

Enclaves in Real-Time Operating Systems

Alex Thomas



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-134

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-134.html>

May 15, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I want to thank my family for all the support and love they have given me. I also want to thank Dayeol Lee, Stephan Kaminsky, and David Kohlbrenner for the initial design discussion of ERTOS. Additionally, I want to thank Stephan Kaminsky for updating the build system of ERTOS, developing IO support for "uart", and for creating a command-line interface for ERTOS. I want to thank Professor Krste Asanovic and Professor John Kubiawicz for mentoring and advising me throughout this year. Finally, I want to thank Debbie Yuen for always believing and encouraging me.

Enclaves in Real-Time Operating Systems

by

Alex Thomas

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering & Computer Science

in the

Graduate Division

of the


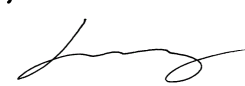
University of California, Berkeley

Committee in charge:

Professor Krste Asanovic, Chair

Spring 2021

The dissertation of Alex Thomas, titled Enclaves in Real-Time Operating Systems, is approved:

Chair		Date	5/14/2021
		Date	5/15/2021
		Date	

University of California, Berkeley

Enclaves in Real-Time Operating Systems

Copyright 2021
by
Alex Thomas

Abstract

Enclaves in Real-Time Operating Systems

by

Alex Thomas

Master of Science in Electrical Engineering & Computer Science

University of California, Berkeley

Professor Krste Asanovic, Chair

With the growing popularity of edge computing and Internet of Things (IoT) devices, there is an increased need for secure computation on embedded devices. Typically, embedded devices have a heterogeneous environment and do not have general security protections compared to hosts on the cloud. As we see more third-party libraries and applications being run on embedded devices, we face the risk of system compromise that even the device's RTOS kernel cannot protect. There is a need for creating Trusted Execution Environments (TEEs) on embedded devices; however, many current TEEs have expensive hardware requirements. We propose using Keystone, a framework for creating customizable TEEs, on RISC-V architectures. The hardware requirement for creating TEEs in Keystone are generally available on standard RISC-V devices as RISC-V already provides PMP registers, the basis of Keystone's isolation. We propose using Keystone with FreeRTOS to implement a module in FreeRTOS for creating efficient and dynamic TEEs on embedded devices. We introduce ERTOS, a new module to FreeRTOS that allows the creation of secure tasks that can be attested and strongly isolated from other tasks using Keystone's security monitor. ERTOS exposes an easy-to-use API that allows developers to create and run enclave-protected tasks. ERTOS adds negligible performance overhead for computation-intensive tasks inside an enclave and introduces optimizations to allow inter-task communication to be more efficient.

Contents

Contents	ii
List of Figures	iii
List of Tables	iv
1 Introduction	1
1.1 Security on Embedded Devices	1
1.2 Trusted Execution Environments on Embedded Devices	2
1.3 Related Work	2
2 ERTOS	6
2.1 Background	6
2.2 Design Overview	8
2.3 Bootloading	9
2.4 Enclave Binary Format	11
2.5 Task Registration	12
2.6 Scheduling	12
2.7 Interrupt Handling	13
2.8 Message Passing	14
2.9 Unprotected Tasks	15
2.10 Implementation	16
3 Evaluation	19
3.1 MicroBenchmarks	19
3.2 General Computation Benchmarks	22
3.3 Synthetic Benchmarks	24
4 Conclusion	29
4.1 Future Work	30
Bibliography	31

List of Figures

1.1	Visualization of ARM TrustZone and Multizone’s IOT framework [21] [22]. Note: The ARM TrustZone model visualized is specifically Cortex-A	4
2.1	Visualization of Keystone’s Software and Hardware Stack	6
2.2	Design Overview of ERTOS	9
2.3	Bootloading Process	10
2.4	Embedded Device Framework Layout	11
2.5	Enclave Scheduling Diagram. E-Task and U-Task refer to Enclave Task and Unprotected Task respectively.	12
2.6	Shared memory for enclave to enclave communication	14
2.7	Message Passing Visualization	15
3.1	Message Passing Results Graph	21
3.2	RV8 Benchmark Graph - Part I	23
3.3	RV8 Benchmark Graph - Part II	23
3.4	Illustration of using a hybrid computing approach, where some planners are executed directly on the robot.	25
3.5	Visualization of the Synthetic Benchmark	26
3.6	RL Benchmark Graph	27
4.1	Hardware Agnostic ERTOS	29

