Memory Efficient Shadow Stack Designs for Real-Time Embedded Systems

by

Avery Smith

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2023

APPROVED:

Professor Robert Walls, Major Thesis Advisor

Professor Craig Shue, Head of Department

Abstract

Kage is a real-time operating system that guarantees return address integrity and control flow integrity for embedded ARMv7-M devices. Kage uses a *parallel shadow stack* for protecting return addresses to minimize instrumentation and, consequently, runtime performance overhead. However, Kage's parallel design incurs a large memory penalty to the device's RAM section. Embedded devices face tighter constraints on memory usage, so memory efficiency becomes a major concern. To address this challenge, we propose two novel shadow stack designs: the *interleaved shadow stack* design and the *shared shadow stack* design. These designs offer similar runtime performance compared to the parallel shadow stack design with significantly higher memory efficiency. For instance, we observed an up to 71.43% improvement to stack usage over the parallel design when running the CoreMark benchmark suite.

Acknowledgments

My utmost thanks go to Professor Walls for doing everything in his power to bring out the best in his students and offering multiple opportunities to explore research just for the fun of it. There is even more to thank of him outside of this project as he was crucial to my growth as a student of cybersecurity, a mentor in life goals in academia, and of course a master of the rock wall.

Thank you to Professor Criswell of the University of Rochester for co-advising and offering a strong analytical lens.

Thank you to Tongwei Ren as one of the major co-authors of this work and for convincing me to try yoga.

The authors of *Kage* and *Silhouette* deserve special recognition as the shoulders of which we currently stand on. Of those authors, a special thanks goes to Yufei Du for answering questions as I started work on this project. I want to recognize Noah Olson, Arthur Ames, Jose Filizzola, Qixing Xue, and Jake Backer who all made contributions to the project at different points in time.

Thank you to the many professors whose enthusiasm for learning continues to inspire me, along with all the individuals who have contributed to my well-being outside of this project. Of these there are too many to list. However, a special recognition goes to Professor Shue who has always found time to help even when he's triple booked, and to Beckley Schowalter for her constant enthusiasm and encouragement.

My master's education was funded through the National Science Foundation's *CyberCorps®: Scholarship for Service.*

This material is based upon work supported by the National Science Foundation under Grant No. 2154415.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded devices are common in daily life, placed in homes in the form of routers, modems, and cars [1]. Software for these devices may be written in memory unsafe languages, such as C, placing these devices at risk of exploitation should a memory safety error be present [2, 3]. There exists literature to address memory safety for these devices with Kage [4] being one of the leading solutions in terms of runtime performance.

Kage is an extension of FreeRTOS [5] and the LLVM [6] compiler to offer control flow integrity and return address integrity with an average 5.2% impact to runtime performance compared to baseline FreeRTOS. However, Kage optimizes for reduced runtime overheads. One of the consequences of this design goal is the use of a parallel shadow stack design. The parallel shadow stack design allows for fast instrumentation for accessing the shadow stack but comes at a cost of high RAM usage relative to baseline FreeRTOS. One of the major constraints of a parallel shadow stack design is that all tasks on the system must have the same size. This causes overallocation to occur when tasks have varying stack needs.

In contrast, we argue memory usage should be a significant consideration in memory safety designs and propose two novel shadow stack designs which offer the security guarantees and runtime performance of Kage while reducing RAM usage.

In summary, we make the following contributions:

- We present the interleaved shadow stack design which, through careful stack placement, relaxes the constraint of Kage which requires all tasks to have the same size.

- We present the shared shadow stack design which allows all tasks to share the same stack. This design requires updates to the task scheduler and inter-process communication mechanisms for functional correctness.

- We evaluated these designs using a STM32L475 Discovery board with the CoreMark benchmark and microbenchmarks. We observe up to 25% memory savings in the interleaved design and 71% memory savings in the shared design when compared to Kage. We observe a 0-0.07% impact to runtime compared to Kage.

Chapter 2

Background

We describe our threat model, targeted architecture, real-time operating system, and the previous work we implement our designs on.

2.1 Threat Model

We assume an attacker can manipulate a memory error in *untrusted code* to alter *control data* located in memory. This includes return addresses, indirect branches, function pointers, and processor state of the currently running thread. Examples of possible memory errors include buffer overflows [2] and dangling pointers [3].

We define untrusted code as all application written code, the libraries they require, and portions of the kernel. Conversely, *trusted code* is assumed to be free of memory errors. We focus on preventing code injection [7, 8, 9] and code-reuse attacks [10, 11, 12, 13, 14] with this threat model. Physical attacks and non-control data attacks [15] are out of scope.

2.2 Architecture

We target the ARMv7-M [16] and ARMv8-M [17] architectures which enables access to two hardware code execution levels, *privileged mode* and *unprivileged mode*. Unprivileged mode is referred to as user-mode in other sources. ARMv7/8-M has no memory management unit (MMU). All memory regions, peripherals, and processor control registers lie in the same physical address space. Instead, ARMv7/8-M provides a memory protection unit (MPU) as an optional feature to set access policies dependent on privilege level. The MPU has a fixed maximum number of regions, usually eight, but the exact value depends on implementation. In each region, read, write, and execute permissions can be set for each privilege level. Notably, the small number of available regions and the following constraint present challenges when designing shadow stacks.

MPU Constraint. *An MPU region must have a size that is a power of two and must be naturally aligned to the size.*

An MPU region larger than 256 bytes may additionally be equally divided into eight *subregions*. By default, every subregion is enabled and may be selectively disabled by the developer. MPU regions have priority levels and can overlap other regions. When an overlap occurs, the highest priority region determines the access policy. Disabling a subregion allows a lower priority region to determine access policy.

ARMv7/8-M offers a unique instruction that interacts with the MPU. The store with translation (`strt`) instruction accesses memory using the MPU's *unprivileged* permissions *regardless of the current code execution level*. These instructions are the basis for store hardening provided by Silhouette [18] which Kage takes advantage of to provide fine-grained intra-address isolation.

2.3 FreeRTOS

When an embedded system requires real-time performance, developers often turn to a *real-time operating system* (RTOS). Kage directly extends FreeRTOS [5], specifically Amazon-FreeRTOS [19]. FreeRTOS is a popular open-source RTOS for microcontrollers and can run on systems with kilobytes of memory. FreeRTOS provides features such as shared queues, preemptive scheduling, and task priority assignment [20]. These features benefit real-time systems as they ensure efficient resource management, timely execution of tasks, and application responsiveness.

FreeRTOS organizes applications as a collection of independent threads called *tasks*. For each task, FreeRTOS maintains a *task control block* (TCB) to store needed runtime data such as a stack pointer and MPU configuration. The TCB remains persistent across the lifetime of a task, which we use to our advantage in the implementation of the shared shadow stack design.

