Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage

Yufei Du, *UNC Chapel Hill and University of Rochester;* Zhuojia Shen, Komail Dharsee, and Jie Zhou, *University of Rochester;* Robert J. Walls, *Worcester Polytechnic Institute;* John Criswell, *University of Rochester*

https://www.usenix.org/conference/usenixsecurity22/presentation/du

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage

Yufei Du[1,2*], Zhuojia Shen[2], Komail Dharsee[2], Jie Zhou[2], Robert J. Walls[3], and John Criswell[2]

[1]UNC Chapel Hill
[2]University of Rochester
[3]Worcester Polytechnic Institute

Abstract

This paper presents *Kage*: a system that protects the control data of both application and kernel code on microcontroller-based embedded systems. Kage consists of a Kage-compliant embedded OS that stores all control data in separate memory regions from untrusted data, a compiler that transforms code to protect these memory regions efficiently and to add forward-edge control-flow integrity checks, and a secure API that allows safe updates to the protected data. We implemented Kage as an extension to FreeRTOS, an embedded real-time operating system. We evaluated Kage's performance using the CoreMark benchmark. Kage incurred a 5.2% average runtime overhead and 49.8% code size overhead. Furthermore, the code size overhead was only 14.2% when compared to baseline FreeRTOS with the MPU enabled. We also evaluated Kage's security guarantees by measuring and analyzing reachable code-reuse gadgets. Compared to FreeRTOS, Kage reduces the number of reachable gadgets from 2,276 to 27, and the remaining 27 gadgets cannot be stitched together to launch a practical attack.

1 Introduction

Embedded systems are becoming increasingly popular and feature-rich. In addition to traditional embedded systems such as routers, modems, and security cameras, new Internet of Things (IoT) 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 to reduce cost and simplify software design [53].

However, simplicity has a cost: most software for microcontroller-based embedded systems is developed in C. Since C is not a memory-safe programming language, these embedded systems can suffer from exploitable memory safety errors [31, 47]. Attempts to protect control-flow despite such errors have been stymied by the severe resource constraints of microcontrollers e.g., memory is often on the order of kilobytes, and the hardware lacks primitives, such as virtual memory, commonly found in desktop systems. These constraints make it difficult to isolate security-critical data structures from untrustworthy code efficiently. For example, RECFISH [51] uses hardware privilege-levels and context switching to isolate shadow stacks from untrusted code at the cost of high overheads (20-30% increase in execution time). More recent defenses, such as Silhouette [55] and μRAI [5], propose far more efficient mechanisms to protect security-critical data, such as return addresses. However, Silhouette and μRAI target bare-metal applications and, consequently, their protections are insufficient for a fully-featured real-time operating system. According to a recent survey [13], 65% of embedded developers use an embedded operating system for their current projects, and within these developers, 42% use an operating system for real-time capability.

Holistic control-flow protection for real-time OSes must also address several additional challenges. First, as these microcontrollers contain no memory management unit [11], all application tasks and the OS kernel share the same physical address space; there is no memory isolation between tasks or between tasks and the OS kernel. Second, in addition to return addresses and function pointers, several additional data structures require protection, including the processor state saved on a context switch and kernel data structures that contain control data. Third, context switching, interrupts, and exceptions complicate control-flow and require careful handling.

This paper presents *Kage*, a software system that protects the control data of a real-time operating system kernel and the application tasks from control-flow hijacking attacks.[1] Kage provides a protected shadow stack for each application task and the kernel, protects saved processor state during context switch and exception handling, and isolates security-critical kernel data structures from untrusted code. Kage leverages and enhances Silhouette [55] to provide efficient intra-address

[*]Part of the work was done when the author was a MS student at University of Rochester.

[1]"Kage," pronounced *kah-geh*, means "shadow" in Japanese.

space isolation, forward-edge control-flow integrity checks, and return-address integrity. We built a prototype of Kage and evaluated its performance and security improvements over a FreeRTOS-based [6] system.

To summarize, our contributions are:

- We designed Kage which provides efficient and holistic control-flow hijacking protection for microcontroller-based embedded systems with a real-time OS. Kage protects return addresses as well as kernel data structures containing control data.

- We built a prototype of Kage consisting of a FreeRTOS-based [6] embedded OS, a compiler based on Silhouette [55], and a binary code scanner.

- We evaluated Kage using a STM32L475 Discovery board [43], the CoreMark benchmark [8], and microbenchmarks. Kage incurs a 5.2% mean runtime overhead on CoreMark. Kage's kernel components add 7 to 273 CPU cycles, with exception handling adding the most overhead. Kage incurred a code size overhead of 49.8%. However, that overhead was only 14.2% when compared to baseline FreeRTOS with the MPU enabled.

- We evaluated the efficacy of Kage's security improvements over a FreeRTOS-based system. Kage adds several security guarantees, including return address integrity (RAI), control-flow integrity (CFI), and protection of control data during interrupts, context switching, and exception handling. Our results show that Kage reduced the number of reachable code-reuse gadgets from 2,276 to 27 for a Kage-compliant binary with CoreMark, and we explain why those remaining gadgets cannot be stitched together to perform practical code-reuse attacks.

2 Background

In this section, we describe Kage's target microcontroller architecture and introduce the real-time OS on which Kage's prototype implementation is based.

2.1 ARMv7-M

While Kage's design applies to multiple ARM architectures [9, 12], our current work targets the market-dominant ARMv7-M architecture [11]. ARMv7-M, which includes the ARM Cortex-M product family, is designed for resource-constrained, energy-efficient, and low-cost microcontrollers [53]. Consequently, ARMv7-M's design differs from general-purpose processors such as X86 [34] and the ARM Cortex-A product family [9, 10].

Processor Modes and Unprivileged Store Instructions There are two hardware privilege levels in ARMv7-M: *privileged mode* and *unprivileged mode*. ARMv7-M supports a special set of *unprivileged store instructions* that always check the unprivileged access permissions regardless of the processor's current execution mode. For example, even if the processor is currently executing in privileged mode, the unprivileged store instructions can only write to memory locations that are writable in unprivileged mode. Attempts to write to a privileged-only memory location will trigger a memory management fault. Unprivileged store instructions are available on several other ARM architectures, e.g., ARMv7-A [9] and ARMv8-M Main Extension [12].

Memory Protections Unlike desktop systems, ARMv7-M does not provide a Memory Management Unit (MMU) and does not support virtual memory. In ARMv7-M, all memory regions, peripherals, and the processor's control registers are in the same address space. To enforce access control policies, ARMv7-M provides a Memory Protection Unit (MPU) as an optional feature. The MPU allows developers to define the start address and length of memory protection regions and the access permissions of each region. The number of protection regions varies for different hardware implementations, e.g., the development board used in this work supports eight regions [44].

Exception Handling ARMv7-M automatically stores a subset of the current processor state on the stack when executing an exception handler and automatically restores it on exception return. The exception handler is responsible for saving the other registers, 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 one will preempt the current exception; the new exception will otherwise remain pending until the handler of the current exception returns.

2.2 FreeRTOS

When an embedded system requires real-time performance, developers often turn to a *real-time operating system (RTOS)*. While RTOS capabilities vary greatly depending on the target system, we focus on real-time operating systems designed for microcontrollers. One such example is Amazon FreeRTOS, which combines the popular FreeRTOS kernel [32] with libraries for connecting to Amazon's web services [6]. FreeRTOS can run on systems with just kilobytes of memory while still providing powerful features such as real-time scheduling, software timers, and shared queues.

FreeRTOS's fundamental abstractions differ significantly from those found in desktop operating systems. For example, application code is divided into a set of *tasks* that are roughly equivalent to a thread in a desktop process. For each task, FreeRTOS maintains a *task control block* to store important data of the task such as the stack pointer and MPU configuration. The FreeRTOS *scheduler* switches between tasks to meet pre-defined timing constraints. Specifically, the scheduler ensures that the processor will always be executing the highest priority task that is ready to execute in the system.

3 Design

Kage protects application and real-time OS kernel code from control-flow hijacking attacks using a *Kage-complaint real-time OS*, a *Kage compiler*, and a *binary code scanner*. We now describe the threat model that Kage addresses, the guarantees it provides to address these threats, and how the aforementioned Kage components provide these security guarantees.

3.1 Threat Model and System Assumptions

We target ARMv7-M-based single-core microcontrollers with an MPU supporting at least seven regions. As ARMv7-M microcontrollers do not support virtual memory [11], all tasks execute within the same address space as the kernel. Further, to improve performance and reduce programming complexity, the tasks and the kernel execute at the same privilege level by default. A consequence of these design decisions is that a memory error in a task (or library) can lead to the system's complete compromise.

We assume the presence of a powerful adversary that attempts to hijack the control flow of the system. The attacker has access to a memory error in *untrusted code* that she can use to manipulate any *control data* stored in memory, including return addresses of both tasks and the kernel, indirect branches, function pointers, and processor state saved on context switches and during exception handling. The untrusted code includes all application tasks, libraries accessible by those tasks, and part of the kernel. An exception is the standard C library and the compiler runtime library such as libgcc [2] or compiler-rt [1]. We assume these libraries have no memory safety errors, but we also assume that attackers may exploit memory safety errors in untrusted code to hijack the control flow to these libraries so as to use their regular store instructions to corrupt privileged memory. We explain how to mitigate this situation in Section 3.4. In summary, our threat model focuses on code-injection and code-reuse attacks. Other attacks, such as non-control data attacks [20], are out of scope.