List of Tables

3.1	Asynchronous Message Passing Benchmarks	20
3.2	Message Passing Optimization Results	21
3.3	RV8 Benchmark Results	22

Acknowledgments

I want to thank my family for all the support and love they have given me. I also want to thank Dayeol Lee, Stephan Kaminsky, and David Kohlbrenner for the initial design discussion of ERTOS. Additionally, I want to thank Stephan Kaminsky for updating the build system of ERTOS, developing IO support for *uart*, and for creating a command-line interface for ERTOS. I want to thank Professor Krste Asanovic and Professor John Kubiawicz for mentoring and advising me throughout this year. Finally, I want to thank Debbie Yuen for always believing and encouraging me.

Chapter 1

Introduction

1.1 Security on Embedded Devices

With the rise of Internet of Things (IoT) devices, there is an increased amount of data being collected, from temperature information to audio input. Because of the ubiquity of IoT devices, a user will be required to trust more and more devices [18]. Moreover, some of these devices are critical to the safety of human lives like automobile or medical devices.

Unfortunately, many IoT devices do not put adequate effort into security and have a wide range of vulnerabilities [13]. We are starting to see attacks targeting embedded devices today. By using a flying drone, a group of researchers were able to compromise a Tesla vehicle using a privileged escalation attack on ConnMan, a commonly used embedded application to manage internet connections [26].

With real-time embedded devices, we are seeing the increased use of third-party applications like crypto libraries, intrusion-detection software, and more [30]. Large manufacturers of smart devices like Google [9], Samsung SmartThings [24], and Amazon [1] have all adopted third-party application support.

To expand in more detail, let us focus on Amazon's smart devices, Amazon Alexa. Alexa allows the use of third-party applications using *Alexa skills*. *Alexa Skills* extends voice functionalities to specific third-party services. As of 2021, Amazon Alexa has over 100,000 skills available on their Amazon Marketplace [1]. With over 100,000 third-party applications available, a strong vetting process is required to protect users from intentional or unintentional bugs that may be harmful; however, detecting bugs on a third-party application is a difficult problem. third-party application with a vulnerability may remain undetected in the vetting process. In fact, it was found that the vetting process was already insufficient to detect developer mistakes for both Google Home and Amazon Alexa for authentication [10].

1.2 Trusted Execution Environments on Embedded Devices

With more platforms supporting third-party applications, there is increased need to isolate these applications. A general purpose OS would typically use a hardware memory management unit to provide memory isolation, but in the more resource-constrained environment of embedded devices, the memory isolation is ignored for more efficient performance.

As we move towards a world with more and more edge devices, we must be more cognizant of the security implications and privacy risks that these devices possess. There is a need for **strong isolation** and **secure computation** on embedded devices. Furthermore, we cannot rely on the security of the RTOS kernel as previous work found protection measures in place for the FreeRTOS kernel deficient [19]. Moreover, an application must inherently trust the large code base of the RTOS kernel. A large privileged code base increases the possibility of finding a vulnerability in the code. A solution to protect embedded device application should also isolate the RTOS kernel and should be protected against an adversary kernel.

We also need a method to verify the identity of an application running on an embedded device. This operation maybe useful to tasks running on an embedded devices to authenticate each other, or for providing proof to the embedded device manufacturer (i.e. Samsung), that a device is running the correct software.

We propose using Trusted Execution Environments on embedded devices. This fits our need for strong isolation and secure computation guarantees. Furthermore, TEEs provide mechanisms to attest an application to verify that the application running inside a TEE is not modified and is running on trusted hardware.

1.3 Related Work

There are several hardware back-ends to provide TEEs, but most of them are not suitable for embedded devices. There are also other memory protection mechanisms that provide similar security guarantees.

Memory Protection Unit

An existing solution for memory isolation is to use a Memory Protection Unit (MPU) to protect tasks running on embedded devices. MPUs can be configured to protect the privileged real-time OS (RTOS) and any non-privileged isolated tasks. MPUs are configured by the privileged RTOS, so any attack, like privilege escalation attacks, that compromises the RTOS kernel will render the MPU useless [32]. What we need is a stronger isolation mechanism where even a compromised RTOS kernel cannot cause harm to our system.

