`, the dispatcher jumps to *dispatch_r*, which restores the execution state and returns from `dispatch`.
- If the task was interrupted by an interrupt and put to sleep by `request_dispatch`, the dispatcher jumps to *ret_int_r*, which restores the execution state and resumes the execution from where it left off.
- If the task was interrupted by a CPU exception and put to sleep by `request_dispatch`, the dispatcher jumps to *ret_exc_r*, which restores the execution state and resumes the execution from where it left off.

3.3.3.2 *The Dispatcher Implementation in the Arm-M Port*

The Arm-M architecture provides PendSV, a kind of software interrupts intended to be used for dispatching. Similarly to many other RTOSes, the Arm-M port of TOPPERS/ASP3 uses PendSV for this purpose.

Using software interrupts is the most efficient way to implement dispatching in Arm-M mainly because of its hard-wired exception handling sequences. The important steps of interrupt handling and dispatching are to saving the contents of registers when interrupting a thread (whether it is a task or interrupt handler) and restoring them

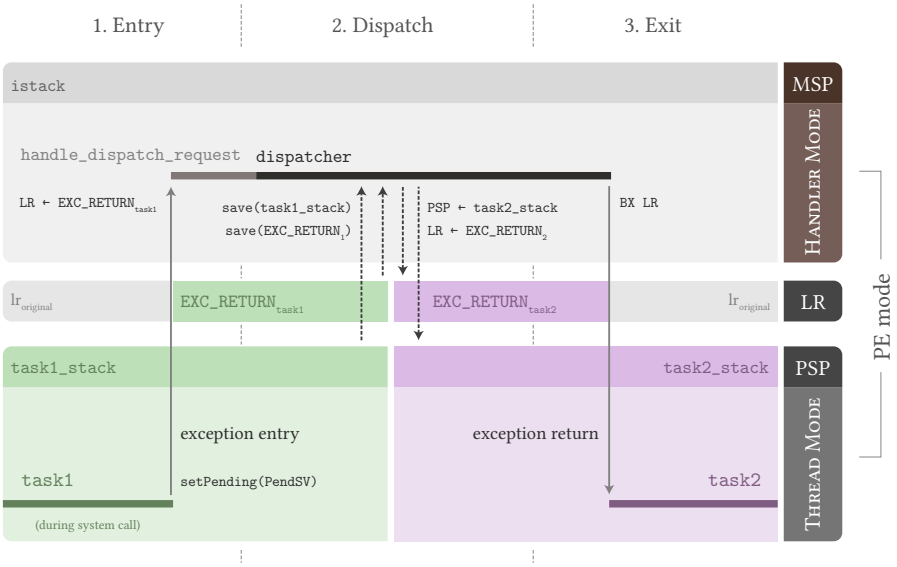


Figure 3.12: An example execution flow of the dispatcher for Arm-M.

when resuming a thread. A non-trivial portion of the porting manual's pseudocode is dedicated to precisely explain when and how such steps are performed. However, in Arm-M, they are entirely taken care of by the hard-wired sequences for normal interrupts[4]. The dispatcher can be implemented as a unique sort of interrupt handler that replaces the background context state.

The Arm-M port's dispatcher code resides in the PendSV handler and the SVC handler. The dispatcher code exchanges the values of the current PSP, `EXC_RETURN`, and the callee-saved registers (`r4`–`r11` and `s16`–`s31`). The hardware exception handling sequences places the exception frame containing the original program counter, the caller-saved registers (`r0`–`r3`, `r12`, and `s0`–`s15`), and the value of the `lr` register above the PSP and reads it from the updated PSP on exception return, so this accomplishes the exchange of whole task states.

The dispatcher entry point function `dispatch` invokes the dispatcher by pending PendSV. `request_dispatch` does the same, but the PendSV handler will not start running until all current exception handlers complete because PendSV is configured with the lowest priority. The other entry points are variants of `dispatch` that perform additional steps or use SVC for invoking the dispatcher, but their specifics are insignificant and thus fall out of this discussion's scope. Consequently, all entry points are ultimately handled uniformly, and so are the exit points; `start_r`, `dispatch_r`, `ret_int_r`, and `ret_exc_r`—all of these map to the dispatcher's register restoration code and the hardware exception return sequence.

4 This, however, does not mean the Arm-M port can be free of interrupt handling code because it has to strictly adhere to the kernel's various requirements.

Fig. 3.12 illustrates the execution flow when a task yields the processor to another task, e.g., because of a blocking system call. In this execution flow, the following events take place:

1. The kernel calls `dispatch`, which causes the activation of the PendSV handler.
2. When the processor activates an exception handler, it pushes an exception frame containing a partial context state to `task1`'s stack (pointed to by PSP). At the same time, it updates the `lr` register with a special value called `EXC_RETURN`, which includes the information necessary for restoring the original execution state, such as whether the exception frame includes floating-point registers.
3. The dispatcher code saves `EXC_RETURN`, PSP, and the rest of the context state (not depicted in the figure) to `task1`'s stack.
4. The dispatcher code restores the same kind of information as the previous step, but this time from `task2`'s stack.
5. The PendSV handler executes a `bx lr` instruction, which triggers the hardware exception return sequence. The processor restores the partial context state from the exception frame stored in `task2`'s stack, resuming `task2`'s execution.

3.3.3.3 *Changes in ASP3+TZ*

We modified the Arm-M dispatcher implementation to support Non-Secure task execution. The required changes were not substantial.

As we explained in Section 3.3.2, each task now has Secure and Non-Secure stacks and stack pointers (PSP). We modified the dispatcher to save and restore both stack pointers. `EXC_RETURN` includes a bit field indicating the security mode in which the exception was taken. The dispatcher preserves `EXC_RETURN`, so it restores the processor to the correct security mode without modification to the dispatcher.

The kernel code executes entirely in Secure mode, and so does the dispatcher. All of the entry points are only called by the kernel code, so for most cases, the dispatcher executes with a Secure background context. The only exception is when `request_dispatch` was called, in which case the processor first returns to the interrupt task and then activates the PendSV handler, so in this case, the background context can be Non-Secure.

3.3.4 *User Interrupt Handlers*

To handle interrupts with user (Non-Secure) interrupt handlers, the kernel must configure NVIC to reassign the interrupt lines to Non-

```
.text :                 /* Secure region */
{
    __text = .;
    *(.vector)
    *_s.o(.text)
} > FLASH

.text.sg :              /* Non-Secure Callable region */
{
    . = ALIGN(32);
    __secure_gateway_start = .;
    *(.text.sg)
    *(.text.sg.*)
} > FLASH
. = ALIGN(32);
__secure_gateway_end = .;

_etext = .;
PROVIDE(etext = .);

.text_nonsecure :       /* Non-Secure region */
{
    . = ALIGN(32);
    __nonsecure_text_data_start = .;
    *_ns.o(.text)
} > FLASH
. = ALIGN(32);
__nonsecure_text_data_end = .;
```

Figure 3.13: An excerpt from the linker script to segregate memory regions by security attribute.

Secure mode. TOPPERS/ASP3 provides a "static API" called `CFG_INT` that allows application developers to configure an interrupt line in the kernel configuration. We extended it to support a new flag `TA_NONSECURE` that, when present, reassigns the interrupt line to Non-Secure mode. Interrupt handlers, which are similarly registered by a static API, are assumed to be belonging to the security mode their interrupt lines are assigned to.

We modified the kernel configurator to generate Secure and Non-Secure exception vector tables. Each table only includes the interrupt handlers belonging to the corresponding security mode. Excluding interrupt handlers from the mismatching mode's table is not strictly necessary but limits the information exposure to Non-Secure code.

3.3.5 *Memory Map and SAU Configuration*

The kernel needs to know the range of addresses occupied by each domain's code and variables to configure SAU and assign appropri-

ate security attributes to memory regions. We modified the linker script to place each domain's symbols in a contiguous memory region and output the range of each memory region (Fig. 3.13). The instruction `. = ALIGN(32)` aligns the range to 32-byte boundaries to comply with SAU's requirement.

The linker provides the range as two symbols representing the endpoints. Fig. 3.14 shows an excerpt from the target-specific initialization code responsible for configuring SAU (Fig. 3.14) as well as registering the Non-Secure exception vector table (Section 3.3.4).

```
void core_initialize(void)
{
    /* ... */

    /*
     *  Setup SAU regions
     */
    extern void *__secure_gateway_start;
    extern void *__secure_gateway_end;
    sil_wrw_mem((void *)SAU_RNR, 0);
    sil_wrw_mem((void *)SAU_RBAR, (uint32_t)&
        __secure_gateway_start);
    sil_wrw_mem((void *)SAU_RLAR, SAU_RLAR_ENABLE | SAU_RLAR_NSC
         | ((uint32_t)&__secure_gateway_end - 32));

    extern void *__nonsecure_text_data_start;
    extern void *__nonsecure_text_data_end;
    sil_wrw_mem((void *)SAU_RNR, 1);
    sil_wrw_mem((void *)SAU_RBAR, (uint32_t)&
        __nonsecure_text_data_start);
    sil_wrw_mem((void *)SAU_RLAR, SAU_RLAR_ENABLE | ((uint32_t)&
        __nonsecure_text_data_end - 32));

    extern void *__nonsecure_bss_start;
    extern void *__nonsecure_bss_end;
    sil_wrw_mem((void *)SAU_RNR, 2);
    sil_wrw_mem((void *)SAU_RBAR, (uint32_t)&
        __nonsecure_bss_start);
    sil_wrw_mem((void *)SAU_RLAR, SAU_RLAR_ENABLE | ((uint32_t)&
        __nonsecure_bss_end - 32));

    /*
     *  Enable SAU
     */
    sil_wrw_mem((void *)SAU_CTRL, SAU_CTRL_ENABLE);

    /*
     *  Setup Non-Secure vector table
     */
    sil_wrw_mem((void*)NVIC_VECTTBL_NS, (uint32_t)
        vector_table_nonsecure);
}
```

Figure 3.14: An excerpt from the kernel's target-specific initialization code.

3.4 Evaluation

We conducted a performance evaluation on ASP3+TZ to verify our hypothesis that a memory-protection-enabled operating system can offer a lower overhead by using CMSE and adopting the proposed Single Binary Image architecture, compared to the traditional ring protection-based approach. To the best of our knowledge, this is the first time a CMSE-based operating system design and the performance quantification of such an operating system are presented.

As we stated in Section 3.2, we intend the proposed design to be implemented as a patch to an existing operating system kernel. The amount of required modification has a profound impact on the engineering cost. Therefore, we also measured the amount of modification we needed to implement ASP3+TZ to show that the proposed design can be implemented with a small modification, hence only incurring a modest engineering cost.

For the performance evaluation, we measured system call execution times and interrupt response times on ASP3+TZ. We did the same measurement on TOPPERS/ASP3, which does not support memory protection, and TOPPERS/HRP2 [118], which implements memory protection using a ring protection mechanism for comparison (see Section 2.1.1 for an overview of TOPPERS kernels). Since TOPPERS/ASP3 does not have the concept of protection domains, we constructed the benchmark program for TOPPERS/ASP3 on the assumption that it has only a system domain.

All the results shown here were taken on Arm Versatile Express Cortex-M Prototyping System (V2M-MPS2+) [22] programmed with *Cortex-M33 Example IoT FPGA Image for MPS2+*, which includes a Cortex-M33 processor and CoreLink SIE-200 System IP. The processor is connected via an AHB5 Memory Protection Controller (MPC) to a total of 8MB of single-cycle SRAM with a 16-bit data bus on the board, on which the benchmark code and data were placed. The built-in UART interface was used for result acquisition.

3.4.1 *Service Call Execution Time*

We measured the execution times of `act_tsk`, a kernel API function defined by the TOPPERS Third-Generation Kernel Specification [100] to activate a task. Calling `act_tsk` has two possible outcomes depending on the activated task's relative priority: if the activated task has a higher priority than the current task, it performs dispatching and transfers the control to the activated task; otherwise, it returns to the calling task without dispatching. For the non-dispatching cases, we measured the duration of the call to `act_tsk`. For the dispatching cases, we measured the duration between the entry to `act_tsk` and

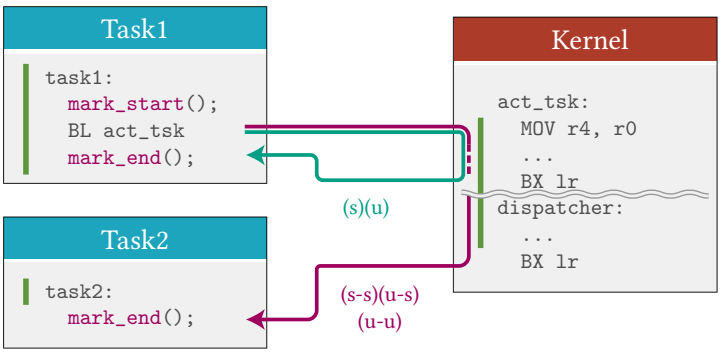


Figure 3.15: The method for measuring the execution time of `act_tsk`.

Table 3.2: The test cases for measuring the execution time of `act_tsk`.

	TASK 1	TASK 2	DISPATCHING?
(s)	system domain		no
(u)	user domain		no
(s-s)	system domain	system domain	yes
(u-s)	user domain	system domain	yes
(u-u)	user domain	user domain	yes

the execution of the second task's first statement (Fig. 3.15). The test cases are listed in Table 3.2.

3.4.1.1 *Result and Discussion*

Fig. 3.16 shows the measured execution times of `act_tsk`. This result shows that ASP3+TZ realizes memory protection at a significantly lower overhead compared to TOPPERS/HRP2, albeit with some functional restrictions.

The key difference between the implementations of ASP3+TZ and TOPPERS/HRP2 is the number of instructions taken to make a system call. In both operating systems, making a system call from a user domain requires a transition to a kernel domain before reaching the called function's core portion. This transition required 33 instructions in TOPPERS/HRP2, not to mention the hardware exception handling sequences involved within, but only five instructions in ASP3+TZ (Fig. 3.17). Despite this striking difference, ASP3+TZ does not weaken security assumptions in any way; for instance, it does not assume the (compiled) binary code's trustworthiness like Safer Sloth's `MPU+itraps` mode does [38] or leave it open to return-to-user attacks like FreeRTOS-MPU's current implementation does [112].

Dispatching in ASP3+TZ exhibited a modest overhead compared to TOPPERS/ASP3 because ASP3+TZ introduces only minimal changes

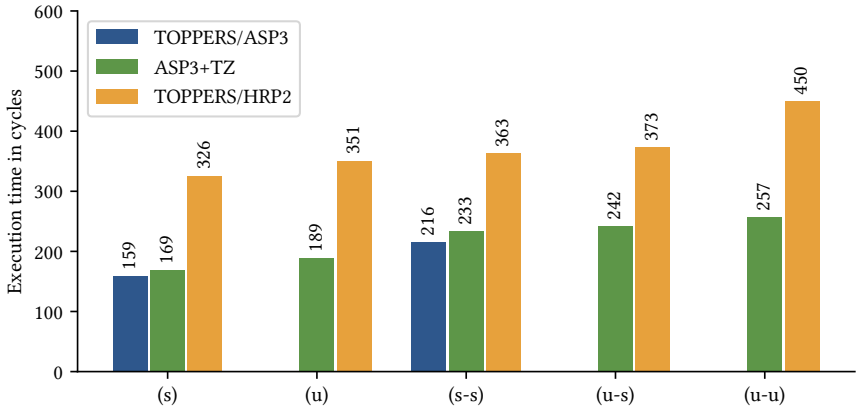


Figure 3.16: The execution time comparison of `act_tsk`.

```
  200044:   f000 b82c   b.w 2000a0 <__ns_act_tsk_veneer>
002000a0 <__ns_act_tsk_veneer>:
  2000a0:   f85f f000   ldr.w   pc, [pc]
10000114 <ns_act_tsk>:
10000114:   e97f e97f   sg
10000118:   b500        push    {lr}
1000011a:   f002 fb89   bl  10002830 <act_tsk>
10002830 <act_tsk>:
```

`__ns_act_tsk_veneer` was generated by the linker because `ns_act_tsk` falls outside the reach of a branch instruction.

Figure 3.17: The instruction sequence executed during the call to `act_tsk` from the user domain in ASP3+TZ.

to the dispatcher, specifically the Non-Secure stack pointer preservation and restoration as explained in Section 3.3. Switching to the correct security mode does not require the dispatcher's awareness because the mode is embedded in `EXC_RETURN`'s bits, and it is the hardware that takes it into account during the exception return sequence.

We stated in Section 3.2 that the supposed smaller overhead of a CMSE-based memory protection implementation would allow more fine-grained partitioning for the same overhead. The result shown here supports this conjecture.

3.4.2 *Interrupt Response Time*

We measured the interrupt response times of an interrupt handler belonging to each domain. The interrupt response times were acquired by reading the timer value in the hardware timer's interrupt handler.

We defined the user interrupt handler in the following ways for each target operating system:

Table 3.3: The test cases for measuring the interrupt response time.

	TASK BELONGS TO	HANDLER BELONGS TO
(s-si)	system domain	system domain
(s-ui)	system domain	user domain
(u-si)	user domain	system domain
(u-ui)	user domain	user domain

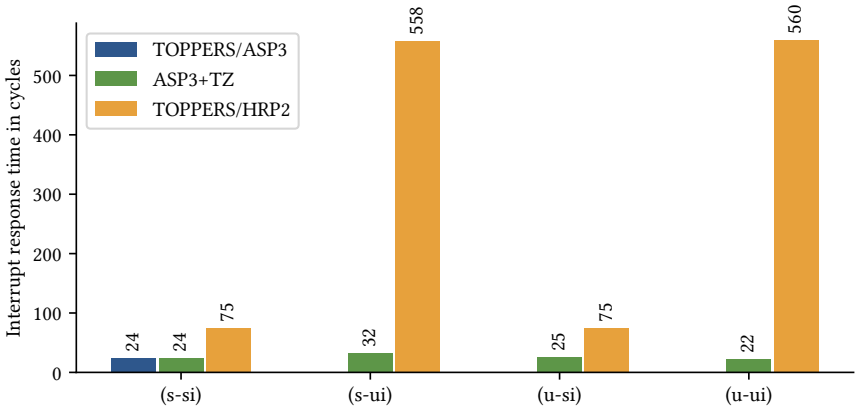


Figure 3.18: Comparison of interrupt response times.

- TOPPERS/ASP3 does not have the concept of domains, so we excluded it from the test cases involving user domains.
- ASP3+TZ natively supports user interrupt handlers. We passed the `TA_NONSECURE` flag to `CFG_INT` and placed the interrupt handler in a Non-Secure memory region.
- TOPPERS/HRP2 does not support user interrupt handlers. We emulated one by defining a user task and having it activated by an interrupt handler.

The test cases are listed in Table 3.3.

3.4.2.1 *Result and Discussion*

Fig. 3.18 shows the measured interrupt response times. The result shows ASP3+TZ realizes user interrupt handlers with response times that are considerably close to those of TOPPERS/ASP3 and far lower than the emulated interrupt handlers on TOPPERS/HRP2. The response times did not vary much depending on the interrupt handler's domain because the hardware takes care of most of the process for all cases. Since ASP3+TZ utilizes the hardware's existing ability to assign interrupt lines to security modes, the existence of user interrupt

Table 3.4: Kernel code additions and deletions, measured in lines.

CODE SECTION	TOPPERS/ASP3	ADDITIONS	DELETIONS
Architecture-specific	1884	419 (+22.2%)	11 (-0.6%)
Common	7502	28 (+0.4%)	4 (-0.1%)

handlers does not influence other interrupt handlers' overhead. The response time for (s-ui) was slightly longer because the hardware had to save all general-purpose registers to the stack and clean them not to expose the execution state of Secure code to Non-Secure code.

3.4.3 *Kernel Code Modification*

We counted the modified lines of code in ASP3+TZ's implementation to evaluate how ASP3+TZ was successful at keeping the engineering cost minimal. The number of original lines were acquired using *cloc* tool[5], which excludes blank and comment lines. The numbers of changed lines were acquired by extracting the modified parts as a patch file using the `git diff` command and then manually removing blank and comment lines from the generated patch file.

Table 3.4 shows the result. From the result, we observe that ASP3+TZ introduced 22.2% and 0.4% increases in the numbers of lines in the architecture-specific part and the common part, respectively. The change in the common part was to support Non-Secure stacks. Based on this result, we confirm that the proposed design only requires a modest amount of modification to the base kernel code.

3.4.4 *Unimplemented Features*

As we stated in Section 3.3, ASP3+TZ's current implementation only supports a subset of the features shown in Section 3.2. In this section, we discuss the expected performance impact and engineering cost of the missing features.

Kernel Object Access Control

An existing technique can be applied to implement kernel object access control. The simplest permission model is a bitmap-based model such as the one used in TOPPERS/HRP2 [118], in which a permission bitmap specifies whether the operation is allowed for each combination of (domain, object, operation). We anticipate that adopting this

5 `https://github.com/AlDanial/cloc`

model has little impact on the system call execution times because reading a permission bit is all it takes.

Multiple User Domains

Supporting multiple user domains requires the dispatcher to reconfigure SAU, which increases the context switching latency.

User interrupt handlers pose a bigger problem. Although CMSE can automatically switch security modes upon taking an interrupt, it does not reconfigure SAU. This restriction means user interrupt handlers cannot be directly taken by Non-Secure mode; the kernel must first take the interrupt in Secure mode, reconfigure SAU for the intended domain, and then call the Non-Secure interrupt handler. This procedure is similar to how user interrupt handlers would be implemented under ring protection. However, since CMSE allows Handler mode execution in Non-Secure mode, this requires fewer mode transitions and incurs a lower overhead.

Unprivileged Functions

Unprivileged functions can be implemented by wrapper code that clears general-purpose registers and calls the inner function by a `blxns` instruction. They are entirely disconnected from other features and do not affect the performance of system calls or interrupt handling.

In conclusion, we expect object access control and unprivileged functions to have little impact on the kernel's runtime overhead and engineering cost. Supporting multiple user domains might be more costly, but we expect it to be more performant than traditional operating system designs. Supporting multiple user domains is not strictly necessary in many systems; for example, one could isolate all untrusted third-party code in a single user domain to protect their safety-critical code. Furthermore, the implementation complexity has a severe impact on the verification cost. Therefore, we argue that supporting only one user domain is still a case that is worth considering.

3.5 Conclusion

IoT devices demand a higher level of security than expected in traditional embedded systems, and it is vital to employ systematic measures, such as software partitioning, to prevent fault propagation and information leakages. Memory protection is often implemented by a processor's ring protection mechanism. However, owing to the ring protection mechanism's highly generic and minimal nature, such as its reliance on an interrupt handling mechanism to enter kernel code, this approach incurs a significant software overhead in terms of execution latency and code size, preventing its use in resource-constrained systems. Therefore, we proposed a new lightweight memory protection scheme that utilizes TrustZone for Armv8-M.

The proposed scheme's implementation is a series of small modifications to a base kernel having no memory protection support. The proposed scheme leverages TrustZone for Armv8-M's hardware support for mode-crossing function calls to alleviate the software overhead associated with service calls and interrupt handling. We applied this scheme on TOPPERS/ASP3 to create an RTOS kernel with memory protection, named ASP3+TZ. We compared its service call overhead and interrupt response time to those of an existing RTOS kernel with memory protection and demonstrated that the proposed scheme incurs far less overhead than a traditional memory protection implementation. Lastly, we presented the number of code lines modified as part of the implementation to show that the proposed scheme only requires little engineering cost.