3.2 Security Guarantees

To mitigate the threats described in Section 3.1, Kage provides the following guarantees:

Guarantee 1. *Return Address Integrity (RAI): Return instructions will always branch to the legal return address saved by the function prologue.*

Guarantee 2. *Control-Flow Integrity (CFI): Indirect function calls will always branch to the beginning of a function.*

Guarantee 1 comes from Kage's use of per-task shadow stacks for storing return addresses. Guarantee 2 is provided by CFI instrumentation inserted at compile time. We discuss these topics further as part of our discussion of the Kage compiler in Section 3.5.

However, Guarantees 1 and 2 alone are insufficient for mitigating control-flow hijacking attacks. As application code runs in the hardware's privileged mode, it can corrupt control data, stack pointers, and other security-critical data maintained by the OS kernel. Kage, therefore, provides the following additional guarantees:

Guarantee 3. *On a context switch, the task's saved processor state will always be the same as it was when the task was taken off of the CPU. When a task first begins execution, its initial processor state, including the task's initial program counter, stack pointer, and* `control` *register, will always be the initial values defined in task initialization. This guarantees that each task begins execution from its main entry point.*

Guarantee 4. *The processor state saved on interrupts and exceptions is never corrupted. When returning from an interrupt or exception, the processor state loaded onto the processor matches the processor's state prior to the interrupt or exception.*

Guarantee 5. *The location and the content of the processor's interrupt vector table cannot be modified by untrusted code.*

These guarantees severely limit the extent to which an attacker can manipulate the control flow of the system. To ensure that the processor executes the correct instructions, Kage provides one more guarantee:

Guarantee 6. *Memory that is writable by untrusted code cannot be executable, and vice versa.*

The foundation of these guarantees is Kage's use of privileged memory regions isolated from untrusted code. For example, Kage's context switching and exception handling mechanisms use these privileged regions to store processor state, thereby providing Guarantees 3 and 4. Further, by defining the location of the interrupt vector table as privileged, Kage ensures Guarantee 5. Finally, as untrusted code can only write to unprivileged regions, Kage configures such regions as non-executable, i.e., Guarantee 6.

3.3 Kage Overview

Kage consists of three components:

- A **real-time OS for microcontrollers** that provides a protected shadow stack for the kernel and each task and protects security-critical data from corruption by memory errors, including processor state saved on a context switch and scheduler and task management data;

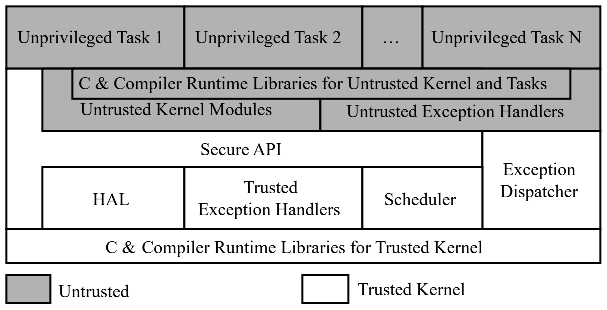
Unprivileged Task 1
Unprivileged Task 2
...
Unprivileged Task N
C & Compiler Runtime Libraries for Untrusted Kernel and Tasks
Untrusted Kernel Modules
Untrusted Exception Handlers
Secure API
Exception Dispatcher
HAL
Trusted Exception Handlers
Scheduler
C & Compiler Runtime Libraries for Trusted Kernel
Untrusted
Trusted Kernel

Figure 1: Architecture of Kage-Compliant Embedded OS

- A **compiler system** that provides efficient intra-address space isolation by transforming all store instructions in untrusted code to ARMv7-M's unprivileged store instructions, saves return addresses to a protected shadow stack, and adds forward-edge control-flow integrity checks;

- A **binary code scanner** that checks whether the binary file generated by the compiler contains any code sequence that may bypass Kage's security guarantees.

3.4 Kage-Compliant Embedded OS

At the core of Kage is an embedded real-time OS. Our design assumes the same task model as FreeRTOS [32], with task data, including task stack pointers, stored in the task control block. Kage divides the code into trusted and untrusted components and the memory into privileged and unprivileged regions. All control data, except function pointers used by untrusted code, are stored in privileged memory. Only trusted components can directly write to privileged memory regions with the exception that function prologues of untrusted components can store return addresses to a privileged region.

Figure 1 illustrates the different components that comprise the Kage architecture. All of the tasks and most kernel components are untrusted. The trusted components are limited to those that must access privileged memory regions.

To mitigate cases wherein control flow is maliciously redirected from untrusted code to the standard C library or the compiler runtime library, Kage supplies two separately compiled versions of each of these libraries. The Kage compiler transforms one copy for safe use by untrusted code, and the other is left unmodified for use by the trusted kernel. Transformed library functions cannot write to privileged memory regions and therefore cannot overwrite control data.

3.4.1 Privileged and Unprivileged Memory

Figure 2 illustrates Kage's unprivileged and privileged memory regions. Untrusted code can only write to the former, while trusted code can write to both types. These access restrictions are implemented, in part, using the store hardening transformation (described in Section 3.5.2) and enforced by the MPU at runtime. Any attempts to write to privileged memory from untrusted code will trigger a fault. While Kage could also restrict loads, load restrictions are not necessary for preventing control-flow hijacking attacks.

Kage's unprivileged memory regions include the unprivileged initialized global data, unprivileged uninitialized global data, unprivileged kernel stack, task stacks, and unprivileged heap regions. The privileged memory regions include all shadow stacks, all control data (except for function pointers in untrusted code), and other security-critical data structures, such as task control blocks and scheduler data structures.

Kage uses separate heaps for trusted and untrusted code. All untrusted code uses the same heap to improve memory utilization. However, Kage could be adapted to use multiple unprivileged heap regions. Kage provides separate dynamic memory allocation and deallocation functions for trusted and untrusted components. The trusted allocation function allocates memory from the privileged heap region, while untrusted code uses the untrusted allocation function to manage memory in the unprivileged heap.

Notably, while all task stacks reside in unprivileged memory, Kage restricts untrusted code to writing only to the stack of the current task. This restriction is necessary to protect control data during context switching (see Section 3.4.3), providing the basis for ensuring Guarantee 3. To detect stack overflows (and underflows), privileged regions surround each task stack.

The System region is also privileged because it contains memory-mapped system registers that must not be accessible to an attacker. For example, a write to this region could be used to change the address of the interrupt vector table [11]; by configuring the System region as privileged, Kage ensures part of Guarantee 5. The Peripheral and Device regions are also privileged.

Finally, the Read-Only Data region, which contains the interrupt vector table, is set to read-only, providing the other part of Guarantee 5. To enforce Guarantee 6, the Trusted and Untrusted Code regions are set to read-only as well and are the only executable regions.

3.4.2 Secure API

The trusted kernel provides a *secure API* for use by the untrusted kernel components and tasks. The secure API allows the untrusted code to perform task management, execute scheduler-related operations, and access the HAL. These operations often require access to privileged memory. The secure API allows unprivileged code to perform these operations without violating Kage's security guarantees.

The secure API functions fall into three broad categories: functions designed for all untrusted code (such as delaying a task, deleting a task, and resuming a task), functions that should only be used by the untrusted kernel (such as raising a

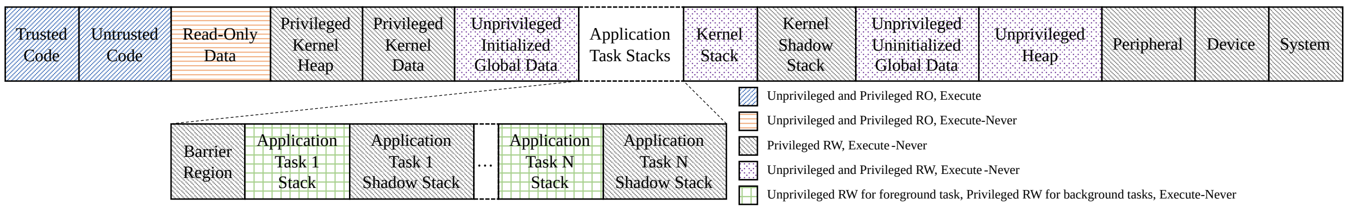
Trusted Code
Untrusted Code
Read-Only Data
Privileged Kernel Heap
Privileged Kernel Data
Unprivileged Initialized Global Data
Application Task Stacks
Kernel Stack
Kernel Shadow Stack
Unprivileged Uninitialized Global Data
Unprivileged Heap
Peripheral
Device
System
Barrier Region
Application Task 1 Stack
Application Task 1 Shadow Stack
...
Application Task N Stack
Application Task N Shadow Stack
Unprivileged and Privileged RO, Execute
Unprivileged and Privileged RO, Execute-Never
Privileged RW, Execute-Never
Unprivileged and Privileged RW, Execute-Never
Unprivileged RW for foreground task, Privileged RW for background tasks, Execute-Never

Figure 2: Memory Regions of Kage

task's execution priority, delaying a task to wait for an event, and resuming a task after the event), and functions used by untrusted exception handlers (such as resuming a task from delay after exception returns).