Software Fault Isolation

Software fault isolation is a form of sandboxing that rewrites an application to perform in a safe way, ensuring that it only accesses or modifies memory within its own memory space. The current state of the art solution for SFI, Google’s Native Client (NaCL), reports from a 5% to up to a 9% overhead in performance [31]. One limitation of SFI is that it increases the code-size of a binary, which can lead to increased instruction cache pressure. Furthermore, there are other memory overhead issues introduced by guard pages and forced instruction alignment, which are problematic in a memory-constrained environment [31]. Ideally, we could run applications with negligible computation overhead while also guaranteeing application confinement.

Intel SGX

In the context of embedded devices, Intel SGX is generally not a popular choice for hardware enforced security due to its significant hardware costs. Intel SGX heavily relies on virtual memory and its **Enclave Page Cache** [7]. The Enclave Page Cache holds pages used specifically for enclaves. The management of these pages are done via the **Enclave Page Cache Map** (EPCM), which contains metadata information of an enclave page such as ownership or page validity. Privacy is guaranteed from the fact that the TLB can only contain entries of the current enclave. Any address access is protected from the EPCM, which can detect whether an enclave page lies outside the current enclave’s page range. If a memory access is to an external enclave page, a modified TLB miss handler will check if the page can be accessed via the EPCM. This is how SGX rejects memory requests that are inside an enclave’s memory range.

Intel SGX relies heavily on virtual memory support and any modifications to remove virtual memory support may result in significant and fundamental changes to the SGX model. Virtual memory is not a popular choice in real-time operating systems because 1) the hardware support to allow virtual memory is expensive and 2) having virtual memory may cause more variability in task completion time, which is simply not suitable in a real-time context. Because of this, SGX is not suitable to be used in a real-time embedded systems context.

ARM TrustZone

ARM has their own TEE framework called TrustZone, which partitions memory into a trusted zone and an untrusted zone. An embedded systems developer can place any privacy preserving computation inside the *Trusted World*, thus all components of an application had to be placed within one zone. Many components are duplicated between the different partitions. For example, ARM TrustZone holds two operating systems, one in the *Trusted World* and another outside of the *Trusted World*.

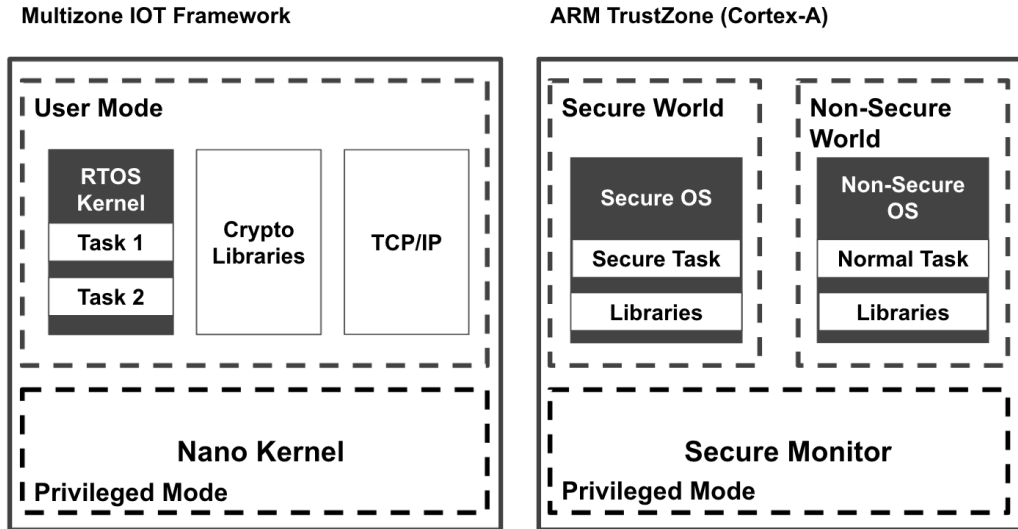


Figure 1.1: Visualization of ARM TrustZone and Multizone’s IOT framework [21] [22]. Note: The ARM TrustZone model visualized is specifically Cortex-A

A secure monitor manages switching between the *Trusted World* and the non-secure world. The processor uses the non-secure bit in the Secure Configuration Registers to determine whether it is in the secure world or not. Entering from the non-secure world to the secure world requires a Secure Monitor Call (SMC). Only a privileged mode can execute a SMC to switch into the secure world [22]. Secure and non-secure isolation is guaranteed because page table entries have a secure bit, which describes whether the page is accessible from the non-secure world. Any address translation from the non-secure world will have its non-secure bit set on, which means any address translation to the secure-world will not be allowed.