4

TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone For Armv8-M

Arbitrary code execution is by far the most serious threat to software. W⊕X is a memory access policy that mitigates such attacks by disallowing memory regions from being writable and executable at the same time and can be efficiently implemented on standard hardware using a ubiquitously-available MMU or MPU. However, there is a class of attack techniques known as Return-Oriented Programming [93], which is resistant to existing mitigation techniques for code injection.

Control-flow integrity [16] is an effective defensive technique against ROP attacks. However, existing CFI techniques mostly presume the existence of specific hardware features, and even so, impose a high runtime overhead, preventing their uses in embedded systems. The latest microcontrollers for small embedded systems include hardware-supported security mechanisms such as TrustZone for Armv8-M [12]. In a small embedded system, an operating system kernel, along with other software components (such as a protocol stack, which requires frequent interaction with networking hardware) having the same privilege level as the kernel, can comprise a substantial portion of the whole system. Nevertheless, existing RTOS-compatible CFI solutions such as [108] did not protect the critical CFI internal state from such components. With the emerging IoT trend, there is now a new level of market demand for operating systems that can operate on microcontrollers [88]. This has motivated us to develop *TZmCFI*, a lightweight CFI implementation for RTOS-based applications running on such hardware.

Unlike desktop and mobile applications, which have often been the subject of previous researches on CFI, embedded applications usually include asynchronous exception handlers (including both external interrupts and CPU-generated exceptions) as part of them. To protect asynchronous exception handlers, we propose *safe shadow exception stacks*, a variant of the traditional shadow stack technique that leverages TrustZone for Armv8-M.

This chapter presents TZmCFI's detailed design and implementation. We sort out the part of the implementation we call *the toolkit* and evaluate the overhead on a now-available mass-produced microcontroller. Furthermore, we provide a thorough, source-code-level discussion on the implementation of a practical RTOS-aware CFI mechanism for microcontroller-based embedded systems. We believe that

the information presented herein will be useful to researchers and practitioners who target Armv8-M as well as other architectures providing a similar mechanism to TrustZone.

The lack of access to source code is a continuing problem in almost every computer science field, hindering validation and code reuse. CFI is no exception; it is not often for CFI publications to be accompanied by released source code aside from a few exceptions, notably [19, 103, 106]. To remedy the problem, we have released TZmCFI's source code at `https://github.com/TZmCFI`, ensuring reproducibility and promoting further research. To the best of our knowledge, this also marks the first time the source code of a CFI mechanism leveraging TrustZone for Armv8-M is made public.

The main contributions of this chapter's work are the following:

- We discuss the implementation techniques used in the build toolchain and the runtime library of TZmCFI in detail (Section 4.4).
- We propose *safe shadow exception stacks*, a variant of the traditional shadow stack technique that leverages TrustZone for Armv8-M and carefully avoids a caveat associated with Armv8-M's exception handling to protect exception handlers (Section 4.2). We implement this technique in TZmCFI.
- We evaluate the prototype system based on TZmCFI from a performance point of view using a mass-production commercial chip, NXP Semiconductors LPC55S69 (Section 4.5).

4.1 Exception Handling in Armv8-M

The exception handling of the Armv8-M architecture is designed for embedded applications written in a high-level programming language. This is accomplished mainly through hardware interrupt sequences that automatically save and restore caller-saved registers to and from the interrupted context's stack and *not* disabling interrupts at interrupt entry to minimize interrupt latency. This design benefits embedded application developers by allowing them to write interrupt handlers without using special compiler directives or writing assembly code and by offering a lower interrupt latency than usually possible by a software-based interrupt entry sequence. When it comes to our CFI solution, this design serves as a double-edged sword.

4.1.1 *The Armv8-M Exception Handling Model*

Armv8-M features an exception handling model in which an interrupt controller, called *Nested Vectored Interrupt Controller (NVIC)*, is tightly integrated into its system-level programming model. Exception sources, including CPU-generated exceptions as well as external interrupt sources, are distinguished by *exception numbers*. NVIC maintains a pending bit for each exception source, which is set in response to the occurrence of the corresponding type of event. When NVIC *activates* a pending exception, it starts a hardware exception entry sequence and transfers the control to the location specified by an exception vector table. An exception is said to be *active* while its exception handler is running.

NVIC tracks and maintains *the current execution priority*. NVIC masks all exception sources whose priorities are configured to be less than the current execution priority. Activating an exception raises the current execution priority to that exception's priority, preventing the activation of lower-priority exceptions while allowing higher-priority ones to be still accepted (this behavior is commonly referred to as *exception nesting*).

The current execution priority can also be manipulated through two software-accessible registers: `FAULTMASK` and `PRIMASK`. When set to a non-default value, they raise the execution priority, effectively disabling exceptions[1]. The `CPS` instruction allows software to manipulate these registers quickly, and using this instruction is the standard way to enable or disable interrupts globally.

The hardware exception entry sequence is comprised of the following steps. Firstly, the processor pushes the current context state onto the current context's stack. The data pushed to the stack is called an *exception frame* and includes the values of some subset of general-

1 NMI cannot be disabled in this way.

purpose registers and the original program counter. The processor updates LR, the ABI-defined return address register, with a special value called EXC_RETURN. Finally, the processor loads a vector address from an exception vector table using the exception number as the offset from the table base address (VTOR) and transfers the control to it.

EXC_RETURN is a special program counter value that triggers an exception return sequence. When the program performs an indirect jump to LR (which is no different from a standard function return), and it contains EXC_RETURN, the processor does not update the program counter but instead initiates an exception return sequence, in which the original context state is restored from the exception frame at the top of the background thread's stack. The processor maintains two stack pointers called MSP (Main Stack Pointer; mainly used in exception handlers) and PSP (Process Stack Pointer; only used in OS tasks). This process utilizes information from bit fields embedded in EXC_RETURN to determine which stack pointer was in use when the execution was interrupted, i.e., which stack the exception frame is located in.

4.1.2 *Exception Entry Chain*

The important thing to notice here is that the exception entry sequence only raises the current execution priority to the current exception's priority. The hardware exception entry sequence executes atomically from a software point of view, and during this process, it never disables exceptions globally at any moment. Higher-priority exceptions can preempt the current one anytime during an exception entry sequence even before the software is given a chance to execute (we refer to this phenomenon as *exception entry chain*). While this is preferable for a lower interrupt latency, this undermines a shadow stack-based CFI scheme's soundness. The invariant of a shadow stack implementation is that it creates a copy of trustworthy code pointers before they can be corrupted by untrusted code. In Section 4.2, we propose a variant of shadow stacks that upholds this invariant even under the presence of exception entry chains, although at a moderate runtime overhead.

The Armv8-M specification [12] is very unclear on the exact conditions in which exception entry chains take place. The experiments we have conducted merely to measure the impact on interrupt latency in this work, and we have failed to capture the naturally-occurring instances of exception entry chains so far. This observation only leads to a question: *Are exception entry chains real?*

It is indeed possible for an exception to be taken on a CPSID I instruction in general. Generalizing this fact, one would assume [87]

```
TopLevel()
  InstructionExecute()
  HandleExceptionTransitions()
    PendingExceptionDetails() → exception_1
    ExceptionEntry(exception_1)
      ExceptionTaken(exception_1, false,
          ...)
TopLevel()
  InstructionExecute()
  HandleExceptionTransitions()
    PendingExceptionDetails() → exception_2
    ExceptionEntry(exception_2)
      ExceptionTaken(exception_2, false,
          ...)
```

Figure 4.1: The execution flow of the pseudocode when handling two exceptions.

that an exception can be taken on the first `CPSID I` instruction of an exception handler, i.e., exception entry chains are unavoidable.

The Armv8-M specification specifies the intended behavior of a processor in terms of pseudocode. We found that, according to the pseudocode, there is a specific condition that causes an exception entry chain. `TopLevel` is the top-level function of the pseudocode, which is called in a loop and calls other functions named `InstructionExecute` and `HandleExceptionTransitions` alternately (Fig. 4.1). This might give a false impression that an exception entry chain would never occur, until one takes a closer look into `HandleExceptionTransitions`. `HandleExceptionTransitions` is responsible not only for exception entry but also for exception return and *tail-chaining*, and all of these three can take place in a single call to `HandleExceptionTransitions`. Tail-chaining is an implementation-specific optimization allowed by the Armv8-M specification, in which a processor skips the usual context state stacking and restoration when an exception return is immediately followed by another exception entry. To implement tail-chaining, `ExceptionReturn` checks an interrupt pending flag, and if it is set, executes the modified exception entry sequence for tail-chaining. This means `HandleExceptionTransitions` can perform two exception entries in a single run (Fig. 4.2), which manifests as an exception entry chain from a software point of view.

We created a small program to exercise the suspected execution flow of the pseudocode, yet failed to produce an exception entry chain. This is because another implementation-specific optimization called *late arrival* was at work. When a processor receives a higher-priority interrupt X while executing the entry sequence for a lower-priority interrupt Y, the processor is allowed to repurpose the already-

```
TopLevel()
  InstructionExecute()
  HandleExceptionTransitions()
    ExceptionReturn()
      PendingExceptionDetails() →
          exception_1
      TailChain(exception_1, ...)
        ExceptionTaken(exception_1, true,
            ...)
    PendingExceptionDetails() → exception_2
    ExceptionEntry(exception_2)
      ExceptionTaken(exception_2, false,
          ...)
```

Figure 4.2: The pathological execution flow of the pseudocode that causes an exception entry chain.

or partially-stacked context state for X instead of taking X as a nested exception. The point in an exception entry sequence until which a higher-priority interrupt is considered late-arriving varies across implementations. Cortex-M3 [3] defines this point as the Execute stage of the first instruction of the lower-priority interrupt handler, making exception entry chains entirely avoidable. On the other hand, Cortex-M1 [4] defines this point at an earlier point, making it susceptible to exception entry chains. The other Arm-M implementations that exist and have already been shipped at the point of writing (M0 [6], M4 [7], M7 [11], and in particular, M23 [9] and M33 [14], which are in our work's scope) are unclear on this matter. The support for CMSE in M23 and M33 makes the exception entry sequence quite complicated and creates many potential execution paths that are hard to cover with a manually-crafted test program.

In the absence of sufficient assurance that exception entry chains can be reliably suppressed by software, we conclude that we cannot dismiss the possibility of exception entry chains.

4.2 Safe Shadow Exception Stack

Enforcing CFI on asynchronous exceptions requires the use of a shadow stack-based technique. In some sense, exceptions and function calls are very alike because they both transfer the control to some point and return to the original location using an indirect jump instruction. For exceptions, W⊕X and vector table address protection (Assumptions 3 and 6 from Section 4.3.1) imply inline checking is required only for exception returns. However, the set of the potential jump targets of an exception return includes every executable instruction in the application, rendering static CFI schemes ineffective. For this reason, shadow stacks must be used to enforce CFI on exception returns. This adaptation of shadow stacks to protect exception returns, which we call *shadow exception stacks*, plays an important role in our CFI solution (Section 4.3.2.2).

Two pieces of code are inserted to the front and back of each exception handler to leverage a shadow stack. They are called *exception trampoline* and *exception return trampolines*, respectively.

There is another difference between exceptions and function calls: exception enter/exit is often handled by a sequence hard-coded onto a processor. This poses a challenge in sound implementation of a shadow stack, especially under the presence of *exception entry chains* (Section 4.1.2), which are specific to Arm-M.

Therefore, we propose *safe shadow exception stacks*, a variant of the traditional shadow stack scheme, for enforcing CFI on exception returns while addressing the issue with exception entry chains.

4.2.1 *Naïve Shadow Stacks*

As noted in Section 4.1.2, Arm-M's exception entry sequence only raises the current execution priority to the exception's priority value and does not entirely disable exceptions. Higher-priority exceptions can preempt the current one anytime during an exception entry sequence even before the software is given a chance to execute. The purpose of having a shadow stack is to create a copy of code pointers written on user-accessible memory before they can be corrupted by untrusted code, so this undermines the soundness of a shadow stack-based CFI scheme. More specifically, a higher-priority exception handler can preempt and corrupt the program counter stored in the current exception's exception frame before it is copied to a protected location by the exception trampoline (Fig. 4.3).

Although it is not directly supported by the hardware, it is possible to avoid the problem by disabling nested exceptions. However, disabling nested exceptions increases the latency of high-priority in-

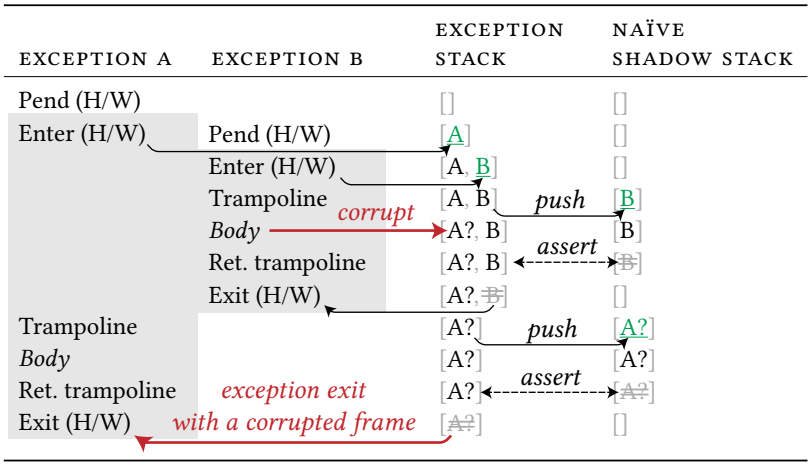


Figure 4.3: The Naïve SES implementation is unsound under the presence of an exception entry chain.

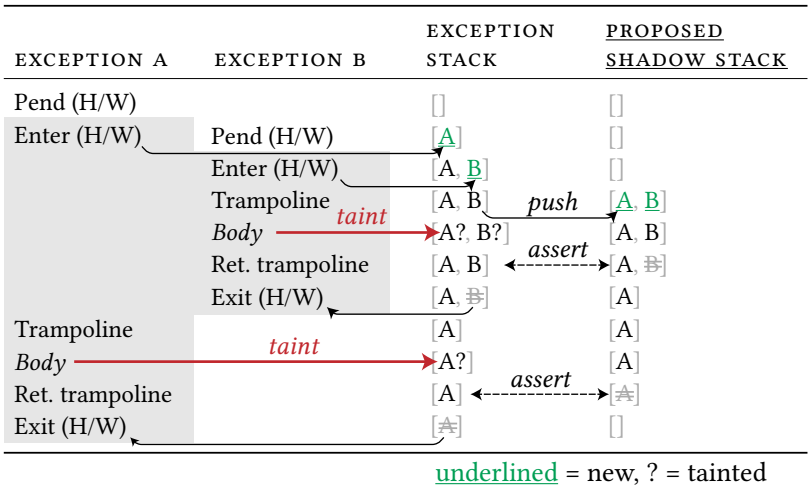


Figure 4.4: The Safe SES implementation handles an exception entry chain correctly. Compare to Fig. 4.3.

terrupts. Therefore, we propose the solution supporting nested exceptions in Section 4.2.2.

4.2.2 *Proposed Solution*

We propose a solution to address the issue described in Section 4.2.1. The basic idea is, whenever the exception trampoline is executed, to protect every exception frame on a stack, not just the latest one. The exception trampoline figures out and adds missing frames to the shadow stack.

Scanning a call stack to locate every active exception frame is inefficient and impractical in general. Actually, the exception trampoline only has to visit a particular subset of the stack, which can be easily

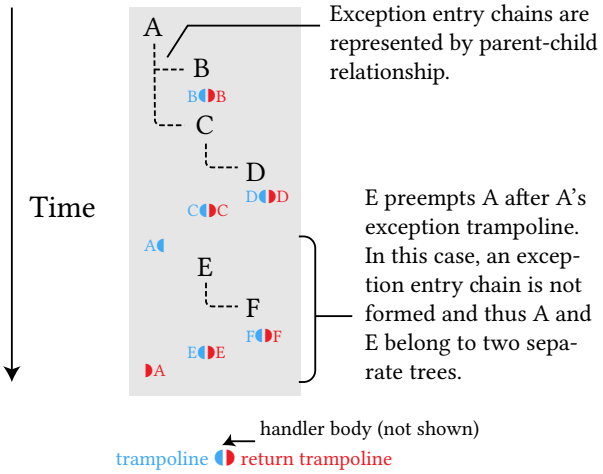


Figure 4.5: An exception entry chaining tree.

found. The subset is defined in terms of a relationship, which we call *exception entry chains*. We define the relationship in this way: two nested exception activations constitute an exception entry chain if they were activated successively without giving the first one's exception handler a chance to execute before the second one's exception entry sequence is started. Fig. 4.4 is an example where A and B form an exception entry chain. Locating an exception frame next to another is easy as long as their originating activations form an exception entry chain because, by the definition of an exception entry chain, the preempted exception handler has not executed any software code, so these two exception frames are adjacent to each other on memory with no intervening stack frames between them. For the reasons explained later in this section, the exception trampoline only has to find exception frames transitively related by exception entry chains to the current exception.

Exception entry chains can be considered as a parent-child relationship. Combined with a temporal relationship, this forms an ordered forest shown in Fig. 4.5. We call the trees in the forest *exception entry chaining trees*. It should be noted that two nested exceptions belong to separate trees if they do not form an exception entry chain (e.g., A and E in Fig. 4.5). The path from a node to the root of the tree it belongs to is a suffix of the exception stack, which we call an *exception entry chaining stack* ($EECSt$).

The rest of this section gives an explanation of how the exception trampoline and the exception return trampoline work in the proposed solution. Here it is assumed that software code does not corrupt exception frames. Such situations are discussed in Section 4.2.3.

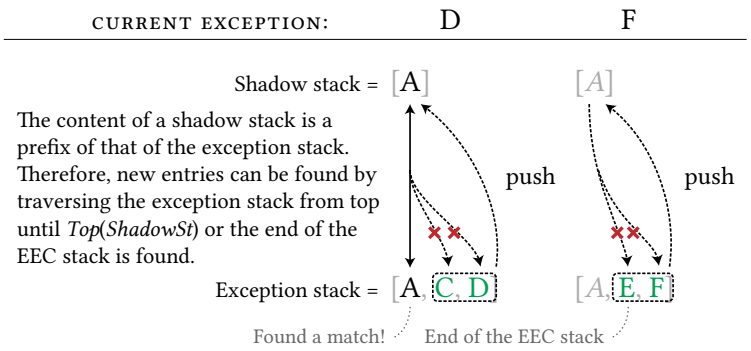


Figure 4.6: The exception trampoline's push loop synchronizes the stacks by scanning an EEC stack and copying newly found entries.

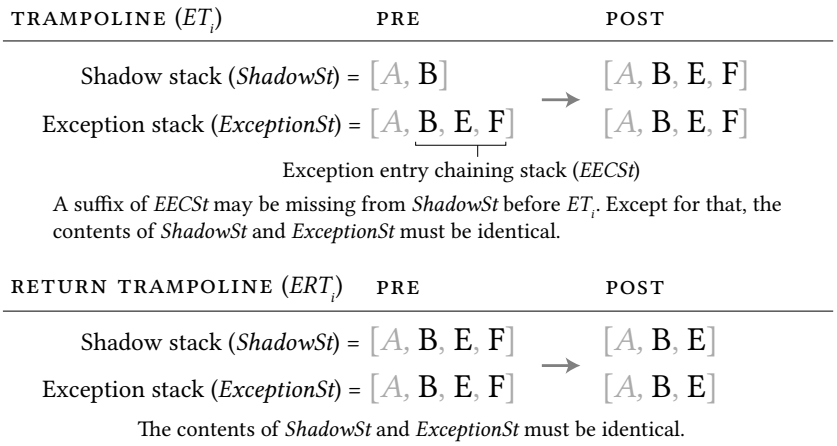


Figure 4.7: The pre/postconditions of the exception trampoline (ET_i) and the exception return trampoline (ERT_i).

For an exception activation i, the pre/postconditions of the exception trampoline (ET_i) and the exception return trampoline[2] (ERT_i) are shown below and illustrated in Fig. 4.7:

$$
\begin{aligned}
Pre(ET_i) &: \exists n.(ShadowSt = ExceptionSt[0:n] \wedge \\
&\quad |ExceptionSt| - |EECSt| \leq n \leq |ExceptionSt|) \\
Post(ET_i) &: ShadowSt = ExceptionSt \\
Pre(ERT_i) &: ShadowSt = ExceptionSt \\
Post(ERT_i) &: ShadowSt = ExceptionSt
\end{aligned}
$$

2 Includes the exception return sequence in this informal proof.

The exception trampoline compares the shadow stack and the exception stack and pushes missing frames to the shadow stack. $Pre(ET_i)$ guarantees that they are found as a suffix of the exception entry chaining stack Fig. 4.6.

The exception return trampoline removes $Top(ShadowSt)$ as the upcoming exception return sequence removes the corresponding exception frame $Top(ExceptionSt)$. The integrity check between $ShadowSt$ and $ExceptionSt$ takes place here. It must check the $\min(2, |ShadowSt|)$ top elements instead of just one (elaborated in Section 4.2.3).

The proof of the preconditions $Pre(ET_i)$ and $Pre(ERT_i)$ is shown. In an exception entry chaining tree, ET_i and ERT_i are called in alternating order: $ET_1, ERT_1, \cdots, ET_n, ERT_n$. The calling order $1, \cdots, n$ obeys the post-order of the tree. Firstly, $ShadowSt$ does not contain any element of $EECSt$ when the exception trampoline is called for the first time in the tree. Therefore, $Pre(ET_1)$ is true. Secondly, non-EEC nested exceptions restore the stack to the original state after returning, thus $Post(ET_1) \rightarrow Pre(ERT_1)$. Finally, $Post(ERT_i)$ implies that $ShadowSt$ is a prefix of $ExceptionSt$ having the top element equal to the lowest common ancestor of i and $i+1$, thus $Post(ERT_i) \rightarrow Pre(ET_{i+1})$.

We implemented the proposed shadow exception stack algorithm as a part of TZmCFI. In the current implementation, each shadow stack entry contains five fields (`PC`, `LR`, `EXC_RETURN`, `R12`, and a pointer to the exception frame). Protecting `R12` is not essential as far as only exception handling is concerned. The reason `R12` is included is that it is used by shadow stack instrumentation code for passing a continuation address (the return address for a monitor call, which should not be confused with the return address of the instrumented function) as a part of its special calling convention, thus corrupting it may lead to a control-flow violation.

4.2.3 *Security Analysis*

A memory region is considered to be in the *tainted* state if potentially-malicious code with write permission on that memory region executes. A tainted memory region can be reverted (decontaminated) to the clean state by checking the contents against a secure copy created when it was known to be clean. Based on our threat model, this means that the entirety of the memory space writable by Non-Secure code is tainted every time untrusted code is executed.

A shadow exception stack-based CFI scheme must maintain the following invariants:

INVARIANT 1 *A tainted exception frame is not inserted into a shadow stack.*

INVARIANT 2 *The processor's exception return sequence does not read a tainted exception frame.*

INVARIANT 3 *The exception trampoline and exception return trampoline's control flow is not determined based on tainted data.*

The exception return trampoline compares $Top(ShadowSt)$ against the actual top exception frame, and should there be a discrepancy, it registers it as a security violation and terminates the program. Therefore, Invariant 1 implies Invariant 2. The exception trampoline scans the stack for new exception frames, which are not tainted by definition, hence Invariant 1. However, the algorithm looks ahead *by one frame* to determine if the stack traversal should be terminated. If the exception return trampoline were to check only the top frame, thus leaving an unchecked frame on the top, it would break Invariant 3. The proposed algorithm avoids this caveat by checking the $\min(2, |ShadowSt|)$ top frames.

4.3 TZmCFI

We developed TZmCFI to investigate the applicability of a shadow-stack-based system-level CFI scheme on embedded systems.