The secure API adheres to the following design principles. First, the secure API should not overwrite control data, unless such control data is not used anymore (e.g., deleting a task). This helps enforce Guarantees 1, 3, and 4. Second, the secure API should not disable or overwrite hardware configurations that Kage uses for protections, such as disabling the MPU, changing Kage's memory access permissions, changing exception priorities, or overwriting the interrupt vector table. Memory protections are critical for enforcing *all* of Kage's guarantees, controlling exception priorities helps enforce Guarantee 4 as Section 3.4.4 explains, and protecting the integrity of the interrupt vector table is critical to enforcing Guarantee 5. Third, secure API functions must write new control data (e.g., when creating a new task) to a privileged memory region, such as a shadow stack or the task control block for enforcing Guarantees 1, 3, and 4. These design principles can be applied to any real-time OS kernel. Appendix A describes how we applied them to AWS FreeRTOS [6] in detail.

Kage includes additional runtime checks to vet arguments that could be controlled by attackers. First, the secure API checks all pointer arguments. For pointers to task control blocks, the API checks the pointer against a table of pointers to valid task control blocks. For other pointers, the API verifies that they point to an unprivileged memory region. These checks prevent attackers from tricking the secure API code from overwriting control data in privileged regions (which could violate Guarantees 1, 3, and 4) or from overwriting memory-mapped system registers (which could disable the MPU protections needed for enforcing all of Kage's guarantees). Second, the API includes functionality that allows untrusted code to provide a new MPU configuration, but it checks if the new configuration violates Kage's base MPU policy. Third, the API ensures that only the system initialization sequence (run at system startup) can call the task creation API function. This check is necessary to prevent an attacker from creating a new task with a stack that overlaps with the shadow stack of another task (which could violate Guarantee 1). Fourth, the API functions for untrusted exception handlers require that the exception handler temporarily raises the execution priority before calling the secure API function such that other untrusted exception handlers cannot preempt the execution. These API functions check the current priority level. If a check fails, Kage executes a code sequence defined by the developer. In our prototype, this code executes an infinite loop. This enforces Guarantee 4 by ensuring the integrity of interrupted program state saved on interrupts and exceptions.

Kage's design is amenable to real-time requirements. Aside from checks on task control blocks, all secure API runtime checks have constant-time performance. Task control block checks are linear time relative to the number of created tasks. As Kage only permits task creation during system initialization, the maximum number of tasks running on a device can be made finite and known at device install time. Consequently, developers can pre-determine the worst-case overhead of task control block checks. The secure API functions for untrusted exception handlers require raising the exception priority, which may cause delays for other untrusted exception handlers to enter. Therefore, real-time systems that include hard deadlines for exception handling need to measure the runtime of these secure API functions and consider the worst-case additional overhead. In situations where exception handlers with a hard deadline are rare and short, developers could also verify that the handlers are bug-free and place them into the trusted computing base.

Finally, untrusted code is only allowed to use direct function calls to access the secure API. Kage does not assign CFI labels to secure API functions; therefore, untrusted code cannot use indirect branches to execute secure API functions. Section 3.5 explains Kage's forward-edge CFI checks.

3.4.3 Context Switching

The kernel needs to store the processor state to memory when context switching between tasks or when an exception occurs [15]. As this state contains control-flow data, Kage must protect the saved state to ensure Guarantee 3. The protected state must also include the task's stack pointer to prevent untrusted code from violating Guarantee 1. Further, as the Secure API places its frames on the stack of the calling task, Kage must also prevent manipulation of this frame data. As described below, Kage provides these protections through a combination of privileged memory regions, MPU configurations, and a purpose-built PendSV handler.

Kage stores the processor state in the current task's shadow stack during context switching and exception handling. The shadow stack is a privileged memory region, so it cannot be modified by untrusted code (as described previously). The processor state 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 operations). In addition, for exceptions, Kage also protects exception return addresses.

ARMv7-M provides the `PendSV` interrupt to switch contexts efficiently [11], and Kage includes a handler for this interrupt. For performance reasons, the processor automatically saves a subset of the processor state to memory before Kage's handler executes. In some cases, this behavior leads to processor state being saved to unprivileged memory (such as a task's stack). Consequently, Kage's `PendSV` handler first copies the registers automatically saved by the processor from the task's stack to the task's shadow stack. The handler then stores the rest of the processor state to the task's shadow stack and transfers control to the kernel's scheduler component.

The scheduler component then checks if the previous task overflowed its stack prior to the context switch. This check may seem unnecessary given that the untrusted task code will trigger a hardware fault if it attempts to write to the privileged memory adjacent to its stack (as described previously in Section 3.4.1). However, a subtle race condition may still allow for an overflow to occur. In particular, if the context switch occurs after the stack pointer is decremented past the end of the task stack *but before* any unprivileged store instructions that would trigger the fault, ARMv7-M's automatic register spilling mechanism could write into the adjacent privileged region—i.e., where the task's shadow stack is placed—bypassing Guarantee 1.

After the scheduler component transfers control back to the handler, Kage restores the saved processor state for the next task from the appropriate shadow stack. Note that for the subset of state stored automatically by the processor, Kage transfers that state back to the appropriate task stack for restoration by the processor.

Kage's `PendSV` handler also reconfigures the MPU to allow unprivileged write access to the stack region of the next task and disallow access to the stack region of the previous task. By disallowing unprivileged write access to the stack region of other tasks, Kage ensures that a task cannot interfere with another task's stack data. As a task stack may include frames from the Secure API, this MPU configuration prevents untrusted code from manipulating the local variables used by the runtime checks of the Secure API.

Kage prevents untrusted code from executing in the middle of a context switch, ensuring the processor state restored to the task stack cannot be corrupted before the handler returns. It does so as follows. First, Kage prevents untrusted exception handlers from preempting the `PendSV` handler's execution (see Section 3.4.4). Second, while a subset of trusted exception handlers can preempt Kage's `PendSV` handler, those trusted exceptions do not transfer control to untrusted code.

As Kage copies, saves, and restores a fixed number of registers on exception dispatch and context switch, and because the stack overflow check is constant time, Kage's protections for saved processor state incur constant overhead and are therefore amenable to real-time system design.

3.4.4 Exception Handling

Kage divides exception handlers into two types. *Trusted exception handlers* are part of the trusted kernel and can access privileged memory. The trusted handlers in our prototype include the system timer `SysTick` handler, the context switching `PendSV` handler, the system call `SVC` handler, the memory protection fault `MemManage` handler, the memory bus fault `BusFault` handler, and the unrecoverable fault `HardFault` handler. Handlers in the HAL library are trusted as well. Further, if a system contains other trap handlers (i.e., exception handlers that cannot be blocked), these handlers must be trusted. All other handlers are *untrusted* and, consequently, can only write to unprivileged memory.

Exceptions are particularly challenging to handle securely because an untrusted exception handler may interrupt trusted code. For example, consider when a task calls the secure API to execute functions in the trusted kernel. The secure API (and any trusted kernel function it calls) uses the unprivileged task stack to store local variables. Under normal control flow, this is not an issue. However, if an exception whose handler is untrusted occurs, the untrusted handler could corrupt the secure API's stack frames on the task stack.

To resolve the above issue, Kage adds a trusted *dispatcher* function to each untrusted exception handler. When an exception whose handler is untrusted occurs, the appropriate dispatcher executes first. 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 regions are read-only (for both trusted and untrusted code). Only then does the dispatcher transfer control to the untrusted exception handler. After the untrusted handler returns, the dispatcher restores the processor state and restores the MPU configuration. Saving the processor state to and restoring it from the protected shadow stack enforces Guarantee 4. Similar to context switching, the exception dispatcher prevents untrusted exception handlers from voiding the runtime checks of the Secure API.

Exception nesting further complicates the dispatcher's behavior. First, the MPU configuration is only restored after all untrusted exceptions have been handled. Second, 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. ARMv7-M requires three instructions to raise the priority.

To prevent another untrusted exception from occurring during the small window of three instructions, the dispatcher first uses a single instruction (`CPS`) to disable all exceptions until it finishes raising its priority. Raising the exception priority prevents untrusted code from another untrusted exception handler from overwriting the processor state before the dispatcher saves them or after the dispatcher restores them, breaking Guarantee 4. Third, all untrusted exception handlers are assigned a lower priority than any of the trusted handlers. This restriction improves performance by removing the need for trusted handlers to spill the processor state onto the shadow stack. Finally, when an untrusted exception handler calls the secure API, the untrusted handler must first temporarily raise its priority to prevent preemption by other untrusted handlers. The secure API functions check that the exception priority is raised. These two restrictions ensure that untrusted code from an untrusted exception handler cannot corrupt trusted code's stack data and use the trusted code to bypass the security guarantees (e.g. making unprivileged memory executable).