Just like SGX, virtual memory reliance is a limitation to TrustZone and makes it unsuitable for real-time systems. Furthermore, the design of TrustZone has several limitations. First, a developer who wishes to modularize their application would find it difficult to do so because of the single-zone architecture. A developer would have to put multiple modules which the application must communicate with in the secure world. This would create a large TCB that is unnecessary. Furthermore, applications within the same zone cannot isolate themselves with each other. This is a significant limitation as we may want to use multiple 3rd party applications in our embedded device that may mutually distrust each other. We provide a diagram of ARM TrustZone’s architecture, specifically Cortex-A, in Figure 1.1 on the right.

Multizone’s IOT Framework

There exists a RISC-V based TEE framework called `Multizone` (created by Hex-5) [21], which improves upon ARM TrustZone and creates multiple zones for each domain of an application. Each zone runs with user privilege. Multizone allows developers to create multiple TEEs that are statically initialized. It introduces a nano kernel, which is responsible for scheduling zones, accessing IO devices, and handling undelegated interrupts [21]. It allows modular development and allows mutual distrusting applications to run on the same system. Different zones can pass messages to each other securely via the nano kernel.

Multizone does not support dynamic TEE creation; all TEEs must be statically defined first. This makes Multizone unsuitable for the scenario where an embedded device might want to download third-party applications from a marketplace like in the case of Amazon Alexa. We provide a diagram of Multizone’s IOT framework in Figure 1.1 on the left.

Chapter 2

ERTOS

2.1 Background

Keystone

Keystone is an open-source framework for creating multiple, customized TEEs based on RISC-V architecture [16]. Keystone uses a privileged *Security Monitor* (SM), which is responsible for creating, deleting, and switching into enclaves dynamically. In order to create TEEs, Keystone utilizes *Physical Memory Protection* (PMP) registers, which act as base and bound registers that seal off memory from other entities including the privi-

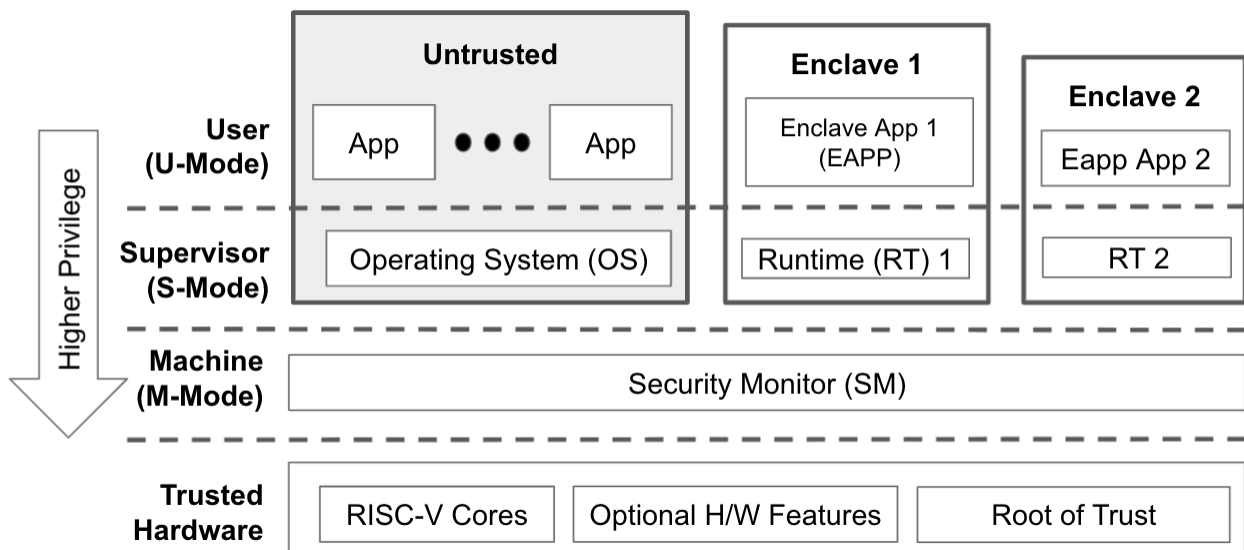


Figure 2.1: Visualization of Keystone's Software and Hardware Stack

leged host OS. These PMP registers are generally available on standard RISC-V machines. Keystone guarantees the confidentiality and integrity of memory within the enclave. We provide a visualization of the different components involved in Keystone in Figure 2.1. Base Keystone does not protect against physical adversaries as pages aren't encrypted and integrity-checked like SGX [7]; however, Keystone does offer software-based page encryption and integrity checks as a module which requires no additional specialized hardware [Andrade2020SoftwareBasedOM].