Applications developed on FreeRTOS follow a paradigm which may be different than other programs.

FreeRTOS Task Design. *Tasks are designed as infinite loops to repeatedly complete a job. Tasks block when they have no work left to do.*

The FreeRTOS kernel maintains lists to determine whether a task is in the *blocked*, *suspended*, *ready*, or *running* state. By default, FreeRTOS uses a preemptive scheduler; *the running task is always the highest priority task in the ready state.* A task can only enter the blocked state during their own execution through yielding. Tasks are removed from the blocked state when the condition of their yield is met. The suspended state is an optional feature which allows halting and resuming the execution of tasks from anywhere inside the application. This paradigm and the management of task states are a major consideration for the updates of the scheduler in the shared shadow stack design to ensure functional correctness.

2.4 Kage

Kage [4] extends the FreeRTOS kernel [5] and LLVM compiler [6] to provide return address integrity (RAI) and control flow integrity (CFI) for embedded ARMv7-M devices. Kage implements return address integrity through the use of *shadow stacks* [21]—an area of trusted memory which can only be written to by store instructions executing in privileged mode. Figure 2.1 shows the stack and shadow stack layout located in RAM. We call this a *parallel shadow stack design*. Kage updates function prologues and epilogues for secure access into the shadow stack using a shadow stack offset as seen in figure 2.2.

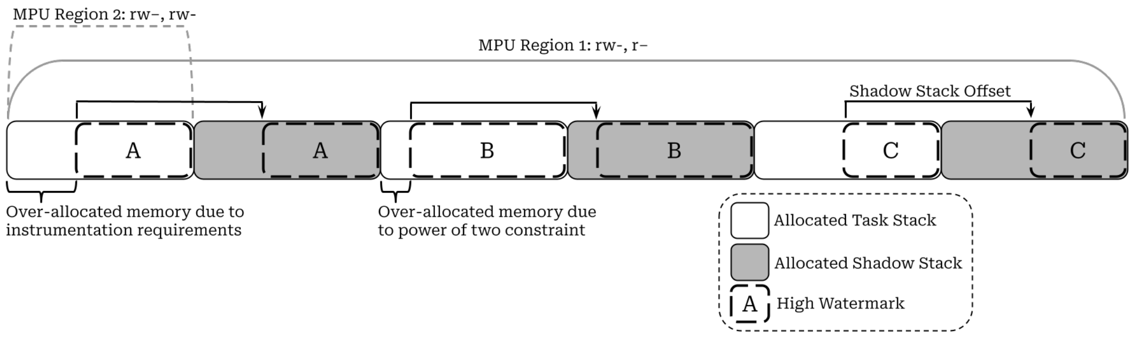


Figure 2.1: The stack layout of the parallel design. The first MPU region protects all of RAM to only allow privileged writes and does not move for the lifetime of the application. On context switch, the second MPU region with a higher priority is moved to allow unprivileged writes to the stack of the task executing.

The Kage compiler uses *store hardening* [18] to transform functions in the untrusted computing base to enforce intra-address space isolation between stacks and shadow stacks. An attacker exploiting a memory error in untrusted code cannot write into the shadow stack and therefore cannot subvert control flow.

The compile time transformations net Kage a huge advantage over other memory protection solutions: Kage incurs an average runtime performance overhead of 5.2% compared to an unmodified FreeRTOS.

Function Prologue

```
push    {r4-r6, lr}
--------------------------------
str.w   lr, [sp, #1020]
sub     sp, #16
strt    r4, [sp]
strt    r5, [sp, #4]
strt    r6, [sp, #8]
strt    lr, [sp, #12]
```

Function Epilogue

```
pop     {r4-r6, pc}
--------------------------------
ldr     r6, [sp, #8]
ldr     r5, [sp, #4]
ldr     r4, [sp]
add     sp, #16
ldr.w   pc, [sp, #1020]
```

Figure 2.2: The Kage compiler transforms function prologues and epilogues in the untrusted computing base to use a privileged store to the shadow stack and store with translation (unprivileged store) for stack writes. The immediate offset of 1020 is the shadow stack offset.

Key Assumption 1. *Kage focuses primarily on runtime performance.*

The efficiency of Kage's runtime performance hinges on the instrumentation added to prologues and epilogues in the untrusted computing base at compile time. Functions are shared among multiple tasks in the application, and the shadow stack offset must work for every task. If every task were the same size, then the shadow stack offset would be the same for all of them. Kage imposes exactly this assumption to the system.

Key Assumption 2. *All tasks must allocate the same amount of stack and shadow stack space.*

Therefore, the allocated amount for each task is the next power of two above the largest high watermark. The allocated space must be a power of two to meet the constraints of the MPU. A high watermark is a recording of the average stack usage for a task. The larger the gap between the largest and smallest high watermark, the more space goes un-utilized in RAM.

Chapter 3

Design

We lead with the intuition for each design and describe the key features which make each design possible. Memory savings are our leading goal and we construct our designs to keep Kage's efficient shadow stack instrumentation.

3.1 Interleaved Shadow Stack

Our first intuition of a design comes from the goal of relaxing Kage's stack size assumption while still retaining the efficient instrumentation to access the shadow stack.

Key Feature 1. *Grouping task stacks together, followed then by grouped shadows stacks, allows each task to use a different stack size while retaining the same instrumentation of a parallel shadow stack.*

The interleaved shadow stack design shown in figure 3.1 has a constant value for the shadow stack offset. The interleaved design no longer wastes space due to a discrepancy in high watermark values and only overallocates to meet the the constraints of the MPU, that is, allocated stacks and shadow stacks are powers of two.

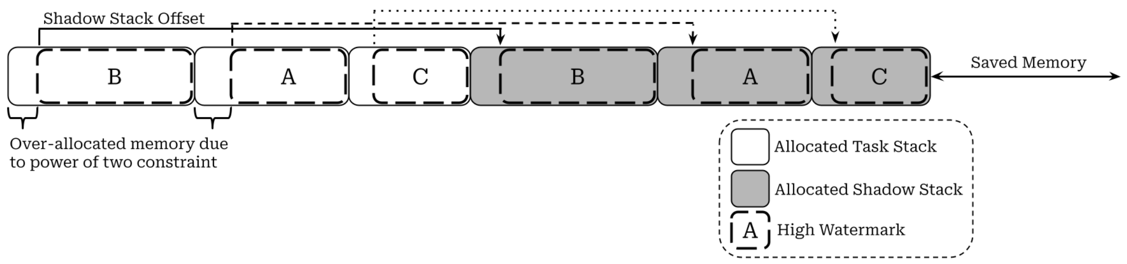


Figure 3.1: The arrangement of stacks allows for all tasks to use the same offset value. This uses the same MPU configuration settings as the parallel shadow stack design.