Kage's exception handling mechanism also checks for potential stack overflows. The reason for this check is a subtle race condition wherein another exception handler could preempt the current handler after the stack pointer is decremented past the end of the kernel stack but before the next store instruction triggers a hardware fault. In such situations, ARMv7-M's automatic register spilling mechanism on exception entry could overwrite privileged memory. We described an analogous race condition and overflow check in Section 3.4.3

Similar to context switching, Kage's exception dispatcher incurs predictable constant-time overhead and is consequently amenable to real-time system design.

3.5 Kage Compiler

Kage leverages and enhances Silhouette [55], an LLVM-based compiler, to efficiently isolate the untrusted components and enforce return address integrity and control-flow integrity on the untrusted code. In particular, Kage stores return addresses in a shadow stack (Section 3.5.1). By combing the store hardening transformation (Section 3.5.2), the CFI instrumentation (Section 3.5.3), and the corresponding memory region configuration (Section 3.4.1), Kage guarantees that a return address will always be saved to, protected in, and retrieved correctly from the shadow stack, and therefore providing return address integrity (Guarantee 1). Kage also uses the CFI transformation to guarantee that an indirect function call will always branch to the beginning of a function (Guarantee 2). We describe these transformations below, but we refer the reader to Section 7.1 of the Silhouette paper [55] for greater detail on how these transformations enforce control-flow and return-address integrity.

3.5.1 Shadow Stack Transformation

Kage uses Silhouette's *shadow stack transformation* to transform the prologue and epilogue of each untrusted function. When entering a function, the return address is saved onto a protected shadow stack. When returning from the function, the system uses the return address from the shadow stack instead of the regular stack. The shadow stacks reside in privileged memory, and the shadow stack instrumentation is considered trusted code. Each task, and the untrusted kernel, use separate shadow stacks.

3.5.2 Store Hardening Transformation

Kage uses Silhouette's *store hardening pass* to transform all store instructions in untrusted code into ARMv7-M's unprivileged store instructions. When combined with the appropriate MPU configuration (see Section 4.3.2), this transformation provides intra-address space isolation, preventing untrusted code from modifying privileged memory. Importantly, store hardening allows Kage to access the shadow stack in every function prologue without the costly hardware privilege mode changes of previous work [51].

3.5.3 CFI Instrumentation

We use CFI [3] instrumentation to protect against forward-edge control-flow hijacking. Specifically, for indirect function calls, Kage inserts CFI labels at the beginning of the legal target functions and inserts instrumentation at all indirect call sites to verify that the target has the correct label at runtime.

Not all functions receive labels in Kage. CFI labels are only assigned to functions in untrusted code that are address-taken or visible to other compilation units. This makes it impossible for untrusted code to, for example, jump into trusted code via an indirect function call.

Globally-Unique Label Generation. Label-based CFI requires that the byte sequence used for CFI labels not appear anywhere else within executable memory i.e., that they be *globally unique* [3]. The Silhouette compiler does not enforce this requirement. Kage addresses this limitation, in part, with a novel scheme for label generation.

Kage's global uniqueness guarantee consists of two parts. First, as instructions on ARMv7-M are either one or two half-words long and aligned at half-word boundaries [11], the encoding of a CFI label must not alias any part of instructions that the compiler may generate. Kage avoids this situation by picking a CFI label consisting of two distinct half-words that are undefined instruction encodings on ARMv7-M. Second, the compiler may embed constant data into code that is identical to a CFI label by coincidence. Kage therefore disallows data embedding within the trusted and untrusted code segments.

3.6 Code Scanner

To ensure that the compiled binary code does not violate Kage's security guarantees, Kage includes a static binary code scanner. If the code scanner finds a violation, it alerts the developer.

Kage's code scanner forbids untrusted code from using the privileged `CPS` and `MSR` instructions [11] as these instructions could change the value of important registers (such as the `CONTROL` register and stack pointers) and undermine Guarantees 1 and 4. However, `MSR` instructions that change the `APSR` or `BASEPRI` register are allowed in untrusted code; changing `APSR` only affects the execution of conditional instructions, and changing `BASEPRI` only disables or enables untrusted exceptions, neither of which impacts Kage's security guarantees. The trusted kernel needs `CPS` and `MSR` instructions, so the code scanner allows the trusted kernel to contain these instructions with any operand.

The code scanner also verifies that the only trusted functions that untrusted code calls are secure API functions. Internal trusted kernel functions should not be available to untrusted code. Since Kage does not add CFI labels to the trusted kernel functions, the code scanner only needs to ensure that no direct function calls from untrusted code to internal trusted functions exist.

4 Implementation

In this section, we describe our modifications to the Silhouette compiler system [55] and the implementation of our Kage-compliant RTOS and the binary code scanner. The Kage RTOS extends Amazon FreeRTOS v1.4.9 [6]. We target the STM32L475 Discovery board [43] because it has an MPU and is officially supported by Amazon FreeRTOS.

4.1 Compiler Implementation

We modified 379 lines of code to adapt the LLVM-based Silhouette compiler,[2] a major part of which improves CFI. To ensure that CFI labels are not embedded within other instructions, we picked `0xf870f871` as our CFI label; both `0xf870` and `0xf871` encode an undefined instruction on ARMv7-M [11]. Kage's CFI checks will jump over the CFI labels when performing the indirect branch as the CFI labels are now undefined instructions.

We set the size of each task's stack and the kernel stack to be 4 KB, allowing multiple stacks to fit in the limited 128 KB of RAM on the discovery board. This stack size also allows for more efficient shadow stack instrumentation. In particular, ARMv7-M's store immediate and load immediate instructions support an immediate offset up to 4 KB [11]. With the stack size (and thereby the shadow stack offset) limited to 4 KB, the shadow stack transformation does not need to encode the shadow stack offset into a free register before accessing the shadow stack. As a result, Kage only needs one instruction in the function prologue to write to the shadow stack.

FreeRTOS provides an optional `privileged_functions` section in the code region to store privileged kernel functions. Kage uses this section to store all trusted kernel functions. A special compiler flag can be used to tell the Kage compiler that all functions within a C source file should be placed in the `privileged_functions` section. When the compiler compiles a function in this section, it skips the store hardening, shadow stack, and CFI transformations.

[2]We measured the line count of modifications to the Silhouette compiler using `git diff`.

4.2 Code Scanner Implementation

We implemented Kage's code scanner as a Python script utilizing Python's `elftools` library. The code scanner scans the untrusted code for encodings of instructions that could potentially undermine Kage's security guarantees as Section 3.6 describes. Our code scanner contains 148 lines of code.[3]

4.3 RTOS Implementation

Our OS prototype adds 2,136 lines of code to AWS FreeRTOS. Like default FreeRTOS, our Kage implementation runs both tasks and the kernel in ARMv7-M's privileged execution mode. Unlike FreeRTOS, Kage enables the MPU and leverages compiler transformations, runtime instrumentation, and the kernel modifications described below to enforce control-flow and return-address integrity. We implemented the secure API following our design. Appendix A explains the secure API implementation in detail.

4.3.1 Trusted and Untrusted Components

In our Kage prototype, the trusted kernel components are the scheduler, the task management module, the kernel list module, the trusted dynamic allocation and deallocation module, and the device-specific support module (including exception handlers for `PendSV`, `SVC`, `MemManage`, and `HardFault`), and the HAL library [46] (including the exception handler for `SysTick` and the default code for unimplemented exception handlers). All other kernel components and all tasks are untrusted. The untrusted kernel components include the untrusted list module, the untrusted allocation module, the queue, the stream buffer, event groups, and the timer modules.

Both trusted and untrusted kernel components need to use the kernel list module to access ready and pending lists of tasks. Therefore, Kage provides two list modules, one for the trusted components and one for the untrusted components.

For the untrusted C library, we transformed Newlib [39], an open source C library designed for embedded systems.

[3]We measured the line count of the code scanner and RTOS prototype using SLOCCount [52].

Hardware Region	**MPU Region**	**Access Permissions**	
		Priv.	**Unpriv.**
FLASH	Entire FLASH region	RO, XN	RO, XN
FLASH	Code segments	RO	RO
RAM2	Entire RAM2 region	RW, XN	RO, XN
RAM2	Unpriv. initialized global data and unpriv. heap	RW, XN	RW, XN
RAM	Entire RAM region	RW, XN	RO, XN
RAM	Unpriv. uninitialized global data and kernel stack	RW, XN	RW, XN
RAM	Stack of current foreground task	RW, XN	RW, XN

Table 1: Kage's MPU Configuration on STM32L475 Discovery board. RW = Read & Write, RO = Read-Only, XN = Execute-Never.

The trusted kernel only uses two C library functions, `memset` and `strlen`, so we manually added untransformed implementations of the two functions in the trusted kernel. Similarly, we transformed LLVM's compiler-rt [1] as the compiler runtime library for untrusted code. The trusted kernel only uses `__aeabi_uldivmod` and `__udivmoddi4`, so we added their untransformed implementations in the trusted kernel.

The FreeRTOS kernel uses a `privileged_data` section for privileged global variables. Kage adopts this section for kernel data that should only be writable by trusted code. In our prototype, only the scheduler and task management data is placed in this section.