4.3.1 *Assumptions*

We presuppose the following assumptions on the instrumented code:

ASSUMPTION 1 *The code executing in Secure Mode and the generated exception trampolines are trustworthy.*

ASSUMPTION 2 *The target hardware does not have a vulnerability capable of defeating the isolation provided by TrustZone.*

ASSUMPTION 3 *The instrumented code is read-only to itself.*

ASSUMPTION 4 *The instrumented code follows a standard ABI.*

ASSUMPTION 5 *The source code of the instrumented code is available.*

ASSUMPTION 6 *The exception vector table offset register is immutable.*

ASSUMPTION 7 *Non-Secure interrupts never preempt Secure exception handlers.*

Assumption 5 imposes restrictions on the choice of third-party libraries. On the other hand, it provides an additional performance advantage by allowing more-efficient ways of inserting inline checks. For example, subroutines doing runtime checks could be invoked through procedure call instructions instead of trap instructions, which usually have a significant overhead associated with exception handling. With regard to Assumption 6, Cortex-M23/33 includes an implementation option for disabling writes to the said register. Secure code can enforce Assumption 7 by using `AIRCR.PRIS` (prioritize Secure exceptions) register.

We consider the threat model in which the attacker can read and write arbitrary memory locations accessible to the code running in Non-Secure Mode. This model is commonly used in previous works on CFI. We further strengthen the model by allowing the exploitation of exception handlers.

4.3.2 *Design*

The primary components of TZmCFI are a modified compiler and a supporting runtime component, *Monitor*, which can be preloaded to the target device. Fig. 4.8 shows how they position themselves within the build pipeline.

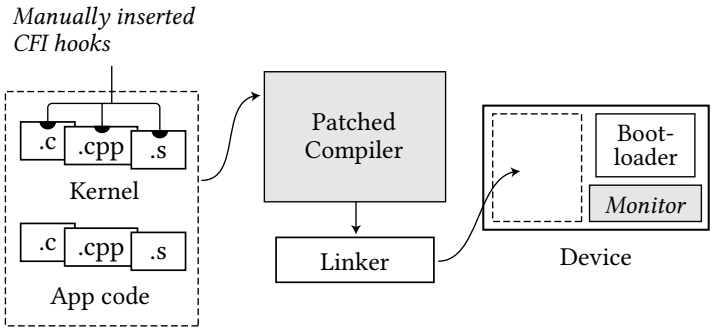


Figure 4.8: The workflow for adding CFI to an application.

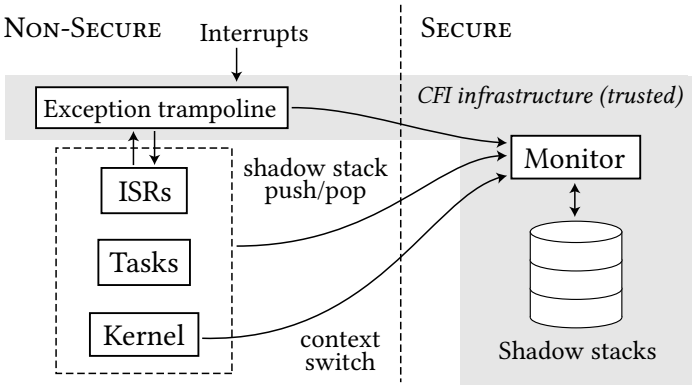


Figure 4.9: The runtime architecture of the proposed system.

We modify the LLVM compiler infrastructure [62] to instrument indirect branches in the compiled program. For forward edges, we leverage the existing forward-edge CFI mechanism of LLVM/Clang [106] to protect the control flow. For backward edges, we use a multitask-aware TrustZone-based implementation of shadow stacks that we developed for TZmCFI (described in Section 4.3.2.1).

To instrument exception handling, the exception vector table is replaced to point to exception trampolines (described in Section 4.3.2.2). The original exception vector table is preserved in a different location and is read by the exception trampolines to invoke the original exception handlers. Some portion of the exception trampolines resides in the Non-Secure code region, while the rest are implemented by Monitor.

Monitor is a software library responsible for maintaining the CFI implementation's internal state data and providing means to interact with it in a controlled way. The state data consists of the current task ID and shadow stacks. Protecting the state data is paramount to ensure TZmCFI's soundness, so we leverage CMSE to isolate the state data from untrusted code (Fig. 4.9). All data accesses to the state data are done through Secure functions in Monitor, all of which can be

called by the instrumented code only through secure gateways. CFI guarantees that all calls to them are legal.

Monitor provides several user-callable functions that the operating system should call through manually-inserted hooks on various occasions. When creating a task, the operating system has to call a specific monitor function named `TCCreateThread` to initialize the task structure internal to Monitor with an initial program counter and obtain a task ID. However, this is vulnerable to data-oriented attacks. To protect the system against such attacks, initializing task structures is permitted only during system startup. The operating system signals that the startup process is complete through a manually inserted hook, after which code pointers generated by untrusted code are no longer trusted. This state persists until a system reset. After a system startup, the operating system must notify context switches through another hook, passing the previously obtained task ID as a parameter.

4.3.2.1 *Shadow Stacks*

The shadow stack instrumentation in TZmCFI is a combination of compiler instrumentation and Secure gateways provided by Monitor. The modified LLVM compiler infrastructure inserts two kinds of shadow stack operations into the compiled code: *Shadow Push* and *Shadow Assert*, both of whose core portion reside in Monitor and execute in Secure mode through their respective Secure gateways.

The implementation of Shadow Assert (Return) comes with two flavors: on function return, the *aborting* flavor compares the current return target against the original return target popped from the stack, and if they are different, aborts the program immediately, while the *non-aborting* flavor simply replaces the current return target with the one from the shadow stack. From a security point of view, they both provide the same CFI guarantee—a code pointer is never created from non-trusted data, though the former is likelier to detect an incorrect behavior. From a performance point of view, the latter provides a lower overhead because only one copy of a return address needs to be preserved.

To support multitasking, each task is associated with its own shadow stack, which is stored as part of Monitor's task structure. When the operating system notifies the occurrence of a context switch through a Monitor hook, the current shadow stack is swapped with a new task's one.

4.3.2.2 *Shadow Exception Stacks*

On an exception[3] entry, the processor loads the program counter with an exception handler's address by fetching it from an exception vector table. Assumptions 3 and 6 guarantee that the vector table fetch is not susceptible to control-flow hijacking. However, the exception handler also needs to preserve the original context state by storing it somewhere vulnerable, often in the background context's stack. Besides a return address, the stored context state might contain additional code pointers that must be protected for comprehensive CFI enforcement. This requires a specialized shadow stack implementation, which we call *shadow exception stack*.

Because of their ability to preempt a program at almost any point, interrupt handlers do not adhere to the standard calling convention used in normal code in general. Fortunately, Arm-M is designed to allow interrupt handlers to be written very similarly to normal functions by performing some part of context stacking and restoration in hardware and treating a special return address (called `EXC_RETURN`) as a signal to start an exception return sequence in hardware. Taking advantage of this design, the shadow stack mechanism for an interrupt handler can be implemented by generating a wrapper function called an *exception trampoline*, which performs shadow stack operations before and after calling the original interrupt handler.

As explained in Section 4.2.1, a naïve implementation of shadow exception stack can leave the protection incomplete because of a peculiarity of Arm-M we call *exception entry chains* (Section 4.1.2). We present multiple solutions to this problem to be chosen based on an application's specific requirements and assumptions:

- *Adopt the naïve shadow exception stack anyway.* An application developer can take this option without risking security vulnerabilities if they can ascertain that the target processor implementation does not allow exception entry chains under any circumstances. We are not aware of any CMSE-capable Cortex-M cores that explicitly meet this criterion.

- *Disallow nested exceptions.* Disabling nested exceptions avoids this problem entirely at the cost of increased interrupt latency.

- *Apply the mitigation technique we proposed in Section 4.2.2.* This option, which we refer to as the *Safe* shadow exception stack implementation in this work, trades off runtime performance for comprehensive protection.

3 Again, *exceptions* refer to both of software-generated traps and hardware-generated interrupts.

MULTITASKING An operating system running on Arm-M usually performs context switching by triggering a specific type of software trap called PendSV. The PendSV exception handler of the operating system swaps the contents of the process stack pointer (PSP) register and several other registers with those of the next task, which were saved when the task was suspended last time. The operating system configures PendSV as the lowest priority exception, so the corresponding exception frame is always stored at the location pointed to by PSP (this would not be the case if PendSV could preempt another exception). Thus, swapping PSP and all of the remaining registers not included in an exception frame achieves context switching.

However, with a shadow exception stack mechanism in place, this procedure is now rejected as a security violation because the shadow exception stack mechanism does not permit the return target to change between exception entry and exit. Therefore, to support multitasking, TZmCFI must provide means to create a shadow exception stack separately for each task and switch them when the operating system invokes the monitor hook for context switching, allowing the PendSV handler to return to the interrupted location of another task.

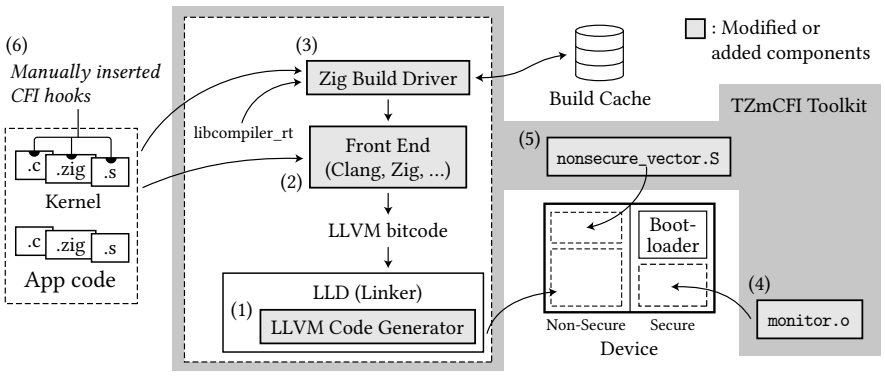


Figure 4.10: Implementation of TZmCFI.

4.4 Implementation

The primary components of TZmCFI are a modified build toolchain and a runtime library. These components can be packaged in a self-contained, distributable form, which we refer to as *TZmCFI Toolkit*.

We chose Zig [105] as the basis of TZmCFI's build toolchain. Besides its primary function as the compiler for the Zig programming language, Zig also serves as a build automation tool supporting C and C++ languages with hash-based automatic build artifact management, which abstracts away the complexities involved with a build process from developers. We made the following changes to the build toolchain. First, we modified the Arm backend of LLVM 10.0 to generate instrumentation code (Section 4.4.1, Fig. 4.10 (1)). Second, we modified the compiler front-end to support a new function attribute for interrupt handlers as well as to add new compiler flags to enable CFI instrumentation (Section 4.4.2, Fig. 4.10 (2)). Lastly, we modified the Zig build system to accommodate the new compiler flags (Section 4.4.4, Fig. 4.10 (3)).

We implemented Monitor, the runtime library for TZmCFI (Section 4.4.3). Monitor is comprised of a Secure static library that is intended to be linked to a bootloader (Fig. 4.10 (4)) and a portion of exception trampolines that is meant to be linked to a Non-Secure application (Fig. 4.10 (5)).

Lastly, Section 4.4.5 (Fig. 4.10 (6)) shows the changes we made to FreeRTOS to integrate it with TZmCFI. We also implemented two optimization techniques to lower the instrumentation overhead, which are explained in Section 4.4.6.

4.4.1 *LLVM Code Generator*

LLVM consumes the LLVM IR code fed by a front end. LLVM IR goes through multiple transformation passes. Numerous passes transform

Without SS instrumentation:

```
function:
  push   {r4, r5, r6, r7, r8, lr}
[...]
  pop    {r4, r5, r6, r7, r8, lr}
  bx lr
```

With SS instrumentation:

```
function:
  push   {r4, r5, r6, r7, r8, lr}
  addw   ip, pc, #8  ; set continuation
  ldr.w  pc, [pc]    ; jump to ...
  .word  __TCPrivateShadowPush
[...]
  pop    {r4, r5, r6, r7, r8, lr}
  ldr.w  pc, [pc]    ; jump to ...
  .word  __TCPrivateShadowAssertReturn
```

Figure 4.11: The generated assembler code of the shadow stack instrumentation in TZmCFI.

abstract input into more a target-specific, more concrete representation (this operation is called "lowering"), while other passes modify input code while retaining its abstraction level to improve certain qualities of the code or to attach analysis data for use by later passes. The last few passes are comprised of target-specific passes which are responsible for generating machine code.

Of such passes, two passes, in particular, are of interest to us: the prologue/epilogue generation pass and the assembly printer pass. The prologue/epilogue generation pass inserts the code that saves or restores callee-saved registers to each entry and exit point of a given function based on register allocation information. The assembly printer pass outputs code to an abstract output stream that can be backed by an assembler source file or directly fed to an assembler to produce an object file. This section shows the changes we introduced to these passes.

The shadow stack instrumentation in the prologue/epilogue generation pass borrows some implementation techniques from ShadowCallStack [103]. For each function, the modified instrumentation pass checks if the return target stored in the `lr` register is spilled to the stack. If that is the case, it inserts the following monitor calls into the function's prologue and epilogue (Fig. 4.11):

- *Shadow Push*, inserted to a prologue, pushes a return target to the current task's shadow stack.
- *Shadow Assert*, inserted to an epilogue, pops a trustworthy return target from the shadow stack, superseding the untrustwor-

thy one from the stack. If this type of monitor call is followed by a function return instruction, they are fused into a *Shadow Assert + Return* monitor call for improved runtime performance.

There was not much room for design choices on regard to where to insert the shadow stack monitor calls because doing it in the prologue/epilogue generation pass gives us access to important information such as the list of spilled registers. The reason knowing spilled registers is important is twofold: Firstly, we want to omit shadow push/return calls if `lr` is never spilled to memory. Secondly, the shadow stack instrumentation routines require scratch registers, and we might be able to get the extra miles by leveraging compile-time information about free registers (see Section 4.4.3.2 for more details).

The monitor calls are inserted in the form of newly added target-specific pseudoinstructions. These pseudoinstructions are expanded to target (real) instructions in the assembly printer pass. The prologue/epilogue generation pass can generate target instructions directly, but in this case, we decided not to do so for the monitor calls because we wanted to use a particular monitor call instruction sequence (Fig. 4.11) we deemed optimal in terms of code size, and the sequence is hard to emit in any other ways.

We borrowed the LLVM function attribute to mark instrumented functions from ShadowCallStack. This approach obviates the need to modify a compiler front-end to support our shadow stack instrumentation, though it might not be the ideal user interface design.

4.4.2 *Compiler Front-End*

Because Clang already supports ShadowCallStack and our shadow stack instrumentation "freeloads" on the same LLVM function attribute, no changes to Clang were needed. As for Zig, we added a new compiler option and modified the compiler to apply the function attribute globally when the option is present.

The forward-edge CFI included in LLVM [106] requires enabling LTO for all linked object files even if they do not have instrumented function calls. We modified Zig to support LLVM bitcode generation and LTO.

4.4.3 *Monitor*

Monitor, the runtime library for TZmCFI is comprised of two components: a Secure static library and a Non-Secure assembler source file containing a substitute Non-Secure exception vector table and exception trampolines.

The Secure portion of Monitor is responsible for managing thread contexts (Section 4.4.3.1) and manipulating shadow stacks (Section 4.4.3.2) and shadow exception stacks (Section 4.4.3.4). Exception trampolines (Section 4.4.3.3) mediate exception handling using the shadow exception stacks.

We implemented the Secure portion of Monitor as a Zig package. This package exports some Secure gateways (Section 4.4.3.5) for use by Non-Secure application as well as some C functions to be called by the Secure bootloader. Although a single compiled binary of the package can be used across almost all Cortex-M33 devices owing to the fact that it is mostly written in a chip-independent way, many aspects of Monitor can be customized by recompiling the package in the manner explained in Section 4.4.3.6.

4.4.3.1 *Thread Management*

Monitor manages thread-specific data blocks, each including the local state of a shadow stack (Section 4.4.3.2) and shadow exception stack (Section 4.4.3.4). The Secure gateway named `TCCreateThread` allocates memory for a new thread-specific data block, initializes it, and then returns the thread identifier assigned to that block. The mapping from thread identifiers to data blocks is maintained in an array with a preconfigured number of elements. The Secure gateway named `TCActivateThread` is used to switch the current thread to another one specified by a given thread identifier.

`TCCreateThread` allocates memory using a bump allocator, which only keeps track of the number of allocated bytes. This suffices because TZmCFI does not support task deletion and require all tasks to be created at startup time.

Some part of the local state of a current thread is cached in global variables. Although this design results in extra memory usage, it substantially lowers the overhead of shadow stack operations by removing extra indirections.

4.4.3.2 *Shadow Stacks*

Monitor exposes the Secure gateways shown in Table 4.1 to allow the instrumentation code to manipulate the shadow stack associated with the current thread. These gateways are, as we call them, *private* in the sense that they are only intended to be called by compiler-generated instrumentation code. Their names are prefixed with `__TCPrivate` to indicate as such. Shadow stack operations are implemented directly within a Non-Secure Callable region (i.e., not using veneer functions) to minimize the performance impact.

There are two primary operations supported by a shadow stack: *Shadow Push* and *Shadow Assert*. Shadow Push pushes the current

Table 4.1: Secure gateways for shadow stack operations.

NAME	CLOBBERED REGISTERS
`__TCPrivateShadowPush`	{r4, r5}
`__TCPrivateShadowAssert`	{r2, r3}
`__TCPrivateShadowAssertReturn`	∅
`__TCPrivateShadowAssertReturnFast`	{r2, r3}

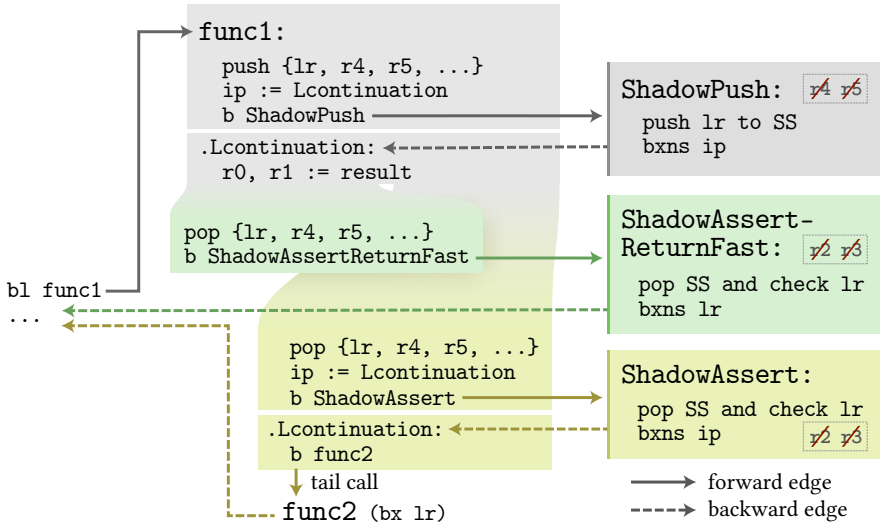


Figure 4.12: The execution flow of shadow stack operations.

value of the `lr` register to the shadow stack. Shadow Assert pops the top element from the stack and compares it to the current value of the `lr` register, and on mismatch, jumps to a panic handler. In the non-aborting flavor, Shadow Assert instead overwrites the `lr` register with the popped value. Shadow Assert and Shadow Push return to an instrumented function by jumping to a *continuation address* passed by the `ip` register.

In addition, our implementation supports the fused Shadow Assert + Return operation as a performance optimization. Shadow Assert + Return returns the control directly to the verified return address. The plain Shadow Assert is used for tail calls.

Fig. 4.12 illustrates the execution flow of the shadow stack operations around an instrumented function named `func1`.

SCRATCH REGISTERS These operations reserve and *clobber* some registers for parameter passing and as scratch registers. The reserved registers are chosen carefully to minimize performance impact. A function prologue saves the contents of callee-saved registers (a subset of `r4-r8`, `r10-r11`, `lr`) to use them as scratch registers during

a current invocation. An instrumented prologue performs Shadow Push right after saving callee-saved registers, meaning these registers are already available for use in Shadow Push. LLVM's register allocator chooses registers with smaller numbers first, so the chances are that `r4` and `r5` are chosen as scratch registers for many functions. Using a register not included in the set of saved callee-registers requires extra register preservation code. Therefore, choosing `r4` and `r5` minimizes the runtime and code overhead.

Shadow Assert follows the same pattern with the difference that it is called from a function epilogue. An instrumented epilogue performs Shadow Assert after restoring callee-saved registers, meaning they are not available for use in Shadow Assert[4]. The standard calling convention uses `r0–r3` for passing a return value, but it is rare to use more than one of them in practice. Therefore, Shadow Assert uses `r2` and `r3` as scratch registers.

As explained above, each shadow stack gateway assumes a specific set of scratch registers are available for use. In a small number of cases wherein this assumption violated, i.e., when the function requires fewer scratch registers or returns a value using more than two registers, our modified compiler emits register preservation and restoration code before and after the gateway calls, thus encompassing all possible cases.

Emitting register restoration code is not possible for Shadow Assert + Return, which returns directly to a caller. For Shadow Assert + Return, we took an alternative approach in which Monitor provides two versions of the gateway, one clobbering `r2` and `r3` and the other one clobbering no registers.

Updating Shadow Stack Each thread has its own copy of a shadow stack. The state of a shadow stack is comprised of an array storing a sequence of trustworthy return addresses and a field `top` pointing to the next element of the current top element. The value of `top` for the current task's shadow stack is cached in a global variable to avoid extra indirections and thus minimize the number of instructions needed by the shadow stack gateways to manipulate it. Implemented in this way, Shadow Push takes no more than four instructions to make a necessary update to a shadow stack (Fig. 4.13).

Overflow and underflow detection is done by utilizing an MPU. In the Armv8-M architecture, an MPU allows defining memory regions, each specified as a range of memory addresses, and assigning memory

4 Shadow Assert must take place at least after restoring `lr`. We could have split out the restoration code of `lr` from the epilogue, but function frames are laid out in a particular way for efficient preservation and restoration by `stmdb` (Store Multiple Decrement Before) and `ldmia` (Load Multiple Increment After) instructions. Any deviation from this way would require extra instructions, increasing the overhead. For this reason, we decided not to take this approach.

```
; Get &g_shadow_stack_top
ldr r5, .L_g_shadow_stack_top_const1
; Get g_shadow_stack_top
ldr r4, [r5]
; g_shadow_stack_top[0] = lr
; g_shadow_stack_top += 1
str lr, [r4], #4
; Store the new value of g_shadow_stack_top
str r4, [r5]
```

Figure 4.13: The portion of the Shadow Push routine responsible for updating a shadow stack.

access permissions (read, write, and execute) for each privilege mode. However, Armv8-M's MPU design only provides a handful of combinations of permissions out of all imaginable combinations based on assumptions such as that all memory addresses should be readable by privileged software. Interrupt handlers always execute in Handler mode, which forces the execution to be in Privileged mode. This means that defining two *guard* MPU regions to surround a shadow stack storage cannot catch all stack overflows and underflows. An interesting property of the MPU design is that it rejects any memory access if it spans over more than one defined memory region. We exploited this property by defining another unprivileged-read-writable memory region so that it overlaps with the guard memory regions, thus effectively turning the guard memory regions into no-access regions.

Monitor reconfigures the MPU whenever a context switch takes place. Because Secure software might use the MPU for other purposes, Monitor delegates MPU reconfiguration to external code through a compile-time interface.