Key Feature 2. *Tasks are arranged in descending order of size to prevent fragmentation caused by natural alignment constraints.*

If a smaller stack were to come before a larger one, padding would be required to place the larger stack in a location that is naturally aligned with its size. By ordering in descending order of size, no padding is required.

3.2 Shared Shadow Stack

The second intuition comes from the idea that some tasks may run infrequently. The previous stack designs require allocating stack space at compile and linkage time. Tasks that run infrequently—such as initialization tasks—will force underutilization of RAM for the life of the embedded device. Our primary insight is to incorporate a shadow stack into a shared stack design. Shared stacks have previously been proposed to increase memory utilization [22]. The shared shadow stack design in figure 3.2 has all tasks share a single stack in memory, along with a single shadow stack. The application keeps a single stack pointer in use instead of one for each task. When a new task preempts a lower priority one, it begins writing at the stack where the previous one left off.

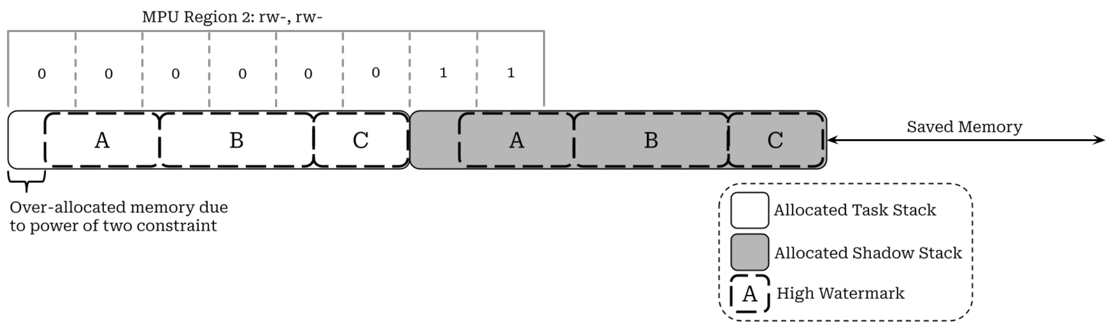


Figure 3.2: The layout of the shared stack. Instead of moving the second MPU region on context switch, the region is now fixed to cover all of the shared stack. To further reduce allocated stack space, extra subregions are disabled using a bitmask in the `MPU_RASR` register. A bit set to `1` indicates a disabled subregion.

As all tasks share the same stack space, it is important to ensure that stack corruption does not occur through normal use. Functional correctness in the shared design can be met with a few extra requirements to the scheduler.

Requirement 1. *Tasks free their stack space when not in the ready state and consume stack space when they run again.*

This requirement ensures that when a higher priority task would halt execution it frees its stack space so that a lower priority task can resume at a higher point in the stack. An example of this can be seen in figure 3.3.

FreeRTOS does not have a mechanism to free stack space when not in use. As such, we add a feature to the shared shadow stack design.

Key Feature 3. *We introduce a task return handler to free stack space when the task returns.*

We add a kernel supervisor call to initiate a task return when a task has completed its work. The kernel frees the used stack space of the task and runs the

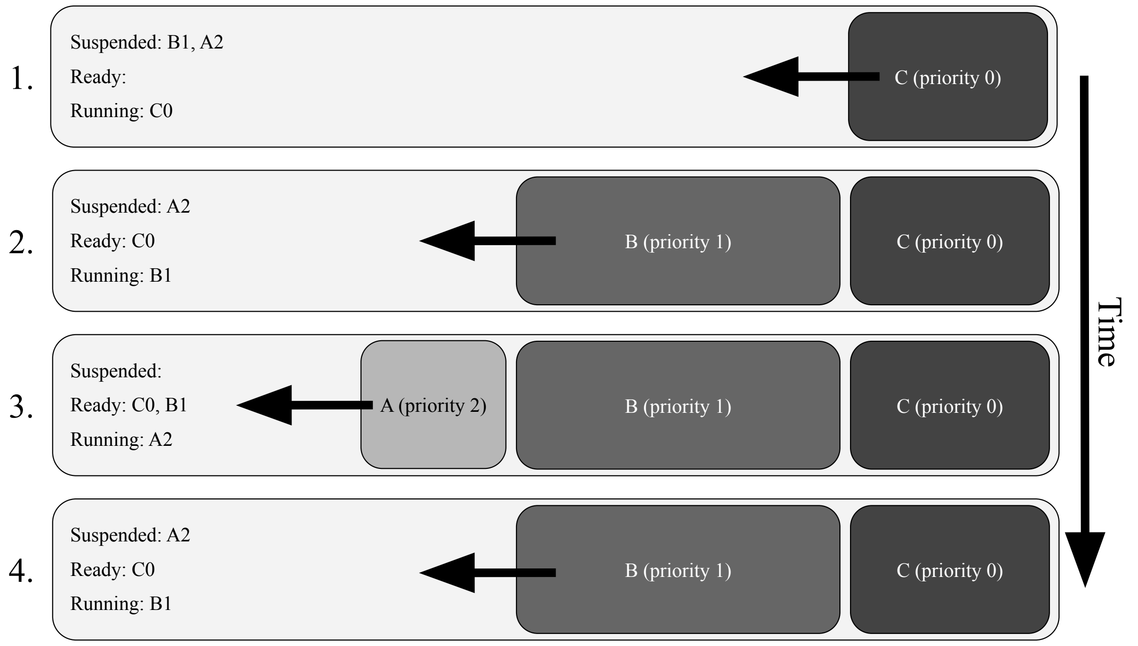


Figure 3.3: This is an example of the usage of the shared stack over time. The stack continually grows downward. When a task moves out of the ready state, it frees the stack space it used. A lower priority task can run again without issue.

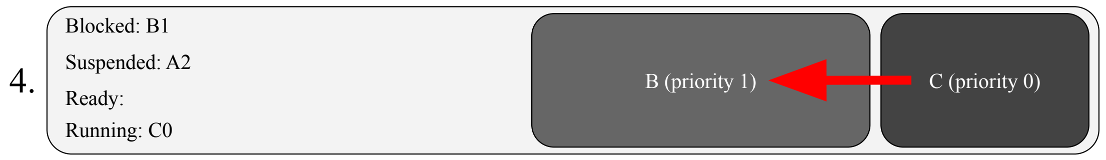


Figure 3.4: A violation of requirement 1 or requirement 2 would cause a lower priority task to corrupt the stack of a higher priority one.

highest priority task in the ready state. This is in direct contradiction to the standard design of a FreeRTOS task. We require that tasks now return when their work is complete instead of looping again. As a consequence, the second requirement now becomes necessary.

