Protecting Applications on Real-Time Embedded Systems from Control-Flow Hijacking with Kage

by

Yufei Du

Submitted in Partial Fulfillment of the
Requirements for the Degree
Master of Science

Supervised by Professor John Criswell

Department of Computer Science
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York
2020

Contents

Biographical Sketch **v**

Acknowledgment **vi**

Abstract **vii**

Contributors **viii**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

2 Background **3**

2.1 ARMv7-M 3

2.1.1 Processor Modes 3

2.1.2 Memory Address Space 4

2.1.3 Memory Protection Unit 4

2.1.4 Hardware Abstraction Library 5

2.1.5 Unprivileged Store Instructions 5

2.1.6 Exception Handling 6

2.2 Silhouette 6

2.3 AWS FreeRTOS 7

3 Design **10**

3.1 Threat Model . . . 10

3.2 Overview of Kage . . . 10

3.3 Overview of KageOS Kernel . . . 11

3.4 Memory Isolation . . . 12

3.4.1 Memory Regions . . . 12

3.4.2 MPU Configuration . . . 14

3.5 Secure API . . . 14

3.6 The Untrusted Kernel . . . 17

3.7 Protecting Processor State . . . 18

3.8 Exception Handling . . . 19

4 Implementation **22**

4.1 System Assumptions . . . 22

4.2 Changes of Silhouette . . . 23

4.3 KageOS . . . 24

4.3.1 Un-trusting the Kernel . . . 24

4.3.2 Isolating Trusted and Untrusted Memory . . . 25

4.3.3 Secure API . . . 26

4.3.4 Context Switching . . . 26

4.3.5 Exception Handling . . . 27

4.4 Limitations of Implementation . . . 28

5 Security Discussion **29**

5.1 Overview . . . 29

5.2 Storing Security-Critical Data . . . 30

5.2.1 Scheduler Data and Task Management Data . . . 30

5.2.2 Secure API Data . . . 31

5.2.3 Control Data . . . 32

5.3 Securing Protected Memory Regions 33
5.3.1 Application Code and Untrusted Kernel Code 33
5.3.2 Runtime Checks of Secure API 34

6 Evaluation **36**
6.1 Performance Experiments 37
6.1.1 Experiment Results 40
6.2 Code Size Experiments 43
6.2.1 Experiment Results 44

7 Related Works **46**
7.1 Control-flow Hijacking Defense on General Purpose Systems 46
7.2 Security Enhancement of Embedded OS 47
7.2.1 Control-flow Hijacking Defense 47
7.2.2 Memory Safety 47
7.2.3 Intra-address Space Isolation 48
7.3 Control-flow Hijacking Defense on Bare-metal Embedded Devices 48

8 Conclusions **49**

Bibliography **51**

Biographical Sketch

The author was born in Shenzhen, Guangdong, China. He attended University of Rochester with a Bechelor of Science degree in computer science. He stayed in University of Rochester to begin his Master program in computer science in 2018. He pursued his research in computer security under the direction of advisor John Criswell.

Acknowledgment

I would like to thank my research advisor, John Criswell. Through chatting with him since my senior year in undergraduate, I shifted from not having any interest in researches to willing to do a Master thesis project. Without him, I would not be conducting researches, writing a thesis, or heading to a PhD program. His passion in system security and his seriousness in "doing things the right way instead of the fast way" have great impacts on me.

I would like to thank Robert J. Walls from Worcester Polytechnic Institute for his support and suggestions on this work. I would like to thank Michael Scott for serving on the committee of this thesis.

The Rochester Security Group has taught me many things. Isaac Richter showed me what system security research is like. Zhuojia Shen, Jie Zhou and Lele Ma taught me how to write high-quality compiler code, how to make research decisions, and how to work efficiently in a team.

The URCS Systems Group has given me much support. The systems lunch has become my favorite Friday activity. Even though the food we had were usually either too spicy or too oily to me, chatting with systems professor and other students on all topics has always been a great joy. The systems seminar has been a great source of learning the ongoing researches from my colleagues, although sometimes I fell asleep during the seminar.

Finally, I would like to thank my parents for their love, their support, and their understanding of my career decisions.

Abstract

This thesis presents *Kage*: a software defense system that protects control data of both applications and the kernel for real-time embedded systems with ARMv7-M microcontroller. Kage uses Silhouette to provide protected shadow stacks and CFI checks to applications and the kernel. Kage combines Silhouette with *KageOS*, an embedded operating system based on AWS FreeRTOS that protects control data in the memory. KageOS ensures that data structures containing control data are stored in a protected memory space that only the small trusted code in the kernel have write access. KageOS also adds runtime checks to the entry points of the trusted kernel to prevent untrusted code from calling trusted functions with corrupted arguments. Comparing to FreeRTOS, Kage incurs 71.57% performance overhead in context switching, 82.76% performance overhead in transferring data between tasks using a the queue API, and 47.70% performance overhead in transferring data between tasks using the stream buffer API. Kage incurs 313 additional CPU cycles in exception handling with an untrusted exception handler, and up to 397 cycles in executing a secure API. Kage incurs 18.62% code size overhead comparing to FreeRTOS.

Contributors

This work was supported by a thesis committee consisting of Professor John Criswell (advisor) and Professor Michael Scott of the Department of Computer Science and Professor Robert J. Walls of the Department of Computer Science of Worcester Polytechnic Institute. All work conducted for the thesis was completed by the student independently.

List of Tables

3.1 Task management secure API for application tasks and untrusted kernel . . . 15
3.2 Task management secure API for untrusted kernel only 15
3.3 Task management secure API for untrusted exception handlers only 15
3.4 Data types of secure API 16

4.1 MPU configurations 26

6.1 Performance overhead of exception handling and secure API incurred by Kage 40
6.2 Performance overhead of context switching and untrusted kernel API 40
6.3 Code size measurements 44

List of Figures

2.1 Reference address space 4

3.1 Architecture of KageOS 11
3.2 Address space of KageOS 12
3.3 RAM layout of KageOS 12

Chapter 1

Introduction

Embedded systems are becoming increasingly popular and feature-rich. In addition to traditional embedded systems such as routers, modems and security cameras, recent developments of Internet of Things devices [14] such as smart sensors, smartwatches and smart door locks allow more embedded systems than ever to connect to the Internet. Today, many embedded systems use microcontrollers instead of general purpose processors for the benefit of low cost and simple software design [43].

The benefit of simple software design comes at a cost, however. Most software for microcontroller-based embedded system is developed in C. Since C is not a memory-safe programming language, these embedded systems can be vulnerable to memory safety errors [26, 39], similar to C programs running on desktop systems. Moreover, by default, embedded operating systems for these systems provides little to no isolation between applications and the kernel [2], meaning an application has full access to kernel memory and the memory of other applications.

Previous work presented Silhouette [44], a software defense that protects against control-flow hijacking attacks on bare-metal software on the ARMv7-M architecture. Silhouette utilizes ARMv7-M's unprivileged store instructions, a set of store instructions that always check the unprivileged access permissions regardless of the current privilege mode. Silhouette efficiently protects the shadow stack it provides with store hardening, an intra-address space

isolation mechanism by transforming all store instructions in application code to unprivileged store instructions. Silhouette instruments the application code to access return addresses from the parallel shadow stack [21], configures the Memory Protection Unit (MPU) to prevent application code to write to the shadow stack and other critical memory regions, and adds forward-edge CFI checks. While it can protect the control-flow of bare-metal programs, Silhouette does not protect all control data in a multi-task system with context switching and interrupts—the processor state needs to be protected as well.

In this thesis, we present *Kage* (the word "kage" means "shadow" in Japanese). Kage is a software defense system that combines Silhouette and *KageOS*, a secure real-time operating system based on Amazon AWS FreeRTOS [2] that ensures that control data of both applications and the kernel are stored in protected memory. Together, Kage protects the operating system and applications from control-flow hijacking attacks.

We summarize this thesis's contributions:

- We built KageOS, a real-time operating system that provides a shadow stack for each application and the kernel to store the processor state during context switch and exception handling, and protected memory regions to securely store other data structures that contain control data including the scheduler data and task management data.

- We have developed Kage, a combination of the Silhouette compiler and runtime system and KageOS real-time operating system, that protects control data of applications and the kernel from corruption.

- We evaluated Kage on a STM32L475 Discovery board [36]. Comparing to an unmodified FreeRTOS system with default configuration, Kage incurs performance overheads of 71.57% in context switching, 82.76% in transferring data using the queue API, and 47.70% in transferring data using the stream buffer API. KageOS's additional protections incur 313 CPU cycles in untrusted exception handling, and up to 397 cycles for the secure API. Kage incurs 18.62% of code size overhead comparing to an unmodified FreeRTOS system with default configuration.

Chapter 2

Background

2.1 ARMv7-M

Kage targets the ARMv7-M architecture [12]. ARMv7-M belongs to the ARM Cortex-M product family, which is designed to be an architecture for low-performance, energy efficient, and low-cost microcontrollers [43]. As a result, ARMv7-M has many different design decisions than architectures designed for general-purpose processors such as X86 [30] or the ARM Cortex-A product family [11, 13]. These differences lead to the need for novel software defenses.

2.1.1 Processor Modes

ARMv7-M supports two processor modes: Thread mode and Handler mode [12]. Applications usually run in Thread mode, whereas exception handlers always run in Handler mode. Thread mode allows two privilege modes: privileged mode and unprivileged mode; Handler mode always runs in privileged mode.

ARMv7-M has a process stack pointer register (`PSP`) and a main stack pointer register (`MSP`). In the context of an embedded operating system, the system initialization code and exception handlers use the main stack; applications and kernel functions called by applications use the process stack.

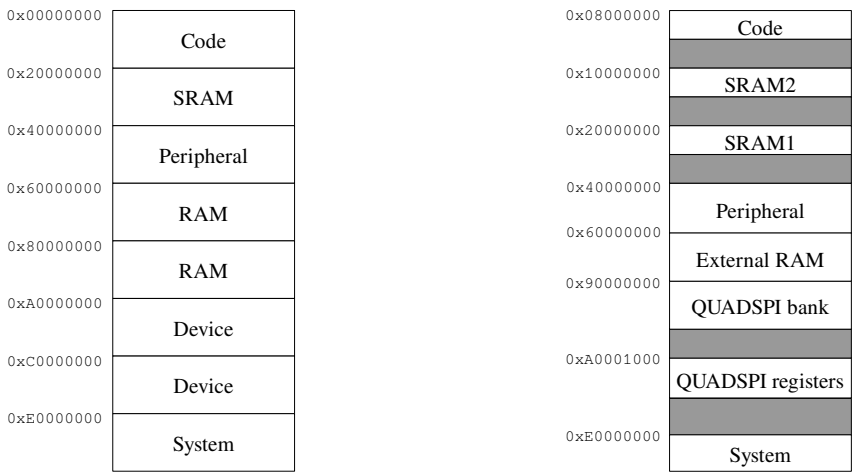

Figure 2.1: Reference address space
of ARMv7-M [12] (left) and address space of STM32L475 Discovery board [36, 38] (right)

2.1.2 Memory Address Space

Unlike desktop systems, ARMv7-M does not provide a Memory Management Unit (MMU); therefore, ARMv7-M does not support virtual memory. All memory regions for both privileged and unprivileged code are in the same address space. Furthermore, ARMv7-M maps all flash, memory, peripherals and the processor's system control registers to the same address space. Figure 2.1 [44] shows the reference address space of ARMv7-M and the address space of the STM32L475 Discovery board [36] used in this thesis. The gray area indicates unmapped addresses. QUADSPI is a feature of the discovery board to access an external NOR FLASH memory. We do not utilize this feature in this thesis. The microcontroller of the discovery board [38] reserves memory space for external RAM support, but it is unused because the board does not contain any external RAM.