As Section 3.4 explains, a Kage-compliant OS needs to provide a privileged heap for the trusted kernel and an unprivileged heap for untrusted kernel and application code. As FreeRTOS only provides one heap and uses the same heap for all dynamically allocated data, for Kage, we replaced all memory allocation calls made outside the trusted computing base to a new untrusted memory allocation component, i.e. a different set of `malloc` and `free` functions that uses the unprivileged heap memory region.

Amazon FreeRTOS also includes a suite of functions for interacting with Amazon's web services. In Kage, all of these functions fall outside of the trusted computing base and are instrumented by the Kage compiler.

4.3.2 MPU Configuration

Our implementation configures the MPU as described in Table 1. Kage uses seven MPU regions of the target board. In addition, Kage enables the default background region of ARMv7-M [11], which forbids unprivileged access to any memory address not listed in the regions above and forbids execution in the peripheral and system regions. The MPU configuration covers two disconnected hardware RAM regions of 32 KB and 96 KB, respectively.

4.4 Limitations

There are several limitations of the Kage prototype. First, our current implementation inherits the parallel shadow stack design of Silhouette [55]. Parallel shadow stacks [17] allow for more processor-efficient instrumentation at the cost of higher RAM usage. While this cost is reasonable for the single-stack bare-metal applications targeted by Silhouette, it limits the number of tasks that Kage can support. Further, the current shadow stack implementation also requires all tasks to use the same stack size. Alternative shadow stack designs [17] can address these issues. Second, Kage relies on the developer to properly configure exception priority. Finally, the Kage compiler does not transform inline assembly code or hand-written assembly source files; we could address this in future versions of Kage by implementing the compiler transformations within the assembler. In our current prototype, we transformed all the untrusted inline assembly blobs and assembly source files by hand, which adds 437 lines of code.

5 Performance Evaluation

In this section, we evaluate Kage's performance. First, we use the CoreMark benchmark [28] to evaluate Kage with realistic application code. Then, we use microbenchmarks to explore the impact of individual Kage components.

As a baseline, we compare Kage to unmodified Amazon FreeRTOS v1.4.9 [6], the same version on which our prototype RTOS is based. We compiled the unmodified FreeRTOS with LLVM 9.0 [36], the same version on which the Silhouette compiler is based [55]. By default, FreeRTOS disables the MPU on the discovery board.

We use an STM32L475 Discovery board [43,45] to run all experiments. This board contains an ARMv7-M [11] 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 FreeRTOS set to run at 80 MHz. As all of our test programs fit within the code memory on our board, we use the `-O3` compiler optimization level to improve the execution time of both the baseline code and Kage.

For each benchmark, we run each configuration three times and record the average value of the three runs.

5.1 Macrobenchmark

To the best of our knowledge, no open-source benchmark exists targeting applications with a real-time kernel. We therefore ported CoreMark [28] to AWS FreeRTOS and Kage and modified the benchmark to utilize FreeRTOS's kernel features. CoreMark is an industry-standard benchmark that is recommended by ARM [8]. CoreMark includes common embedded operations such as linked list manipulations, matrix multiplications, and state machine operations. In short, our modified benchmark performs the CoreMark computations using multiple tasks that are preempted and context switched and that communicate their outputs to a main task via a queue.

5.1.1 Benchmark Setup

Our port of CoreMark complies with the license and the instructions included in the CoreMark repository [29]. Namely, we only modified files whose name includes `portme`.

More specifically, we edited the architecture-dependent source files of CoreMark to use the system calls of FreeRTOS and the secure API of Kage so that the CoreMark code would run on AWS FreeRTOS and Kage. We configured CoreMark to use its multi-threaded code path so that it creates multiple benchmark tasks, requiring the OS kernel to context switch between tasks. We configured all the benchmark tasks to have the same priority, causing FreeRTOS to use a round robin scheduling policy. This invokes far more context switches than the default priority-based scheduling algorithm.

We also modified CoreMark so that a main task initializes the system and starts execution of all the subtasks that perform the benchmark computations. This main task also creates a FreeRTOS queue for each subtask to send its output to the main task. The main task will verify that the outputs are correct and measures the start and end times of the benchmark. Our results include the time to create all of the tasks and inter-task communication queues, the time for the tasks to perform their computations, the time needed by FreeRTOS to context switch between tasks, and the time needed to delete all the tasks once all computations are complete. Finally, we modified the function calls to `malloc` and `free` such that the benchmark uses the untrusted heap allocation API of Kage.

CoreMark [28] reports throughput in iterations of computation per second. We report results for our modified version of CoreMark in these units as well. Overhead was measured as a decrease in the number of iterations per second.

5.1.2 Benchmark Results

Table 2 summarizes the performance of the baseline FreeRTOS, Kage with only the kernel modifications (i.e., without any transformations of untrusted code), and the complete Kage system. As Kage requires its OS mechanisms, the compiler transformations, and the code scanner to enforce the security guarantees, we only include separate results for the OS mechanisms to elucidate Kage's overhead sources.

For these experiments, we set the number of iterations to 2,000 per benchmark task and varied the number of tasks from one to three. Note that the single-task configuration uses the single-threaded code path of CoreMark, which runs the benchmark algorithms in the main routine without creating any benchmark task. For each configuration, we run the benchmark 3 times.

Kage incurs 5.2% mean overhead compared to the baseline FreeRTOS. In the single-threaded configuration, Kage incurs the lowest overhead at 4.6%. This is because the single-threaded code path uses only a single main task without creating any benchmark task; as a result, the single-threaded result has no context switching or inter-task queue overhead. With two and three benchmark tasks, Kage incurs 5.5% and 5.5% overhead, respectively. This minor increase demonstrates that the overhead Kage adds via the secure API, context switching, and unprivileged inter-task queue is likely to have only a minor impact on the performance of real world applications. With FreeRTOS's default priority-based scheduling, which invokes context switches less often, we expect the impact of Kage's context switching to be reduced further.

Configuration	**FreeRTOS (Iter/s)**	**Kage's OS Mechanisms (Iter/s)**	**Kage (Iter/s)**
Single-threaded (no benchmark task)	182.20	179.31	173.88
Double-threaded (2 benchmark tasks)	183.07	178.42	172.99
Triple-threaded (3 benchmark tasks)	183.07	178.41	173.01

Table 2: Macrobenchmark Results using Modified CoreMark

Configuration	**FreeRTOS (Iter/s)**	**Kage (Iter/s)**
Single-threaded (no benchmark task)	104.76	96.14
Double-threaded (2 benchmark tasks)	102.60	95.77
Triple-threaded (3 benchmark tasks)	102.59	95.77

Table 3: Macrobenchmark Results without Caching

The primary source of Kage's overhead was the transformation of untrusted code. For example, the mean overhead was 2.2% when only Kage's kernel mechanisms were enabled. In other words, the shadow stack transformation, store hardening, and forward-edge CFI checks account for the rest of the 5.2% mean overhead. Note that Kage relies on all of these mechanisms for its security guarantees, i.e., none of these components are optional.

In both the baseline and Kage, we observed that running more benchmark tasks would unexpectedly increase performance in some situations. Namely, for the baseline, the double-threaded configuration has higher performance than the single-threaded configuration, and Kage, the triple-threaded configuration has higher performance than the double-threaded configuration. To explore the source of speedup, we re-ran the experiments for the baseline and for Kage but with the instruction and data caches disabled. Table 3 shows the results with no caching. Without caching, running more tasks decreases the performance for both the baseline and Kage, as expected. This result leads us to conclude that the minor speedup is due to caching.

5.1.3 Code Size Results

We also measured the code size of the binary files of the triple-threaded configuration to evaluate the impact of Kage on code

Section	FreeRTOS (Bytes)	Kage's OS Mechanisms (Bytes)	Kage (Bytes)
Trusted code	50,584	18,950	18,950
Untrusted code	0	53,410	56,802
Total	50,584	72,360	75,752

Table 4: Code Size Results

Microbenchmark for Secure API	Time (cycles)
Checking pointer of task control block	62
Checking other types of pointer	75
Checking a new MPU region configuration	106
Checking the current exception priority	7

Table 5: Secure API Overhead

size. As the benchmark tasks are identical, the code size of other configurations is nearly identical.

Table 4 shows the sizes of the trusted and untrusted code of FreeRTOS and Kage. Comparing to the baseline FreeRTOS, Kage incurs an overhead of 49.8% in code size. However, the bulk of this overhead comes from enabling the MPU in FreeRTOS, which is not directly from Kage's extensions. For example, the code size of the same FreeRTOS with the MPU enabled is 66,704 bytes, 31.1% larger than that of the baseline FreeRTOS. Compared to this MPU-enabled version, Kage incurs an overhead of only 14.2%.

Kage also incurs a substantial code size overhead because it includes two versions (an untrusted version and a trusted version) of the C and compiler runtime library functions, the heap allocation functions, and the kernel list API functions.

5.2 Microbenchmarks

We designed and built a set of microbenchmarks to measure the number of additional processor cycles introduced by various Kage components. Specifically, we measured the cycle counts of Kage's secure API checks, context switching, exception handling, and untrusted kernel code. We used a combination of handwritten assembly and the KIN1 library [30] to access the cycle counter.