Requirement 2. *Tasks must run to completion and can not block.*

Blocking would immediately cause the blocked task to have their stack corrupted by a lower priority task as the blocked task exists lower on the stack. Disallowing blocking is a severe restriction to a real-time operating. An alternative must exist in order to allow event driven scheduling. To fill this gap, we add the concept of *task predicates*.

Key Feature 4. *We introduce task predicates to ensure that a task only runs when its requirements are fulfilled.*

If a task were to run and block on a requirement, the requirement must now be satisfied before the task runs. For example, if a task needs data from a queue, the kernel only schedules the task once the queue has an object in it. The kernel moves tasks from the suspended state to the ready state when the predicates for a task evaluate to true.

With these requirements met, the scheduler will never schedule a task such that it can write into a different task's stack space [22]. As a direct consequence of the requirements, round-robin scheduling is not supported; two tasks cannot be the same priority.

Key Feature 5. *Disabling select MPU subregions further reduces the amount of over-allocated stack space.*

The shared design has the same implementation technique of placing an MPU region over the task stack space to allow unprivileged writes. However, instead of covering the individual tasks, this region covers the entire shared stack. This then subjects the shared stack to the same constraints of the MPU. This presents a potentially large amount of wasted space when rounding up to the next power of two.

Let w be the high watermark for the shared stack and the next power of two be 2^n. In the worst case, w is one greater than a power of two ($w = 2^{n-1} + 1$) and the shared stack must allocate an extra $2^{n-1} - 1$ bytes which will go unused. We can reduce this waste by disabling extra subregions. Instead of allocating the full 2^n, we allocate to the next multiple of the subregion size. If our implementation has eight subregions, each subregion has size $2^n/2^3 = 2^{n-3}$. In the worst case, we waste almost a full subregion: $2^{n-3} - 1$.

This is only a constant improvement when viewed asymptotically, but since embedded devices have less memory available, we see this as a dramatic improvement in practice.

It is important to note that this technique cannot be applied to the parallel or interleaved design. We look at the high watermark for task i, w_i. We must still apply a region of size 2^{n_i}, but then we only allocate space which is a multiple of the subregion $2^{n_i-3}k,\ k \in \{1, 2, \ldots 7\}$ (assume that k is not zero or eight, as in both cases no memory is saved with this technique). When we go to place an MPU region on the task that follows it in memory, j, if $w_i = w_j$ then we have broken the alignment necessary to place the MPU region over j, as j's stack explicitly begins between the 2^{n_j} aligned bytes as $0 < 2^{n_i-3}k < 2^{n_i} = 2^{n_j}$.

Chapter 4

Implementation

In this section, we describe the implementation of our designs.

4.1 Interleaved Shadow Stack

To enforce stack size sorting in memory, we updated the linker file to place stack and shadow stack buffers in RAM by their defined size. Stack and shadow stack buffers are statically allocated and marked with variable attributes. The linker file uses the attributes to determine where to place the buffer in memory. For our prototype, since we know stack sizes statically, we manually computed the shadow stack offset by taking the sum of all task stack sizes. We then pass this value as a flag to the Kage compiler for use in shadow stack instrumentation.

4.2 Shared Shadow Stack

To meet the requirements of the shared design's functional correctness, we implemented task return mechanisms and scheduling predicates.

4.2.1 Task Return Mechanisms

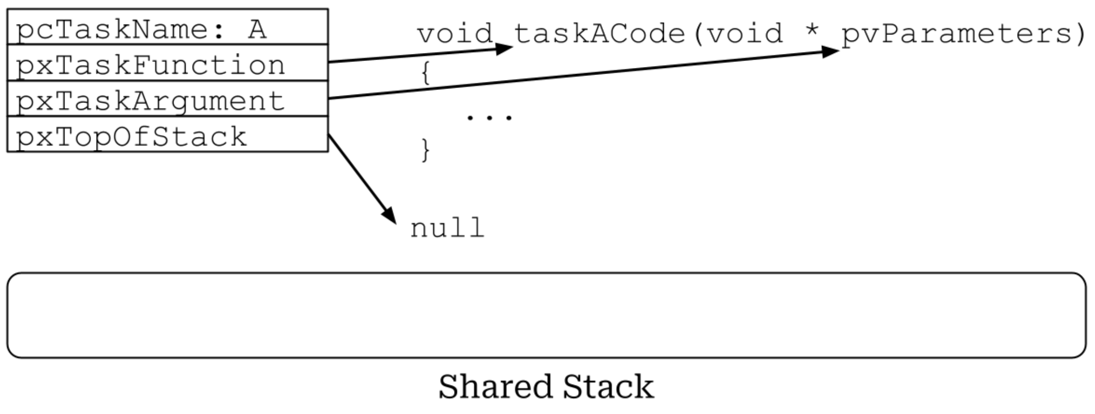
In FreeRTOS on task initialization, the stack frame for a task is prepared as if it were context switched out of. Eight words are placed at the top of the stack which determines the initial processor state of the task. These eight words are called the basic frame, and are managed on exception entry and exit. The eight words placed are `R0`, `R1`, `R2`, `R3`, `R12`, `R14` (the link register), the start address of task code, and `xPSR`. This ensures that context switching into the task works even if that task has never run before. The link register holds the return value for the task code entry function and task initialization sets this value to null. This generates a hard fault on tasks attempting to return from their task code.

For the shared shadow stack design, tasks do not have stack space allocated ahead of time, so this stack frame preparation happens when the task first runs. We add two initial processor state values onto the task control block for use in this stack frame preparation—the pointer to task code and the pointer to the task's parameters (`R0`). We re-purpose the pointer to the top of the stack by initially setting it to null. The task control block sits in privileged data and belongs to the trusted kernel, so our security guarantees are not violated. An example of this is seen in figure 4.1.