2.1.3 Memory Protection Unit

Although ARMv7-M does not contain a MMU, the architecture provides a Memory Protection Unit (MPU) as an optional feature [12]. Microcontroller manufacturers can choose

to implement this feature in their products. The MPU allows developers to define memory protection regions. For each region, developers can configure the base address, length, and access permissions of this region. ARMv7-M supports up to 256 protection regions, but the exact number of protection regions supported varies in different hardware implementations. The MPU supports defining sub-regions within a protection region and allows a sub-region to have different access permissions than the region that contains it. This feature saves the number of MPU regions required and simplifies the process of configuring the MPU. When the MPU is disabled, ARMv7-M applies a default access control configuration, which sets the Peripheral, Device, and System (which contains system registers) memory regions to execute-never. This default configuration can also optionally be enabled when the MPU is enabled to serve as a background region. The System region is always only writable in privileged mode regardless of the MPU configuration.

2.1.4 Hardware Abstraction Library

In the world of ARMv7-M microcontrollers, manufacturers provide a Hardware Abstraction Library (HAL) [35] for applications to access the hardware. The HAL also includes the default interrupt handlers and part of the startup code such as setting the processor clock.

2.1.5 Unprivileged Store Instructions

ARMv7-M supports a special set of store instructions—the unprivileged store instructions—that always check the unprivileged access permission, regardless of the current processor mode. Even if the processor is currently in privileged mode, the unprivileged store instructions can only successfully write to memory locations that are configured as writable in unprivileged mode; attempting to write to a privileged only memory location will trigger a memory management fault. This set of instructions is also available on several other ARM architectures such as ARMv7-A [11] and ARMv8-M Main Extension [29].

2.1.6 Exception Handling

ARMv7-M automatically stores a subset of the current processor state when executing an exception handler and automatically restores it on exception return [12]. Depending on the current processor mode, on exception entry, the processor stores a subset of the processor state on either the process stack or the main stack. The other registers are up to the exception handler to save them if needed. ARMv7-M allows exception chaining when an exception occurs while another exception handler is running: if the new exception has higher priority than the current exception, then the new exception will preempt the current exception; otherwise, the new exception will be pending until the handler of the current exception returns.

2.2 Silhouette

Silhouette [44] is a software system that protects bare-metal embedded applications from control-flow hijacking attacks. Silhouette uses store hardening, a technique that provides intra-address space isolation without the expensive switches between privileged and unprivileged mode using unprivileged store instructions. We describe this technique in more detail in the next paragraph. Silhouette uses store hardening to prevent application code from writing to the shadow stack. Combined with label-based forward-edge control-flow integrity checks [8], a privileged code scanner, and a secure MPU configuration, Silhouette guarantees return address integrity: a function always returns to the correct return address stored on the shadow stack, and the return address cannot be corrupted.

Silhouette consists of four compiler passes to transform and check the application code during linking stage. Silhouette's compiler passes are:

1. **Shadow Stack Transformation**: Silhouette instruments function prologue and epilogue of each application function such that when entering a function, the return address is saved on the shadow stack, and when returning from the function, the system uses the return address from the shadow stack instead of the regular stack.

2. **Store Hardening**: In order to provide intra-address space isolation while running both trusted and untrusted code in privileged mode, Silhouette transforms all store instructions of the application code into ARMv7-M's unprivileged store instructions such that the application code only has write permission to unprivileged memory regions.

3. **CFI Transformation**: With the shadow stack transformation pass protecting the system from backward-edge control-flow hijacking, Silhouette also protects forward-edge control flow with CFI. For indirect jumps and indirect function calls, Silhouette inserts CFI labels to the beginning of the destination basic blocks and inserts checks before the jump or the function call to verify that target location has the correct label.

4. **Privileged Code Scanner**: In order to protect ARMv7-M's control registers, which contains critical data structures such as the location of stack pointer, Silhouette scans the compiled native code of application code to ensure that the application code cannot overwrite control registers under any circumstances.

Silhouette applies the following MPU policies. First, Silhouette sets the code segment to be readable, executable but non-writable for both unprivileged and privileged accesses. The code segment is the only memory region with executable permission. This ensures that an attacker cannot inject code to the device's memory and execute it. Second, Silhouette sets the shadow stack region to be writable for privileged accesses only in order to prevent application code to overwrite values stored on the shadow stack. Third, Silhouette sets the peripheral and device regions to be writable in privileged mode only. Finally, Silhouette enables the background MPU regions to make the peripheral, device, and system regions non-executable. ARMv7-M always rejects unprivileged write to the system region, so no special configuration is needed to protect the system region.

2.3 AWS FreeRTOS

Amazon AWS FreeRTOS [2] is a popular open-source real-time operating system that supports a variety of embedded architectures including ARMv7-M, ARMv7-R, MIPS and RISC-

V [4]. The STM32L475 Discovery board [36] is one of the officially supported devices for AWS FreeRTOS. At its core, AWS FreeRTOS uses the FreeRTOS kernel [3], another open-source project. FreeRTOS kernel provides core functionalities of the OS, and AWS FreeRTOS provides additional features. The rest of this section focuses on the ARMv7-M version of AWS FreeRTOS with default configurations for STM32L475 Discovery board, which may be different from other versions or devices.

In FreeRTOS, applications run as tasks. A task in FreeRTOS is conceptually similar to a process in desktop OS. When creating a new task, the kernel creates a Task Control Block (TCB), which is similar to a process control block, and add the TCB to the ready list for the scheduler. A TCB contains important data for the task including the stack pointer, MPU configuration, the task's execution priority, notification bits and notification state. By default, the kernel stores the TCB in the heap. FreeRTOS provides one heap for both application tasks and the kernel.

By default, application tasks and the kernel both execute in privileged mode with the MPU disabled. It is a common practice to run all code in privileged mode in embedded world for better performance and less programming complexity [17, 44]. AWS FreeRTOS assumes that everything runs in privileged mode, but the FreeRTOS kernel provides an option and a set of kernel functions to support using different privilege modes and the MPU. However, regardless of the privilege mode or the MPU configuration, FreeRTOS does not fully isolate application memory and kernel memory. For example, FreeRTOS uses the one and only heap for both application tasks and the kernel. When using dynamic allocations to create an application task, FreeRTOS stores the TCB on the heap and allocates the task stack on the same heap. All dynamic allocations in application tasks are also using the same heap.

While FreeRTOS is an operating system, it has far less features than a desktop operating system. Most notably, both the OS and applications tasks are compiled at the same time into one binary executable file. The FreeRTOS kernel only provides the scheduler, task management API, kernel exception handlers, a heap, queue API, semaphores, timer API, and

a stream buffer API for communication between an exception handler and a task. The AWS FreeRTOS adds device-specific initialization, a logging task, Wi-Fi support, and networking protocols to communicate with Amazon Web Service.

Chapter 3

Design

3.1 Threat Model

Our threat model assumes that an attacker can exploit memory errors in code of any task running on the system and the kernel, except for a small trusted portion, to write malicious data to any memory location, with the goal of maliciously changing the control data, which includes the return address, the processor state and the stack pointer, of any application or the operating system and therefore hijacking the control flow. The trusted portion of the kernel should be verified to be free of memory errors.

3.2 Overview of Kage

Kage combines the Silhouette compiler and runtime system [44] and KageOS to protect the control-flow of application tasks and the OS. KageOS is a real-time operating system based on Amazon AWS FreeRTOS [2].

By default, AWS FreeRTOS executes both tasks and system functions in privileged mode. KageOS keeps this default configuration, as Silhouette allows unprivileged code to execute in privileged mode while protecting the control flow of the system. Also, AWS FreeRTOS is mainly designed for single-processor devices. The FreeRTOS kernel does not support scheduling for multiprocessor devices, so on these devices, each processor needs to run an

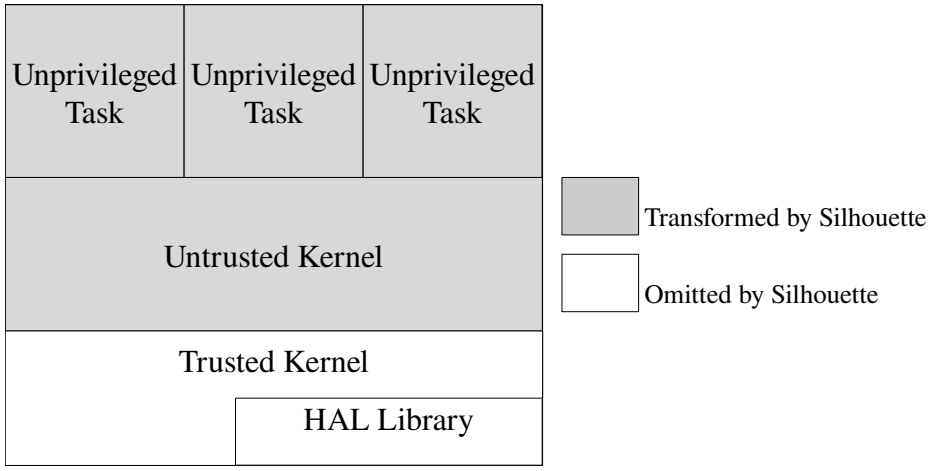

Figure 3.1: Architecture of KageOS

instance of FreeRTOS [6]. KageOS inherits this limitation of FreeRTOS and focus on single-processor devices. KageOS requires the device to support a MPU with at least eight memory regions.

Figure 3.1 shows the architecture of KageOS. Kage uses Silhouette to transform tasks and the untrusted kernel, meaning that tasks and the untrusted kernel save return addresses on the parallel shadow stack, use unprivileged store instructions in function bodies, contain CFI [8] labels in the beginning of each basic block, and perform CFI checks before indirect branches or indirect jumps.

3.3 Overview of KageOS Kernel

The kernel of KageOS consists of two isolated sections: a trusted portion that needs privileged permission to access protected memory regions and system registers, and the rest of the kernel that does not need such permission. Kage uses Silhouette to transform the untrusted kernel. The trusted kernel includes device-specific functions to access system registers, kernel exception handlers, and the task management module, which contains the scheduler code and functions that access the task control blocks, which contain critical data of each task including the task stack pointer and task-specific MPU configuration.