4.4.3.3 *Exception Trampolines*

The exception trampolines and return trampolines of TZmCFI are comprised of the following stages (Fig. 4.14):

- The *Stage0* routines are the first stage of the trampolines and execute in Non-Secure mode. There is a separate instance of the Stage0 routine for each exception number. Stage0 masks all interrupts and invokes Stage1, passing `EXC_RETURN` and the corresponding Stage2 routine's address.
- The *Stage1* routine is entered by a Secure gateway named `__TCPrivateEnterInterrupt`. The shadow push operation of the shadow exception stack implementation (Section 4.4.3.4) takes place here.

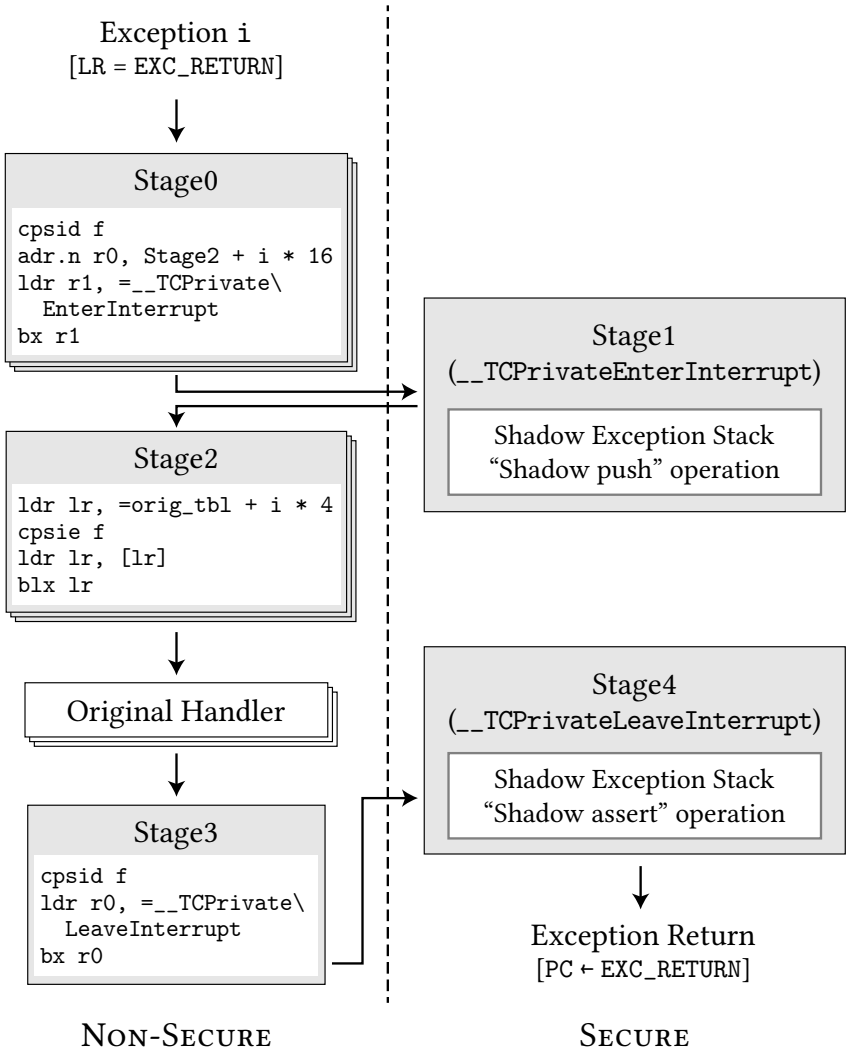


Figure 4.14: The execution flow of the exception trampolines and exception return trampolines.

- The *Stage2* routines re-enable interrupt globally, read the original exception vector table, and the original exception handler corresponding to a current exception. The `lr` register is updated to point to the Stage3 routine. The original value of `EXC_RETURN` is passed by the `r0` register.

  As with Stage0, there is a separate instance of the Stage2 routine for each exception number.

- The original exception handler returns the control to the *Stage3* routine, which unmasks interrupts and transfers the control to Stage4.

- The *Stage4* routine is entered by a Secure gateway named `__TCPrivateLeaveInterrupt`. The shadow assert and pop operation of the shadow exception stack implementation (Section 4.4.3.4) takes place here. Stage4 triggers the hardware-based exception return sequence by updating the program counter with `EXC_RETURN`, which was just popped from the shadow exception stack.

The first instructions of the *Stage0* routines mask all interrupts, but another interrupt can preempt even at these instructions as part of the phenomenon we call exception entry chain (Section 4.1.2). Detecting an exception entry chain is an essential step in the Safe shadow exception stack algorithm (Section 4.4.3.4). To facilitate the implementation of this step, the Stage0 routines are arranged exactly 16 bytes apart from each other. This way, the algorithm can quickly determine whether an exception activation is involved in an exception entry chain by comparing the interrupted program counter to the address range of the Stage0 routines and checking the alignment of the program counter. Using a bit rotation trick, we implemented this check by two computational operations and one comparison (Fig. 4.15).

The Non-Secure part of the trampolines is defined in an assembly source file `nonsecure_vector_ses.S`, which is intended to be linked to a Non-Secure application. The assembly source file is parameterized by the number of exception numbers, which affects the number of replicas of the Stage0 and Stage2 routines. The assembly source file also includes an exception vector table to be used in place of the application's original exception vector table. The original table is now read by the Stage2 trampoline routine, referred to under a specific symbol name.

The substitute exception vector table (Fig. 4.16) is structurally compatible with a standard Arm-M exception vector table but includes some changes specific to TZmCFI and extra information to be consumed by Monitor.

```
const ExecptionEntryPCSet = struct {
    start: usize = 0, // first Stage0 routine
    len: usize = 0, // # of exception numbers
    // [...]
    fn contains(self: *const Self, pc: usize) bool {
        // -%: subtraction with wrap-around
        const rel = pc -% self.start;

        // return rel < (self.len * 16) and rel % 16 == 0;

        // Use a bit rotation trick to do alignment and boundary
            checks at the same time.
        return rotr(usize, rel, log2(16)) < self.len;
    }
};
```

Figure 4.15: The code that determines if an exception entry chain has occurred by examining the interrupted program counter.

- The first entry, which usually contains an initial stack pointer, is replaced with a custom header indicating the number of entries contained in the table.
- The reset vector points to the *reset handler trampoline* defined in the same assembly source file. The reset handler trampoline simply calls the original reset handler after configuring the stack pointer, using the information from the original vector table.
- The remaining entries point to their respective Stage0 routines.

As is usually the case, the substitute exception vector table must be placed at a fixed location known by a Secure bootloader. When initializing Monitor, the bootloader must pass the table's address to Monitor's initialization function, which will validate the table's structure and use the information from the TZmCFI-specific header embedded in the table to construct an efficient representation of a set of program counter values signifying an exception entry chain (Fig. 4.15).

4.4.3.4 *Shadow Exception Stacks*

The Secure gateways `__TCPrivateEnterInterrupt` (shadow push) and `__TCPrivateEnterInterrupt` (shadow assert) implement the core algorithm of the shadow exception stack operations. They are invoked by the exception trampolines (Section 4.4.3.3) and execute in Secure mode. They use custom veneer functions (Section 4.4.3.5) because of their irregular calling convention.

The `fillFrameAddress` function leverages information from `EXC_RETURN` to locate the newest exception frame in one of the

```
  // How many entries do we generate?
  .set TableSize, 124 + 16

// This assembler source defines the "real" Non-Secure exception
     vector table.
.section .text.isr_vector
TCProtectedExceptionVectorTable:
  // Initial stack pointer - we don't use it so store the number
       of entries instead
  .word TableSize | TC_VEC_TABLE_HDR_SIGNATURE
  .word Reset

  .set i, 0
  .rept TableSize - 2
    .word ExceptionTrampolinesStage0 +
        TC_VEC_TABLE_TRAMPOLINE_STRIDE * i
    .set i, i + 1
  .endr
```

Figure 4.16: The substitute exception vector table.

```
fn fillFrameAddress(self: *Self) void {
    if ((self.exc_return & EXC_RETURN.S) != 0) {
        self.frame = @ptrCast([*]const usize, &dummy_exc_frame
            [0]);
    } else if ((self.exc_return & EXC_RETURN.SPSEL) != 0) {
        self.frame = @intToPtr([*]const usize, self.psp);
    } else {
        self.frame = @intToPtr([*]const usize, self.msp);
    }
}
```

Figure 4.17: The code to locate a recently-pushed exception frame.

four banked stack pointers of Armv8-M. `EXC_RETURN.SPSEL` indicates whether it is located in a Main stack or in a Process stack. `EXC_RETURN.S` indicates the security state of the interrupted context, which also affects the location of the exception frame. The exception frame stacked in a Secure context is considered trustworthy based on Assumption 1. Assumption 7 implies that no vulnerable exceptions frame are present beyond that point. Also, the exception frame in this case uses a different format to contain all general-purpose registers (i.e., including callee-saved ones), which would complicate the reading process of an exception frame. Therefore, this function returns a dummy exception frame address in such a case. Fig. 4.17 shows the implementation of `fillFrameAddress`.

Our shadow exception stack implementation comes in three flavors: *Naïve*, *Unnested*, and *Safe*, each corresponding to one of the

solutions presented in Section 4.3.2.2. These implementations can be chosen by Monitor's build option (Section 4.4.3.6).

NAÏVE SHADOW STACKS The Naïve implementation is a straightforward realization of the shadow stack technique; the shadow push operation pushes the newest exception frame, and the shadow pop operation pops the top element.

Each shadow entry contains five fields (`pc`, `lr`, `EXC_RETURN`, `ip`, and a pointer to the original exception frame). Protecting `ip` is not essential as far as only exception handling is concerned. The reason `ip` is included is that it is used by linker-generated trampolines as well as by the shadow stack instrumentation code (Section 4.4.3.2) for passing a continuation address as part of its special calling convention, and therefore the corruption of `ip` can result in a control-flow violation.

UNNESTED SHADOW STACKS The nuisance with exception entry chains (Section 4.2.1) can be avoided entirely by prohibiting exception nesting. The Unnested implementation is the specialization of the Naïve implementation for this approach that holds only one element in the stack.

When using this implementation, Non-Secure code is responsible for configuring interrupt priorities and the priority grouping not to cause exception nesting. Our attacker model and assumptions (Section 4.3.2) do not eliminate the possibility of the corruption of interrupt priority registers. Therefore, the push operation of this implementation checks `EXC_RETURN` and, if the Mode bit indicates the exception was taken in Handler mode, registers a CFI violation with the reason that the current exception preempted another one in breach of this implementation's assumption.

A slimmed-down version of exception trampolines that do not include any `cpsid` instructions may be used in place with this shadow exception stack implementation.

SAFE SHADOW STACKS An exception entry chain, the situation in which the processor takes two exceptions E_1, E_2 in succession without giving the exception handler of E_1 a chance to execute, undermines the soundness of the Naïve implementation. In such a situation, the Naïve implementation only protects the exception frame at the top corresponding to E_2 and leaves the exception frame of E_1 open to memory corruption.

The Safe implementation implements the safe shadow exception stack algorithm we presented in Section 4.2.2 to mitigate this issue. The basic idea is that it walks through the stack and pushes as many exception frames as possible until the state of the shadow exception

stack is consistent with that of the exception stack. The key component is the `moveNext` function, which locates the previous exception frame based on the information from a current exception frame and updates the state of the exception frame iterator accordingly.

Locating a next exception frame is inefficient and impractical in general because of the need to skip generic function frames, which would require frame pointers or DWARF information. However, if we limit the exception frame traversal to an exception entry chain (EEC) stack (a sequence of exception frames between which exception entry chains are formed), this problem becomes easy. It has been shown that the Safe SES algorithm only needs to be able to traverse within an EEC stack for correctness. The `moveNext` function exploits the specific Stage0 trampoline layout (Section 4.4.3.3) to check if the current exception frame is involved in an exception entry chain. If that is the case, the previous frame is located just above the original stack pointer. Otherwise, `moveNext` signals the end of an EEC stack.

The push loop terminates when it hits the end of an EEC stack (Fig. 4.6, F). If the SES is non-empty, the loop also terminates when it encounters an exception frame that is identical to the current top element of the SES (Fig. 4.6, D).

The Safe shadow assert implementation is similar to that of the Naïve implementation, except that it must validate the last *two* exception frames for the algorithm's soundness. It uses `moveNext` to locate the second last exception frame.

4.4.3.5 *Secure Gateways*

Secure gateways are Secure functions exposed to Non-Secure code. When compiling Secure code, they are marked with special symbols whose names are prefixed by `__acle_se_`. A linker supporting CMSE is supposed to detect such symbols and generate another output file named a *CMSE import library*, which only includes the addresses of Secure gateways. The officially maintained GNU Arm Embedded Toolchain provides a version of binutils meeting this criterion, but that was not an option for us because the LLVM indirect call CFI requires LTO, which meant we had to use LLD for linking. LLD was not capable of generating a CMSE import library, so we wrote a tool to generate a CMSE import library in Rust [104]. Fig. 4.18 shows an excerpt from the CMSE import library generated by our tool.

Secure gateways must be contained by an NSC (*Non-Secure Callable*) region defined by SAU (Security Attribute Unit) or IDAU (Implementation Defined Attribute Unit) for them to be actually callable [12]. The common way to ensure this is to output Secure gateways to a section named `.gnu.sgstubs` and to configure SAU or IDAU to mark this section as an NSC region in the initialization code

```
.syntax unified
/* [...] */
.type TCActivateThread function
.set TCActivateThread, 0x140000e5
.global TCActivateThread
.type __TCPrivateShadowPush function
.set __TCPrivateShadowPush, 0x14000107
.global __TCPrivateShadowPush
/* [...] */
```

Figure 4.18: The CMSE import library generated by our tool.

```
.gnu.sgstubs :
{
    . = ALIGN(32);
    __nsc_start = .;
    KEEP(*(.gnu.sgstubs*))
    . = ALIGN(32);
    __nsc_end = .;
} > CODEMEM
```

Figure 4.19: The linker script that allocates an NSC region.

of a Secure bootloader. SAU can assign security attributes to any continuous address ranges specified in 32-byte granularity, which makes it straightforward to allocate an address range for an NSC region using a linker script, as shown in Fig. 4.19.

Most general-purpose registers are not banked between security states, meaning their values are simply carried over during a security mode transition. Carelessly implemented Secure gateways could leak sensitive information to Non-Secure code through caller-saved registers. Furthermore, the valid entry points of Secure gateways are indicated by the 32-bit bit pattern of the `SG` instruction, and the chance of arbitrary data in an NSC region colliding with this bit pattern increases proportionally with the size of the region. To mitigate these issues, Secure gateways are usually implemented in two parts: inner code and *veneer functions*. The veneer functions are minimal code that resides in an NSC region and proxies the calls to the inner code of Secure gateways, which resides in a Secure region. On function return, the veneer code clears caller-saved registers and returns to the caller using the `BXNS` instruction, which is a special instruction for branching to a Non-Secure region. The veneer functions can be written by hand or generated automatically by a supporting compiler by marking a Secure gateway with `__attribute__((cmse_nonsecure_entry))` attribute [8]. A programming language with macro processing or programmatic symbol

generation can generate veneer functions without special compiler support, as we did in Zig (Fig. 4.20).

4.4.3.6 *Instantiation*

The Secure portion of Monitor is implemented as a Zig package. An application developer "instantiates" this package by invoking it from the source code of a Zig static library. The package provides a compile-time interface to customize various aspects.

Fig. 4.21 shows a minimal implementation of the static library. `@import` is a built-in function that imports an external Zig source file at compile time. In Zig's compile-time evaluation scheme, exporting symbols (Secure gateways and C APIs in this case) in Zig is a side-effectful operation. Putting `@import` in a `comptime` block forces the evaluation of `@import` along with its side effects.

Monitor looks in the root source file for item definitions for compile-time customization. The `tcSetShadowStackGuard` function configures the MPU for shadow stack bounds check (Section 4.4.3.2). Optional items recognized include but are not limited to:

- `TC_ABORTING_SHADOWSTACK` controls the flavor of the shadow stack implementation (Section 4.4.3.2).
- `TC_SHADOW_EXC_STACK_TYPE` specifies which shadow exception stack implementation (Section 4.4.3.4) to use.
- Monitor uses the "panicking" mechanism of Zig to report a security violation. The default panic handler executes a software breakpoint instruction. An application developer can provide a custom panic handler by creating a global function named `panic`.

4.4.4 *Build Toolchain*

A compiled program often requires a target-specific runtime library (e.g., libgcc in GCC, compiler-rt in LLVM) to realize unsupported or complex operations on the target processor. Such a library should be instrumented as well for comprehensive protection, but the instrumented version of the library should only be used when building a Non-Secure application, not when building the Secure bootloader or Monitor. The process of building and configuring the correct runtime library for an application can be tedious and error-prone and should preferably be hidden from an application developer, as is usually the case for an uninstrumented program.

To deal with this issue, our build toolchain leverages Zig's on-the-fly compiler-rt compilation feature, which, during a linking process,

```
pub fn exportNonSecureCallable(comptime name: []const u8,
    comptime func: extern fn (usize, usize, usize, usize) usize)
     void {
    const Veneer = struct {
        fn veneer() callconv(.Naked) void {
            @setRuntimeSafety(false);
            asm volatile (
                \\ .cpu cortex-m33
                \\ sg
                \\
                \\ push {lr}
                \\ bl %[func]
                \\ pop {lr}
                \\
                \\ # Clear registers
                \\ mov r1, lr
                \\ mov r2, lr
                \\ mov r3, lr
                \\ mov ip, lr
                \\ msr apsr, lr
                \\
                \\ # Return
                \\ bxns lr
                : // no output
                : [func] "X" (func)
            );

            unreachable;
        }
    };
    @export(Veneer.veneer, .{ .name = name, .linkage = .Strong,
        .section = ".gnu.sgstubs" });
    @export(Veneer.veneer, .{ .name = "__acle_se_" ++ name, .
        linkage = .Strong, .section = ".gnu.sgstubs" });
}

comptime {
    arm_cmse.exportNonSecureCallable("TCActivateThread",
        TCActivateThread);
}
```

Figure 4.20: Leveraging Zig's compile-time facilities to generate a veneer function.

```
const arm_m = @import("arm_m");
comptime {
    _ = @import("tzmcfi-monitor");
}

pub const TC_SHADOW_EXC_STACK_TYPE = .Unnested;

pub fn tcSetShadowStackGuard(stack_start: usize, stack_end:
    usize) void {
    const mpu = arm_m.mpu;
    const Mpu = arm_m.Mpu;
    mpu.regRnr().* = 0;

    // `stack_start - 32 .. stack_start`
    mpu.regRbar().* = (stack_start - 32) | Mpu.RBAR_AP_RW_ANY;
    mpu.regRlar().* = (stack_start - 32) | Mpu.RLAR_EN;

    // `stack_end .. stack_end + 32`
    mpu.regRbarA(1).* = stack_end | Mpu.RBAR_AP_RW_ANY;
    mpu.regRlarA(1).* = stack_end | Mpu.RLAR_EN;
}
```

Figure 4.21: The Zig source code for instantiating the Secure portion of Monitor.

automatically builds compiler-rt using the build options matching to the compiled program. The compiled object files of compiler-rt are stored in a global cache directory, indexed by the hashes of their build options. We added the CFI instrumentation options to Zig's build artifact hashing inputs. The result is that an application developer can be oblivious on regard to whether the correct version of compiler-rt is compiled in.

Whether to use this feature is entirely up to an application developer. To sum up, an application developer who is currently using a build pipeline based on a makefile is given the options shown below. Note that, in any case, the developer will need to use LLD as a linker explicitly or implicitly for the reason explained in Section 4.4.3.5. The first two options are illustrated in Fig. 4.22. TZmCFI does not support the third option.

1. Rework the makefile as a Zig build script.
2. Replace the linker invocation with `zig build-exe`.
3. Manually build compile-rt and then link it when invoking LLD.

```
1. (in build.zig)
  const exe =
    b.addExecutable("hello", "main.zig");
  exe.setLinkerScriptPath("nonsecure.ld");
  exe.setTarget(try CrossTarget.parse(.{
    .arch_os_abi = "thumb-freestanding-eabi",
    .cpu_features = "cortex_m33-fpregs",
  }));
  exe.enable_lto = true;
  exe.enable_shadow_call_stack = true;
  exe.addCSourceFile("startup.S",
    &[_][]const u8{});
  exe.addCSourceFile("Monitor/secure_implib.s",
    &[_][]const u8{});

2. (in Makefile)
  zig build-exe main.zig --name hello \
    --c-source TZmCFI/src/nonsecure_vector.S \
    --c-source startup.S \
    --c-source Monitor/secure_implib.s \
    --lto -fsanitize=shadow-call-stack \
    --release-small \
    -target thumb-freestanding-eabi \
    -mcpu=cortex_m33-fpregs \
    --linker-script nonsecure.ld
```

Figure 4.22: Using Zig to link a Non-Secure application. `main.zig` file can be empty.

4.4.5 *FreeRTOS*

TZmCFI Monitor exposes some Secure gateways which are supposed to be called by a Non-Secure operating system kernel through manually inserted hooks at appropriate moments. There are two such gateways: `TCCreateThread` and `TCActivateThread`. The kernel should also allocate a space in the task control block to store the thread ID returned by `TCCreateThread`.

An operating system kernel supporting Armv8-M with TrustZone, i.e., designed for Non-Secure mode execution may already have context switch hooks in place. Non-Secure code may call Secure gateways, which require a Secure stack to operate. Non-Secure interrupts can preempt their execution in controlled ways, and a Non-Secure operating system can perform context switching in the midst of them, so the Non-Secure operating system is also responsible for managing Secure contexts. Since Secure state registers cannot be accessed by Non-Secure code, Secure code is supposed to expose Secure gateways to implement Secure context switching, through an interface standardized by CMSIS-Core [2], though a modified convention may also be used. Therefore, a short implementation path can be achieved by choosing the RTOS port that already supports Armv8-M with TrustZone as the starting point as was the case with our FreeRTOS example application, and then inserting new hooks to the same place as where the existing hooks are. Simply put, such a kernel may already have done some part of the job for us.

We used the `CORTEX_MPU_M33F_Simulator_Keil_GCC` demo application included in FreeRTOS version 10.2.0 as the starting point. This demo application configures FreeRTOS to use an MPU wrapper. But more importantly, its implementation of the FreeRTOS portability layer utilizes the aforementioned Secure context switching interface, albeit with different function names such as `SecureContext_LoadContext` as opposed to `TZ_LoadContext_S`[5].

In the original implementation (the one that comes with the demo application) of the portability layer, Secure contexts are only allocated on-demand. We changed it to create one instantly as a task is created, as it is unlikely for a task not to make any subroutine calls. This change enabled us to get rid of some branches that checked the existence of a Secure context associated with a task. We also modified the Secure context management function to supply the created task's entry point.

Inserting the context switch hook did not require much change. We defined `SecureContext_LoadContext`

5 Note that, because Secure gateway calls are transparent, Non-Secure code could define `SecureContext_LoadContext` as a function delegating to `TZ_LoadContext_S`.

as a tail call to `TCActivateThread`. We implemented `SecureContext_SaveContext` as an empty function because it was not needed for TZmCFI. Because Secure gateway calls are transparent, `SecureContext_{LoadContext, SaveContext}` can be defined in Non-Secure code, and we did so, but care had to be taken when doing this: a shadow stack must be initialized as if the thread was preempted by an interrupt at the first instruction of the entry point. This means that, if `SecureContext_LoadContext` were implemented in a way that it pushes an entry to a shadow stack, the initial shadow stack would have to include the entry for `SecureContext_LoadContext`, which in turn would mean that `TCCreateThread` would have to able to accept a sequence of return addresses to initialize a shadow stack with. We wanted to keep the interface simple, not to mention the overhead that comes with flexibility, so we decided not to support this use case. Thus `SecureContext_LoadContext` had to be implemented in a way that it does not push any shadow stack entries.