Table 5 shows the cycle counts for the secure API runtime checks. These components have no equivalents in FreeRTOS, so there are no baseline numbers. Table 6 shows the performance overhead of other Kage mechanisms compared to the baseline FreeRTOS and FreeRTOS with the MPU enabled.

Kage's overhead on the microbenchmarks stands in contrast to its low overhead on the CoreMark macrobenchmark. CoreMark's heavy computation and dearth of system calls are the primary reasons for this disparity. Besides the overhead of the compiler transformations, context switching is the component that most consistently incurs overhead in CoreMark.

Secure API Runtime Checks: The secure API runtime

Microbenchmark	FreeRTOS (cycles)	FreeRTOS with MPU (cycles)	Kage (cycles)
Context switching	190	213	322
Exception dispatcher	46	46	319
Queue: create	533	704	819
Queue: send and receive	2,012	2,676	3,636
Stream buffer: create	621	723	904
Stream buffer: send and receive	2,043	2,737	3,200

Table 6: Context Switching, Untrusted Exception, and Untrusted Kernel Overhead

checks incur overhead ranging from 7 to 106 cycles. The amount of overhead added to each secure API function depends on the subset of checks the function uses. For example, only one secure API function calls the 106-cycle MPU configuration check.

Most of the secure API functions contain only one runtime check, either the task control block check or the generic pointer check. Only nine of the thirty-one secure API functions include multiple runtime checks. The worst case is the secure API function that re-configures the task-specific MPU configuration, which has the runtime checks for task control block, generic pointer, and MPU configuration.

Context Switching: To measure the cycle count of context switching, we added code to the beginning of the `PendSV` [11] handler to reset the cycle counter and added code to read the cycle counter immediately before the handler returns. We made the same changes to the two FreeRTOS configurations.

Kage's context switching adds 132 cycles over the baseline FreeRTOS, primarily due to the cost of saving processor state to the shadow stack. This translates to 1.65 microseconds of execution per context switch. That is an overhead of 69.5% relative to the baseline FreeRTOS and 51.2% relative to the MPU-enabled FreeRTOS. Enabling the MPU increases FreeRTOS's context switching latency (and decreases the relative overhead of Kage) as the kernel must read the MPU configurations of the next task from the task control block and write it into the MPU control registers.

Exception Dispatching: This microbenchmark measures the execution time of a `HardFault` exception triggered by a divide-by-zero operation. We configured the `HardFault` handler to use Kage's exception dispatcher. The actual handler contains only a small amount of code that clears the divide-by-zero status bit. The baseline contains the same code that performs a divide-by-zero operation and the same handler code that clears the status bit, but it does not use the dispatcher.

Kage's exception dispatcher adds 273 cycles of overhead. As with context switching, the exception dispatcher's overhead is primarily due to reconfiguring the MPU and saving register state. The overhead is larger for exception dispatching than for context switching because the baseline FreeRTOS does not save any register state when handling exceptions (aside from the registers the hardware saves automatically).

In contrast, the baseline FreeRTOS does save register state during context switching for registers that are not automatically spilled by the hardware.

Untrusted Kernel: Our untrusted-kernel-code microbenchmarks measure the overhead Kage adds to the untrusted kernel API. Specifically, we measured the performance of the queue and the stream buffer API [32] by measuring the execution time of creating a queue or a stream buffer and the execution time of transferring a 32-bit value between tasks.

We selected the queue and stream buffer modules over other untrusted modules for two reasons. First, our macrobenchmark uses the queue API for inter-task communication. Second, their performance does not depend on unpredictable factors such as network connectivity and timer events.

For the queue and the stream buffer APIs, Kage incurs an overhead of 53.7% in queue creation and 80.7% in transferring data in queue; Kage incurs an overhead of 45.6% in stream buffer creation and 56.6% in transferring data in stream buffer. A major contributor to the overhead is the existing FreeRTOS code for supporting the MPU. For instance, Kage's overhead relative to the MPU-enabled FreeRTOS is 16.3% for queue creation, 35.9% for queue data transfer, 25% for stream buffer creation, and 16.9% for stream buffer data transfer. The remaining overhead comes from additional secure API checks and Silhouette's compiler transformations.

6 Security Evaluation

To evaluate Kage's security, we consider the actions available to an attacker when all of Kage's protections have been applied. In particular, given the security guarantees listed in Section 3.2, we consider the following question: is it possible for an attacker to manipulate the control flow in a way that Kage permits and still perform a useful attack? We conclude that useful code-reuse attacks are impossible for the CoreMark application benchmark.

6.1 Summary of Kage Protections

By design, Kage ensures the integrity of return addresses (Guarantee 1); restricts the set of legal targets of forward-edge control-flow branches (Guarantee 2); protects control data during interrupts, exceptions and context switching (Guarantees 3, 4, and 5); and prevents modification of the existing code and injection of new code (Guarantee 6). Most code-reuse attacks [19, 33, 35, 41, 49] are thwarted by this set of security guarantees. Manipulation of non-control-data is not prevented by Kage, but as our threat model (Section 3.1) states, such attacks fall outside of this paper's scope.

The only possible control-flow hijacking hazard in Kage arises from the manipulation of forward-edge control-flow branches. This is due to Kage's coarse-grained CFI instrumentation (Section 3.5.3), which restricts the set of legal targets but does not prevent the manipulation. In particular, an attacker can manipulate a function pointer (in untrusted code) to branch to any function that starts with a valid CFI label. This is a known weakness of label-based CFI instrumentation [3], but as past studies have observed, this issue is largely mitigated by using a shadow stack for protecting return addresses [3, 18]. We confirm these observations also hold for Kage by analyzing the set of reachable code-reuse gadgets in a Kage-compliant OS with the CoreMark benchmark [8].

6.2 Code-Reuse Gadget Analysis

We used ROPgadget [42], an automated tool that is commonly adopted by security researchers and practitioners [16, 21, 23, 54], to search for gadgets within two binaries. One was the baseline which comprised FreeRTOS, CoreMark, and all required supporting libraries; the other was a Kage-compliant system consisting of all Kage's components listed in Figure 1 with the unprivileged CoreMark application tasks. This is the same setup we used in Section 5.

Kage's security guarantees directly render two broad categories of code-reuse gadgets as unreachable: gadgets found in trusted code, and gadgets that do not start at the beginning of a function or immediately after a call instruction. Gadgets in trusted code are not reachable because Kage does not add CFI labels to functions in trusted code, and Kage's labeling scheme ensures that a valid CFI label can never unintentionally appear within trusted code. Consequently, any attempts by the attacker to call trusted code from a manipulated function pointer will fail the CFI check. Untrusted code can call the secure API via direct function calls (which require no CFI labels or CFI checks), so it is possible for an attacker to reuse an existing direct call instruction within a gadget to call the entry point of a secure API function. However, due to the checks performed by the secure API code (see Section 3.4.2), the code within the secure API function cannot be used to bypass Kage's guarantees. Finally, gadgets within trusted code cannot be reached via corrupted function returns because Kage guarantees the integrity of return addresses, and trusted code never calls functions in untrusted code.

Within untrusted code, the only reachable gadgets are those that start at the beginning of a function or start immediately after a call instruction. The former are reachable either directly through function-pointer manipulation or as a part of a larger gadget that contains a direct call instruction. As Kage guarantees return address integrity, the latter can only be reached when a function returns to the callsite preceding the gadget and the callsite is the dynamic caller of the function.

Table 7 summarizes the results. ROPgadget discovers $2,276$ gadgets in the baseline. Because there are no control flow protections in the baseline binary, an attacker can use all of the gadgets. In contrast, ROPgadget finds $1,605$ gadgets in the Kage-compliant binary, but after filtering out unreachable gadgets according to the rules described above, only 27 gad-

Binary	Found (#)	Reachable (#)	Priv. Store (#)	Stitchable (#)
FreeRTOS	2,276	2,276	1,031	1,908
Kage	1,605	27	0	0

Table 7: Gadgets Found in Baseline FreeRTOS and Kage

gets remain reachable under Kage's restricted control flow. Specifically, the 27 gadgets fall into three categories. First, 17 gadgets start at the beginning of a function and end with a return instruction to its caller. Second, 4 gadgets start right after a direct function call and end with a return. The remaining 6 gadgets start after a direct call and end with a direct branch to a static address that is either in the same function or is the entry point of another function.

We believe this significantly reduced set of 27 gadgets makes control-flow hijacking attacks impractical. First, none of the 27 gadgets have a privileged store instruction; therefore, an attacker cannot use these gadgets to corrupt security-critical memory regions (such as the one for MPU configuration). Second, even though attackers can cause control flow to divert to these gadgets, they cannot combine and execute these gadgets in arbitrary order i.e., an attacker cannot cause a gadget to jump to another gadget of the attacker's choosing, rendering the gadget not *stitchable*. All 27 gadgets terminate with a return, a direct tail call to a function, or a direct branch to code within the same function. Because of Kage's return address integrity guarantee, those terminating with a return instruction cannot be stitched together with other arbitrary gadgets within the set of 27 gadgets. Those ending with a direct tail call or direct branch do not jump to another gadget within the set and therefore cannot be stitched together, either. With these restrictions, we see no way to construct a working attack from these gadgets with existing techniques. By comparison, $1,031$ gadgets found in baseline FreeRTOS contain privileged store instructions and $1,908$ gadgets are directly stitchable (because they end in either an indirect branch, branch to another gadget, or a write to the program counter).