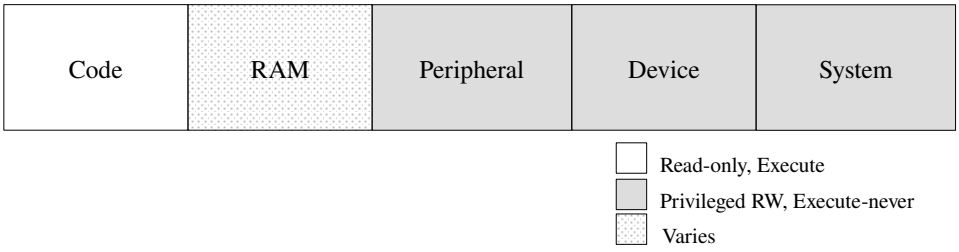

Figure 3.2: Address space of KageOS

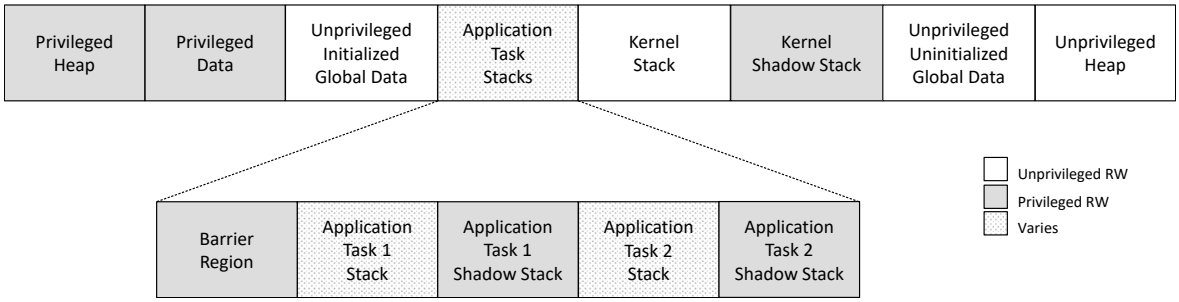

Figure 3.3: RAM layout of KageOS

3.4 Memory Isolation

In order to protect control data and data of the trusted kernel, KageOS needs to isolate data that only the trusted kernel can access and data that the untrusted kernel and tasks can access. KageOS separates privileged and unprivileged data into different memory regions and utilizes the MPU to protect each memory region. Figure 3.2 shows the access permissions of KageOS's overall address space. Figure 3.3 shows the layout and access permissions of KageOS's RAM.

3.4.1 Memory Regions

KageOS divides the RAM into eight memory regions.

The Privileged Data region, which is only writable in privileged mode, contains critical data for the trusted kernel including data structures to control the scheduler and the current state of tasks. The Unprivileged Initialized Global Data region and the Unprivileged Uninitialized Global Data region contain other global data for trusted kernel, untrusted kernel, and application tasks. These two regions do not contain any control data.

Similarly, KageOS has two heap regions. The trusted kernel stores dynamically allocated data in the Privileged Heap region. Notably, the Privileged Heap contains the task control block of all tasks. As Chapter 3.3 states, the task control block contains critical data of each task including the stack pointer and MPU configurations. Dynamically allocated data of the untrusted kernel and application tasks is stored in the Unprivileged Heap region. Similar to AWS FreeRTOS, all tasks and the untrusted kernel share the same unprivileged heap. Since dynamically allocated data of tasks and the untrusted kernel contains no control data or critical data, there is no need to create a heap for each task and the untrusted kernel.

The Application Task Stacks region contains the stack and the shadow stack of each task. The stack of the current foreground task is writable in unprivileged mode; the shadow stack, as well as all stacks and shadow stacks of other tasks, are writable in privileged mode only. Therefore, the current task cannot write to any shadow stack or the stacks of other tasks. Since KageOS uses the same parallel shadow stack as Silhouette, the shadow stack pointer is calculated from the stack pointer, so a stack overflow could cause the shadow stack to overflow into unprivileged stack region. However, as Figure 3.3 shows, the memory region before any task stack region is always either the shadow stack of the previous task or the special Barrier Region, whose sole purpose is to prevent stack overflow. Both these two kinds of regions are always not writable in unprivileged mode. Therefore, overflow of task stack is not possible in KageOS.

Finally, the kernel stack and the kernel shadow stack are in separate memory regions. In KageOS, the system initialization code and exception handlers use the kernel stack. Since a portion of the initialization code is untrusted, and since KageOS allows developers to call untrusted functions in exception handlers (See Chapter 3.8 for more details), there needs to be a kernel shadow stack, and the kernel stack needs to be writable in unprivileged permission. Every time when the system switches from the kernel stack to the task stack, KageOS resets the kernel stack pointer to the initial address, ensuring that no data is left on the kernel stack.

3.4.2 MPU Configuration

In addition to the access permissions of each RAM region, KageOS also configures the MPU to protect the Code, Peripheral, Device, and System regions in the address space. The MPU configuration of these regions are identical to that of Silhouette [44]. KageOS sets the Code region to be read-only and Peripheral and Device regions to be writable in privileged permission only. KageOS also enables the processor's default background MPU configuration to set Peripheral, Device, and System regions to be execute-never. Finally, as Chapter 2.1.3 explains, the System region is always writable in privileged permission only regardless of MPU configuration.

3.5 Secure API

The trusted kernel provides a secure API for untrusted kernel and application tasks to use. Table 3.1 lists the secure API for task management available to both application tasks and the untrusted kernel, excluding untrusted exception handlers. Table 3.2 lists the secure API only available to the untrusted kernel, also excluding untrusted exception handlers. Finally, Table 3.3 lists the secure API only available to untrusted exception handlers. Table 3.4 explains the data types of the arguments and returns.

To enforce Kage's security policy, KageOS performs runtime checks in the secure API to ensure that the arguments cannot cause the secure API or functions of the trusted kernel called by the secure API to corrupt any control data.

KageOS limits `xTaskCreateRestricted` such that only the system initialization code can create new tasks. Without any restriction, this API can be dangerous because an attacker can corrupt the argument to change the task stack to overlap with a current task and corrupt the stack and the shadow stack of the task. The system initialization code includes trusted code that initializes the hardware, trusted code that initializes and starts the scheduler, untrusted code that creates the system timer task, and untrusted code that the system timer task calls exactly once that creates all application tasks. The system initialization code only executes

Name(argument types): return type	Description
xTaskCreateRestricted(TaskParameters_t*, TaskHandle_t*): BaseType_t	Create a task with given parameters. Write the task control block of the task created to second argument. Return true if the task is successfully created, false otherwise.
vTaskDelete(TaskHandle_t): void	Delete given task.
vTaskDelayUntil(TickType_t*, TickType_t): void	Delay current foreground task for given ticks relative to the first argument and write the wake up tick to it.
vTaskDelay(TickType_t): void	Delay current foreground task for given ticks relative to the tick this function is called.
vTaskPrioritySet(TaskHandle_t, UBaseType_t): void	Set the priority of given task to given value.
vTaskSuspend(TaskHandle_t): void	Suspend given task.
vTaskResume(TaskHandle_t): void	Resume given task. Do nothing if the task is not suspended.
vTaskAllocateMPURegions(TaskHandle_t, MemoryRegion_t*): void	Change the MPU configuration of given task to given memory region configuration.
ulTaskNotifyTake(BaseType_t, TickType_t): uint32_t	Block current foreground task and use notification as semaphore until given ticks. Set notification value to 0 if first argument is true, decrement value if false. Return notification value before it changes. .
xTaskNotifyWait(uint32_t, uint32_t, uint32_t*, TickType_t): BaseType_t	Block current foreground task until notified or for given ticks. First argument is notification value bits to clear before block; second argument is bits to clear after receiving notification. Write notification value bits received to third argument. Return true if notification is received, false on timeout.
xTaskGenericNotify(TaskHandle_t, uint32_t, eNotifyAction, uint32_t*): BaseType_t	Unblock given task and optionally update its notification value bits. Second and third arguments specify how to update given task's notification value bits. Write the old notification value bits to fourth argument. Return true if notification bits are updated successfully, false otherwise.
xTaskNotifyStateClear(TaskHandle_t): BaseType_t	Clear notification state of given task without clearing notification value bits. Return true if this function changes the notification state; return false if no action is needed.

Table 3.1: Task management secure API for application tasks and untrusted kernel

Name(argument types): return type	Description
vTaskMissedYield(void): void	Request for a context switch.
xTaskPriorityInherit(TaskHandle_t): BaseType_t	Raise the priority of given task to that of the current foreground task.
xTaskPriorityDisinherit(TaskHandle_t): BaseType_t	Reset the priority of given task to its original value.
xTaskPriorityDisinheritAfterTimeout(TaskHandle_t, UBaseType_t): void	Set the priority of given task to given priority value if the task's current priority is lower than the value.
pvTaskIncrementMutexHeldCount(void): TaskHandle_t	Increment the mutex count of current foreground task. Return its task control block.
vTaskSuspendAll(void): void	Stop the scheduler.
xTaskResumeAll(void): BaseType_t	Resume the scheduler. Return true if a context switch is scheduled.
vTaskPlaceOnEventList(List_t*, TickType_t): void	Delay current foreground task for given ticks and add it to the given event waiting list. Store the task priority and sort the list by it.
vTaskPlaceOnEventListRestricted(List_t*,TickType_t, BaseType_t):void	Same as vTaskPlaceOnEventList, but use the third argument to determine whether to delay indefinitely or not.
vTaskPlaceOnUnorderedEventList(List_t*, TickType_t, TickType_t): void	Delay current foreground task for given ticks and add it to the given event waiting list. Store second argument and do not sort the list.
xTaskRemoveFromEventList(List_t*): BaseType_t	Resume the first task in the list and remove the task from the event list. Return true if a context switch is required, false otherwise.
vTaskRemoveFromUnorderedEventList(ListItem_t*, TickType_t): void	Resume the task associated with given list item. Store second argument to the value of list item.

Table 3.2: Task management secure API for untrusted kernel only

Name(argument types): return type	Description
xTaskResumeFromISR(TaskHandle_t): BaseType_t	Resume given task. Do nothing if the task is not suspended. Can only be called in exception handlers. Return true if a context switch is required; return false otherwise.
xTaskGenericNotifyFromISR(TaskHandle_t, uint32_t, eNotifyAction, uint32_t*, BaseType_t*): BaseType_t	Same as xTaskGenericNotify, but write to fifth argument whether a context switch is needed after unblocking given task.
vTaskNotifyGiveFromISR(TaskHandle_t, BaseType_t*): void	Unblock given task, using notification as semaphore. Write the second argument to true if the unblocked task has higher priority than current foreground task. Can only be called in exception handlers. The normal version of this API is defined as a macro of xTaskGenericNotify.

Table 3.3: Task management secure API for untrusted exception handlers only

Data Type	Internal Data Type	Description
`TaskHandle_t`	`tskTaskControlBlock*`	The pointer to a task control block
`TickType_t`	`uint32_t`	Number of ticks
`BaseType_t`	`long`	General data
`UBaseType_t`	`unsigned long`	Unsigned general data
`List_t`	N/A	A struct representing a doubly linked list
`ListItem_t`	N/A	A struct representing a node in `List_t`
`MemoryRegion_t`	N/A	A struct to store a memory region configuration
`TaskParameters_t`	N/A	A struct to store configurations of a task

Table 3.4: Data types of secure API

once, and it executes before malicious party could interfere with the system. KageOS defines a macro for developers to specify the finite number of tasks on the system. KageOS tracks the number of tasks created in total and adds check to `xTaskCreateRestricted` so that additional tasks cannot be created. This number of tasks created is not decremented when deleting a task, so an attacker cannot create a new task even if they manage to maliciously delete a task.

For other task management secure API, KageOS adds runtime checks to functions that takes in pointers in its arguments. Since secure API uses privileged store instructions, a malicious pointer could cause the API to overwrite control-flow data such as return addresses on the shadow stack. Within all pointers in the arguments of secure API, `TaskHandle_t`, which points to a task control block, and `ListItem_t`, which points to a list item struct inside a task control block, are the only types of pointer that points to protected memory; all other pointers point to unprivileged memory with no control-flow data. Therefore, KageOS performs two types of runtime checks.