The exception vector table was replaced with the one from TZmCFI Toolkit (Section 4.4.3.3). The old vector table was renamed to `raw_exception_vectors`, the name the exception trampolines expect.

The existence of exception trampolines changes the calling convention of exception handlers: exception handlers now receive `EXC_RETURN` by `r0`, `lr` register now contains the address of the exception return trampoline, and exception handlers now have to return to `lr` instead of `EXC_RETURN`, or alternatively, to the exception return trampoline directly. This does not change the Arm-M's property that most interrupt handlers can be implemented as normal C functions, thus had no impact on regular interrupt handlers, though they can be further optimized by the use of a TZmCFI-specific function attribute (we call this optimization "Trampoline Shortcut"; see Section 4.4.6.2) or by hand. The only affected places were the PendSV handler because `EXC_RETURN` includes some portion of the background context's execution state, and the SVC handler, which needs `EXC_RETURN` to locate the background context's stack pointer.

4.4.6 *Optimization*

We implemented two kinds of optimization to alleviate the overhead of shadow exception stacks.

4.4.6.1 *Accelerated Privilege Escalation*

When memory protection is enabled, an operating system requires a mechanism to temporarily enter Privileged mode so that kernel ser-

```
# Declare a Secure gateway. When branching to a
# Non-Secure Callable region, the processor make
# sures the target is a SG instruction
# (otherwise it will cause a fault). The SG
# instruction itself is a no-op.
sg

# Clear the nPRIV bit of CONTROL_NS.
mrs r0, control_ns
bic r0, #1
msr control_ns, r0
mov r0, #0

# Return to a Non-Secure caller.
bxns lr
```

Figure 4.23: The implementation of `TCRaisePrivilege`.

vices invoked from a user task can access the operating system's critical data structure. One way to do this is to invoke an SVC handler, wherein the `CONTROL` register is updated to transition the calling task into Privileged mode, but this greatly increases CPU utilization because all Non-Secure exception handlers are subject to shadow exception stack instrumentation. For this reason, we took an alternative approach and replaced this mechanism with a Secure mode implementation.

The `CONTROL` register, which controls the current privilege level, is banked between security states. This means that a Secure function can update `CONTROL_NS`, the Non-Secure version of this register, regardless of the current Non-Secure privilege level. We took advantage of this property by adding a Secure gateway named `TCRaisePrivilege`, which updates `CONTROL_NS` to enter the privileged mode (Fig. 4.23). Assuming CFI is in place, this approach does not hinder security because CFI prevents it from being called from a disallowed location.

For FreeRTOS, switching to this approach is as easy as to point the macro `portRAISE_PRIVILEGE` to `TCRaisePrivilege`.

4.4.6.2 *Trampoline Shortcut*

Since an exception handler is just a plain function, its return address (`EXC_RETURN` in this case) is subject to protection by shadow stack. The shadow exception stack instrumentation replaces the return address with a constant function pointer to the return trampoline. This means that it is actually unnecessary to preserve or protect the return address provided that the return instruction is replaced with a direct jump to the return trampoline. We implemented this optimization

Without TC_INTR:

```
handleTimer1:
  addw    ip, pc, #8  ; set continuation
  ldr.w   pc, [pc]  ; jump to ...
  .word   __TCPrivateShadowPush
  stmdb   sp!, {r4-r9, sl, lr}
[...]
  ldmia.w sp!, {r4-r9, sl, lr}
  ldr.w   pc, [pc]  ; jump to ...
  .word   __TCPrivateShadowAssertReturn
```

With TC_INTR:

```
handleTimer1:
  stmdb   sp!, {r4-r9, sl}
[...]
  ldmia.w sp!, {r4-r9, sl}
  cpsid   f
  ldr.w   pc, [pc]  ; jump to ...
  .word   __TCPrivateLeaveInterrupt
```

Figure 4.24: The generated assembler code of an exception handler with and without Trampoline Shortcut.

technique as a new LLVM calling convention TC_INTR. Fig. 4.24 is a concrete example that illustrates this technique.

4.5 Evaluation

We conducted an experiment to measure the runtime overhead of the proposed CFI mechanism.

The experiment was performed on an NXP LPC55S69 MCU [77]. The LPC55S69 MCU we used in the experiment includes two Cortex-M33 processors clocked at 100MHz, 640KB of on-chip flash ROM, and 320KB of on-chip SRAM. We used only the first processor (the only one equipped with TrustZone), leaving the other one left unused. Our test suite is comprised of the following three test programs: *Interrupt Latency* (Section 4.5.1), *FreeRTOS-MPU System Calls* (Section 4.5.2), and *CoreMark* (Section 4.5.3), all of which are bare-metal applications except for FreeRTOS-MPU System Calls.

TZmCFI Monitor and the test programs were built using the Zig programming language to facilitate the development and test process. Third-party software written in C, such as FreeRTOS and CoreMark, was compiled using LibClang 9.0 integrated into Zig's build system. The code was compiled using *ReleaseFast* and *ReleaseSmall* build modes provided by Zig, which are hard-coded to map to the code optimization flags `-O3` and `-Os`, respectively.

A 32KB portion of the on-chip SRAM, called SRAMX, is connected to the code bus for faster code execution. During the experiment, we loaded TZmCFI Monitor onto this portion of SRAM and all other code onto the flash ROM. We believe this configuration reflects a potential real-world use case very well considering that the performance of TZmCFI Monitor has a massive impact on the overall application performance and that SRAMX is too small to store normal application code.

Throughout this section, the following abbreviations are used to refer to the various CFI mechanisms used during the experiment:

- *Ctx* — The use of the context management API, including task creation and context switching. This is technically not a CFI mechanism by itself but rather a prerequisite for other mechanisms. Note that even without TZmCFI, multi-tasking applications are still required to use the API (with a slimmer implementation) to preempt the execution of Secure functions correctly. Also, task deletion is never performed because it is not supported in TZmCFI.

- *SES* — Shadow exception stacks (Section 4.4.3.4).

- *APE* — Accelerated privilege escalation (Section 4.4.6.1).

- *SS* — Our multi-task-aware TrustZone-based implementation of shadow stacks (Section 4.4.3.2). The two flavors of the im-

plementation, *Aborting* and *Non-Aborting* are evaluated separately.

For the experiment, we wrote a statistical profiler that counts the number of monitor calls that take place while running the benchmarking code. The profiler counts the following types of monitor calls:

- *EntInt* and *LeaInt* represent the execution of an exception trampoline and an exception return trampoline. This pair of operations is executed every time an exception is handled to keep track of valid exception return targets on SES.
- *ShPush* and *ShAsrt* represent push and pop operations on SS, which are inserted to function prologues and epilogues by the compiler. Some leaf functions do not have them if the return target is not spilled to memory.
- *ShAsrtRet* is a fused operation of ShAsrt and a function return, used for performance optimization.

To obtain the execution trace and a detailed breakdown of the execution time, we wrote a simple CPU profiler that collects a program counter on each cycle. On each run of a test program, this CPU profiler configures a Secure hardware timer to fire at a *target cycle* specified in relative to the start of the run. The timer interrupt handler, configured at a priority higher than the Non-Secure priority range, collects the original program counter on which the interrupt was taken. This process is repeated, each time starting a fresh run of the test program and incrementing the target cycle until the acquired data points cover the entirety of the measured time interval.

4.5.1 *Interrupt Latency*

We created a benchmark program to assess the impact on the interrupt response time. Two timers, Timer0 and Timer1, both having interrupt handlers, were configured to fire almost simultaneously. The interrupt priority of Timer1 was set to higher than that of Timer0. We measured the interrupt response time of Timer1 by reading the timer value in Timer1's interrupt handler.

The timing difference between them was controlled by the *skew* parameter. The variation in the skew changes the behavior of the handling order of the interrupts and the code path taken by the exception trampoline's implementation. For example, a positive skew causes Timer1 to fire while Timer0's handle is still running, in which case the exception trampoline iterates through the stack until it encounters Timer0's exception frame. On the other hand, a negative

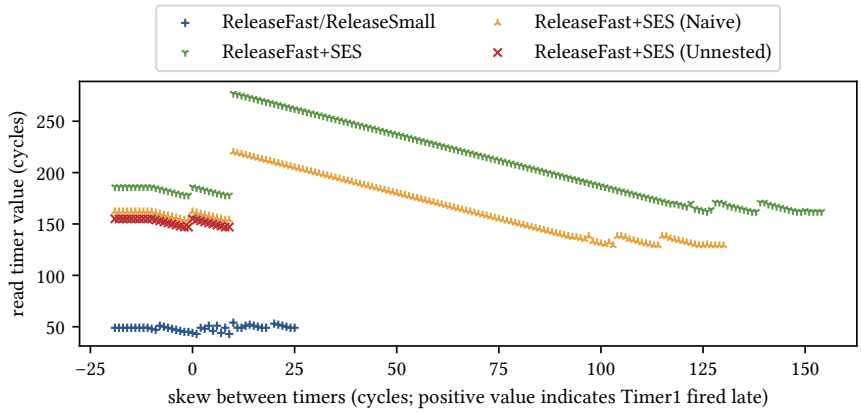


Figure 4.25: The interrupt response time of Timer1.

Table 4.2: The interrupt response time of Timer1. All numbers are in cycles.

	RELEASEFAST		RELEASESMALL	
SKEW	BASELINE	INSTRUMENTED	BASELINE	INSTRUMENTED
< -9	49	185 (+136)	49	187 (+138)
0	44	186 (+142)	44	188 (+144)
10	54	277 (+223)	54	277 (+173)

one makes Timer1 the top-level exception, in which case the exception trampoline proceeds to the code path where it just pushes every frame in the exception entry chain stack. Depending on the processor implementation, a specific skew value induces an exception entry chain, in which case the exception trampoline will save the exception frames of both Timer0 and Timer1. We swept the skew parameter through a range and observed the changes in the interrupt response time.

Fig. 4.25 and Table 4.2 show the result of the measurement. The overall increase in the interrupt response time in the cases where the exception trampoline ran only once ($skew < -9$) was 136 cycles. In the case where Timer1 fired late ($skew = 10$), the increase was twofold due to the exception trampoline being Timer0 handler's critical section.

In this experiment, we could not find a datapoint indicating the presence of an exception entry chain. To obtain the interrupt response time of the case including the occurrence of an exception entry chain, we performed an additional experiment identical to the previous one except that an `ldm sp, {r0-r1}` instruction was inserted before the first instruction of the Stage0 trampoline. An exception entry chain was observed at $skew = 10$, and the interrupt response time was 236 cycles (*ReleaseFast*).

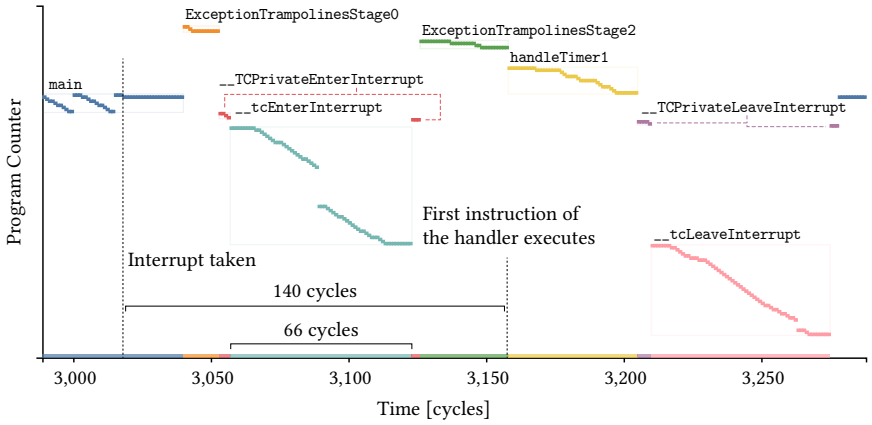


Figure 4.26: The execution trace of an instrumented interrupt handling sequence.

Although it is clear from the result that the Safe implementation is most costly, the performance difference between the SES implementations has proven unsubstantial. The CPU profiler has given us some insight into this matter and the proportions of overhead contributors. Fig. 4.26 is the execution trace of the processor executing an interrupt handling sequence in an instrumented program[6]. According to the execution trace, a major portion of the exception trampoline's execution time was spent on tasks irrelevant to the core portion of the SES implementation, viz., instruction fetch of the exception trampoline and reading the original exception vector table, both of which cost about 10 cycles every time they took place. Furthermore, comparing the similarly captured traces with different SES implementations revealed that the variable part of the SES implementation accounted only for an insignificant portion of the execution time of the Stage1 trampoline. This result presents plenty of opportunities for optimizing the code organization around the exception trampoline, which we leave for future work.

4.5.2 *FreeRTOS-MPU System Calls*

We constructed a benchmark application to measure the execution times of FreeRTOS's API functions. All functions are called from restricted tasks (i.e., tasks with memory protection) executing in Unprivileged mode.

Table 4.3 shows the measured execution times of FreeRTOS's API functions with various CFI mechanisms turned on or off. The inter-

6 This execution trace was captured on a program similar to but not exactly identical to Interrupt Latency benchmark program. It is by no means directly comparable to Fig. 4.25 or Table 4.2. It merely serves to support the point made here.

esting part is shown as a bar chart in Fig. 4.27. Table 4.4 shows the number of monitor calls taking place during FreeRTOS system calls.

Task creation and deletion are intended to be done only during system initialization, and task deletion is not supported and treated as no-op (hence the lack of contribution of Ctx to DelTask). Even uninstrumented, the "Ctx" portion in an orthodox application is not exactly zero (but usually less than TZmCFI's Ctx) for the reason explained earlier in Ctx's definition. Since Ctx dominates the execution times of task creation/deletion functions, this fact makes it challenging to evaluate the overhead in a fair manner. Nevertheless, their results are still shown here for completeness.

SES added 245 cycles on average for each trigged exception. All measured API functions use software traps for entering Privileged mode as well as for context switching. APE proved to be an effective approach to eliminate the use of software trap for entering Privileged mode along with its associated overhead and sometimes even completely offset all the overhead caused by other CFI mechanisms in use.

Non-aborting shadow stacks were faster than aborting shadow stacks, though the difference was not significant. Similarly to Section 4.5.1, the performance difference between the SES implementations was not substantial.

The overall overhead with all CFI mechanisms and optimization techniques turned on was -7–35%, varying primarily based on whether a context switch occurred or not.

4.5.3 *CoreMark*

Fig. 4.28 shows the CoreMark scores (iterations per second). Table 4.5 shows the number of monitor calls taking place during a single iteration of CoreMark. The measured overhead for the ReleaseFast build was 9%. ReleaseSmall fared worse, most likely because of more conservative function inlining, which increased the number of shadow stack operations.

Based on this result, we can estimate the number of cycles spent on every pair of shadow stack operations. For ReleaseFast + Non-Aborting SS, it is calculated as:

$$\frac{SystemCoreClock}{ShPushCountPerIteration} \cdot$$

$$\left(\frac{1}{Score} - \frac{1}{BaselineScore} \right) \approx 52.74 \,[\text{cycles}]$$

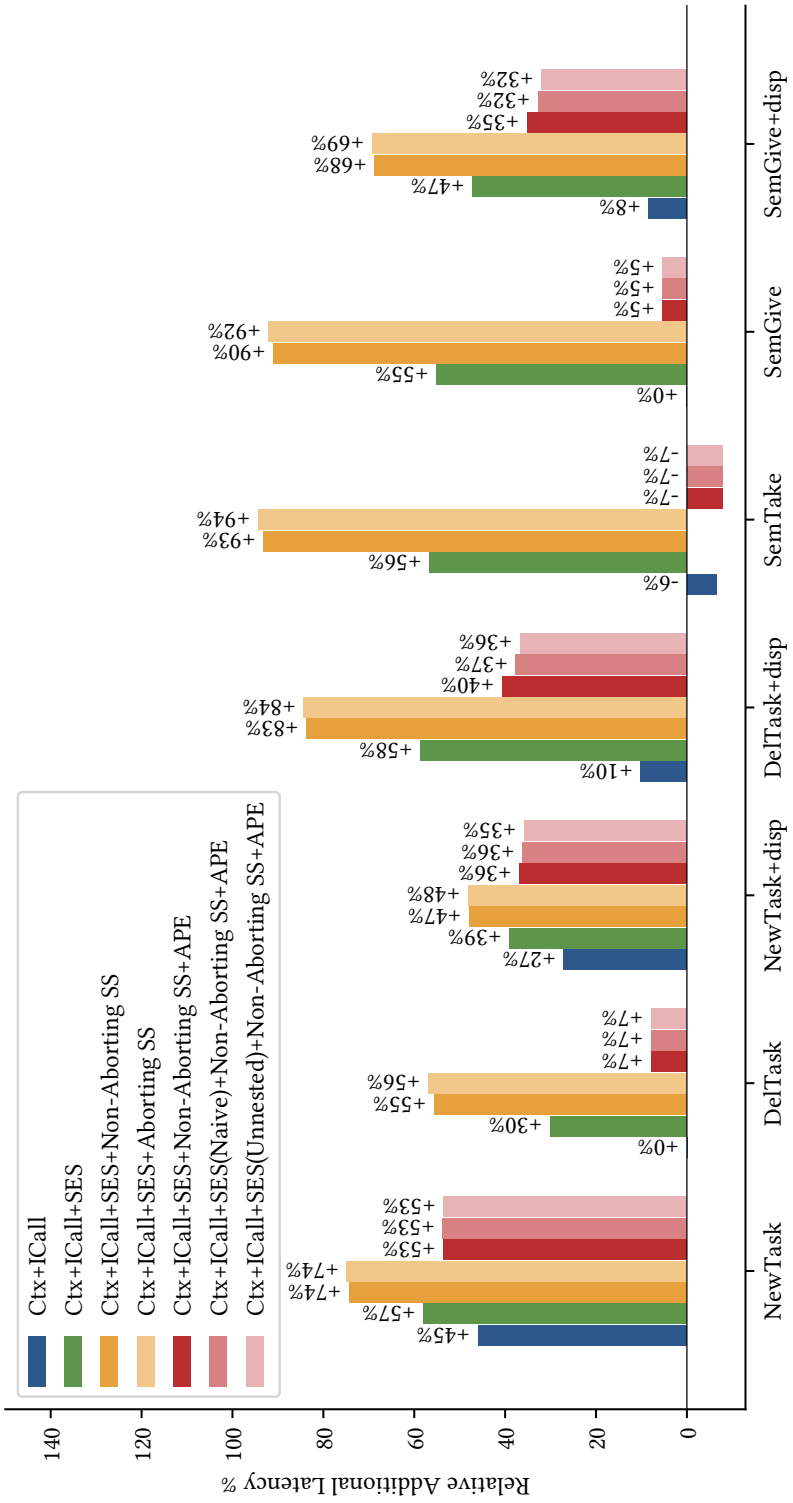


Figure 4.27: The relative overhead for each type of FreeRTOS system call. The build mode used here is *ReleaseFast*.

Table 4.3: The execution time for each type of FreeRTOS system calls. All values within are shown in cycles.

BUILD MODE	CTX	ICALL	SES	APE	SHADOW STACK	OVERHEAD[A]	NEWTASK	DELTASK	NEWTASK +DISP	DELTASK +DISP	SEMTAKE	SEMGIVE	SEMGIVE +DISP
ReleaseFast						33	1996	850	3750	964	411	488	1211
	✓					34	2914	849	4767	1062	410	487	1309
	✓	✓				34	2914	849	4767	1062	384	489	1313
	✓	✓		✓		34	2818	754	4671	966	308	403	1238
	✓	✓	✓			34	3153	1106	5216	1529	644	757	1783
	✓	✓	✓	✓		34	2800	754	4863	1176	308	403	1448
	✓	✓	✓		Non-Aborting	66	3480	1323	5547	1771	794	932	2043
	✓	✓	✓	✓	Non-Aborting	66	3065	917	5132	1356	379	514	1637
	✓	✓	✓		Aborting	67	3490	1334	5556	1777	799	937	2050
	✓	✓	✓	✓	Aborting	67	3072	925	5138	1359	381	516	1641
	✓	✓	Naive	✓	Non-Aborting	66	3069	917	5107	1327	379	514	1608
	✓	✓	Unnest	✓	Non-Aborting	66	3064	917	5093	1318	379	514	1599
ReleaseSmall						18	2061	846	3807	939	407	518	1274
	✓					26	3058	846	4911	1038	407	518	1373
	✓	✓				26	3058	846	4911	1038	412	522	1385
	✓	✓		✓		26	2964	745	4817	944	313	429	1294
	✓	✓	✓			26	3332	1108	5405	1520	674	780	1864
	✓	✓	✓	✓		26	2980	745	5053	1164	313	429	1514
	✓	✓	✓		Non-Aborting	76	3800	1453	5847	1730	805	904	2044
	✓	✓	✓	✓	Non-Aborting	76	3382	1041	5437	1310	383	493	1628
	✓	✓	✓		Aborting	77	3812	1468	5857	1737	810	909	2051
	✓	✓	✓	✓	Aborting	77	3391	1053	5444	1314	385	495	1632
	✓	✓	Naive	✓	Non-Aborting	76	3394	1041	5418	1279	383	493	1597
	✓	✓	Unnest	✓	Non-Aborting	76	3381	1041	5396	1270	383	493	1588

Table 4.4: The number of monitor calls for each type of FreeRTOS system calls.

BUILD MODE	EVENT	OVERHEAD[A]	NEWTASK	DELTASK	NEWTASK+DISP	DELTASK+DISP	SEMTAKE	SEMGIVE	SEMGIVE+DISP
ReleaseFast	EntInt	0	1	1	2	2	1	1	2
	LeaInt	0	1	1	2	2	1	1	2
	ShPush	1	8	6	9	5	4	4	6
	ShAsrt	0	1	3	1	1	1	1	1
	ShAsrtRet	1	7	3	6	4	3	3	5
ReleaseSmall	EntInt	0	1	1	2	2	1	1	2
	LeaInt	0	1	1	2	2	1	1	2
	ShPush	1	10	8	11	5	4	4	6
	ShAsrt	0	1	4	1	1	1	1	1
	ShAsrtRet	1	9	4	7	5	3	3	5

[a] The raw measurement for an empty code section. This value is already subtracted from the rest of the measurements.

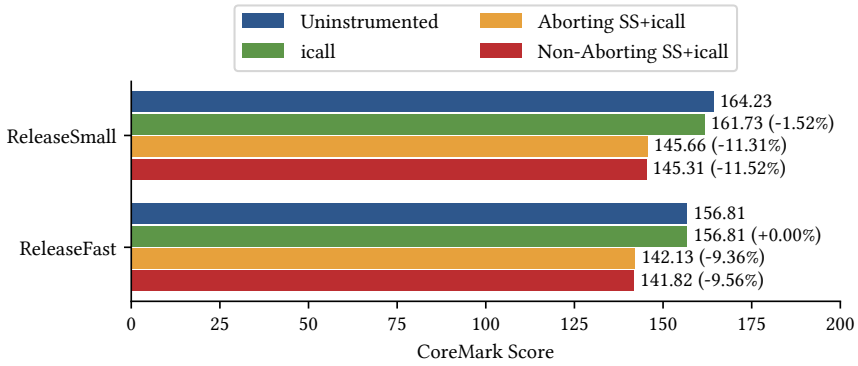


Figure 4.28: The CoreMark scores.

Table 4.5: The per-iteration event statistics for CoreMark.