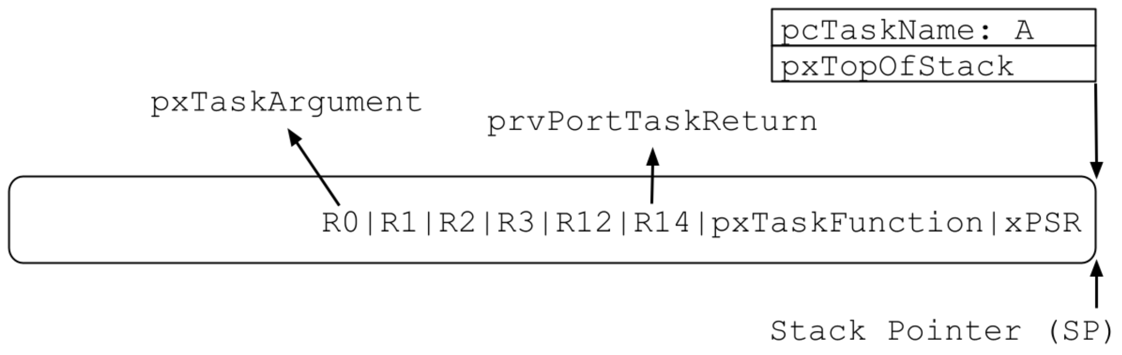
As a result of our scheduler constraints, we have two cases to address for swapping tasks.

Case 1. *A task is preempted. A context switch occurs.*

Because tasks never block, context switches are *always* from a lower priority task to a higher priority task. We save the context of the current task and set up the stack for the new higher priority task. We set the pointer to the stack in the new task's TCB to the current stack pointer. We then push the basic frame onto

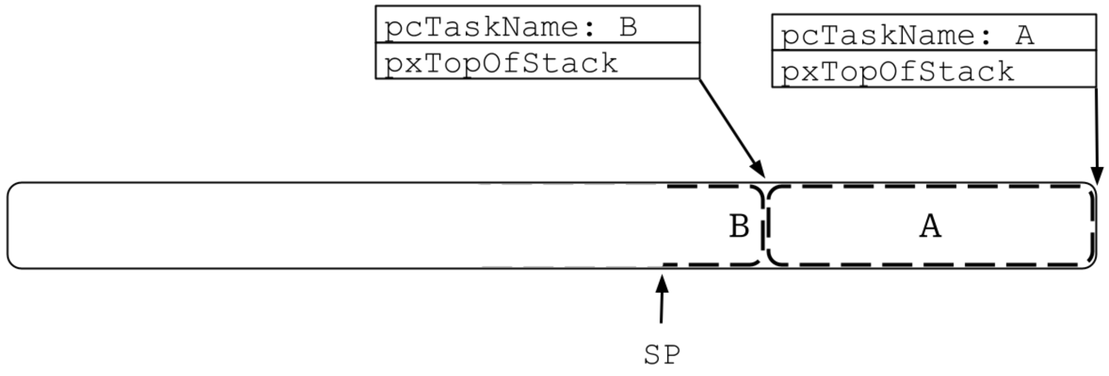


(a) Here, no task has yet run so the shared stack is empty. The task control block contains two new pointers, `pxTaskFunction` and `pxTaskArgument`, and `pxTopOfStack` is set to null.

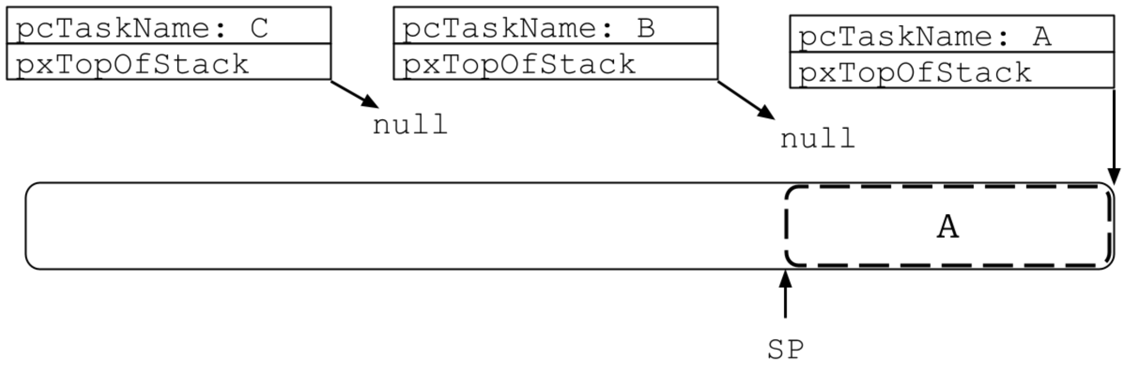


(b) When task A is scheduled, the basic frame is prepared and `pxTopOfStack` is set in the TCB. The stack pointer points to the top of the shared stack as no task has run yet.

Figure 4.1: Visual representation of the task return mechanism.



(c) Here, task B has preempted task A. The stack pointer follows B as the stack grows down. `pxTopOfStack` points to the top of B's stack usage and the bottom of task A's usage.



(d) If Task B finishes, the kernel resets the stack pointer to the value of task B's `pxTopOfStack`. If A were to be scheduled next, the kernel checks `pxTopOfStack`, finds it to be non-null, and loads the context from the stack pointer which is saved at the bottom of A's stack usage. If C were to be scheduled next, the kernel sees `pxTopOfStack` is null and prepares the stack frame like in 4.1b.

Figure 4.1: Visual representation of the task return mechanism (cont.).

the stack. Instead of null for the link register (`R14`), we place the address of a new function, `prvPortTaskReturn`, the task return handler. This task return handler generates a supervisor call (SVC) to the kernel indicating that it must handle a task return event.

Case 2. *A task completes. The task returns to the kernel.*

In a task return event, we reset the stack pointer to the stack address in the task's TCB. This brings the pointer back to before the task had run. We zero out the stack address in the TCB, and suspend the task if it had a queue predicate placed on it. We then have another two possible cases.

Case 2.1. *The highest ready task is the task currently lowest on the stack.*

Case 2.2. *A new task has become ready that is of priority lower than the completed task, but higher than the task saved on the stack.*

FreeRTOS's preemptive scheduler gives us the task with the highest priority which is ready, but we do not initially know which of the two cases the task belongs to. We determine if a task has run—and therefore has stack space in use—by checking the pointer to the stack in the new running task's TCB. If it is null, the task has not been run and must have the stack frame prepared. If the pointer is non-null, it indicates the task has stack space allocated so we restore the context at the current stack pointer. Our scheduler requirements ensure that the running task must be the lowest on the stack, so we can safely resume from the current stack pointer.

4.2.2 Task Predicates

Currently, the only task predicate we have implemented is a queue predicate. Queue predicates are implemented using FreeRTOS's existing functionality. We add a function called `xQueuePredicate` which takes as arguments a task handle and a queue. This moves the task to the suspended list, and adds the task to the queue's list of tasks waiting to receive. When the queue receives data, it moves the dependent task out of the suspended list and into the ready list. This requires no update to code to send data through a queue. As a consequence of queue predicates, when calling `xQueueReceive`, data will always exist in the queue. This allows us to remove many of the checks and yielding code from within `xQueueReceive`.