For `TaskHandle_t`, KageOS verifies if the pointer points to a valid task control block. The trusted kernel maintains a table of pointers to task control block of task created. When `xTaskCreateRestricted` successfully creates a task, `xTaskCreateRestricted` adds a pointer to the task control block to the table. When a task is deleted, KageOS removes the pointer to its task control block right before the task control block is de-allocated. With this task control block table, KageOS could effectively check if a pointer points to a valid task control block. `ListItem_t` requires an additional step. `ListItem_t` contains a pointer to the data

structure it represents. While this pointer is not always a task control block, this is a requirement of `vTaskRemoveFromUnorderedEventList`, the only secure API function that takes in a `ListItem_t` pointer. Therefore, KageOS checks if the owner pointer of the `ListItem_t` points to a valid task control block.

For other types of pointers, KageOS checks if they point to an address within an unprivileged memory region by comparing its address and its size with all privileged memory regions. To perform the check, KageOS needs to know the size of the data. Thankfully, almost all secure API functions have deterministic size of the pointer's data, with only one exception, `vTaskAllocateMPURegions`, which takes in an array of memory region configurations. However, this function does not write any data to this pointer. Therefore, it is not capable of overwriting any control-flow data stored in trusted memory.

The `vTaskAllocateMPURegions` API performs an additional check. As the MPU provides hardware protection to privileged memory regions, it is important to prevent attackers from altering the MPU configuration. Therefore, `vTaskAllocateMPURegions` checks if the new MPU configuration violates any of KageOS's MPU configuration.

The secure API for untrusted exception handlers also needs an additional runtime check. When an untrusted exception handler calls a secure API, the handler must raise its priority such that it cannot be interrupted by another exception with untrusted handler. To enforce this requirement, secure API available to untrusted exception handlers verifies that the current exception priority is set to the highest configurable priority.

For all runtime checks, KageOS executes a pre-configured failing routine if a check fails. By default, the routine is an infinite loop.

3.6 The Untrusted Kernel

KageOS considers all kernel modules that do not directly modify tasks, the scheduler, or access the peripheral or system regions as untrusted and applies Silhouette [44] to make them unprivileged. Untrusting a large portion of the kernel means that only a small part of the kernel needs to be verified as memory error-free. The data these modules use are stored in

their corresponding unprivileged memory regions. Namely, the untrusted kernel modules use the unprivileged heap for dynamic memory allocations. Different from FreeRTOS, KageOS stores the event list item of each task outside of its task control block. Instead, the event list items of all tasks are stored separately in the unprivileged uninitialized global data region because they are not control data. The event list item data structure of each task includes data such as a pointer to the task control block and a 32-bit value. Untrusted kernel modules such as the queue use this data structure to maintain a list of tasks waiting on data or resources. Other data of kernel modules that are now untrusted is stored either in the unprivileged initialized data region or in the unprivileged uninitialized data region. This configuration reduces the number of secure API required and the size of the trusted data.

The untrusted kernel also includes an unprivileged version of the C library. Since the trusted kernel, the untrusted kernel, and application tasks all use the C library, KageOS needs to provide two sets of the C library such that the trusted kernel calls the default C library with regular store instructions, and the untrusted kernel and tasks call the unprivileged C library with unprivileged store instructions. The untrusted kernel and tasks should not call the default C library functions because the default C library functions such as `memset` could overwrite protected data if the pointer argument is corrupted and points to a privileged memory region. Duplicating the C library inevitably inflates the code size. The exact amount of code size increase depends on the number of C libraries used by tasks because when building the binary, the linker only links C library functions that are used.

3.7 Protecting Processor State

In an operating system with context switching and exception handling support, the kernel needs to store the processor state to memory during context switch or when an exception occurs [15]. The processor state contains critical control-flow data including the current program counter and the stack pointer. The processor state needs to be protected in order to prevent control-flow hijacking.

The processor state KageOS protects includes all general purpose registers, the `LR` link

register, the program status register, the `CONTROL` register, the stack pointer, and all floating point registers if the processor supports floating point, which is optional on ARMv7-M architecture [12]. For exceptions, KageOS also protects the exception return address.

KageOS stores processor state in privileged memory regions during context switching and exception handling. ARMv7-M provides a `PendSV` interrupt to efficiently switch context [12]. As Chapter 2.1.6 explains, the processor automatically saves a subset of the processor state on exception entry. During a context switch, the `PendSV` handler of KageOS first copies the registers that are automatically spilled by the processor from the task's stack to the task's shadow stack. Then, the handler stores the rest of the processor state to the task's shadow stack and calls the scheduler function to decide the next task to run.

The scheduler function from FreeRTOS [3]also checks for stack overflow. This stack overflow check is necessary because in a situation where the task first decrements the stack pointer and then store data to the stack, a context switch may occur between these instructions. As Chapter 2.1.6 states, the processor automatically pushes a subset of registers to the current stack. If decrementing the stack pointer causes stack overflow, the automatic register spilling and the trusted `PendSV` handler may corrupt the memory region before the task stack region, which could be a task shadow stack, without violating the MPU configuration.

After the scheduler function returns, the handler loads the processor state of the next task that are not automatically saved by the processor from its shadow stack. Finally, the handler copies the other processor state from the task's shadow stack to its stack and returns. No untrusted code could execute during `PendSV` handler's execution (See Chapter 3.8), so the processor state restored to the task stack could not be corrupted before the handler returns.

3.8 Exception Handling

The kernel exception handlers, which include the handler of `SysTick`, `PendSV`, `SVCall`, `MemManage`, `BusFault` and `HardFault`, are in the trusted kernel because they either need to access privileged memory regions or only execute when an error occurs.

KageOS allows developers to configure hardware-specific exception handlers and the `UsageFault` handler to execute functions in the untrusted kernel in order to avoid adding code to the trusted kernel unnecessarily. However, these exceptions must have lower exception priority than any of the exceptions whose handler is in the trusted kernel. Currently, Kage relies on the developer to correctly configure exception priority. Since trusted code in exception handlers use the unprivileged kernel stack to store local variables, this restriction prevents untrusted code from corrupting data of trusted kernel saved on the kernel stack. Moreover, because this restriction prevents untrusted code to preempt exception handlers in the trusted kernel, these trusted exception handlers do not need to spill processor state on the shadow stack: the system will not run any untrusted code when a trusted exception handler executes, so processor state saved on the regular stack cannot be corrupted. By the time that untrusted code can execute, the trusted exception handler has already returned and popped the saved processor state.

KageOS's exception handling mechanism needs to prevent the kernel stack from overflow, with similar reasons that the scheduler function in context switching performs stack overflow check. While the MPU policy of the task shadow stack before the kernel stack prevents untrusted exception handlers to write to an overflowed stack, an untrusted exception handler may decrement the stack pointer first and then write to it. If a trusted exception handler or an exception dispatcher preempts the current handler after the stack pointer is decremented but before the store instruction, then ARMv7-M's automatic register spilling mechanism on exception entry could overwrite protected data in the task shadow stack region.

As a task can call secure API to execute functions in trusted kernel, the secure API and trusted kernel function it calls use the task stack to store local variables. This behavior can also cause a potential security risk: when the task calls a secure API and is executing code in trusted kernel, an exception whose handler is untrusted may occur. In this situation, the task stack is still writable in unprivileged mode, so the untrusted handler could corrupt local variables of the secure API on the task stack.

To mitigate these two issues, KageOS disables privileged write access to task stacks and

task shadow stacks when entering an untrusted exception handler. Since all exception handlers use the kernel stack, disabling write access to task stacks and task shadow stacks does not affect the execution of any exception handler, trusted or untrusted. After the untrusted handler finishes, KageOS restores the MPU to the original configuration. In the case of nested exceptions with untrusted or trusted exception handlers preempting an untrusted exception handler, KageOS restores the MPU to the original configuration after all exception handlers finish.

Beside a task, an untrusted exception handler can also call secure API. In this case, the untrusted exception handler must raise its priority to prevent other untrusted exception handlers from preemption.

To protect the processor state when executing an untrusted exception handler and to perform the above MPU re-configuration, KageOS adds a trusted "dispatcher" function to each untrusted exception handlers. When an exception whose handler is untrusted occurs, KageOS would call this dispatcher function first instead of directly calling the untrusted handler. The dispatcher function saves all processor state to the shadow stack and configures the MPU such that the entire task stack and task shadow stack region is read-only in both privileged and unprivileged mode. Then, the dispatcher calls the untrusted exception handler. After the handler returns, the dispatcher restores the processor state. If this exception does not preempt any other untrusted exception, the dispatcher restores the MPU configuration. When saving and restoring the processor state, the dispatcher temporarily sets its priority to the maximum configurable priority, preventing other untrusted exception handlers from preempting it.

Chapter 4

Implementation

Our implementation of Kage includes two main components: the Silhouette compiler system and the KageOS real-time OS. We modified the original prototype of Silhouette [44], and we built KageOS by modifying and extending Amazon AWS FreeRTOS v1.4.9 [2]. We choose to target STM32L475 Discovery board [36] for this prototype instead of the more powerful STM32F469 Discovery board [37] used in Silhouette project because the STM32L475 board is officially supported by AWS FreeRTOS, and the STM32F469 board is not [4]. We modified less than 20 lines of code in Silhouette implementation. We measured the line count of KageOS and an unmodified AWS FreeRTOS v1.4.9 using SLOCCount [42]. KageOS adds 1129 lines of code to AWS FreeRTOS.

4.1 System Assumptions

We make two assumptions on our implementation. First, identical to the original Silhouette [44], we assume that the Hardware Abstraction Layer (HAL), provided by the hardware manufacturer, is part of the trusted computing base and is not transformed by Silhouette's compiler passes. Then, we assume that the hardware includes a memory protection unit (MPU), as Silhouette relies on the MPU to protect shadow stacks and other security-critical memory regions.

```
1  str.w   lr, [sp, #4092] // Save LR to mem[sp + 4092]
```

Listing 4.1: Instructions to Spill Return Address

```
1  add     sp, #4          // Restore the stack pointer
2  ldr.w   pc, [sp, #4092] // Lode PC from mem[sp + 4092]
```

Listing 4.2: Instructions to Restore Return Address

4.2 Changes of Silhouette

Because of the difference between bare-metal applications and applications running in KageOS, we need to modify the implementation of Silhouette in order to use it as a component of Kage.

Our prototype of Kage restricts the stack size of each task and the kernel stack to be 4KB, due to extremely limited 128KB of RAM on the STM32L475 Discovery board. Since ARMv7-M's store immediate and load immediate instructions support an immediate offset up to 4KB, Silhouette's shadow stack transformation pass no longer need to insert the shadow stack offset to the `IP` register before storing to the shadow stack or loading from the shadow stack. Silhouette now inserts only one instruction in function prologue before saving callee-saved registers to the regular stack. Listing 4.1 shows the instruction added in function prologue. Since Silhouette keeps the original store instruction of `LR`, Silhouette does not decrement the stack pointer after the store instruction. Silhouette inserts two instructions in function epilogue after restoring other callee-saved registers. Listing 4.2 shows the instructions added in function epilogue. As Silhouette removes the original load instruction of return address, Silhouette needs to increment the stack pointer.

As Chapter 3.5 explains, part of the kernel is trusted and should not be transformed by Silhouette. FreeRTOS provides an optional `privileged_functions` section in the Code region to store privileged kernel functions. KageOS uses this section to store all trusted kernel functions. Silhouette detects this attribute and skips the function if this attribute exists.