BUILD MODE	EVENT	COUNT	BUILD MODE	EVENT	COUNT
ReleaseFast	ShPush	1249	ReleaseSmall	ShPush	1671
	ShArtRet	1249		ShArtRet	1671

The result is more favorable than [108], which reported a decrease of about 21% in the CoreMark score. The improvement is attributed to the use of a CMSE secure gateway in place of an expensive software trap to enter the processor mode to access a shadow stack. [81] provides results for Dhrystone and two custom microbenchmarks but not for CoreMark. Their result indicated a 513% increase in Dhrystone's execution time, even though the monitor was called only for 34 times during a single run, which took 0.15 milliseconds uninstrumented. These numbers amount to about 900 cycles spent for every pair of shadow stack operations, which is 17 times larger than the estimation for our solution calculated by the formula above. The overhead of a protected shadow call stack reported by [16] varies across different programs in the range of 5–21%, which spans roughly the same range as our result does.

4.6 Conclusion

With the emergence of IoT and smart devices and the increase in the complexity of embedded systems, it is increasingly important to employ defensive security techniques such as CFI on embedded applications as well. However, the adoption and development of such mechanisms on embedded systems are still in infancy.

In this chapter's work, we introduced *TZmCFI*, a holistic CFI solution for microcontroller-based embedded systems. TZmCFI is built upon several CFI techniques, namely shadow stacks, shadow exception stacks, and LLVM forward-edge CFI for comprehensive protection. The toolset for TZmCFI, which we call *TZmCFI Toolkit*, includes a modified LLVM-based build toolchain for producing an instrumented Non-Secure binary and a supporting runtime library that monitors exception handlers and provides secure storage for the CFI mechanism's internal state. We presented design decisions and choices made throughout its implementation, notably alternative options to the safe but slow shadow exception stack implementation from our previous work. We have also shown the process of retrofitting FreeRTOS-MPU for CFI enforcement.

We conducted a performance measurement of TZmCFI using three benchmark applications: *Interrupt Latency*, *FreeRTOS-MPU System Calls*, and *CoreMark*. The shadow exception stack instrumentation's measured overhead was moderate if not significant, although a major portion of the overhead was caused by jumping between numerous stages of exception trampolines, leaving a further opportunity for optimization. As for the shadow stack instrumentation, the overhead was clearly lower than previous works that use shadow stacks and target similar systems.

5

Componentizing an Operating System Feature Using a TECS Plugin

With ever-widening use cases of microcontroller-based systems and their further integration, there is an increasing demand for efficient software development methods that can deal with the arising complexity. Component-based development (CBD) is a development methodology that encourages code reusability by dividing the system into separate components. This allows for large and complex software to be constructed efficiently from reusable components, significantly reducing development costs and time [28].

CBD is realized with the help of a component system. Although many component systems are available, few can be used for embedded system development, owing to the tight design constraints (e.g., runtime/memory overhead) imposed by embedded systems [37]. Among these component systems, the TOPPERS Embedded Component System (TECS) is specifically designed for embedded systems [24].

TECS offers a feature that allows for the real-time operating system's (RTOS's) kernel objects (e.g., tasks) to be treated as components [27]. In many RTOSs, kernel objects are created statically by writing static API statements in a kernel configuration file. The component definitions for the TECS include *factory* blocks that can be used to generate static API statements automatically. By utilizing this feature, developers can access the RTOS's features inside the component framework to enjoy the benefits of CBD. However, those static API statements that can be generated by TECS are somewhat limited insofar as it cannot generate certain statements, including the API of the TOPPERS/ASP3 real-time kernel's time event notifications [100] (cyclic/alarm notification), which allows one of eight notification methods to be specified for each notification object. The parameter formats for each notification method differ from one another, which prevents the kernel feature from being componentized.

This chapter proposes a cell-type plugin, which is one type [79] of plugin supported by the TECS generator [26]. This is used to componentize the time event notification without sacrificing usability and to show the usefulness of the proposed plugin.

In this study, we first designed the component model for the time event notification, considering usability and execution efficiency. Then, we developed the TECS generator plugin, called `NotifierPlugin`, according to the designed component model. Fi-

nally, we evaluated the runtime overhead by comparing the interrupt processing time of the componentized time event notification to that of the time event notification created without TECS or our plugin. The results demonstrate the feasibility of the proposed plugin.

Our contributions in this chapter are as follows:

- We design a TOPPERS/ASP3 time event notification component for TECS, considering usability and execution efficiency (Sections 5.2 and 5.4).
- We design and implement a TECS generator plugin to realize the component design (Sections 5.2 and 5.3).
- We evaluate the componentized time event notification, showing that the runtime overhead is considerably low (Section 5.5).

5.1 Time Event Notifications

Time event notifications, as defined in the TOPPERS Third-Generation Kernel Specification, are a kernel feature that provides a method for notifying the user when a certain time has elapsed. There are two types of time event notifications: *cyclic notifications*, which fire periodically; and *alarm notifications*, which fire at a certain time specified by the application.

Time event notifications replace *time event handlers* from the previous kernel generation. Time event handlers' handler functions are called from a timer interrupt handler and thus always execute in the kernel domain. Therefore, there was no way to create time event objects belonging to a user domain. Time event notifications address this issue by providing additional notification methods other than a handler function.

Time event notifications support eight notification methods:

- Call handler function (`TNFY_HANDLER`): The kernel's timer interrupt handler calls the specified function pointer of type `void(*)(intptr_t)`. This notification method is compatible with the legacy timer event handlers.
- Assign variable (`TNFY_SETVAR`): The kernel updates the specified variable of type `intptr_t`.
- Increment variable (`TNFY_INCVAR`): The kernel increments the specified variable of type `intptr_t`.
- Activate task (`TNFY_ACTTSK`): The kernel activates the specified task.
- Wake up task (`TNFY_WUPTSK`): The kernel wakes up the specified task.
- Signal semaphore (`TNFY_SIGSEM`): The kernel signals the specified semaphore.
- Set eventflag (`TNFY_SETFLG`): The kernel sets a flag pattern on the specified eventflag.
- Send to dataqueue (`TNFY_SNDDTQ`): The kernel sends the specified value to the specified dataqueue.

Similarly, time event notifications support eight error notification methods:

- None: The kernel silently ignores any errors that occurred while applying the specified method.

- Assign variable (TENFY_SETVAR): The kernel updates the specified variable of type intptr_t with the error code returned by the notification method.
- Increment variable (TENFY_INCVAR): Identical as the normal notification method.
- Activate task (TENFY_ACTTSK): Identical as the normal notification method.
- Wake up task (TENFY_WUPTSK): Identical as the normal notification method.
- Signal semaphore (TENFY_SIGSEM): Identical as the normal notification method.
- Set eventflag (TENFY_SETFLG): Identical as the normal notification method.
- Send to dataqueue (TENFY_SNDDTQ): The kernel sends the error code returned by the notification method to the specified dataqueue.

Time event notifications are created by static APIs CRE_CYC or CRE_ALM in the kernel configuration file.

- Cyclic notification are created by: CRE_CYC(<ID>, { <attributes>, {<method>}, <period>, <phase> });
- Alarm notifications are created by: CRE_ALM(<ID>, { <attributes>, {<method>} });

<method> contains one or more comma-separated values. The first value specifies the bit-wise OR of the normal and error notification methods to use. The remaining values specify the parameters of the specified notification methods, and their number varies based on which methods are selected. For example, TNFY_SETVAR takes two values: the variable to update and the assigned value. On the other hand, TNFY_ACTTSK only takes one value to specify the task to activate. The error notification method's parameters follow those of the normal notification method. Fig. 5.1 shows an example of creating a cyclic notification that activates the task TASK_1, and, in case of a failure, sets the eventflag object FLG_1 with the flag pattern 0x0001U.

```
CRE_CYC(CYC_ID, { TA_NULL,
  { TNFY_WUPTSK|TENFY_SETFLG, TASK_1, FLG_1, 0x0001U },
  50, 0 });
```

Figure 5.1: Example of a kernel configuration for creating a cyclic notification.

5.2 Design

5.2.1 *Considerations*

We considered the following points while designing the component model for the componentized time event notification.

The first point pertains to the execution speed and ROM/RAM consumption. Because hardware in embedded systems is often designed to be minimal, even a slight increase in the execution time or memory consumption can impact the hardware costs of the system. Therefore, overhead imposed by the component must be minimal—or zero, if possible.

The second point pertains to usability (e.g., an easy-to-use interface, error detection, the backward compatibility). The lack of usability not only makes it more difficult to use the component, but also affects productivity and even the quality of the final product. Hence, the usability of the component is of paramount importance.

5.2.2 *Creating a Time Event Notification*

Users can create a cyclic notification or alarm notification by instantiating `tCyclicNotifier` or `tAlarmNotifier`, respectively. This method is comparable to that of existing components, including time event handler components (viz., `tCyclicHandler` and `tAlarmHandler`) and `tTask`.

5.2.3 *Controlling a Time Event Notification*

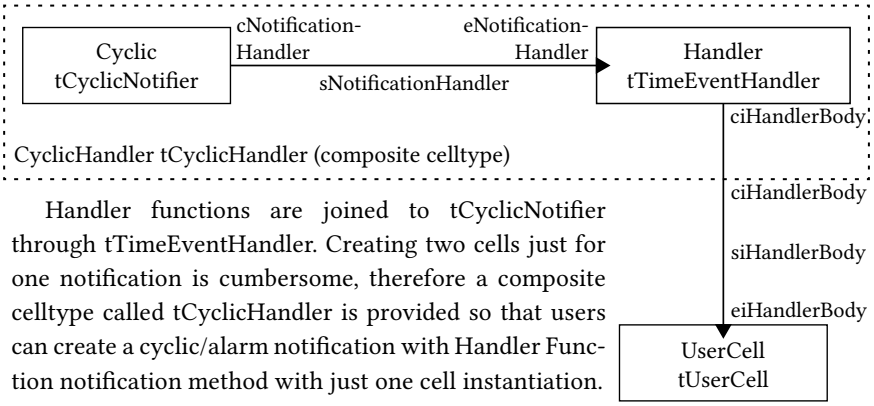
The TECS binding for the older version of the TOPPERS/ASP kernel, called ASP+TECS [99], included `tCyclicHandler` and `tAlarmHandler`. These could be controlled via the entry ports `eCyclic`, `eAlarm`, `eiAlarm` with the respective signatures `sCyclic`, `sAlarm`, `siAlarm`. We followed this design when creating the specification for our time event notification components, as shown in Fig. 5.2.

5.2.4 *Notification Methods*

The notification target for each time event notification can be specified by joining a cell to `ciNotificationHandler` or `ciErrorNotificationHandler`. These call ports are known as *common call ports*. The notification target can also be specified by the attribute values. The reason to prefer the use of common call ports is because of the simplicity of the interface. The signature associated

```
// signature to control cyclic notifications
signature sCyclic {
    ER start(void);
    ER stop(void);
    ER refer([out]T_RCYC *pk_cyclicHandlerStatus); // query
};
// signature to control alarm notifications
signature sAlarm {
    ER start([in] RELTIM alarmTime);
    ER stop(void);
    ER refer([out]T_RALM *pk_alarmStatus); // query
};
// (for non-task context)
[context("non-task")] signature siAlarm {
    ER start([in] RELTIM alarmTime);
    ER stop(void);
};
```

Figure 5.2: Signature definitions for the time event notification component.



Handler functions are joined to tCyclicNotifier through tTimeEventHandler. Creating two cells just for one notification is cumbersome, therefore a composite celltype called tCyclicHandler is provided so that users can create a cyclic/alarm notification with Handler Function notification method with just one cell instantiation.

Figure 5.3: Joining a handler function to the time event notification component.

with common call ports is `siNotificationHandler`, which has no function headers. Moreover, no actual TECS calls are made through this signature.

The notification method is determined with the following process: (a) each notification method requires a specific combination of the joined celltype and attributes (as shown in Table 5.1), and they are compared with the actual specification such that the exact match is used; (b) any missing or redundant specifications result in an error.

5.2.5 *Handler Functions*

User-provided handler functions are joined via `tTimeEventHandler`, as shown in Fig. 5.3. To remove the need for developers to de-

Table 5.1: Notification methods and accepted combinations of the joined celltype and attributes.

METHOD	CELLTYPE	ATTRIBUTE
None[1]	None	None
Handler Function[2]	`tTimeEventHandler`	None
Set Variable	None	`setVariableAddress`
		`setVariableValue`[3]
Increment Variable	None	`incrementedVariableAddress`
Activate Task	`tTask`[4]	None
Wake Up Task	`tTask`[5]	None
Signal Semaphore	`tSemaphore`	None
Set Eventflag	`tEventflag`	`flagPattern`
Send To Dataqueue	`tDataqueue`	`dataQueueSentValue`[3]

[1] Only applicable for the error notification method.
[2] Not applicable for the error notification method.
[3] Not applicable for the error notification method because the error code of the normal notification is used instead.
[4] Must be joined to the entry port `eiActivateNotificationHandler`.
[5] Must be joined to the entry port `eiWakeUpNotificationHandler`.

fine extra cells, composite celltypes called `tCyclicHandler` and `tAlarmHandler` are provided. Their interface is similar to that of the components defined in ASP+TECS with the same name, providing compatibility for legacy codes.

5.2.6 *Rejected Designs*

We considered several other designs but rejected them in favor of the presented design. This section lists some of them, along with their rejection reasons.

5.2.6.1 *One Celltype for Each Notification Method*

We considered a trivial component design where every combination of (alarm or cyclic, normal notification method, error notification method) is given a unique celltype. This design does not require a TECS generator plugin to implement, but the celltype names would become too long and hurt usability. Furthermore, this design would result in a total of 86 celltypes and severely curtail maintainability.

5.2.6.2 *One Call Port for Each Notification Method*

We also considered providing one call port for each notification method, i.e., an application developer specifies which notification

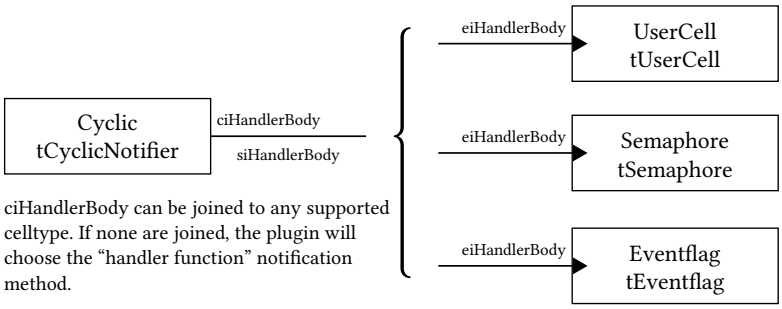


Figure 5.4: The rejected design proposal in which `ciHandlerBody` serves all notification methods.

method to use by creating a join on an appropriate call port and leaving others unconnected. This approach is superior to Section 5.2.6.1 in that the celltypes need not be duplicated for a large number of combinations. However, it is less handy than providing one call port for all notification methods because application developers would have to be aware of which call port to use for each notification method. Furthermore, having extra call ports would clutter the component diagram, such as those rendered by TECSCDE [72].

5.2.6.3 *Use `siHandlerBody` for All Notification Methods*

Finally, we considered using the existing `ciHandlerBody` call port (of signature `siHandlerBody`), which was used for registering a handler function in the original ASP+TECS, for all notification methods (Fig. 5.4).

In this design, the celltypes that can be used as a notification target gain a new entry port of signature `siHandlerBody`. When the `ciHandlerBody` call port is connected to an arbitrary entry port, the plugin would choose Handler Function notification method and generates code similar to ASP+TECS. However, when the call port is connected to one of such newly-gained entry ports, the plugin would instead choose the notification method matching the entry port's containing celltype.

Notification methods involving parameter passing, namely Set Variable (the value to assign), Set Eventflag (the bit pattern to set), and Send To Dataqueue (the word to send), are problematic in this approach. The signature `siHandlerBody` has the sole function signature `void main(void)`, which is fundamentally incapable of passing a parameter value. This itself does not impede the function of time event notifications because the parameter is implicitly passed by the kernel or specified by `CRE_CYC` or `CRE_ALM`. Thus, the `siHandlerBody` joins are converted into kernel configuration directives, and the original joins cease to exist at runtime in this case. What

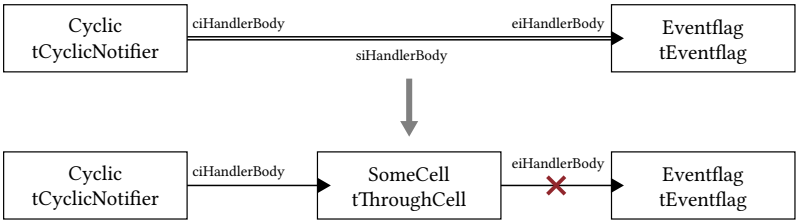


Figure 5.5: A through plugin turns a legal `siHandlerBody` connection into illegal one.

is problematic is that these newly-gained entry ports are mere entry ports and could be connected to arbitrary cells that do not receive this special treatment.

To prevent these entry ports' misuses, we would have to write another TECS generator plugin to prohibit illegal connection to these ports. Further confusion ensues when *through plugins* enter the picture. Through plugins are a type of TECS generator plugins that transform joins, e.g., to realize remote procedure calls [25] and inter-partition calls [53]. They usually work by replacing the original join with an external mechanism and capping each end with a proxy cell. This transformation can turn a legal `siHandlerBody` connection into an illegal one (Fig. 5.5). Not all connections become illegal by this transformation; those associated with Increment Variable, Activate Task, and others are still legal after this transformation because they do not need an implicit parameter. This causes mild confusion to application developers because some uses of a through plugin work, but other uses produce hard-to-understand errors blaming the through plugin's generated cells.

Another variation of this approach is to provide two call ports: one of `siHandlerBody` for Handler Function method and the other one for all remaining notification methods. This addresses the illegal connection issue but leaves some call ports unconnected like the approach in Section 5.2.6.2 does.

Whichever variation we choose, this approach unnecessarily consumes memory. Let us consider a hypothetical application design with three `tCyclicNotifier` cells C_1, C_2, and C_3. C_1 and C_2 use Handler Function method, each connected to a different target. C_3 uses some other method. Since C_1 and C_2 connect to different targets, the TECS generator implements their call ports by indirect calls and stores the respective entry port descriptors in their call INIBs. The layout of INIBs must be uniform across all instances of the celltype, so C_3 includes a slot for the entry port descriptor, too. However, C_3

uses a different notification method, and therefore will never use this entry port description, wasting memory[1].

5.2.7 *Error Notification*

The set of attributes for normal notifications is provided for error notifications as well, with the suffix `ForError`. Not all attributes are available for error notifications, as shown in Table 5.1. Furthermore, some special considerations are needed for error notifications:

- When a normal notification method is a Handler Function, Set Variable, or Increment Variable, the corresponding error notification method cannot be specified, because these notification methods never fail.
- On the other hand, if the specified normal notification method can fail, then omitting the error notification method will cause the notification error to go unnoticed. To prevent this, warnings are generated unless errors are ignored explicitly by specifying the `ignoreErrors` attribute.

1 Actually, this memory consumption issue can be addressed by the optimization method described in Section 5.3.5. However, this does not address other issues.

5.3 Plugin Implementation

We componentized the ASP3 time event notification based on the design presented in Section 5.2.

Our proposed celltype plugin `NotifierPlugin` plays a central role in the componentization of the time event notification. The plugin's main operation is to generate a static API statement for each time event notification cell. This process involves the following steps:

1. Perform the following steps for each handler (in doing so, the normal notification and error notification are called internally):
   a) Check the join of the common call port.
   b) Decide the handler type (which is a concept similar to the notification methods but involves differentiation between normal notifications and error notifications) according to the joined celltype and specified attributes.
   c) Raise an error if no handler type matches.
   d) Check whether the error notification method specification is missing and raise a warning appropriately.
   e) Generate part of the static API statement for the handler being processed.
2. Finally, the complete static API statement is generated according to the specified format string and written to the output file.

5.3.1 *Handlers*

Handlers are a concept introduced by our plugin to refer to each slot of normal notifications and error notifications. They are represented by two global instances of class `Handler`: `EVENT_HANDLER` for normal notifications and `ERROR_HANDLER` for error notifications. When something is dependent on which handler is in question in a particular context, it compares the current handler to `EVENT_HANDLER` and `ERROR_HANDLER`. Tasks common to both handlers can be implemented by iterating over an array of the handler objects.

5.3.2 *Handler Types*

The objective is to decide the notification method for each handler. Most notification methods are common to both handlers, but some differ in their behaviors between the handlers and take different sets of parameters. We introduce the concept of *handler types* to capture such differences. Table 5.2 lists all handler types and their corresponding notification methods.

Table 5.2: Handler types, their corresponding notification methods, and their supported handlers.

HANDLER TYPE	NOTIFICATION METHOD	HANDLER	
		NORMAL	ERROR
`ActivateTaskHandlerType`	Activate Task	✓	✓
`WakeUpTaskHandlerType`	Wake Up Task	✓	✓
`SetVariableHandlerType`	Set Variable	✓	
`SetVariableToErrorCodeHandlerType`	Set Variable		✓
`IncrementVariableHandlerType`	Increment Variable	✓	✓
`SignalSemaphoreHandlerType`	Signal Semaphore	✓	✓
`SetEventflagHandlerType`	Set Eventflag	✓	✓
`SendToDataqueueHandlerType`	Send To Dataqueue	✓	
`SendErrorCodeToDataqueueHandlerType`	Send To Dataqueue		✓
`UserHandlerType`	Handler Function	✓	
`NullHandlerType`	None		✓

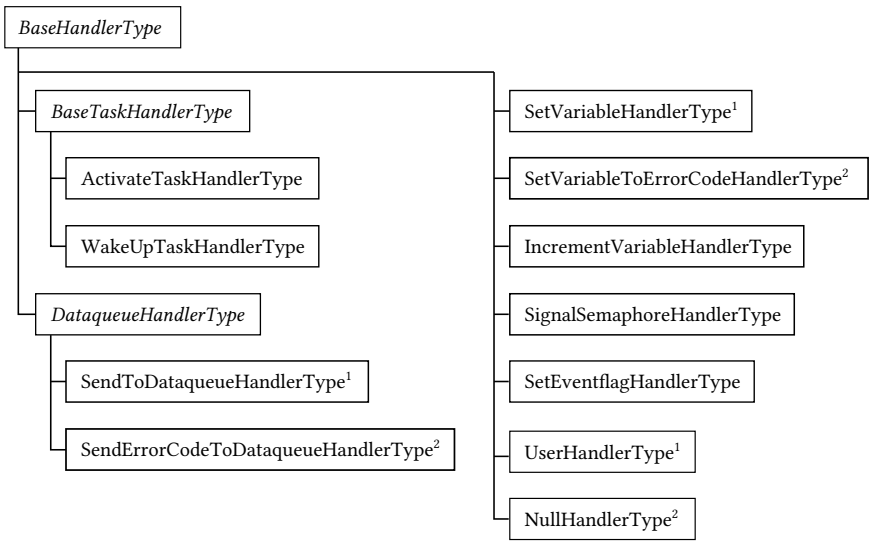
Our plugin represents each handler type as a *handler type class* and its singleton instance. Fig. 5.6 shows the class hierarchy of the defined handler type classes. These classes provide the following methods:

- `validate_join` returns a flag indicating whether a given combination of a joined celltype, the notification cell's attributes, and the current handler is appropriate for the handler type.
- `gen_cfg_handler_type` outputs the constant symbol representing the handler type's notification method to comprise the first value of `{<method>}`.
- `gen_cfg_handler_paremeters` outputs the parameters specific to that notification method to include them in the remaining part of `{<method>}`.
- `might_fail` returns a flag indicating whether the handler type is fallible (Section 5.3.3).