For multiple task predicates, the only currently supported operand is `OR`. More boolean expressions may be added with some extra engineering time.

4.3 Other Changes to Kage

We optimized the hand-coded assembly used to store processor state to the shadow stack in a context switch. Kage uses many individual `ldr` and `str` instructions which we convert to `ldm` (load multiple) and `stm` (store multiple) by changing the ordering of context store and restore. This optimization reduces the total number of instructions from the previous work. We apply this optimization to all three designs.

We also fix a compilation issue which applies Kage compiler's store protections to the linked Newlib library, even on the baseline FreeRTOS benchmarks.

Chapter 5

Methodology

To evaluate the performance of our designs, we utilize Kage's modified version of the CoreMark benchmark [23]. We manually measure the memory usage for each trial and compare between designs. In order to further understand the performance implications of our implementations, we create and run our own microbenchmarks which record cycle counts with the KIN1 library [24].

5.1 Updates to Kage's CoreMark

CoreMark provides an effective measurement of holistic runtime performance on an average embedded application behavior. Kage takes the CoreMark benchmark and updates it to include measurements of FreeRTOS features such as inter-process communication and context switching. Kage runs all benchmark code inside FreeRTOS's Daemon task. This task manages timers and thus allocates more stack space than needed for benchmarks. We move benchmark code out of the Daemon task and into a new, empty task.

We use four different types of tasks when evaluating with CoreMark: FreeRTOS's idle task, Amazon FreeRTOS's logging task, a coordination task, and Kage's reformulation of benchmark tasks, which adds IPC and context switching. The idle task is standard in all FreeRTOS applications and handles cleanup of deleted tasks. The idle task has the lowest priority of `0`. The logging task handles console output and sends results over UART. The logging task has the highest priority at `6`, but will only run when a task has sent a string to print to the logging task's queue. The coordination task creates the benchmark tasks, times their execution, records the results, and sends strings to the logging task. The coordination task has the second highest priority at `5`. The benchmark tasks run the CoreMark evaluation. The number of benchmark tasks depends on the requested number of threads. CoreMark 1-thread only creates one benchmark task, CoreMark 2-thread has two benchmark tasks, and CoreMark 3-threads has three benchmark tasks. The priority of these benchmark tasks are assigned in descending order at `4`, `3`, and `2` respectively. This means the benchmark tasks *do not round-robin*. We choose this due to requirements of the shared scheduler. In order to generate a fair evaluation, we do not want round-robin scheduling to hinder the performance of the baseline, parallel, and interleaved designs. We apply one change to the shared design not present in the other evaluations. The coordination task is split into two separate tasks: coordination start and coordination end. The coordination start task has the same priority at `6`, creates the benchmark tasks, starts the timer, and then it returns. The coordination end task has a lower priority than all of the benchmark tasks at `1` and waits from them to finish. The coordination end task then stops the timer, collects the results and passes them to the logging task. The reason for this is to fully utilize the shared stack design and run tasks as sequentially as possible. An analysis of this reasoning is in section 6.1.

5.2 Deciding Stack Usage

To decide the stack space that must be allocated to each task, we run a high watermark analysis on the benchmark. We fill the stacks with known debugging bytes and take the average usage after multiple runs. Using these recorded high watermarks, we reduce that stack allocations to as small as possible such that the program still runs correctly.

5.3 Microbenchmark Selection

For purposes of understanding the impacts to specific parts of FreeRTOS that our implementation makes, we want to record cycle counts for specific code regions. In particular, we have made large changes to queues and context switching in the shared design, small improvements to context switching in the parallel and interleaved design, and other minor revisions from the previous prototype. We use the KIN1 library [24] to record cycles.

5.4 Replacing `printf`

We find the `printf` function acts improperly on our device. Primarily, floating point operations with `printf`'s float format specifier do not utilize the board's existing floating point hardware and instead use software computations. This introduces error during console printing which propagates to make the output meaningless. We replace calls to `printf` by sending static strings and raw data over the board's universal asynchronous receiver-transmitter (UART). This has the knock-on effect of dramatically reducing the code size and stack usage of the benchmark binaries.

Chapter 6

Evaluation

In this section, we evaluate the performance of our two shadow stack designs. First, we use the CoreMark benchmark [23] to simulate realistic application code. Then, we use microbenchmarks to explore the impact of individual components.

We chose Amazon FreeRTOS v1.4.9 [19] and LLVM 9.0 [6]. We use an STM32L475 Discovery board [25] for all experiments. This board contains an ARMv7-M [16] 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.

6.1 Stack Usage Analysis

To compare the stack usage of different designs, we calculated the total allocated stack and shadow stack space for each design. Table 6.1 shows those values and table 6.2 shows the percentage difference from the parallel design. The parallel design and interleaved design both grow linearly in allocation when adding a new benchmark task, so the difference remains constant. The shared design however has a constant allocation size, so the difference grows linearly.

Stack Allocations in Bytes					
	Parallel	**Interleaved**		**Shared**	
	Allocated	**Allocated**	**Saved**	**Allocated**	**Saved**
One task	5120	3840	1280	2048	3072
Two tasks	6144	4864	1280	2048	4096
Three tasks	7168	5888	1280	2048	5120

Table 6.1: We record the total sum of allocated stack and shadow stack space including the kernel stack and shadow stack. We then record the the amount of bytes saved compared to the parallel shadow stack design.

Percent Savings Compared to Parallel Design		
	Interleaved	**Shared**
One task	25.00	60.00
Two tasks	20.83	66.67
Three tasks	17.86	71.43

Table 6.2: Since the amount saved remains constant for the interleaved design, the percent goes down as the total size goes up. Conversely, the shared design saves more as the total goes up.

Observation 1. *The task workflow and memory requirements affect the stack usage of different designs and should be considered when selecting a design.*

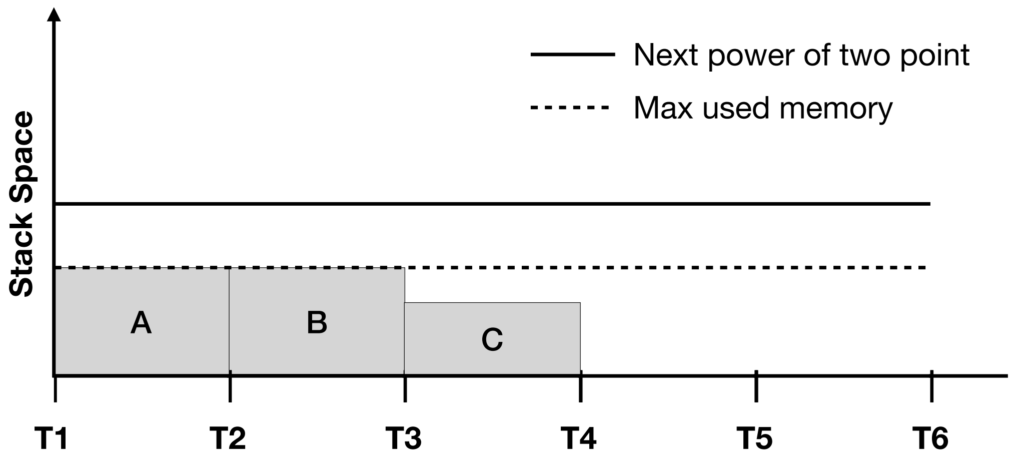
The number of tasks concurrently on the shared stack significantly impacts the stack allocation requirements. Task workflows primarily determine this number, as shown in Figure 6.1.