4.3 KageOS

As Chapter 2.3 explains, AWS FreeRTOS executes both application code and the kernel in privileged mode and disables the MPU by default. First, we configured FreeRTOS to use the MPU. Since Silhouette allows all code to execute in privileged mode while providing protections, KageOS keeps the AWS FreeRTOS's default configuration of running everything in privileged mode. We modified FreeRTOS such that the prototype matches the design in Chapter 3.

4.3.1 Un-trusting the Kernel

The FreeRTOS kernel annotates privileged kernel functions with `privileged_functions` attribute. In KageOS, only the scheduler, task management module, the kernel list module, the trusted dynamic allocation and de-allocation module, and the device-specific support module (including exception handlers) are trusted. We removed the `privileged_functions` attribute from all other kernel functions. These functions include the queue, stream buffer, event groups, and the timer modules.

Both trusted and untrusted kernel use the kernel list module to access list data structure. KageOS includes another copy of list module for untrusted kernel such that trusted kernel can call the trusted list module to access protected memory while the untrusted kernel uses the untrusted list module. Similarly, KageOS provides two dynamic allocation and deallocation modules. The trusted kernel calls the trusted version, which allocates memory from the Privileged Heap memory region, and untrusted kernel and application tasks call the untrusted version, which allocates memory from the Unprivileged Heap region.

For C library, KageOS includes the default pre-compiled C library, but only the trusted kernel uses it. For the untrusted kernel and tasks, KageOS includes another C library from `Newlib` [1], a C library designed for embedded systems. In our prototype of KageOS, we only imported functions that the untrusted kernel needs.

The FreeRTOS kernel uses another attribute, `privileged_data`, to annotate privileged global variables. In KageOS, only the scheduler data and task management data have this

attribute. All other data in the kernel are now untrusted, so we removed this attribute from these variables.

AWS FreeRTOS assumes that everything runs in privileged mode, and kernel functions added by Amazon AWS do not have the privileged attribute. This means that Silhouette transforms them by default, and they are untrusted.

In KageOS, there are 66 functions in the trusted kernel (excluding pre-compiled library functions), 1042 functions in the untrusted kernel, and 834 functions in the HAL library [35].

4.3.2 Isolating Trusted and Untrusted Memory

As Chapter 3.4.1 explains, KageOS provides a privileged heap for the trusted kernel and an unprivileged heap for untrusted kernel and application code. FreeRTOS only provides one heap and uses the same heap for all dynamically allocated data. We added another set of heap API for unprivileged data and replaced all function calls to the heap API in untrusted code with the unprivileged heap API.

KageOS configures the MPU as Figure 3.3 and Chapter 3.4.2 describes. Table 4.1 shows the detailed MPU configuration of KageOS prototype. The STM32L475 Discovery board has two disconnected RAM regions. The first region, RAM2, is 32KB, and the other region, RAM, is 96KB. ARMv7-M supports overlapping MPU regions [12]. Therefore, KageOS configures the entire RAM region as unprivileged Read/Write first and then configures the privileged regions as privileged only Read/Write. The only difference between the implementation of our prototype and the design is that the implementation does not include the barrier region before the stack of the first task. Since the application task stacks region is at the beginning of RAM hardware region, and as Figure 2.1 shows, a stack overflow of the first task will cause the system to write to the unmapped region between RAM2 and RAM. This will trigger a `BusFault` and will fail. The number of configurable MPU regions varies by hardware. The STM32L475 Discovery board only allows eight MPU regions, and KageOS uses all eight regions to protect all security-critical regions. Therefore, the `vTaskAllocateMPURegions` secure API, which allows an application task to change its MPU configuration, will not

Hardware Region	MPU Region	Access
FLASH	Code	Privileged RO, Unprivileged RO
RAM2	Privileged heap and privileged data	Privileged RW, Unprivileged RO, XN
RAM2	Unprivileged initialized global data	Privileged RW, Unprivileged RW, XN
RAM	Entire RAM region	Privileged RW, Unprivileged RW, XN
RAM	Application task stacks	Privileged RW, Unprivileged RO, XN
RAM	Kernel shadow stack	Privileged RW, Unprivileged RO, XN
RAM	Stack of current foreground task	Privileged RW, Unprivileged RW, XN
Peripheral	Peripheral	Privileged RW, Unprivileged RO, XN

Table 4.1: MPU configurations

RW = Read/Write, RO = Read-only, XN = Execute-Never

succeed in any condition in our prototype.

4.3.3 Secure API

The trusted kernel provides a set of secure API for untrusted kernel and application tasks. We added `xVerifyTCB` and `vVerifyUntrustedData` API to check TCB pointer and other types of pointer, respectively. Each secure API function that takes one or more pointer arguments calls these two functions in the beginning of the function to verify the pointers. We added `ulPortGetBASEPRI` API to check the current exception priority. Secure API for untrusted exception handlers calls this function to verify that the current exception priority is set to the highest configurable priority.

For `vTaskAllocateMPURegions`, which takes in a new set of MPU configuration, we added checks in the beginning of the function to ensure that the new MPU configuration does not violate the MPU policy of KageOS. However, since our prototype uses all eight MPU regions that our board supports, this API function can never succeed in the prototype.

For all runtime checks, KageOS uses FreeRTOS's `configASSERT(cond)` macro to check conditions. If `cond` is false, then KageOS executes the failing routine defined in the `configASSERT` macro. By default, the failing routine is an infinite loop.

4.3.4 Context Switching

ARMv7-M uses an interrupt, PendSV, to handle context switching. The PendSV handler

in FreeRTOS saves the processor state of current foreground task its task stack, calls the scheduler function to select the next task with highest priority from the list of ready tasks, and restores the processor state of the next task from its task stack. KageOS stores processor state to the task's shadow stack instead of the its regular stack.

While the scheduler function in FreeRTOS contains stack overflow check, it only checks whether the last 16 bytes on the stack is overwritten. KageOS has an additional condition to the check: it checks whether the task stack pointer is smaller than the beginning of the task stack.

4.3.5 Exception Handling

Our prototype only contains exception handlers required for the system to operate, which are the kernel exception handlers in the trusted kernel. As trusted exception handlers do not back up the processor state to the shadow stack, the exception handlers from FreeRTOS work as intended without modification.

To evaluate the overhead of untrusted exception handlers, we implemented a dummy exception handler and its dispatcher. The exception handler itself does not contain any code, and this exception is not declared in the vector table. However, the dispatcher is fully functional. The dispatcher first sets its priority to the highest configurable priority to prevent other untrusted exception handlers from preempting it. Then, the dispatcher copies registers that are saved by the processor to either the task shadow stack or the kernel shadow stack, depending on where these registers are spilled, and it spills other registers, the current value of the `control` register, the `lr` register that contains data on how this exception returns, and the current active stack pointer to the kernel shadow stack. After spilling all processor state, the dispatcher configures the MPU to set the entire task stack and shadow stack region to read-only in both privileged and unprivileged mode. Then, the dispatcher restores the priority of the exception and calls the exception handler. After the exception handler returns, the dispatcher sets the priority to the highest configurable priority again. Then, the dispatcher checks if it is in a set of nested exceptions, preempting another exception. If not,

the dispatcher restores the MPU configuration. It then restores all processor state it saves that are not automatically saved by the processor from the kernel shadow stack. Finally, the dispatcher copies the registers that are saved by the processor from the corresponding shadow stack to the stack, restores the exception priority, and returns.

4.4 Limitations of Implementation

There are a number of limitations in our current implementation of Kage.

First, we have not made changes to Silhouette's privielged code scanner. As the trusted kernel of KageOS needs to use privileged instructions for context switch and exception handling, the privileged code scanner needs to be modified to allow trusted kernel functions to contain the pivileged instruction. The privileged code scanner should also check the usage of secure API in tasks to ensure that tasks only call secure API functions they are allowed to call. Since the privileged code scanner is not required for evaluation and does not affect evaluation results, we currently disable it in Kage.

Second, our current implementation keeps the parallel shadow stack [21] design of Silhouette. While parallel shadow stack works well in bare-metal applications on embedded systems, it causes much higher RAM usage in an embedded OS as it requires double the stack memory of each task. Also, our implementation of the shadow stack requires all tasks to have stacks with the same size in order to properly access the parallel shadow stack.

Finally, our current implementation only imports required untrusted C library functions for the untrusted kernel. Since KageOS needs both the default C library for the trusted kernel and an untrusted C library for the untrusted kernel and tasks, we would need to rename untrusted C library functions such that they can exist with their trusted counterparts.

Chapter 5

Security Discussion

In this chapter, we discuss Kage's security guarantees and how Kage enforces them.

5.1 Overview

We list the security guarantees of Kage:

1. On function return, the return instruction will always branch to its legal return address saved during function prologue.

2. On context switch, the processor state of the upcoming foreground task will always be the same values they were immediately before the task is previously switched out. When a task first starts executing, its initial processor state, including the program counter and the `control` register, will always be the initial values defined in task initialization.

3. On exception handling return, the processor state will always be the values immediately before the exception occurs.

4. Only the trusted kernel code can access task management data and scheduler data, including task control blocks, which contain control data, and scheduler's ready and pending list, which contain pointers to task control blocks.

Kage inherits the first guarantee from Silhouette [44]. Silhouette enforces this security guarantee with its protected shadow stack and forward-edge CFI checks. The same CFI check mechanism applies to application code and untrusted kernel code in KageOS. The protected shadow stack guarantees that function prologue code saves the return address of the caller function on the shadow stack, function epilogue code loads the return address from the shadow stack, and the shadow stack cannot be corrupted. The CFI checks ensures that a function cannot jump to the middle of a function, corrupting the stack pointer with unmatched stack pushing or popping. The second and third guarantees both require protecting processor state when it is spilled to the memory. The last guarantee requires task managent and scheduler data to be protected.

In this chapter, we first discuss how Kage stores all of the above security-critical data in a protected memory region. Then, we discuss how Kage protects the protected memory regions from corruption.

5.2 Storing Security-Critical Data

Kage ensures that scheduler data, task management data and control data are saved in protected memory regions through a combination of memory configuration, compile-time transformation, and spilling at runtime. Chapter 3.4 explains how Kage prevents untrusted code to change data saved in protected memory regions. To prevent the trusted kernel from unintended behaviors, Kage also ensures that the local and global data of the secure API cannot be corrupted by untrusted code.

5.2.1 Scheduler Data and Task Management Data

Kage protects all scheduler-related data and task management data. Scheduler-related data includes scheduler lists (ready lists, the delay list, the suspend list, and the pending list) that consist of pointers to task control block of tasks in different status, the pointer of the current running task, and the status of the scheduler. Task management data includes all task control blocks, and the list of pointers to initialized tasks.

Kage needs to protect these scheduler data and task management data. The task control block contains security-critical data including the stack pointer of the task and the task-specific MPU configuration. The runtime checks of secure API that takes an argument of a task control block pointer uses the list of pointers to initialized tasks to verify the legality of the argument. The scheduler lists include pointers to task control blocks that will be executed. Without protecting the scheduler lists, an attacker could corrupt the ready list and point the pointer to garbage data such that the scheduler would switch context to execute arbitrary code pointed by the garbage data.