It may seem counterintuitive that ROPGadget found fewer gadgets in Kage than it found in FreeRTOS even though the former has a larger code size than the latter (see Section 5.1.3). There are two main factors that contribute to the gadget count difference. First, Kage's shadow stack transformation changes `pop {..., pc}` to `ldr pc, [sp, #4092]`, eliminating a large number of potential gadgets. Specifically, when the shadow stack transformation was disabled, ROPgadget found 1,828 gadgets in Kage (compared to 1,605). Second, FreeRTOS uses `libgcc` as the runtime library while Kage uses a store-hardened version of `compiler-rt` [1]. As a result, many gadgets found in the `libgcc` library in FreeRTOS do not exist in Kage. For example, ROPgadget found 112 gadgets in `__adddf3` in FreeRTOS. While the `compiler-rt` library also has this function, ROPgadget did not find any gadgets in the `compiler-rt` version.

Finally, we acknowledge that our evaluation's completeness depends on two factors. First and foremost, it depends on ROPgadget's ability to locate gadgets. Given that it found 2,276 gadgets in our baseline, we believe that ROPgadget is capable of finding many gadgets. Second, our evaluation depends on the applications used for evaluation. Different programs will have different numbers and types of gadgets. However, ROPgadget found many gadgets in the CoreMark code and even more in common code (the C library, the compiler-rt run-time library, and the untrusted portion of FreeRTOS) that will be linked into any application compiled for Kage.

7 Related Work

Control-Flow Protections on Embedded Systems. Several studies have proposed schemes for protecting bare-metal embedded systems (i.e., those without an OS kernel) from control-flow hijacking, including Silhouette [55], μRAI [5], EPOXY [22], and CaRE [40]. Our system, Kage, leverages Silhouette's store hardening technique to isolate privileged memory. μRAI [5] protects the return address of a function by saving all return addresses in the code segment at compile time and reserving a register to indicate the proper entry for current function, combining that technique with forward-edge CFI checks. EPOXY [22] protects the return address by moving all potentially unsafe stack operations to a separate unsafe stack. Both the safe and unsafe stack are writable in unprivileged mode, so a strong attacker could still potentially overwrite return addresses on the safe stack. Also, EPOXY relies on static analysis techniques to identify unsafe stack operations, which is more challenging for larger programs such as an embedded OS with multiple application tasks, each using its own stack. CaRE [40] provides a protected shadow stack and forward-edge CFI for ARMv8-M [12] by providing a branch monitor that accesses the ARM TrustZone-M-protected secure memory [12] to handle control-flow transfers.[4] Unlike these systems, Kage includes and protects a full real-time kernel.

Some studies have considered embedded systems with a real-time OS. Of those, RECFISH [51] is the most directly relevant to our work. RECFISH [51] provides CFI checks and protected shadow stacks for tasks on FreeRTOS. Kage addresses several RECFISH limitations. First, RECFISH includes the entire RTOS kernel in the TCB; application tasks can trick kernel functions into overwriting control data by passing in bad pointer arguments. Kage, in contrast, removes most kernel code from the TCB and prevents misuse of the secure API with runtime checks. Second, RECFISH does not protect the processor state of exception handlers from untrusted exception handler code, whereas Kage does this through the exception dispatcher. Third, RECFISH uses SVC

[4] As TrustZone-M is only available on the ARMv8-M architecture, CaRE cannot be adapted to the widely-deployed ARMv7-M architecture.

to switch privilege modes when tasks must write to their shadow stacks, incurring high overhead (20-30% on average) in application code. Kage uses store hardening to access the shadow stack efficiently.

μArmor [4] is a compiler-based protection system targeting Zephyr RTOS that relies on diversification and data separation. For data separation, μArmor ensures that code and data are in separate regions and data and pointers are separated within a stack frame. For diversification, μArmor applies various diversification techniques to the binary, such as randomizing register order during a function call, inserting `NOP` instructions, and reordering functions inside the binary. As with other diversification-based defenses, μArmor requires that the attacker cannot obtain the binary or leak memory. In contrast, information leaks do not weaken Kage's security guarantees.

Memory Safety on Embedded Systems. Embedded SAFECode [27] is a compiler that enforces memory safety on embedded programs written in C. SAFECode guarantees the safety of pointer references to the stack and heap and the safety of array accesses. Due to the additional restrictions on pointer operations and array accesses, developers need to modify the OS kernel and application source code to meet SAFECode's requirements. In some cases, it is impractical to pass all the checks. Kage's security guarantees are weaker than full memory safety, but Kage provides strong protection against control-flow hijacking attacks with low overhead and without requiring application source code changes.

nesCheck [38] 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 code areas potentially vulnerable to memory bugs and adds runtime checks to these locations. As nesCheck adds a runtime check to every location in the code that may cause a memory error, programs containing many memory operations could see high overhead. While the store hardening mechanism used by Kage also incurs overhead on store-heavy programs, Kage can transform many types of store instructions with no overhead.

Tock [37, 48] is an embedded operating system with its kernel written in Rust. Tock takes advantage of features of the Rust programming language to enforce memory safety and type safety in its kernel. Unlike Kage, Tock uses hardware privilege levels for isolation. Tock is not a real-time operating system, as it uses a round-robin scheduler instead of a real-time priority-based scheduler. Moreover, Tock requires device manufacturers or OS maintainers to re-write existing HAL libraries in Rust for each device.

Intra-address Space Isolation on Embedded Systems. Mbed OS [7] 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 [11] and ARMv8-M [12] microcontrollers, where Kage supports single-core ARMv7-M microcontrollers.

General-Purpose OS Control-Flow Integrity. SVA [25, 26] is a compiler-based virtual machine that enforces control-flow integrity, memory safety, and type safety on applications and the OS kernel. KCoFI [24] uses the SVA infrastructure to enforce CFI for operating systems. KCoFI provides similar protections as Kage such as protecting the processor state during context switch and exception entry, and additional enforcements required for general-purpose systems with virtual memory. However, Kage and KCoFI have two key differences. First, KCoFI protects its privileged memory using software fault isolation [50] while Kage utilizes store hardening. Second, KCoFI is a compiler-based virtual machine that is designed to be agnostic to the OS kernel's design. Consequently, KCoFI must maintain its own metadata for information that is already present within the OS kernel. In contrast, Kage splits the OS kernel into a small trusted component and an untrusted component, allowing Kage to avoid maintaining redundant copies of security critical metadata outside the kernel.

8 Conclusions and Future Work

We presented the design, implementation, and evaluation of a software system that protects microcontroller-based embedded systems from control-flow hijacking. Collectively called Kage, our work includes techniques to isolate untrusted components, protect control data from corruption by memory errors, and securely handle context switching, interrupts, and exceptions. Kage is open-sourced at https://github.com/URSec/Kage.

Kage's implementation builds on FreeRTOS [6] and extends the Silhouette compiler [55]. Porting existing code written for FreeRTOS to Kage should be relatively straightforward as Kage preserves the kernel features of FreeRTOS (such as scheduling behavior, intertask communication, etc). Further, we designed Kage's secure API to be compatible with the task API of FreeRTOS.

Kage's security guarantees come at a minor cost when executing application code in tasks. For CoreMark [28], Kage incurred an average runtime overhead of 5.2% compared to unmodified FreeRTOS. In addition, Kage added an overhead of 49.8% in code size, but the main source of that overhead was from the additional code in FreeRTOS needed to manage the MPU. In particular, Kage increased the code size by just 14.2% over baseline FreeRTOS with the MPU enabled. Kage also reduced the reachable code-reuse gadgets from 2,276 to 27. Further, those remaining gadgets are not sufficient for conducting a practical attack.

There are several promising directions for future work. First, we plan to extend Kage to support additional security policies. For example, with minor modifications to the secure API, Kage can also protect the integrity of task scheduling. That is, the system could guarantee the real-time execution of tasks. Second, we intend to implement alternative shadow stack designs to improve memory utilization and allow more flexibility in setting the stack size for individual tasks. Finally, we would like to investigate techniques for reducing Kage's code size overhead.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Brent Kang, for their insightful feedback. This work was supported by ONR Award N00014-17-1-2996.

References

[1] "compiler-rt" runtime libraries. https://compiler-rt.llvm.org.

[2] The gcc low-level runtime library. https://gcc.gnu.org/onlinedocs/gccint/Libgcc.html.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security 13*, 1 (Nov. 2009), 4:1–4:40.

[4] ABBASI, A., WETZELS, J., HOLZ, T., AND ETALLE, S. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy* (Stockholm, Sweden, 2019), EuroSP '19, IEEE Computer Society, pp. 31–46.

[5] ALMAKHDHUB, N. S., CLEMENTS, A. A., BAGCHI, S., AND PAYER, M. μRAI: Securing embedded systems with return address integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium* (San Diego, CA, 2020), NDSS '20, Internet Society.

[6] Amazon FreeRTOS. https://aws.amazon.com/freertos.