Observation 2. *The interleaved and shared designs both offer significant stack space savings compared to the parallel design.*

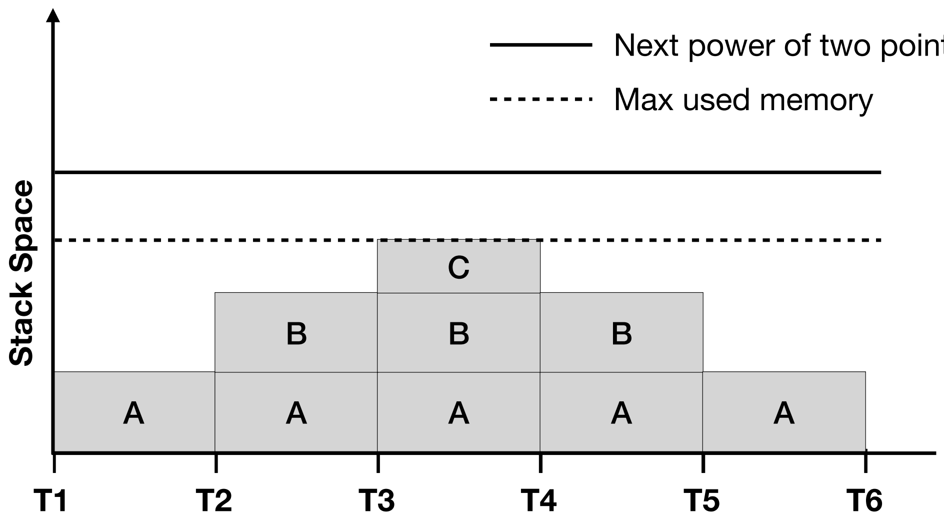
Specifically, the interleaved design reduces stack usage by at least 17.86% and up to 25.00%, while the shared design saves between 60.00% and 71.43% of stack space. The CoreMark benchmark task requires 512 bytes memory in the parallel and interleaved design. Increasing the number of benchmark tasks does not change the memory savings of the interleaved design. The experiments match our best case scenario in figure 6.1a for the shared design which allows the allocated space to remain constant, regardless of how many benchmark tasks we add.

6.2 CoreMark Runtime Comparison

We report results in terms of the number of iterations per second (Iter/sec) where an iteration represents a unit of work completed by CoreMark. Each CoreMark benchmark task computes 2000 iterations. Overhead was measured as a decrease in the number of iterations per second. For each shadow stack design, we show two sets of results, one with store hardening and one without. *Store hardening is not optional.* A user cannot obtain the security guarantees offered by these designs without store hardening. We record separate measurements in order to understand what directly contributes to performance overhead.



(a) In this task workflow, each task completes before another task runs. The shared stack's usage is then determined by the largest task.



(b) In this workflow, task A and B are preempted before they can complete. The shared stack's usage then becomes the sum of all tasks.

Figure 6.1: These two task execution patterns produce different stack allocation requirements as they change the high watermark of the shared stack.

Iterations per Second							
	FreeRTOS	**Parallel**		**Interleaved**		**Shared**	
	(Baseline)	**No SH**	**SH**	**No SH**	**SH**	**No SH**	**SH**
One task	183.49	179.24	173.90	179.24	173.85	179.24	173.91
Two tasks	183.62	179.12	173.91	179.12	173.89	179.24	173.88
Three tasks	183.61	179.12	173.91	179.12	173.89	179.23	173.91

Table 6.3: The recorded iterations per second for the baseline unmodified FreeRTOS, and each design with and without store hardening (SH). Caching is enabled in these evaluations.

Percent Overhead Compared to Baseline						
	Parallel		**Interleaved**		**Shared**	
	No SH	**SH**	**No SH**	**SH**	**No SH**	**SH**
One task	2.31	5.23	2.31	5.25	2.31	5.20
Two tasks	2.46	5.29	2.46	5.30	2.39	5.31
Three tasks	2.45	5.29	2.44	5.30	2.38	5.29

Table 6.4: The runtime overhead percents are all very similar for each design.

Table 6.3 presents a summary of the results obtained from the CoreMark benchmark. It also includes the performance of the baseline FreeRTOS as a comparison.

The main factor contributing to run time overhead is store hardening. For instance, the average runtime overhead amounted to 2.39% without store hardening so the remaining 2.88% of the 5.27% overhead can be attributed to store hardening. It is crucial to emphasize that all designs' security guarantees are dependent on the collective functioning of these mechanisms, rendering none of these components optional.

Observation 3. *Parallel and interleaved designs exhibit similar performance overhead, with differences ranging from 0.0% to 0.02%.*

This is because the instrumentation that accesses the shadow stack is exactly the same in both designs. The arrangement of the interleaved design does not add extra computation over the parallel design except in a specific edge case.

Observation 4. *The shared design has similar performance overhead as the parallel and interleaved designs, with differences ranging from 0.0% to 0.07%.*

Again, the shared design uses the same instrumentation to access the shadow stack as the parallel design. To further understand why we end up with the same runtime values, we examine microbenchmarks.

6.2.1 Shadow Stack Offset Runtime Concerns

Store instructions can only accept an immediate offset between 0-4095. When the shadow stack offset value is greater than 4095 bytes, an additional instruction must be inserted into each prologue and epilogue in untrusted code as seen in figure 6.2. This adds an approximately 2% runtime performance penalty when two instructions added to each untrusted function.