KageOS stores the task control blocks in the Privileged Heap region. When creating a new task, KageOS allocates the memory for the task control block dynamically on the privileged heap. For other scheduler and task management data, KageOS stores them in the Privileged Data memory region by statically allocating these data in this region at compile time.

5.2.2 Secure API Data

Kage protects the data of secure API from corruption. As part of the trusted kernel, secure API uses the Privileged Data memory region for global data and the Privileged Heap region for dynamically allocated data. However, since application tasks and the untrusted kernel can call secure API, secure API uses either the task stack or the kernel stack for local data, meaning that the local data could be corrupted by untrusted code. Kage takes several measures to prevent this situation.

First, all functions that a secure API calls are either functions from the trusted kernel or untrusted kernel functions that do not store any data to the memory. These untrusted kernel functions simply return data requested by its caller and do not contain any store instruction other than its prologue or epilogue. An attacker cannot use these untrusted functions to corrupt the stack data of a secure API.

Second, during context switch, KageOS disables unprivileged write permission of the stack of all tasks other than the upcoming foreground task. This ensures that when a context switch occurs when a task calls a secure API, the secure API's local data cannot be corrupted by

the other tasks.

Third, when an exception with untrusted handler occurs when a task calls a secure API, the handler's dispatcher configures the MPU to temporarily disable unprivileged write access of the task stack. This prevents an untrusted exception handler from corrupting the stack data of a secure API called by a task.

Finally, untrusted exception handlers must raise their priority before calling a secure API such that the exception cannot be preempted by another exception with untrusted handler. This prevents an untrusted exception handler from corrupting the stack data of a secure API called by an untrusted exception handler.

5.2.3 Control Data

Kage ensures that all control data is saved in protected memory regions. This includes the return address and processor state for both applications and the kernel.

Kage uses Silhouette's shadow stack transformation [44] to store return addresses to a shadow stack. For application code, return addresses are stored in the shadow stack of its task. For kernel code called by a task, return addresses are stored in the shadow stack of the calling task. For trusted kernel code in system initialization and exception handling, return addresses are stored in the kernel shadow stack region. Silhouette's shadow stack transformation loads the return address from the corresponding shadow stack in the function epilogue to ensure that the function returns to its correct return address.

In both context switching and exception handling, processor state is saved on one of the shadow stacks, with the exception of the task stack pointer, which is in the task control block. The kernel stack pointer is never saved to the memory: when the kernel transfers control back to a task, it resets the kernel stack pointer to the initial value. For the rest of this section, processor state refers to the processor state beside the stack pointer.

When initializing a task, KageOS initializes the processor state values on the task's shadow stack and initializes the stack pointer. During context switch, KageOS's `PendSV` handler saves the processor state of current foreground task to the task's shadow stack, saves the

stack pointer to the task control block, loads the stack pointer of the next task from its task control block, and loads other processor state of the next task from the task's shadow stack.

For exception handling, all untrusted exception handlers use their dispatcher to spill the processor state to a shadow stack. Each untrusted exception handler has a corresponding dispatcher. The dispatcher saves processor-spilled registers to either the task's shadow stack or the kernel shadow stack, depending on the execution mode before the exception occurs. The dispatcher saves the other processor state, along with the task stack pointer, to the kernel shadow stack. To prevent corruption of the processor-spilled registers when they are on the stack, the dispatcher raises its priority to prevent other untrusted exception handlers from preempting it when saving or restoring the processor state.

Trusted exception handlers do not spill the processor state to a shadow stack because when a trusted exception occurs, no untrusted code could execute until the exception returns: the priority of any exception whose handler is trusted is higher than the priority of any exception whose handler is untrusted, and all functions called by trusted exception handlers are trusted.

5.3 Securing Protected Memory Regions

With critical data in the protected memory regions, Kage ensures that an attacker cannot overwrite data in protected memory regions. First, Kage configures the MPU to only allow write access in privileged permission. Then, Kage transforms application code and untrusted kernel code to only use unprivileged write instructions, and KageOS performs runtime checks in trusted secure API that unprivileged code can call to ensure that unprivileged code cannot take advantage of the secure API to corrupt protected memory regions or nullify the MPU configuration.

5.3.1 Application Code and Untrusted Kernel Code

Kage uses Silhouette's store hardening pass [44] to transform the application code and the untrusted kernel code. The store hardening pass uses ARMv7-M's unprivileged store instructions, which always checks the unprivileged access permission when writing to the memory

regardless of the current permission mode. Silhouette replaces all store instructions of application code and untrusted kernel code with unprivileged store instructions, except for the store instruction to spill return address to the shadow stack in function prologue. This means that store instructions in the function body cannot overwrite data stored in protected memory regions. However, the privileged write instruction to save the return address could potentially overwrite privileged memory if the current stack pointer is corrupted.

Silhouette guarantees the integrity of stack pointer for bare-metal applications. First, Silhouette ensures that the stack pointer is never spilled to the memory. Second, it prevents stack overflow and stack underflow. Finally, it uses forward-edge CFI checks to prevent indirect call from branching into the middle of a function. Kage directly inherits the last enforcement. With additional mechanisms to prevent stack overflow in context switching (See Chapter 3.7) and exception handling (See Chapter 3.8), Kage persists the second enforcement as well. However, since KageOS supports context switching, KageOS has to save the stack pointer of a task to the memory when it is not running. As Chapter 2.3 states, the stack pointer of each task is saved in the task's task control block. Task control blocks are saved in the Privileged Heap region, which is protected and only writable in privileged permission. Therefore, an attacker cannot use the store instructions in function body to overwrite the value of the stack pointer of any task. The privileged store instruction that saves the return address to the shadow stack saves to an address relative to the stack pointer, so unless the stack pointer is already corrupted, this privileged store instruction cannot be used to alter the stack pointer. The secure API of the trusted kernel performs checks at runtime to prevent writing into incorrect address (See Chapter 5.3.2). Therefore, the attacker cannot overwrite the stack pointer saved in task control blocks. Conbining this property with stack overflow and underflow prevention and CFI checks, Kage guarantees the integrity of stack pointer.

5.3.2 Runtime Checks of Secure API

The trusted kernel of KageOS provides a list of secure API for application code and the untrusted kernel to call. Since the secure API has full privileged permission with regular store

instructions, it could potentially overwrite data in protected memory regions if the arguments are altered by an attacker. KageOS adds runtime checks to secure API for different types of arguments.

For secure API that takes a pointer to task control block, KageOS checks if the pointer is pointing to a valid task control block. For other types of pointer, KageOS checks if it points to an unprivileged memory region. For MPU re-configuration, KageOS checks if the new MPU configuration violates any of KageOS's MPU policies. Finally, for secure API available to exception handlers, KageOS checks if the exception priority is the highest configurable priority value such that during the execution of the secure API, another untrusted exception handler cannot preempt the current exception. If a check fails, KageOS executes a code sequence defined by the developer. By default, the system stops by executing an infinite loop.

Chapter 6

Evaluation

We evaluate the performance and code size overhead of Kage. For performance overhead, we use microbenchmarks to measure the overhead that Kage introduces by untrusting the majority of the RTOS kernel, adding runtime checks to the secure API, and protecting processor state in context switch and exception handling. For code size overhead, we measure the code size of the kernel and evaluate the overhead incurred by KageOS's changes to the kernel and Silhouette's transformations. For baseline, we measure the code size of the same task running on an unmodified AWS FreeRTOS [2], compiled with an unmodified LLVM 9.0 compiler [31], the same version that Silhouette [44] prototype uses. By default, the FreeRTOS for our discovery board disables the MPU, and we use this configuration as the baseline. To better understand Kage's overhead, we also use a configuration of FreeRTOS with the MPU enabled.

We decided to use only microbenchmarks in our evaluation because currently there is no benchmark suite that can represent the performance of the FreeRTOS kernel. Common benchmark suites for embedded systems such as BEEBS and CoreMark [10] are designed such that they can run on bare-metal devices without an OS. They make no use of multitasking, system calls, or exception handling. Therefore, running these benchmark suites in Kage is meaningless beside showing the overhead of Silhouette. Running real-world FreeRTOS applications in Kage as a benchmark is also not feasible. Most commercial applications that

use FreeRTOS are proprietary and are limited to their specific hardware. While few open-source FreeRTOS projects are available [22,23,28], they either target different hardware with features specific to that hardware or require additional peripherals. More importantly, most of these programs are hobby projects or learning projects. They cannot represent real-world programs, and they have almost no documentation. We could not find any open-source FreeRTOS project that requires no special hardware, represents real-world applications, and has sufficient documentation.

We use STM32L475 Discovery board [36,38] to run all experiments. This board contains an ARMv7-M [12] microcontroller capable of running up to 80 MHz with MPU support, 128 KB of SRAM and 1 MB of flash memory. We use the default configuration of AWS FreeRTOS to run at 80 MHz.

For all experiments, we use the `-O3` optimization level to better simulate real-world usages.

6.1 Performance Experiments

We designed microbenchmarks to measure the overhead incurred by Kage. In each microbenchmark written in C, we measure the processor cycles using the KIN1 library [25]; for benchmarks written in assembly, we added code to manually reset the cycle counter and code to read current cycle count.

We designed microbenchmarks to evaluate performance overhead of Kage in four aspects: exception handling, secure API, context switching, and untrusted kernel modules.

For exception handling, our microbenchmark measures the overall overhead of the dispatcher function that copies and spills the processor state to the shadow stack and restore them from the shadow stack. In this microbenchmark, we use the dummy exception handler and its dispatcher as Chapter 4.3.5 describes. As this dummy exception is not declared in the vector table, we added a function call to call the dispatcher from the function that creates application tasks, `vApplicationDaemonTaskStartupHook`. In the beginning of the dispatcher, we added code to simulate an exception by assigning the `lr` register with the exception properties and added code to reset the cycle counter before the dispatcher code. After the

dispatcher code finishes, we added code to restore the value of `lr` to properly return. After the function call to the dispatcher returns, the caller function then prints the cycle count. Since the dummy exception handler itself is simply an empty function, we believe that this experiment can accurately show the performance overhead of the dispatcher.

For secure API, our microbenchmark measures the overhead of all types of runtime checks. There are four types of runtime checks in the secure API of Kage: pointers to TCB, pointers to other data types, MPU configuration, and exception priority for the secure API for untrusted exception handlers. In our prototype, we implemented the first two checks as their own functions, so they are easy to evaluate. We added a new dummy secure API. This API would reset the cycle counter, call the TCB pointer check function with the pointer to the current foreground TCB, read and print the cycle counter, and repeat this procedure for the function that checks other types of pointer. The runtime check for MPU configuration is integrated into the trusted function that applies the MPU configuration. As Chapter 4.3.2 states, our prototype uses all eight MPU regions available on our board, so the secure API that changes the MPU configuration can never succeed. However, this trusted function that applies the MPU configuration is also called when creating a new task. While a task cannot declare its custom MPU regions in our prototype, we configured the arguments when creating the task such that this trusted function would check the MPU region of the task's stack. We added code to reset the cycle counter in this trusted function immediately before the checks, and we added code to read the cycle counter after the checks. For the runtime check of exception priority, we added the microbenchmark in the `SVC` handler. We defined a new `SVC` number, and the `SVC` handler would execute the microbenchmark sequence if the `SVC` number matches. Similar to other runtime checks, we added code to reset the cycle counter before calling the check and code to record the cycle count after the check.