The plugin loops over all handler type objects and checks if the current cell's configuration and the current handler are suitable for each handler type. It adopts the first matching handler type. If none matches, it produces an error message and moves on to the next handler.

5.3.3 *Missing/Excess Error Notifications*

As we described in Section 5.2.7, some normal notification methods are fallible and can emit error notifications. Others are not, and the



[1] Not applicable for the error notification method.

[2] Only applicable for the error notification method.

Classes shown in *italics* are abstract.

Figure 5.6: The class hierarchy of the handler type classes.

kernel configurator disallows specifying an error notification method in this case.

Our plugin reports a warning if the error notification method is missing in the former case. Application developers may use the `ignoreErrors` attribute to suppress this warning if it is intentional. Conversely, it reports an error if the error notification method is specified in the latter case. To implement these rules, the handler type classes' have a method named `might_fail` to indicate if the handler type is fallible.

5.3.4 *Kernel Configuration Generation*

This plugin produces kernel configuration fragments. This process is comprised of three steps (Fig. 5.7):

Mapping Handler Types to Parameters

In the first step, the plugin generates the basic elements of the final configuration fragments. There are two elements for each handler: (a) a *notification mode*, which is a constant expression, such as `TNFY_HANDLER` and `TENFY_SETFLG`, that specifies a notification method, and; (b) *notification parameters*, which are additional pa-

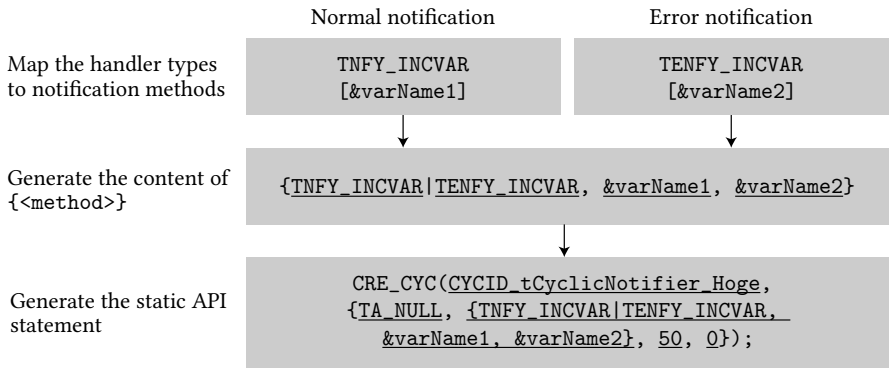


Figure 5.7: The steps of generating kernel configuration fragments.

rameters specific to each handler type, such as a word to send to a dataqueue.

The handler type classes' `gen_cfg_handler_type` and `gen_cfg_handler_paremeters` methods are responsible for generating these elements. How they generate notification parameters varies between handler types:

- `ActivateTaskHandlerType` and `WakeUpTaskHandlerType` read the joined task cell's `id` attribute value and output it as the notification parameter.
- `SetVariableHandlerType`, `SetVariableToErrorCodeHandlerType`, and `IncrementVariableHandlerType` outputs the specified variable value as the notification parameter.
- `SignalSemaphoreHandlerType` reads the joined semaphore cell's `id` attribute value and outputs it as the notification parameter.
- `SetEventflagHandlerType` reads the joined eventflag cell's `id` attribute value and outputs it as the notification parameter.
- `SendToDataqueueHandlerType` reads the joined dataqueue cell's `id` attribute value and outputs it as the first notification parameter. It also outputs the specified value as the second notification parameter.
- `SendErrorCodeToDataqueueHandlerType` reads the joined dataqueue cell's `id` attribute value and outputs it as the notification parameter.
- `UserHandlerType` outputs the name of an *adapter function* (Section 5.3.5) as the notification parameter.

Generating {<method>}

In the second step, the plugin combines the basic elements generated in the previous step to form the {<method>} parameter, ready to be embedded in the final kernel configuration directive.

Generating CRE_CYC or CRE_ALM

Finally, the plugin generates a CRE_CYC or CRE_ALM directive according to the format string specified by a plugin parameter, embedding the {<method>} parameter generated in the previous step and attribute values. Like the standard factory blocks (Section 2.4.2.4, the plugin replaces special tokens such as id. After that, the plugin writes the generated directive to the specified output file.

5.3.5 *Interfacing Handler Functions to the Kernel*

Handler functions for the signature siHandlerBody can be joined to the time event notification cell through the cell tTimeEventHandler. The static API of a time event notification expects the C function signature of handler functions to be void(*)(intptr_t). However, the C function signature of the corresponding entry port function never matches the expected one, and even changes depending on whether the entry port is an array or whether its celltype is defined as a singleton. This raises the need for *adapter functions*, which convert calls from the kernel to handler functions. A simple way to do this would be using a single adapter function for all time event notifications, letting TECS's interface code perform the remaining routing work. However, we found that this method, when done without call port optimization, increases the interrupt processing time by about 20 cycles. This might seem insignificant but is enough to discourage the use of the component.

To minimize the overhead, we adopted another approach: we made the plugin generate an adapter function for each entry port automatically, bypassing TECS's interface code. Adapter functions must pass up to two parameters to entry port functions: CELLIDX (non-singleton celltype only) and int_t (entry port array only). However, adapter functions can only receive one intptr_t parameter via the static API specification. Therefore, the remaining parameter value is coded in the function body, and different functions are generated for each distinct parameter value.

```
[context("non-task")]
signature siNotificationHandler {};
```

Figure 5.8: The signature for the common call ports.

5.4 Component Definition

We created a component description to realize the time event notification component. The defined celltypes use the TECS generator plugin proposed in Section 5.3.

5.4.1 *Signatures*

We defined a signature for the common call ports (Section 5.2.4). Fig. 5.8 shows the signature's definition. Since the common call ports are only used to convey the relationship between cells to the plugin and not to make TECS calls through them, the signature includes no functions.

5.4.2 *Celltypes*

We defined the celltypes `tCyclicNotifier` and `tAlarmNotifier` by using the celltypes `tCyclicHandler` and `tAlarmHandler` from ASP+TECS [99] as the starting point. They are for cyclic notifications and alarm notifications, respectively. Fig. 5.9 shows the definition of `tCyclicNotifier`.

We replaced `ciHandlerBody` with the common call ports `ciNotificationHandler` and `ciErrorNotificationHandler`. We added attributes to supply parameters to the notification methods. These attributes' values are only read by the plugin at build time, so the `[omit]` specifiers on these attributes prevent the TECS generator from reifying them, saving memory. The `factory` blocks to generate kernel configuration directives were superseded by our plugin, which is activated by the `[generate(...)]` specifiers on the celltypes.

We defined the celltype `tTimeEventHandler` that proxies Handler Function notification method's notification target. Fig. 5.10 shows its definition. Its entry port `eiNotificationHandler` connects to a time event notification cell's common call port, and its call port `ciHandlerBody` connects to a user-provided handler cell. The plugin implements handler calls by the method described in Section 5.3.5, so the `[omit]` specifier is used to suppress the generation of any runtime structures associated with the call port, such as port descriptors.

```
[active, generate(NotifierPlugin,
  "factory=\"CRE_CYC({{id}}, { {{attribute}}, { "
  "{{_handler_params_}} }, {{cycleTime}}, {{cyclePhase}}"
  " });\", output_file=tecsgen.cfg")]
celltype tCyclicNotifier {
  [inline] entry  sCyclic     eCyclic;

  call            siNotificationHandler   ciNotificationHandler;
  [optional] call siNotificationHandler
      ciErrorNotificationHandler;

  attr {
    ID              id = C_EXP("ALMID_$id$");
    [omit] ATR      attribute = C_EXP("TA_NULL");
    [omit] bool_t   ignoreErrors = false;
    [omit] RELTIM   cycleTime;
    [omit] RELTIM   cyclePhase = 0;
    // TNFY_SETVAR: Set Variable
    [omit] intptr_t *setVariableAddress = 0;
    [omit] intptr_t setVariableValue = 0;
    // TNFY_INCVAR: Increment Variable
    [omit] intptr_t *incrementedVariableAddress = 0;
    // TNFY_SETFLG: Set Eventflag
    [omit] FLGPTN flagPattern = 0;
    // TNFY_SNDDTQ: Send To Dataqueue
    [omit] intptr_t dataqueueSentValue = 0;
    // TENFY_SETVAR: Set Variable (on error)
    [omit] intptr_t *setVariableAddressForError = 0;
    // TENFY_INCVAR: Increment Variable (on error)
    [omit] intptr_t *incrementedVariableAddressForError = 0;
    // TENFY_SETFLG: Set Eventflag (on error)
    [omit] FLGPTN flagPatternForError = 0;
  };

  FACTORY { write("$ct$_factory.h",
    "#include \"kernel_cfg.h\""); };
};
```

Figure 5.9: The cyclic notification celltype.

```
celltype tTimeEventHandler {
  entry   siNotificationHandler   eiNotificationHandler;
  [omit] call siHandlerBody       ciHandlerBody;

  FACTORY {
    write("tecsgen.cfg", "#include \"$ct$_tecsgen.h\"");
  };
};
```

Figure 5.10: The celltype used for Handler Function notification method.

Table 5.3: The entry ports added to kernel object celltypes.

CELLTYPE	METHOD	ENTRY PORT
`tTask`	Activate Task	`eiActivateNotificationHandler`
	Wake Up Task	`eiWakeUpNotificationHandler`
`tSemaphore`	Signal Semaphore	`eiNotificationHandler`
`tEventflag`	Set Eventflag	`eiNotificationHandler`
`tDataqueue`	Send To Dataqueue	`eiNotificationHandler`

We added entry ports to some kernel object celltypes to allow them to be specified as notification targets. Table 5.3 lists the added entry ports.

5.4.3 *Composite Celltypes*

It would be cumbersome to instantiate two cells to use Handler Function notification method. Therefore, we defined the composite celltypes `tCyclicHandler` and `tAlarmHandler`, which combine `tTimeEventHandler` with their respective `t*Notifier` celltypes. Fig. 5.11 shows the definition of `tCyclicHandler`. Their interfaces are deliberately designed to imitate those of the identically-named celltypes from ASP+TECS to maximize compatibility.

```
[active]
composite tCyclicHandler {
  entry   sCyclic             eCyclic;
  [omit] call siHandlerBody   ciHandlerBody;

  attr {
    ID              id = C_EXP("ALMID_$id$");
    [omit] ATR      attribute = C_EXP("TA_NULL");
    [omit] RELTIM   cycleTime;
    [omit] RELTIM   cyclePhase = 0;
  };

  cell tCyclicNotifier CyclicNotifier {
    id = composite.id;
    attribute = composite.attribute;
    ciNotificationHandler = TimeEventHandler.
        eiNotificationHandler;
    cycleTime = composite.cycleTime;
    cyclePhase = composite.cyclePhase;
  };

  cell tTimeEventHandler TimeEventHandler {
    ciHandlerBody => composite.ciHandlerBody;
  };

  eCyclic => CyclicNotifier.eCyclic;
};
```

Figure 5.11: The cyclic handler celltype.

5.5 Experimental Evaluation

To evaluate the runtime overhead of our proposed time event notification component, we measured the difference between the interrupt processing time of our time event notification component and compared it with that of hand-written static API statements.

We performed the measurements on a TOPPERS/ASP3 kernel ported to a NUCLEO-F401RE Arm Cortex-M4 development board with an STMicroelectronics STM32F401RE MCU [89].

To measure the interrupt processing time of the time event notification, we created a cyclic notification firing every 50 microseconds. Its notification method was configured to the Handler Function. We selected the Handler Function because, whereas a static API statement is indeed sufficient for other methods, a bridge function is generated with the Handler Function and this can be the source of considerable runtime overhead. A function with a single call to `wait_randomly` (details for which are provided below) was specified as the handler function of the cyclic notification. We ran a busy loop while the cyclic notification was active. We measured the time required for each iteration, and obtained a histogram. The occurrence of an interrupt event during this busy loop renders the iteration time longer, resulting in two conspicuous peaks in the histogram. Therefore, the interrupt processing time can be estimated by measuring the distance between these two peaks.

5.5.1 *Test Cases*

We performed the measurements with the following three test cases:

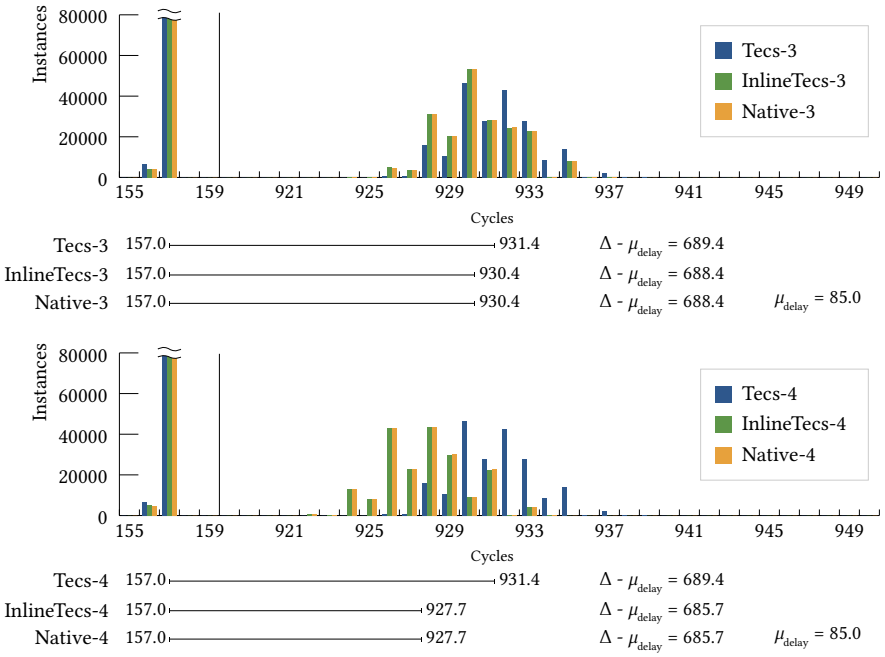
Tecs We created a cyclic notification using `tCyclicHandler`, whose call port is joined to an entry port of the singleton cell of the `tMain` celltype.

InlineTecs This case is identical to Tecs, except that the joined entry port is implemented as an inline function.

Native We created a cyclic notification by writing the `CRE_CYC` static API statement directly.

Each test case included two subcases: one with two dummy cyclic notifications (suffixed by `-3`), and the other with three dummy cyclic notifications (suffixed by `-4`).

During the experiment, we found that the processing time among the test cases was very close to two distinct values, as though they were "quantized". Whereas we could not identify the cause of this phenomenon, it could affect the accuracy of the results. There is a



For each test case, a histogram of the iteration time is shown. Each chart has two peaks, whose expected values are shown under the chart. The interrupt processing time Δ can be obtained by measuring the distance between the two peaks. The values obtained by subtracting the average execution time μ_{delay} from `wait_randomly` are shown on the right side.

Figure 5.12: Histograms of measured iteration times.

method known as *dithering* that involves adding a noise signal to ensure accurate estimations of the expected value of the original signal under the existence of the quantization effect [109]. Therefore, we wrote a function named `wait_randomly`, which waits for a random period of time following a normal distribution. We then added a call to this function into the handler function. We measured the average execution time μ_{delay} of `wait_randomly` beforehand to ensure that this was not included in the final results.

Fig. 5.12 shows histograms of the measured loop iteration time. As depicted in Fig. 5.12, our time event notification component's handler function invocation overhead was less than four cycles. Furthermore, the overhead was completely eliminated in the test case (InlineTecs) where the entry port was inlined.

5.6 Conclusion

In this chapter, we proposed a TECS generator plugin and showed our approach to componentize RTOS features requiring complex static API generation using the proposed plugin. Furthermore, we demonstrated that the runtime overhead of our time event notification component is small and can even be eliminated when the handler's entry port is inlined. We established this by measuring the interrupt processing time when a handler function was called via our component.

6 Conclusion

With their ever-increasing processing power and requirements, microcontroller-based embedded systems are now faced with all the problems which were only possessed by larger systems, such as development costs, quality, and security issues. Despite the ongoing trend of the increasingly blurring border between traditional embedded systems and connected systems, embedded systems have lagged in these aspects because of their unique requirements and hardware. This dissertation took on three research topics, each primarily focusing on partitioning, control-flow integrity, and component-based development, respectively, to advance the state of microcontroller-based embedded system development. We proposed techniques that are applicable to such systems, availing ourselves of microcontrollers' unique hardware features. Meanwhile, we have identified several potential research avenues and goals, which we list under each topic's conclusion.

Lightweight RTOS Utilizing TrustZone for Armv8-M

In Chapter 3, we proposed a new lightweight memory protection scheme that utilizes TrustZone for Armv8-M. The proposed scheme's implementation is a series of small modifications to a base kernel having no memory protection support. We implemented this scheme on TOPPERS/ASP3 to create an RTOS kernel with memory protection, named *ASP3+TZ*. We conducted an experimental evaluation to demonstrate that the proposed scheme incurs far less overhead than a traditional memory protection implementation. Finally, we presented the number of code lines modified as part of the implementation and showed that the proposed scheme only requires little engineering cost.

Full-fledged Partitioning We demonstrated in Chapter 3 that CMSE allows efficient implementation of memory protection. However, the proposed operating system traded other partitioning features, such as time protection and kernel object access control, for simplicity, thus leaving them on the table. This approach is comparable to FreeRTOS-MPU, which uses an MPU wrapper to retrofit the kernel with memory protection. An alternative would be to use an operating system kernel designed with memory protection in mind as the starting point. However, it introduces many problems because

of the mismatch between CMSE's automatic mode transition behavior and the operating system's intended hardware behavior. We are not suggesting that we should reconcile this disparity; rather, we consider it to be beneficial to lower the barriers of memory protection. Although the retrofitting approach's implementation in FreeRTOS-MPU has been shown to be insecure [112], this problem is most likely not fundamental to the approach. It would be interesting to see how well this approach's extension could handle full-fledged partitioning in a sound manner.

Code Sharing Between Secure and Non-Secure Worlds A problem with CMSE-based memory protection is that there is no trivial way to share code, e.g., a C runtime library, between Secure and Non-Secure worlds. In CMSE, the execution mode is strictly tied to the currently-running code region's security attribute. If we put it in a Non-Secure Callable region, it would allow Non-Secure callers to circumvent security checks. If we put it in a Non-Secure region, it would not be able to access Secure data. In our prototype, we sidestepped the problem by not using a C runtime library from Non-Secure code.

Interrupt-driven Multitasking An emerging design approach in embedded operating systems is interrupt-driven multitasking [38, 40, 43, 49, 50, 66], in which the kernel maps tasks to unused interrupt lines and leverages interrupt handling hardware to implement priority-based scheduling and dispatching. This approach has been successfully applied to support blocking system calls [49], memory protection [38], asynchronous programming [40], and real-time systems [43, 66] and offers a significantly lower overhead compared to traditional approaches without using specialized hardware, provided that its limited scalability is not a problem.

We expect incorporating interrupt-driven multitasking into CMSE-based memory protection to be a relatively uncomplicated task.

TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone For Armv8-M

In Chapter 4, we introduced *TZmCFI*, a holistic CFI solution for microcontroller-based embedded systems. TZmCFI is built upon several CFI techniques, namely shadow stacks, shadow exception stacks, and LLVM forward-edge CFI for comprehensive protection. We conducted a performance measurement of TZmCFI. The measured overhead of the shadow exception stack instrumentation was moderate if not significant, although a major portion of the overhead was caused by jumping between numerous stages of exception trampolines, leaving a further opportunity for optimization. As for the shadow stack

instrumentation, the overhead was clearly lower than previous works that use shadow stacks and target similar systems.

INTERRUPT-DRIVEN MULTITASKING As explained above, interrupt-driven multitasking is an emerging approach in embedded operating systems and offers a significantly lower overhead compared to traditional approaches. As interrupt-driven multitasking uses a processor's interrupt processing mechanism in an unusual way, supporting it in TZmCFI requires an extension to its control-flow model.

POST-BOOT TASK MANAGEMENT TZmCFI supports task creation only at boot time. This limitation is a deliberate choice we made to mitigate data-oriented attacks that take place after the boot process but can be inconvenient. We leave for future work the question of what post-boot task operations it can support safely.

COMPRESSED SHADOW STACKS μRAI [19] uses a compressed shadow stack representation, which we did not consider for use in TZmCFI because of its performance indeterminism. Barring that, it would be a beneficial approach and could be further improved if implemented by CMSE. In fact, the same argument about performance indeterminism applies to whole-program optimization. Since μRAI is an extension of shadow stacks, we expect it to be straightforward to replace TZmCFI's shadow stacks with that. This evokes the following research questions: (1) *What is the extent of its indeterminism?* (2) *Can we overcome the issue?* For example, a system can have a real-time portion and a non-real-time portion, and we could mitigate the indeterminism by allowing developers to mark functions that are ineligible for optimization manually.

EXCEPTIONAL CONTROL FLOWS The shadow stack instrumentation used in TZmCFI assumes every inter-procedural control transfer is either a function call or return and causes false positives for exceptional control transfers, such as `setjmp`, C++ exceptions, and Rust unwinding panics. Addressing this limitation is easy, but doing so while maintaining the soundness and a reasonable overhead is likely challenging.

Modern programming languages, such as C++20, Rust [104], and Zig [105], support coroutines to enable *asynchronous programming*. These programming languages implement this by converting functions into explicit state machines during compilation. Thus, the state data essentially serve as running tasks' program counters. However, this data is not protected by CFI per se, leaving this technique open to data-oriented attacks.

Dynamic Linking TZmCFI does not support dynamic linking. This requires a dynamic linker's cooperation to handle the information used by LLVM forward-edge CFI. Although dynamic linking is rarely used in microcontroller-based systems, it does have some use cases [42, 65].

Binary Instrumentation Like many other previous works [19, 80, 106, 111], the proposed CFI solution uses a modified LLVM compiler stack. However, this severely limits the choice of compilers. This approach also assumes that source code is available and poses a considerable problem for third-party software, which is not always available in a source code form. The alternative solution that does not have this limitation is binary instrumentation, but layout-modifying binary instrumentation requires a sophisticated binary instrumentation framework like the one used by the original CFI paper [16] and involves some heuristics. Preserving a binary layout is possible but comes at a tremendous runtime overhead [81].

An intermediate solution is to instrument assembler code[1] [64]. Compiled assembler code is almost as low-level as machine code and is thus harder to reverse-engineer than the source code. Therefore, we would expect third-party software vendors to be more forthcoming about releasing their middleware in this form, though releasing software in this form is indisputably unconventional.

CFI as a Right, Not a Privilege The largest obstacle of any binary exploit mitigations is reaching developers and gaining ubiquity and popularity. It has only been successful in the situations where it is driven by a large company having a near-pervasive control over a platform. The major consumer computing platforms except for Linux [82] have been progressive in this matter, as exemplified by Microsoft's Control Flow Guard [101], Android system CFI [85], and the use of Armv8.3-A pointer authentication [13] in Apple platforms [21]. Compared to this, we are only aware of *one* proprietary solution [92] for embedded systems. What is common among these examples of consumer platforms is that they are mostly transparent; most users and even developers are entirely unaware of these mechanisms' existence (maybe, except for gamers looking to maximize performance). This observation suggests development platforms and ecosystems take a vital role in the widespread deployment of binary exploit mitigations.

1 We considered this option at some point while designing TZmCFI but eventually moved away toward compiler instrumentation.

Componentizing an Operating System Feature Using a TECS Plugin

Lastly, in Chapter 5, we proposed a TECS generator plugin and showed our approach to componentize RTOS features requiring complex static API generation using the proposed plugin. Furthermore, we demonstrated that the runtime overhead of our time event notification component is small and can even be eliminated when the handler's entry port is inlined. We established this by measuring the interrupt response time when a handler function was called via our component.