Observation 5. *All designs will encounter an approximately 2% runtime performance penalty when the shadow stack offset becomes larger than 4095 bytes. The interleaved design is more likely to go above this value and the parallel design is least likely.*

The shadow stack offset in the parallel design is determined by the stack space allocated to all tasks, which is in turn determined by the stack with the largest requirements. The interleaved design's shadow stack offset is determined by the sum of all task stack allocations. *There are no instances in which the interleaved design would have a smaller shadow stack offset than the parallel design.* Conversely, there

```
Function Prologue

str.w   lr, [sp, #4092]
------------------------------
mov     ip, #4096
str.w   lr, [sp, ip]
```

Figure 6.2: The above store can access the shadow stack with one instruction, while the below one must use two instructions. 4092 is used because it is 4-byte aligned.

exists scenarios where the parallel design uses one shadow stack access instruction and the interleaved uses two.

The shadow stack offset of the shared stack design depends on the task workflow as seen in figure 6.1. In the best case scenario, it can potentially be smaller than the parallel design due to key feature 5. In the worst case it is equivalent or worse than the interleaved design.

6.3 Microbenchmarks

We built a set of microbenchmarks and measure cycle counts using the KIN1 Library [24] to further understand our holistic runtime evaluation. Table 6.5 shows the cycle counts for segments of FreeRTOS that were updated to make the designs possible.

Observation 6. *The shared design does not perform equally well in every area comparatively, it only matches holistically.*

One measurement that immediately stands out is the dramatically reduced cycle count for queue sending and receiving. Because we use queue predicates to ensure that queues always have data before a task executes, we can remove all of the code that checks if the queue has data and all of the code that causes the task to

Cycle Counts								
	FreeRTOS		**Parallel**		**Interleaved**		**Shared**	
	No MPU	**MPU**	**No SH**	**SH**	**No SH**	**SH**	**No SH**	**SH**
Context switching	174	195	259	259	259	259	177	177
Queue: create	543	720	748	838	748	838	745	845
Queue: send and receive	2053	2698	3386	3590	3365	3558	287	307
Task return: resume	NA	NA	NA	NA	NA	NA	294	351
Task return: new task	NA	NA	NA	NA	NA	NA	304	321

Table 6.5

block if the queue does not have data. The queue predicate then requires extra instrumentation which we see reflected in the task return benchmarks. Context switching appears to be significantly faster for the shared design, but a task in the shared design must also eventually return. On average, swapping in and out a task is slower in the shared design. These areas of increased and reduced runtime performance average out in the CoreMark benchmark.

Chapter 7

Related Works

We examine previous work from SoK [26], Zipper Stack [27], and μRAI [28].

SoK proposes and evaluates the parallel shadow stack design along with a compact shadow stack design which uses either a register, segment, or global variable to point to the shadow stack. The compact design offers significantly condensed shadow stacks, but memory accesses through segments and global variables are considerably slower.

Not only are our shadow stack designs novel compared to SoK, but the difference in architectural targets mean our shadow stacks work on embedded devices without a MMU, which SoK relies on.

Zipper Stack takes a similar approach to focusing on memory savings by doing away with a shadow stack entirely. Zipper Stack cryptographically verifies return addresses at runtime using hardware hashing modules. Performance of Zipper Stack heavily depends on the hardware module, but achieves good results when it is available. Zipper Stack also targets x86, but extensions to ARM are theoretically possible using Pointer Authentication (PAC).

μRAI targets embedded devices and also provides an alternative to a shadow stack. Instead, return addresses are saved in the code segment at compile time and a reserved register points to the correct return address for the currently executing function. This technique does move the memory constraint from RAM into Flash, but in doing so, incurs a memory access penalty to performance.

Chapter 8

Conclusion

We proposed, implemented, and evaluated two novel shadow stack designs for use in embedded memory safety designs targeting ARMv7/8-M. The interleaved shadow stack design allows for stacks of different sizes through an alternative shadow stack placement than Kage's parallel shadow stack design. The shared stack design has all tasks use a single stack and, despite changes to scheduling, matches the performance overheads of Kage. Our designs save between 17%-71% memory in RAM compared to Kage's parallel shadow stack design. At the same time, we observe minimal impact to runtime performance with differences of 0-0.07%. We also observe that to maximize memory safety, no one design fits all. The best choice of a shadow stack design depends on the task workflow of the application and should be evaluated at development time.

Future directions for the work include extending the task predicate capabilities of the shared design to support alternative predicates and arbitrary boolean expressions, optimizations to the queue predicate functionality to reduce cycle counts in the task return mechanism, and other shadow stack designs which reduce the size of the shadow stack allocated for each task.

Bibliography

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, November 1996.

[3] J. Afek and A. Sharbani, "Dangling pointer: Smashing the pointer for fun and profit," 2007.

[4] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow protection on Real-Time embedded systems with kage," in *31st USENIX Security Symposium (USENIX Security 22)*, (Boston, MA), pp. 2281–2298, USENIX Association, Aug. 2022.

[5] A. W. Services, "FreeRTOS," 2023. https://www.freertos.org/.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, (San Jose, CA, USA), pp. 75–88, Mar 2004.

[7] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, (New York, NY, USA), p. 179–190, Association for Computing Machinery, 2012.

[8] T. Giannetsos, T. Dimitriou, I. Krontiris, and N. R. Prasad, "Arbitrary code injection through self-propagating worms in von neumann architecture devices," *The Computer Journal*, vol. 53, no. 10, pp. 1576–1593, 2010.

[9] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), p. 15–26, Association for Computing Machinery, 2008.

[10] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (USA), p. 385–399, USENIX Association, 2014.

[11] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*, pp. 575–589, 2014.

[12] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 957–972, USENIX Association, Aug. 2014.

[13] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, mar 2012.

[14] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, (Berlin, Heidelberg), p. 121–141, Springer-Verlag, 2011.

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, (USA), p. 12, USENIX Association, 2005.

[16] A. Holdings, "Armv7-m architecture reference manual," 02 2021.

[17] A. Limited, "Armv8-m architecture reference manual," 12 2022.

[18] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," 2019.

[19] "Amazon FreeRTOS." `https://aws.amazon.com/freertos/`.

[20] R. Barry, *Mastering the FreeRTOS™ Real Time Kernel*. Real Time Engineers Ltd., 2016. `https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf`.

[21] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, may 2019.

[22] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, pp. 67–99, 1991.

[23] "Coremark: An EEMBC benchmark." `https://www.eembc.org/coremark/`.

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

[25] STMicroelectronics, *RM0351 Reference Manual: STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm-based 32-bit MCUs*, 9 ed., Jun 2021.

[26] L. Szekeres, M. Payer, T. Wei, and D. X. Song, "Sok: Eternal war in memory," *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, 2013.

[27] J. Li, L. Chen, Q. Xu, L. Tian, G. Shi, K. Chen, and D. Meng, "Zipper stack: Shadow stacks without shadow," 2020.

[28] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "µrai: Securing embedded systems with return address integrity," in *Network and Distributed System Security Symposium*, 2020.