For context switching, our microbenchmark measures the overall overhead of a context switch. This overhead includes overhead of copying and spilling processor state to the shadow stack and restoring processor state from the shadow stack. ARMv7-M uses `PendSV` interrupt to switch context. The `PendSV` handler of Kage is written entirely in assembly, so we had to

add code to manually write to the cycle counter to reset it, in the beginning of the handler. Immediately before the handler returns, we added code to read the cycle counter and store the value to a global variable. We made the same changes on AWS FreeRTOS and compare the differences in cycle counts.

Finally, for untrusted kernel modules, our microbenchmark measures the overhead Silhouette incurs to kernel API that are now untrusted. Specifically, we evaluate the performance of the queue and the stream buffer API [3]. We selected these two modules for evaluation because we can control their performance factors easily: they do not depend on network connectivity, they only require minimal amount of arguments, and they are not triggered by timer events. Both modules provide features to transfer data between different scopes, but they are not the same. The queue API provides a queue that multiple senders and receivers can transfer data; the stream buffer API provides a buffer that exactly one sender can send data to exactly one receiver. In both cases, the sender and the receiver can be either a task or an exception handler, but only the stream buffer can transfer data between tasks executing on different cores on a multi-core system. We programmed our microbenchmark such that it can use either a queue or a stream buffer to transfer some dummy data. We use two tasks as the sender and receiver in both cases. We first created a new queue or stream buffer before creating the sender and receiver tasks, and we added code to measure the amount of cycles to create the queue or the stream buffer. Then, we create the tasks such that the receiver task has higher execution priority than the sender task, ensuring that the receiver task will execute first. When the receiver task starts, it resets the cycle counter and calls `xQueueReceive()` or `xStreamBufferReceive()` to wait for data. Then, as the receiver is blocked waiting for the data, the sender task starts and calls `xQueueSend()` or `xStreamBufferSend()` to send data. The corresponding API will then unblock the receiver task, and since the receiver task has higher priority, the system will perform a context switch back to the receiver task. The receiver task then records the cycle count and prints it.

For application tasks, we believe that Kage incurs similar performance overhead as Silhouette [44], with the addition of the overheads of FreeRTOS features discussed above. In

Microbenchmark	**Time (cycle)**
Exception dispatcher	313
Secure API: task control block	70
Secure API: other pointers	72
Secure API: MPU configuration	258
Secure API: Exception priority	5

Table 6.1: Performance overhead of exception handling and secure API incurred by Kage

Microbenchmark	**FreeRTOS (cycle)**	**FreeRTOS w/ MPU (cycle)**	**Kage (cycle)**
Context switching	197	222	338
Queue: create	573	712	871
Queue: send and receive	2094	2736	3827
Stream buffer: create	653	730	957
Stream buffer: send and receive	2413	2750	3564

Table 6.2: Performance overhead of context switching and untrusted kernel API

summary, Silhouette incurs a geomean of 1.2% performance overhead in CoreMark-Pro benchmark suite [24] and a geomean of 3.6% performance overhead in BEEBS benchmark suite [32].

6.1.1 Experiment Results

Table 6.1 and Table 6.2 summarize the results of the microbenchmarks. Table 6.1 shows performance overhead of new security checks Kage provides that FreeRTOS does not have, so there is no baseline data. Table 6.2 shows performance overhead of mechanisms where Kage differs from FreeRTOS.

The exception handling dispatcher incurs 313 cycles of overhead due to backing up the processor state, changing the MPU configuration, and restoring the processor state. As Chapter 3.8 discusses, only untrusted exception handlers need to use the dispatcher. Trusted exception handlers that execute at a high frequency, such as the `SysTick` handler and the `PendSV` handler, do no have this overhead.

The secure API runtime checks incurs minimal overhead overall. Within the four types of runtime checks, most secure API uses only the runtime check of task control block and the check of other pointers. Nineteen of 26 secure API functions call the runtime check of

task control block. Ten of the secure API functions call the runtime check of other pointers. Within these secure API, eight functions call both the runtime check of task control block and other pointers. Of the 26 secure API functions, nineteen of them contain the runtime check of task control block. Ten of them contain the runtime check of other pointers. Only one secure API function calls the check of MPU configuration. The three secure API functions for untrusted exception handlers contain the runtime check of exception priority. Nine of the secure API functions contain more than one runtime check. Within these nine functions, five contain both runtime checks of task control block and other pointers. One of them contains both runtime checks of task control block and exception priority. Two functions contains the runtime checks of task control block, other pointers, and exception priority. Finally, one function contains the runtime checks of task control block, other pointer, and the MPU configuration, adding up to a total of 397 cycles of overhead in runtime checks.

Context switching incurs high performance overhead. Comparing to FreeRTOS with default configuration, the context switching mechanism of Kage slows down by 71.57%; comparing to FreeRTOS with MPU enabled, the performance overhead is 52.25%. The difference in context switching code between FreeRTOS with MPU and FreeRTOS without MPU is that during a context switch, the kernel would read the MPU configurations of the next task from the task control block and write them to the MPU control registers, taking slightly more cycles. Comparing to FreeRTOS with MPU, the main sources of performance overhead in KageOS's context switching code are additional instructions to backup and restore processor state that are automatically spilled by the hardware. In ARMv7-M, the hardware automatically saves seven general-purpose registers and the exception return address to the stack on exception entry [12]; if the processor supports floating point operations, the hardware further saves sixteen floating-point registers and the `fpscr` register to the stack. As Chapter 3.7 states, KageOS copies this part of the processor state to the shadow stack.

The queue and the stream buffer untrusted API also incur relatively high performance overheads. When creating a new queue, Kage incurs an overhead of 52.01%, comparing to baseline, and 22.33% comparing to FreeRTOS with MPU. In the case of sending and

receiving data in the queue, Kage incurs an overhead of 82.76% comparing to baseline and 39.88% comparing to FreeRTOS with MPU. To better understand the source of the overhead, we ran the queue experiment with KageOS and an unmodified LLVM 9 [31] compiler. This time, creating a queue took 776 cycles, and sending and receiving data took 3559 cycles. This shows that Silhouette is the main source of overhead when creating the queue but not when sending and receiving data. The main sources of overhead when sending and receiving data in the queue are calls to the secure API and context switching. When a task calls `xQueueReceive` to receive data from the queue, since the queue is empty at that time, the function calls a secure API, `vTaskPlaceOnEventList`, to set the status of the task to delayed and waiting. This secure API that contains two of the above runtime checks, the check of task control block and the check of other pointers. Then, the function requests a context switch to wait for the data to appear in the queue. After that, the receiver task calls `xQueueSend` to send data to the queue. This function calls the `xTaskRemoveFromEventList` secure API to remove the receiver task from delayed and waiting status. This secure API also contains the same set of runtime checks as `vTaskPlaceOnEventList`. Finally, the function requests a context switch to resume the receiver task.

The stream buffer untrusted API incurs similar performance overhead with the queue API. When creating a new stream buffer, Kage incurs a performance overhead of 46.55% comparing to baseline, and 31.10% comparing to FreeRTOS with MPU. When sending and receiving data in the stream buffer, Kage incurs 47.70% of performance overhead comparing to baseline and 29.60% comparing to FreeRTOS with MPU. Similar to the queue API, we also ran the stream buffer microbenchmarks with an unmodified LLVM 9 compiler: creating a new stream buffer took 849 cycles, and sending and receving data took 3358 cycles. Another source of performance overhead when creating a new stream buffer is the C library. While the queue creation API does not contain any call to the C library, the stream buffer initializes the buffer using C library function `memset`. We ran the stream buffer creation experiment again with the default `memset` library function instead of the Newlib [1] library function and with an unmodified LLVM 9 compiler, and the execution time of creating a stream buffer was

reduced to 790 cycles, indicating that the `memset` Newlib library function is slower than the function in the default C library. Sending and receiving data in the buffer stream has similar sources of overhead with sending and receiving data in the queue. While the exact secure API is different, `xStreamBufferSend` and `xStreamBufferReceive` both call secure API functions that contain runtime checks, and these two functions also cause context switches.

For real-time operating systems, one of the most important features is predictability [7]. While Kage incurs high performance overhead comparing to FreeRTOS, Kage still persists predictability in all its modules. For example, excluding the impact of code layout, on the same hardware and with the same configuration, the exception dispatcher and context switching should take the same number of cycles accross different runs. While the performance of some of the features varies depending on task configurations, the performance is still overall predictable. For example, the secure API runtime check of task control block depends on the number of tasks created, but with the same number of tasks, this runtime check should always take the same amount of time to run.

6.2 Code Size Experiments

For code size overhead, we use similar methodology to measure code size as Silhouette [44]. We added code to Silhouette's shadow stack pass, the first compiler pass of Silhouette, to measure the code size of the untrusted code before Silhouette's transformations and the code size of the trusted code. We then added code to Silhouette's CFI pass to measure the code size of the untrusted code after Silhouette's transformations. We measure the code size of each function using the `getFunctionCodeSize` API in Silhouette. For baseline and configurations without Silhouette compiler passes, we still compile them with Silhouette, but we only measure the code size data before applying Silhouette's transformations.

The code size evaluation contains four configurations: unmodified AWS FreeRTOS with default configuration, unmodified AWS FreeRTOS with MPU enabled, Kage, and KageOS without Silhouette compiler passes. Our measurement of each configuration contains the RTOS kernel, library functions in the kernel, and the HAL library [35]. Contrary to the code

Function Type	FreeRTOS (bytes)	FreeRTOS w/ MPU (bytes)	Kage (bytes)	KageOS w/o Silhouette (bytes)
Privileged functions	0	27254	19526	19526
HAL library	139460	139460	139460	139460
Other functions	235934	214930	286292	235346
Total	375394	381644	445278	394332

Table 6.3: Code size measurements

size evaluation of Silhouette, we include the code size of the HAL library because the main purpose of our code size evaluation is to evaluate the overall code size of Kage, which requires the HAL library.

We choose to measure the code size during compilation instead of measuring the size of the compiled binary file because while the compiler would compile every function in the source code, the linker would exclude unused functions in the final binary file. As we could not find a capable benchmark suite or real-world program that uses any of the FreeRTOS API, we believe that measuring the code size of the binary file could not accurately show the overhead of Kage.

For application tasks, we believe that Kage incurs the same code size overhead as Silhouette because Kage uses Silhouette to transform application code and does not add additional code to it. In summary, Silhouette [44] incurs a geomean of 21.4% code size overhead in CoreMark-Pro benchmark suite [24] and a geomean of 25.5% code size overhead in BEEBS benchmark suite [32]. Note that, as Chapter 4.4 states, our prototype only includes untrusted C library functions that the untrusted kernel requires. Since all C library functions that tasks use also need to have an untrusted version, tasks that use more C library functions will have higher code size overhead.

6.2.1 Experiment Results

Table 6.3 shows the code size measurement results.

In total, Kage incurs a code size overhead of 18.62% comparing to FreeRTOS with default configuration and an overhead of 16.67% comparing to FreeRTOS with MPU configuration.

Most of the overhead comes from Silhouette transformations. Without Silhouette, KageOS incurs only 5.04% of overhead comparing to FreeRTOS with default configuration and 3.32% comparing to FreeRTOS with MPU.