NOTIFICATION BY VARIABLE Chapter 5 showed our approach to componentize the time event notifications defined in the TOPPERS Third-Generation Kernel Specification. However, the method for specifying a variable as the notification target still requires reconsideration because, in the current design, the target variable is set by explicitly specifying the address of the variable via an attribute. To leverage the advantages of the component system, it is more desirable for this connection to be represented as a join. We shall investigate this matter in future research.

[1] ARM Ltd. *Application Note 291: Word final word Using TrustZone on ARMv8-M*. http://www.keil.com/appnotes/docs/apnt_291.asp. ARM Ltd.

[2] ARM Ltd. *CMSIS — Arm*. https://www.arm.com/why-arm/technologies/cmsis. Retrieved on Nov 3, 2020.

[3] ARM Ltd. *Cortex-M3 Technical Reference Manual (Issue E)*. ARM Ltd., 2007.

[4] ARM Ltd. *Cortex-M1 Technical Reference Manual (Issue D)*. ARM Ltd., 2008.

[5] ARM Ltd. *ARM Security Technology - Building a Secure System using TrustZone Technology (issue C)*. https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. ARM Ltd., 2009.

[6] ARM Ltd. *Cortex-M0 Technical Reference Manual (Issue C)*. ARM Ltd., 2009.

[7] ARM Ltd. *Cortex-M4 Technical Reference Manual (Issue B)*. ARM Ltd., 2010.

[8] ARM Ltd. *ARMv8-M Security Extensions: Requirements on Development Tools*. https://developer.arm.com/documentation/ecm0359818/latest. ARM Ltd., 2015.

[9] ARM Ltd. *Cortex-M23 Technical Reference Manual (Issue C)*. ARM Ltd., 2016.

[10] ARM Ltd. *ARM CoreLink SIE-200 System IP for Embedded Technical Reference Manual (Revision G)*. https://developer.arm.com/documentation/ddi0571/g. ARM Ltd., 2017.

[11] ARM Ltd. *Cortex-M7 Technical Reference Manual (Issue F)*. ARM Ltd., 2018.

[12] ARM Ltd. *Armv8-M Architecture Reference Manual (Version B.f)*. ARM Ltd., 2019.

[13] ARM Ltd. *Armv8 Architecture Reference Manual, for Armv8-A architecture profile (Issue F.c)*. https://developer.arm.com/documentation/ddi0487/fc/. 2020.

[14] ARM Ltd. *Cortex-M33 Technical Reference Manual (Issue 0100-03)*. ARM Ltd., 2020.

[15] AUTOSAR. *Specification of Operating System*. http://www.autosar.org/standards/classic-platform/release-43/software-architecture/system-services/. Automotive Open System Architecture GbR.

[16] Martin Abadi, Mihai Budiu, and Úlfar Erlingsson. "Control-Flow Integrity." In: *ACM Conference on Computer and Communication Security (CCS)*. Alexandria, VA, 2005, pp. 340–353. URL: https://www.microsoft.com/en-us/research/publication/control-flow-integrity/.

[17] A. Abbasi et al. "Challenges in Designing Exploit Mitigations for Deeply Embedded Systems." In: *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 2019, pp. 31–46. DOI: 10.1109/EuroSP.2019.00013.

[18] Tigist Abera et al. "C-FLAT: Control-Flow Attestation for Embedded Systems Software." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 743–754. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978358. URL: http://doi.acm.org/10.1145/2976749.2978358.

[19] Naif Saleh Almakhdhub et al. "μRAI: Securing Embedded Systems with Return Address Integrity." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS20)*. Feb. 2020.

[20] Amazon Web Services, Inc. *FreeRTOS - Real-time operating system for microcontrollers - AWS*. https://aws.amazon.com/freertos/. Retrieved on Mar 30, 2020. 2020.

[21] Apple Inc. *Preparing Your App to Work with Pointer Authentication*. https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication. Retrieved on Dec 1, 2020. 2020.

[22] Arm Ltd. *Arm MPS2+ FPGA Prototyping Board*. https://www.arm.com/products/development-tools/development-boards/mps2-plus. Retrieved on May 14, 2019. 2019.

[23] Armis Inc. *URGENT/11 Leaves Billions of Devices Open to Cyber Security Risks*. https://www.armis.com/urgent11/. Retrieved on Nov 27, 2020. 2019.

[24] T. Azumi et al. "A New Specification of Software Components for Embedded Systems." In: *Proceedings of 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*. 2007, pp. 46–50. DOI: 10.1109/ISORC.2007.7.

[25] Takuya Azumi, Hiroshi Oyama, and Hiroaki Takada. “A Realization of RPC in Embedded Component Systems.” In: 情報処理学会研究報告. *SLDM, [システム LSI 設計技術]*. Vol. 134. Information Processing Society of Japan (IPSJ), 2008, pp. 97–102.

[26] Takuya Azumi, Hiroshi Oyama, and Hiroaki Takada. “Memory allocator for efficient task communications by using RPC channels in an embedded component system.” In: *Proceedings of the 12th IASTED International Conference on Software Engineering and Applications*. 2008, pp. 204–209.

[27] Takuya Azumi et al. “Wheeled inverted pendulum with embedded component system: a case study.” In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2010, pp. 151–155.

[28] Debayan Bose. “Component Based Development.” In: *CoRR* abs/1011.2163 (2010).

[29] Jacques Brygier and Mehmet Oezer. “Safety and security for the internet of things.” In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. TOULOUSE, France, Jan. 2016. URL: https://hal.archives-ouvertes.fr/hal-01292301.

[30] Erik Buchanan et al. “When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC.” In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS ’08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 27–38. ISBN: 9781595938107. DOI: 10.1145/1455770.1455776. URL: https://doi.org/10.1145/1455770.1455776.

[31] Stefan Bunzel. *AUTOSAR architecture expands safety and security applications*. https://www.eetimes.com/autosar-architecture-expands-safety-and-security-applications/. Retrieved on Nov 27, 2020. 2011.

[32] Nicholas Carlini and David Wagner. “ROP is Still Dangerous: Breaking Modern Defenses.” In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[33] Stephen Checkoway et al. “Comprehensive experimental analyses of automotive attack surfaces.” In: *USENIX Security Symposium*. Vol. 4. San Francisco. 2011, pp. 447–462.

[34] International Electrotechnical Commission. *IEC61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. 2010.

[35] John Criswell, Nathan Dautenhahn, and Vikram Adve. "KCoFI: Complete control-flow integrity for commodity operating system kernels." In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 292–307.

[36] John Criswell et al. "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems." In: *SOSP '07: Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*. Stevenson, WA, 2007.

[37] Ivica Crnkovic. "Component-based Software Engineering for Embedded Systems." In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. ACM, 2005, pp. 712–713.

[38] Daniel Danner et al. "Safer Sloth: Efficient, hardware-tailored memory protection." In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014, pp. 37–48. DOI: 10.1109/RTAS.2014.6925989.

[39] Lucas Davi et al. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." In: *Network and Distributed System Security Symposium (NDSS)*. Jan. 2012.

[40] *Drone | An Embedded Operating System for writing real-time applications in Rust*. https://www.drone-os.com. Retrieved on Nov 30, 2020. 2020.

[41] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. "A component model for hard real-time systems: CCM with ARINC-653." In: *Software: Practice and Experience* 41.12 (2011), pp. 1517–1550. DOI: https://doi.org/10.1002/spe.1083. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1083. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1083.

[42] Adam Dunkels et al. "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks." In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys '06. Boulder, Colorado, USA: Association for Computing Machinery, 2006, pp. 15–28. ISBN: 1595933433. DOI: 10.1145/1182807.1182810. URL: https://doi.org/10.1145/1182807.1182810.

[43] J. Eriksson et al. "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives." In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2013, pp. 110–113. DOI: 10.1109/SIES.2013.6601482.

[44] Aurélien Francillon and Claude Castelluccia. "Code Injection Attacks on Harvard-Architecture Devices." In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 15–26. ISBN: 9781595938107. DOI: 10.1145/1455770.1455775. URL: https://doi.org/10.1145/1455770.1455775.

[45] Mike Frysinger. *vDSO - overview of the virtual ELF dynamic shared object*. http://man7.org/linux/man-pages/man7/vdso.7.html.

[46] Le Guan et al. "TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone." In: *CoRR* abs/1704.05600 (2017). URL: http://arxiv.org/abs/1704.05600.

[47] Randy Harr and Jim Ready. "A Conversation with Jim Ready." In: *Queue* 1.2 (Apr. 2003), pp. 6–9. ISSN: 1542-7730. DOI: 10.1145/644254.644261. URL: https://doi.org/10.1145/644254.644261.

[48] Grant Hernandez et al. "Smart nest thermostat: A smart spy in your home." In: *Black Hat USA*. 2014.

[49] W. Hofer, D. Lohmann, and W. Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads." In: *2011 IEEE 32nd Real-Time Systems Symposium*. 2011, pp. 67–77. DOI: 10.1109/RTSS.2011.14.

[50] W. Hofer et al. "Sloth: Threads as Interrupts." In: *2009 30th IEEE Real-Time Systems Symposium*. 2009, pp. 204–213. DOI: 10.1109/RTSS.2009.18.

[51] トロン協会 ITRON 仕様検討グループ. *μITRON4.0 仕様*. http://www.ertl.jp/ITRON/SPEC/mitron4-j.html.

[52] *Is Embedded Behind*. https://wiki.c2.com/?IsEmbeddedBehind. Retrieved on Nov 28, 2020. 2012.

[53] T. Ishikawa et al. "HR-TECS: Component technology for embedded systems with memory protection." In: *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. 2013, pp. 1–8. DOI: 10.1109/ISORC.2013.6913200.

[54] Georges-Axel Jaloyan et al. "Return-Oriented Programming on RISC-V." In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 471–480. ISBN: 9781450367509. URL: https://doi.org/10.1145/3320269.3384738.

[55] Patrick Koeberl et al. "TrustLite: A Security Architecture for Tiny Embedded Devices." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 10:1–10:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592824.

[56] Constantinos Kolias et al. "DDoS in the IoT: Mirai and other botnets." In: *Computer* 50.7 (2017), pp. 80–84.

[57] P. Koopman. "Embedded system security." In: *Computer* 37.7 (2004), pp. 95–97. DOI: 10.1109/MC.2004.52.

[58] Philip Koopman, Jennifer Black, and Theresa Maxino. "Position paper: Deeply embedded survivability." In: *ARO Planning Workshop on Embedded Systems and Network Security*. Raleigh NC, Feb. 2007.

[59] Tim Kornau et al. "Return oriented programming for the ARM architecture." PhD thesis. Master's thesis, Ruhr-Universität Bochum, 2010.

[60] Ihor Kuz et al. "CAmkES: A component model for secure microkernel-based embedded systems." In: *Journal of Systems and Software* 80.5 (2007). Component-Based Software Engineering of Trustworthy Embedded Systems, pp. 687 –699. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2006.08.039. URL: http://www.sciencedirect.com/science/article/pii/S016412120600224X.

[61] R. Langner. "Stuxnet: Dissecting a Cyberwarfare Weapon." In: *IEEE Security Privacy* 9.3 (2011), pp. 49–51. DOI: 10.1109/MSP.2011.67.

[62] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation." In: *International Symposium on Code Generation and Optimization*. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[63] Xavier Leroy. "Formal verification of a realistic compiler." In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: http://xavierleroy.org/publi/compcert-CACM.pdf.

[64] J. Li et al. "ABCFI: Fast and Lightweight Fine-Grained Hardware-Assisted Control-Flow Integrity." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3165–3176. DOI: 10.1109/TCAD.2020.3012640.

[65] Yixiao Li, Yutaka Matsubara, and Hiroaki Takada. "EV3RT: A Real-time Software Platform for LEGO Mindstorms EV3." In: *Computer Software* 34.4 (2017), pp. 91–115. ISSN: 0289-6540. DOI: 10.11309/jssst.34.4_91. URL: https://ci.nii.ac.jp/naid/130006855221/en/.

[66] Per Lindgren et al. "Abstract Timers and Their Implementation onto the ARM Cortex-M Family of MCUs." In: *SIGBED Rev.* 13.1 (Mar. 2016), pp. 48–53. DOI: 10.1145/2907972.2907979. URL: https://doi.org/10.1145/2907972.2907979.

[67] Frédéric Loiret et al. "Component-Based Real-Time Operating System for Embedded Applications." In: *Component-Based Software Engineering*. Ed. by Grace A. Lewis, Iman Poernomo, and Christine Hofmeister. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 209–226. ISBN: 978-3-642-02414-6.

[68] Real Time Engineers Ltd. *FreeRTOS-MPU - ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit support in FreeRTOS*. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[69] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution." In: *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: ACM, 2013, 10:1–10:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368.

[70] Charlie Miller and Chris Valasek. "A survey of remote automotive attack surfaces." In: *Black Hat USA*. 2014.

[71] João Batista Corrêa Gomes Moreira. "Protection mechanisms against control-flow hijacking attacks." PhD thesis. University of Campinas, 2016.

[72] NPO 法人 TOPPERS プロジェクト. *TOPPERS プロジェクト / TECS*. http://www.toppers.jp/tecs.html.

[73] NPO 法人 TOPPERS プロジェクト. *TOPPERS プロジェクト / TOPPERS/ASP3 カーネル*. https://www.toppers.jp/asp3-kernel.html.

[74] NPO 法人 TOPPERS プロジェクト. *TOPPERS プロジェクト / TOPPERS/HRP2 カーネル*. https://www.toppers.jp/hrp2-kernel.html.

[75] NPO 法人 TOPPERS プロジェクト. *TOPPERS プロジェクト / TOPPERS/JSP カーネル*. https://www.toppers.jp/jsp-kernel.html.

[76] NPO 法人 TOPPERS プロジェクト. *TOPPERS プロジェクト*. http://www.toppers.jp/.

[77] NXP Semiconductors. *LPC55S6x: High Efficiency Arm® Cortex®-M33-based Microcontroller Family*. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc5500-cortex-m33/high-efficiency-arm-cortex-m33-based-microcontroller-family:LPC55S6x. Retrieved on Nov 28, 2020.

[78] NXP Semiconductors. *i.MX RT1170 Crossover MCU Family - First GHz MCU with Arm® Cortex®-M7 and Cortex-M4 Cores*. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1170-crossover-mcu-family-first-ghz-mcu-with-arm-cortex-m7-and-cortex-m4-cores:i.MX-RT1170. Retrieved on Nov 28, 2020.

[79] Yuki Nagahara et al. "Distributed Intent: Android Framework for Networked Devices Operation." In: *Proceedings of 10th IEEE International Conference on Embedded Software and Systems (ICESS 2013)*. 2013, pp. 651–658. DOI: 10.1109/CSE.2013.101.

[80] Ben Niu and Gang Tan. "Per-Input Control-Flow Integrity." In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 914–926. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813644. URL: http://doi.acm.org/10.1145/2810103.2813644.

[81] Thomas Nyman et al. "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers." In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Marc Dacier et al. Cham: Springer International Publishing, 2017, pp. 259–284. ISBN: 978-3-319-66332-6.

[82] Bjorn Pagen. *State Of Linux Desktop Security*. https://bjornpagen.com/blog/linux-security. Retrieved on Sep 19, 2020. 2020.

[83] A. Pinho, L. Couto, and J. Oliveira. "Towards Rust for Critical Systems." In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 19–24. DOI: 10.1109/ISSREW.2019.00036.

[84] Jonathan Pollet and Joe Cummins. "Electricity for free? The dirty underbelly of scada and smart meters." In: *Black Hat USA*. 2010.

[85] Android Open Source Project. *Control Flow Integrity*. https://source.android.com/devices/tech/debug/cfi. Retrieved on Dec 1, 2020. 2020.

[86] RTCA, Inc. *RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[87] Realtime Rik (https://stackoverflow.com/users/6517894). *Avoiding CortexM Interrupt Nesting*. Stack Overflow. https://stackoverflow.com/a/52887239 (version:2018-10-19).

[88] Chris Rommel. *The Battle for the IoT is Being Fought in the MCU Software Ecosystem*. https://www.vdcresearch.com/News-events/iot-blog/the-battle-for-the-iot-is-being-fought-in-the-mcu-software-ecosystem.html. Retrieved on Nov 29, 2020. 2015.

[89] *STM32F401xD/E Datasheet*. http://www.st.com/web/en/resource/technical/document/reference_manual/DM00096844.pdf. STMicroelectronics.

[90] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not." In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[91] A. Sadeghi, C. Wachsmann, and M. Waidner. "Security and privacy challenges in industrial Internet of Things." In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2747942..

[92] Karamba Security. *Control Flow Integrity (CFI)*. https://www.karambasecurity.com/products/cfi. 2020.

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

[94] K. Z. Snow et al. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 574–588. DOI: 10.1109/SP.2013.45.

[95] International Organization for Standardization. *ISO 26262:2011, Road vehicles –Functional safety*. 2011.

[96] L. Szekeres et al. "SoK: Eternal War in Memory." In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[97] *TECS Specification*. V1.0.2.32. http://www.toppers.jp/download.cgi/tecs_package-20120608.tar.gz. TOPPERS Project, Inc. 2011.

[98] *TOPPERS New Generation Kernel Specification*. Release 1.7.1. https://www.toppers.jp/docs/tech/ngki_spec-171.pdf. TOPPERS Project, Inc. 2015.

[99] TOPPERS Project, Inc. *TECS Combined Packages*. https://www.toppers.jp/tecs.html#e-package.

[100] *TOPPERS Third Generation Kernel (ITRON-based) Specification*. Release 3.0.0. https://www.toppers.jp/docs/tech/tgki_spec-300.pdf. TOPPERS Project, Inc. 2016.

[101] Jack Tang. *Exploring Control Flow Guard in Windows 10*. https://www.trendmicro.com/en_us/research/15/a/exploring-control-flow-guard-in-windows-10.html. Retrieved on Nov 28, 2020. 2015.

[102] Imagination Technologies. *MIPS-VZ Security Features as Compared to ARMv8-M CMSE*. https://www.mips.com/downloads/mips-vz-security-features-as-compared-to-armv8-m-cmse/.

[103] The Clang Team. *ShadowCallStack — Clang 11 documentation*. https://clang.llvm.org/docs/ShadowCallStack.html. Retrieved on Mar 30, 2020. 2020.

[104] The Rust team. *Rust Programming Language*. https://www.rust-lang.org. Retrieved on Sep 30, 2020.

[105] The Zig community. *The Zig Programming Language*. https://ziglang.org. Retrieved on Sep 30, 2020.

[106] Caroline Tice et al. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 941–955. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[107] Robert Wahbe et al. "Efficient software-based fault isolation." In: *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 1993, pp. 203–216.

[108] Robert J Walls et al. "Control-Flow Integrity for Real-Time Embedded Systems." In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[109] Bernard Widrow and István Kollár. *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge, UK: Cambridge University Press, 2008. ISBN: 9780521886710.

[110] XMOS. *XC Specification (X5965A)*. https://www.xmos.ai/download/XC-Specification(X5965A).pdf. XMOS, 2011.

[111] Jie Zhou et al. "Silhouette: Efficient Protected Shadow Stacks for Embedded Systems." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1219–1236. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie.

[112] Wei Zhou et al. *Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems*. 2019. arXiv: 1908.03638 [cs.CR].

[113] Victor van der Veen et al. "Practical Context-sensitive CFI." English. In: *CCS 2015 - Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Vol. 2015-October. Association for Computing Machinery (ACM), Oct. 2015, pp. 927–940. DOI: 10.1145/2810103.2813673.

[114] トロン協会バージョンアップ WG. *μITRON4.0* 仕様保護機能拡張. http://www.ertl.jp/ITRON/SPEC/px-j.html.

[115] 中嶋 健一郎 et al. "セキュリティ支援ハードウェアによるハイブリッド OS システムの高信頼化." In: 情報処理学会研究報告. *EMB,* 組込みシステム 2008.116 (2008), pp. 1–7. ISSN: 09196072. URL: http://ci.nii.ac.jp/naid/110007099117/.

[116] 高田 広章. "静的 OS のための静的 API とコンフィギュレータ." In: コンピュータソフトウェア 37.3 (2020), pp. 45–66. DOI: 10.11309/jssst.37.3_45.

[117] 本田晋也, 岡部亮, and 攝津敦. 車載システム向けの仮想マシンの割込み応答性向上手法. Tech. rep. 14. 名古屋大学大学院情報科学研究科, 三菱電機株式会社情報技術総合研究所, 三菱電機株式会社情報技術総合研究所, 2017.

[118] 石川拓也, 本田晋也, and 高田広章. "静的なメモリ配置を行うメモリ保護機能を持ったリアルタイム OS." In: コンピュータソフトウェア 29.4 (2012), pp. 161–181.

Publications

Journal Papers

- Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. "Design and Implementation of RTOS-Aware Control-Flow Integrity Mechanism for Microcontroller-Based Systems" in Japan Society for Software Science and Technology, *Journal on Computer Software* (under review).
- Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. "TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone For Armv8-M" in Springer, *International Journal of Parallel Programming*, Jul 2020. DOI: 10.1007-/s10766-020-00673-z
- Tomoaki Kawada and Shinya Honda. "Lightweight RTOS Utilizing TrustZone for ARMv8-M" in Information Processing Society of Japan, *Journal of Information Processing*, Vol.59, No.2, pp. 762–774, Feb 2018 (in Japanese).

International Conferences (Peer-Reviewed)

- Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. "Shadow Exception Stacks: Control-Flow Integrity for Asynchronous Exceptions Using TrustZone for Armv8-M" in *the 6th International Embedded Systems Symposium (IESS 2019)*, Friedrichshafen, Germany, Sep 2019.
- Tomoaki Kawada, Takuya Azumi, Hiroshi Oyama, and Hiroaki Takada. "Componentizing an Operating System Feature Using a TECS Plugin" in *the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA 2016)*, Nagoya, Japan, Oct 2016.

Domestic Conferences

- Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. "Shadow Exception Stacks: TrustZone-M-based CFI for Asynchronous Exceptions," in *the Symposium on Cryptography and Information Security (SCIS2019)*, Jan 2019 (in Japanese).

- Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. "Study on Multitasking-aware Control-Flow Integrity based on TrustZone for Armv8-M," in *the Embedded System Symposium 2018 (ESS2018)*, Aug 2018 (in Japanese).
- Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. "Multitasking-aware Control-Flow Integrity based on TrustZone for Armv8-M," poster presented at *the 20th Summer Workshop on Embedded System Technologies (SWEST20)*, Aug 2018 (in Japanese).
- Tomoaki Kawada and Shinya Honda. "Lightweight RTOS Utilizing TrustZone for ARMv8-M," in *ETNET2017*, Mar 2017 (in Japanese).
- Tomoaki Kawada, Takuya Azumi, Hiroshi Oyama, and Hiroaki Takada. "Componentization of Operating System Feature Using a TECS Plugin," poster presented at 第 41 回組込みシステム合同研究発表会, Jun 2016 (in Japanese).
- Tomoaki Kawada, Takuya Azumi, Hiroshi Oyama, and Hiroaki Takada. "Componentization of Operating System Feature Using a TECS Plugin," in *ETNET2016*, Mar 2016 (in Japanese).

Award

- Received Information Processing Society of Japan Tokai Branch, Incentive Award (2018) for the paper "Lightweight RTOS Utilizing TrustZone for ARMv8-M."

Colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

This document was typeset on Nix using Tectonic, a modernized XeTeX engine.

Final Version as of January 14, 2021 (Version 1.0).