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

[8] ARM HOLDINGS. *CoreMark Benchmarking for ARM Cortex Processors*, July 2013. DAI 0350A.

[9] ARM HOLDINGS. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, Mar. 2018. DDI 0406C.d.

[10] ARM HOLDINGS. *Arm Architecture Reference Manual: Armv8, for A-profile architecture*, July 2021. DDI 0487G.b.

[11] ARM HOLDINGS. *ARMv7-M Architecture Reference Manual*, Feb. 2021. DDI 0403E.e.

[12] ARM HOLDINGS. *Armv8-M Architecture Reference Manual*, Sept. 2021. DDI 0553B.q.

[13] ASPENCORE. 2019 embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[14] ATZORI, L., IERA, A., AND MORABITO, G. The internet of things: A survey. *Computer Networks 54*, 15 (Oct. 2010), 2787–2805.

[15] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, 2005.

[16] BROWN, M. D., AND PANDE, S. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *Proceedings of the 12th USENIX Workshop on Cyber Security Experimentation and Test* (Santa Clara, CA, 2019), CSET '19, USENIX Association.

[17] BUROW, N., ZHANG, X., AND PAYER, M. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy* (San Francisco, CA, 2019), SP '19, IEEE Computer Society, pp. 985–999.

[18] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium* (Washington, DC, 2015), Security '15, USENIX Association, pp. 161–176.

[19] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, 2014), Security '14, USENIX Association, pp. 385–399.

[20] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD, 2005), Security '05, USENIX Association, pp. 177–191.

[21] CLEMENTS, A. A., ALMAKHDHUB, N. S., BAGCHI, S., AND PAYER, M. ACES: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium* (Baltimore, MD, 2018), Security '18, USENIX Association, pp. 65–82.

[22] CLEMENTS, A. A., ALMAKHDHUB, N. S., SAAB, K. S., SRIVASTAVA, P., KOO, J., BAGCHI, S., AND PAYER, M. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy* (San Jose, CA, 2017), SP '17, IEEE Computer Society, pp. 289–303.

[23] COFFMAN, J., KELLY, D. M., WELLONS, C. C., AND GEARHART, A. S. ROP gadget prevalence and survival under compiler-based binary diversification schemes. In *Proceedings of the 2016 ACM Workshop on Software PROtection* (Vienna, Austria, 2016), SPRO '16, ACM, pp. 15–26.

[24] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Berkeley, CA, 2014), SP '14, IEEE Computer Society, pp. 292–307.

[25] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th USENIX Security Symposium* (Montreal, QC, Canada, 2009), Security '09, USENIX Association, pp. 83–100.

[26] CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07, ACM, pp. 351–366.

[27] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems 4*, 1 (Feb. 2005), 73–111.

[28] CoreMark: An EEMBC benchmark. https://www.eembc.org/coremark.

[29] CoreMark benchmark Github repository. https://github.com/eembc/coremark.

[30] ERICHSTYGER. McuOnEclipse processor expert components and example projects. https://github.com/ErichStyger/mcuoneclipse.

[31] ERLINGSSON, Ú., YOUNAN, Y., AND PIESSENS, F. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer-Verlag, Berlin/Heidelberg, Germany, 2010, pp. 633–658.

[32] FreeRTOS real time operating system. http://www.freertos.org.

[33] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (San Jose, CA, 2014), SP '14, IEEE Computer Society, pp. 575–589.

[34] 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*, June 2021. Order Number: 325462-075US.

[35] KEMERLIS, V. P., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. ret2Dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, 2014), Security '14, USENIX Association, pp. 957–972.

[36] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, CA, 2004), CGO '04, IEEE Computer Society.

[37] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, 2017), SOSP '17, ACM, pp. 234–251.

[38] MIDI, D., PAYER, M., AND BERTINO, E. Memory safety for embedded devices with nesCheck. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates, 2017), ASIACCS '17, ACM, pp. 127–139.

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

[40] NYMAN, T., EKBERG, J.-E., DAVI, L., AND ASOKAN, N. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses* (Atlanta, GA, 2017), RAID '17, Springer-Verlag, pp. 259–284.

[41] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information Systems Security 15*, 1 (Mar. 2012), 2:1–2:34.

[42] SALWAN, J., AND WIRTH, A. Ropgadget: Gadgets finder and auto-roper, 2011. http://shell-storm.org/project/ROPgadget.

[43] STMICROELECTRONICS. *UM2153 User Manual: Discovery kit for IoT node, multi-channel communication with STM32L4*, Oct. 2019. Rev 5.

[44] STMICROELECTRONICS. *PM0214 Programming Manual: STM32 Cortex-M4 MCUs and MPUs programming manual*, Mar. 2020. Rev 10.

[45] STMICROELECTRONICS. *RM0351 Reference Manual: STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm-based 32-bit MCUs*, June 2021. Rev 9.

[46] STMICROELECTRONICS. *UM1884 User Manual: Description of STM32L4/L4+ HAL and low-layer drivers*, Sept. 2021. Rev 9.

[47] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Berkeley, CA, 2013), SP '13, IEEE Computer Society, pp. 48–62.

[48] Tock embedded operating system. http://www.tockos.org.

[49] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection* (Menlo Park, CA, 2011), RAID '11, Springer-Verlag, pp. 121–141.

[50] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, 1993), SOSP '93, ACM, pp. 203–216.

[51] WALLS, R. J., BROWN, N. F., LE BARON, T., SHUE, C. A., OKHRAVI, H., AND WARD, B. C. Control-flow integrity for real-time embedded systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems* (Stuttgart, Germany, 2019), ECRTS '19, Schloss Dagstuhl–Leibniz-Zentrum füer Informatik, pp. 2:1–2:24.

[52] WHEELER, D. A. Sloccount version 2.26, 2004. https://dwheeler.com/sloccount.

[53] YIU, J. ARM Cortex-M for beginners: An overview of the ARM Cortex-M processor family and comparison.

[54] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium* (Washington, DC, 2013), Security '13, USENIX Association, pp. 337–352.

[55] ZHOU, J., DU, Y., SHEN, Z., MA, L., CRISWELL, J., AND WALLS, R. J. Silhouette: Efficient protected shadow stacks for embedded systems. In *Proceedings of the 29th USENIX Security Symposium* (Boston, MA, 2020), Security '20, USENIX Association, pp. 1219–1236.

A Secure API Implementation

This appendix shows the secure API details in our prototype of Kage. The secure API is the subset of API functions, from

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 8: Data Types of Secure API

Name(Argument Types): Return Type	Description
`xTaskCreateRestricted(TaskParameters_t*, TaskHandle_t*): BaseType_t`	Create a task with given parameters. 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.
`vTaskFinishInit(void): void`	Mark the end of task creation. After this point, Kage does not allow creating new task anymore.
`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. 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. 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. 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.
`vMainUARTPrintString(char*): void`	Print given string to serial output using HAL library function that accesses corresponding peripherals.

Table 9: Task Management Secure API for Application Tasks and Untrusted Kernel

the FreeRTOS [32] scheduler and task management modules, that write to protected data structures, with one exception: we added one additional API function, `vTaskFinishInit`, to declear the end of the initialization sequence. Table 9, Table 10, and Table 11 list the individual secure API functions. Beside `vTaskFinishInit`, all functions in the list use the same arguments and return types as the corresponding functions in FreeRTOS.

Table 8 lists the argument data types of the secure API. All secure API functions that have an argument of type `TaskHandle_t` contain the runtime check that verifies the task control block's validity. All secure API functions with other types of pointer arguments use a different runtime check to verify that the pointer points to an unprivileged memory region and that writing to the address would not cause an overflow.

Table 9 lists the secure API functions available to both application tasks and untrusted kernel components. These functions are available for untrusted components to control the state of tasks such as deleting a task, changing the MPU configuration of a task, and task synchronization. The `vTaskAllocateMPURegions` contains the runtime check verifying that the new task-specific MPU configuration does not violate Kage's MPU policy. The `xTaskCreateRestricted` function is only available to the system initialization sequence. To enforce this restriction, we require developers to call the `vTaskFinishInit` function that marks the end of the initialization sequence. After calling `vTaskFinishInit`, Kage no longer allows task creation.

Table 10 lists the secure API functions only available to untrusted kernel components. These functions allow untrusted kernel components such as the queue API and the timer API to manage the interactions of tasks. For example, when a task calls the queue API to receive a data structure,

Name(Argument Types): Return Type	Description
`vPortEnterCritical(void): void`	Enter critical section that temporarily disables context switching and exception handling.
`vPortExitCritical(void): void`	Exit critical section.
`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 10: 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. 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. The normal version of this API is defined as a macro of `xTaskGenericNotify`.

Table 11: Task Management Secure API for Untrusted Exception Handlers Only

but the queue is empty, the queue API function calls the `vTaskPlaceOnEventList` secure API function to add the task to the list of tasks waiting for data on this queue and to delay the task from executing until the data arrives or a timeout occurs.

Table 11 lists the secure API functions available to untrusted exception handlers. They allow untrusted exception handlers to resume or unblock a task and optionally send signals to the task through the notification bits. These secure API functions contain the runtime check that verifies the current exception priority.