Some of the code size results require explanations. First, FreeRTOS with default configuration does not assign the `privileged_function` attribute to privileged kernel functions as it does not use the MPU and always use the privileged mode. Therefore, it has no "privileged function" in the measurement. Second, Kage has less privileged kernel functions than unmodified FreeRTOS with MPU because KageOS only keeps a portion of FreeRTOS's privileged kernel functions as trusted, as Chapter 3.6 explains. Other kernel functions are now untrusted and are in the "other functions" category. Finally, as Chapter 3.1 discusses, the HAL library is part of the trusted computing base, so it is untouched by KageOS or Silhouette. Therefore, the code size of HAL library is consistent across all configurations.

Chapter 7

Related Works

7.1 Control-flow Hijacking Defense on General Purpose Systems

SVA [19, 20] is a compiler-based virtual machine that enforces control-flow integrity, memory safety, and type safety on applications and the kernel. At its core, SVA uses its virtual machine to manage low-level data and provides a set of API for the OS to call at low-level operations. SVA requires the OS to be instrumented and compiled into a virtual ISA. KCoFI [18] uses the SVA infrastructure to enforce control-flow integrity for the operating system. KCoFI provides similar protections as Kage such as protecting the processor state during context switch and on exception entry, and KCoFI has additional security enforcements that are required for general purpose systems with virtual memory and memory management unit. However, Kage and KCoFI use different approaches. KCoFI is a separate middleware between the operating system and the hardware, and the operating system requires little to no modification other than being compiled in a virtual ISA. KCoFI does not trust the OS kernel. While both KCoFI and the OS execute in privileged mode, KCoFI protects its memory space from being overwritten by the OS using software fault isolation [40]. KCoFI stores control data in its memory space such that neither application code nor the OS could corrupt it. On the other hand, Kage enforces its security guarantees by modifying and adding runtime checks

to the FreeRTOS kernel. Kage does not use a virtual machine or a middleware. Instead, it moves a portion of the OS kernel to unprivileged and hardens the portion of the kernel that is still privileged and trusted. Kage stores control data in protected memory regions such that only the trusted kernel has write access to them. Kage uses only the MPU to protect privileged memory space from being overwritten by untrusted code.

OS-level Address space randomization (ASR) [27] is a defense that relies on randomizing the address space of the entire system to hide sensitive data from a malicious party. ASR continuously randomizes the address space such that an attacker cannot realistically perform an attack through brute forcing. However, for embedded systems, ASR cannot effectively hide sensitive data by randomization because embedded systems have significantly smaller memory and no virtual memory. An attacker has a much higher probability of accessing the correct data by guessing.

7.2 Security Enhancement of Embedded OS

7.2.1 Control-flow Hijacking Defense

RECFISH [41] provides CFI checks and protected shadow stack to FreeRTOS. However, RECFISH uses the traditional privileged and unprivileged modes to isolate tasks and the kernel. When accessing the shadow stack, the system needs to use an `SVC` call to switch to privileged mode. Kage, on the other hand, uses Silhouette's mechanisms [44] that allows both privileged and unprivileged code to run in privileged mode. Moreover, RECFISH does not protect the scheduler from other kernel modules or have runtime checks in the kernel API available to application code. Another minor difference is that RECFISH's prototype targets ARMv7-R [11] where Kage's prototype targets ARMv7-M [12].

7.2.2 Memory Safety

nesCheck [33] is a compiler that enforces memory safety on programs written in nesC, a C dialect used in applications for TinyOS. nesCheck uses whole-program static analysis to detect

memory bugs vulnarible locations in the code and adds runtime checks to these locations. As nesCheck adds runtime check to every location in the code that may cause a memory error, complex programs that contain a large amount of memory operations could have high overhead. While Silhouette's store hardening mechanism also incurs overhead on store-heavy programs, many types of store instructions can be directly converted into unprivileged store instructions with no overhead. Also, Kage does not provide full memory safety and only protects control data.

7.2.3 Intra-address Space Isolation

Mbed OS [5] provides a secure partition manager in its Platform Security Architecture, allowing each application to create independent secure partitions. However, the Platform Security Architecture of Mbed OS only supports multi-core ARMv7-M [12] and ARMv8-M [29] microcontrollers, where Kage supports single-core ARMv7-M microcontrollers.

7.3 Control-flow Hijacking Defense on Bare-metal Embedded Devices

Beside Silhouette [44], various other works focus on preventing control-flow hijacking attacks on bare-metal embedded devices. uRAI [9] protects the return address of a function by saving all return addresses in the code segment in compile time and reserving a register to indicate the proper entry for current function, combining with a forward-edge CFI. CFICaRE [34] provides a protected shadow stack and forward-edge CFI for ARMv8-M [29] by providing a branch monitor that accesses the ARM TrustZone-M-protected secure memory [29] to handle control-flow transfers. As TrustZone-M is only available to ARMv8-M architecture, CFICaRE does not support ARMv7-M.

Chapter 8

Conclusions

In this thesis, we presented Kage: a software defense that protects the operating system and applications against control-flow hijacking attacks. Kage combines Silhouette [44] compiler and runtime system and KageOS, a hardened real-time operating system that protects control data of applications and the kernel. Kage protects the return address, processor state, and all data that the trusted kernel uses from corruption by memory errors. Existing FreeRTOS [2] programs should be relatively simple to be ported to Kage, since the secure API of Kage uses the same interface as the task API of FreeRTOS, and other API has the same interface as FreeRTOS as well even though they are now untrusted. With the exception dispatcher, Kage allows developers to add untrusted exception handlers. We evaluated the performance and code size overhead of Kage. Comparing to FreeRTOS with default configuration, Kage incurs 71.57% performance overhead in context switching, 82.76% performance overhead when transferring data using the queue API, 313 additional CPU cycles in untrusted exception handling, and up to 258 additional CPU cycles in runtime checks for secure API functions. In code size, Kage incurs 18.62% overhead comparing to FreeRTOS with default configuration.

In its current state, Kage has a number of weaknesses. First, Kage uses significantly more memory than FreeRTOS. While KageOS adds only few global variables to track the number of tasks created and pointers to task control blocks, its parallel shadow stack doubles the stack size of each task and the kernel. The parallel shadow stack also requires the stack size

of all tasks and the kernel to be the same, meaning that if one task needs a very large stack, all tasks and the kernel need to match the size, taking even more memory. As embedded systems have limited amount of memory, parallel shadow stack puts limits on the number of tasks a device can run. Using a different shadow stack approach [16] could improve the high memory consumption of Kage, with potential sacrifices such as slower shadow stack access and lack of `setjmp`/`longjmp` support. Second, since the HAL library [35] runs at the same level as the trusted kernel, the entire HAL library needs to be trusted. To reduce the amount of trusted code from the HAL library, one could either manually identify functions in the HAL library that requires privileged access or use static analysis to automatically skip functions that requires privileged access during compilation. Third, Kage requires tasks and the untrusted kernel to use a separate untrusted C library instead of the pre-built library provided by the manufacturer, which is used by the trusted kernel. In the current prototype, we only implemented functions that the untrusted kernel uses. Since both C libraries are used, all functions in the untrusted library need to be renamed. Finally, since Kage applies store hardening but does not transform load instructions, application tasks in Kage are not fully in unprivileged execution mode. Although they cannot write to privileged memory or execute privileged instructions, they use regular load instructions and can read from memory regions configured to be readable only in privileged mode.

Bibliography

[1] Newlib. https://sourceware.org/newlib/.

[2] Amazon FreeRTOS. https://aws.amazon.com/freertos/, 2019.

[3] FreeRTOS real time operating system. http://www.freertos.org/, 2019.

[4] Aws partner device catalog. https://devices.amazonaws.com/search?page=1&sv=freertos, 2020.

[5] Free open source IoT OS and development tools from Arm — Mbed. https://os.mbed.com, 2020.

[6] FreeRTOS - multicore (dual core) inter core communication example on STM32H745I Discovery board from ST. https://www.freertos.org/STM32H7_Dual_Core_AMP_RTOS_demo.html, 2020.

[7] Why RTOS and What is RTOS? https://www.freertos.org/about-RTOS.html, 2020.

[8] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.

[9] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. μRAI: Securing Embedded Systems with Return Address Integrity. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, 2020.

[10] Arm Holdings. *CoreMark Benchmarking for ARM Cortex Processors*. ID071213.

[11] Arm Holdings. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, 2014. ARM DDI 0406C.c.

[12] Arm Holdings. *ARMv7-M Architecture Reference Manual*. Arm Holdings, 2014. ARM DDI 0403E.b.

[13] Arm Holdings. *Arm ® Architecture Reference Manual: Armv8, for Armv8-A architecture profile*, 2020. ARM DDI 0487F.b.

[14] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[15] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[16] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, SP '19, pages 985–999, San Francisco, CA, 2019. IEEE Computer Society.

[17] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting Bare-Metal Embedded Systems with Privilege Overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303, May 2017.

[18] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14.

[19] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th USENIX Security Symposium*, Security'09, pages 83–100, 2009.

[20] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In

Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 351–366, 2007.

[21] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, 2015.

[22] dlebed. STM32 FreeRTOS firmware for IR Thermometer board, 2012. `https://github.com/dlebed/stm32l_freertos_ir_thermo`.

[23] DouglasXie. WiFi camera device application firmware project, MCU: STM32F4, IDE: IAR v7.8, 2018. `https://github.com/DouglasXie/WiFi_Camera_Firmware`.

[24] EEMBC. CoreMark-Pro Benchmark Github Repository. `https://github.com/eembc/coremark-pro`.

[25] ErichStyger. McuOnEclipse Processor Expert components and example projects, 2020. `https://github.com/ErichStyger/mcuoneclipse`.

[26] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, pages 633–658. Springer, 2010.

[27] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security12, page 40, USA, 2012. USENIX Association.

[28] hexabitzinc. Hexabitz Audio Speaker and Headphone Jack Module Firmware, 2019. `https://github.com/hexabitzinc/H07R3x-Firmware`.

[29] Arm Holdings. *ARMv8-M Architecture Reference Manual*. Arm Holdings, 2019. ARM DDI 0553B.f.

[30] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes:1,2A,2B,2C,2D,3A,3B,3C,3D and 4*, 2019. Order Number: 325462-070US.

[31] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[32] mageec. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms, 2019. `https://github.com/mageec/beebs`.

[33] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory Safety for Embedded Devices with nesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 127–139, New York, NY, USA, 2017. ACM.

[34] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. CFI CaRE: Hardware-supported call andreturn enforcement for commercial microcontrollers. In *Research in Attacks, Intrusions, and Defenses*, pages 259–284. Springer International Publishing, 2017.

[35] STMicroelectronics. Description of STM32L4/L4+ HAL and low-layer drivers, September 2017. UM1884 Rev 7.

[36] STMicroelectronics. *Discovery kit for IoT node, multi-channel communication with STM32L4*, March 2018. UM2153 Rev 4.

[37] STMicroelectronics. *Discovery kit with STM32F469NI MCU*, June 2018. DocID028183 Rev 2.

[38] STMicroelectronics. *STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm ®-based 32-bit MCUs*, April 2020. RM0351 Rev 7.

[39] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[40] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP'93, 1993.

[41] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. Ward. Control-flow integrity for real-time embedded systems. In *31st Conference on Real-Time Systems (ECRTS'19)*, July 2019.

[42] David A. Wheeler. SLOCCount Version 2.26, 2004.

[43] Joseph Yiu. ARM Cortex-M for Beginners: An overview of the ARM Cortex-M processor family and comparison. 2017.

[44] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient protected shadow stacks on embedded systems, 2019.