RISC-V Privileged ISA

RISC-V provides several privilege levels to provide protection across several layers of the software stack. Keystone relies on **Machine**, **Supervisor**, and **User** mode [28].

Machine (M) mode is the highest privilege and is responsible for interacting with hardware, which includes setting up and configuring the PMP registers. This is what privilege the **Security Monitor** executes on. There is no virtualization on this privilege, thus all memory accesses in this mode use physical addresses (assuming the default configuration).

Supervisor (S) mode is the privilege the host OS typically executes on. Keystone's **runtime** runs on supervisor privilege.

User (U) mode is what regular applications without any privilege execute on. All enclave applications run in this mode.

Keystone Components

Security Monitor (SM): This is a small, trusted component, responsible for managing all the enclaves and for creating verifiable reports to prove that an application is running inside an enclave.

Runtime (RT): This component runs in S-mode and provides an interface for communicating with the SM from the user application. The runtime resides inside the enclave and is responsible for setting up the enclave application's environment (i.e. page table initialization).

Enclave Application (EAPP): This is the client's application that runs inside the enclave. For any resources the client requires from the host, it must interact with the runtime, which calls the SM.

Host Operating System: This is the OS of the host that uses the Keystone driver to interact with the SM to create or interact with the enclaves. This component is untrusted.

Virtual Memory Reliance

In Keystone, page tables are managed and configured by the **runtime**, which is part of the enclave. We can completely remove the page table mapping altogether and perform no address translation. In this case, there is no reason to still have a **runtime**. This does not affect the memory isolation of the TEE because the isolation is provided by the PMP registers, which does not assume anything about the virtual memory of an enclave.

This makes modifying Keystone to not use virtual memory a far easier task than SGX or TrustZone as Keystone’s architecture does not rely heavily on virtual memory support.

FreeRTOS

FreeRTOS is an open-source real time operating system owned by Amazon and is a popular choice for embedded device developers. Amazon provides libraries to assist developers in connecting their device to use Amazon Web Services [25]. FreeRTOS has a very light memory footprint; in fact, there are only three source files in the kernel [17]. Furthermore, it provides user applications with useful data abstractions like queues for message passing between tasks and synchronization primitives like a semaphore. FreeRTOS also provides several libraries for creating a TCP/IP stack or for IO support.

Scheduler

FreeRTOS assumes a single core system, thus only one task can be scheduled at a time. [17]. FreeRTOS uses the term *task* to refer to a thread. The scheduler is responsible for context switching the tasks and saving/restoring state upon a context switch. All task control blocks are stored in a list data structure inside the RTOS kernel.

Each task is assigned a priority as FreeRTOS by default uses a preemptive priority scheduling. For tasks that require a hard deadline, the user should give it a high priority. Tasks that have a soft deadline should be given a low priority. In the case of no active user tasks, the idle task will run with priority 0. Although FreeRTOS uses strict priority scheduling, it can be modified to support other scheduling policies like earliest deadline first (EDF).

2.2 Design Overview

In order to create a framework for creating enclaves on embedded devices, we require a scheduler that can meet real time requirements. Keystone was not designed to be a scheduler. Its sole purpose is to create, delete, and switch into an enclave context. One can borrow several components of an RTOS kernel and port it to the SM, but that defeats the original point of the SM as being a light and trusted component that can be easily verified.

We wanted to keep the SM lightweight as it was originally intended, while also providing the user with an RTOS kernel to schedule their tasks. Moreover, we wanted to protect all TEEs even from a compromised RTOS kernel. What we wanted was something similar to the original Keystone design, where the host OS would be responsible for providing the enclaves with system resources and scheduling.

Because we removed virtual memory support, we only have Machine and User mode. This would mean the FreeRTOS kernel, which was originally designed to run in S-mode, must run in user mode. This guarantees that even though the FreeRTOS kernel might be

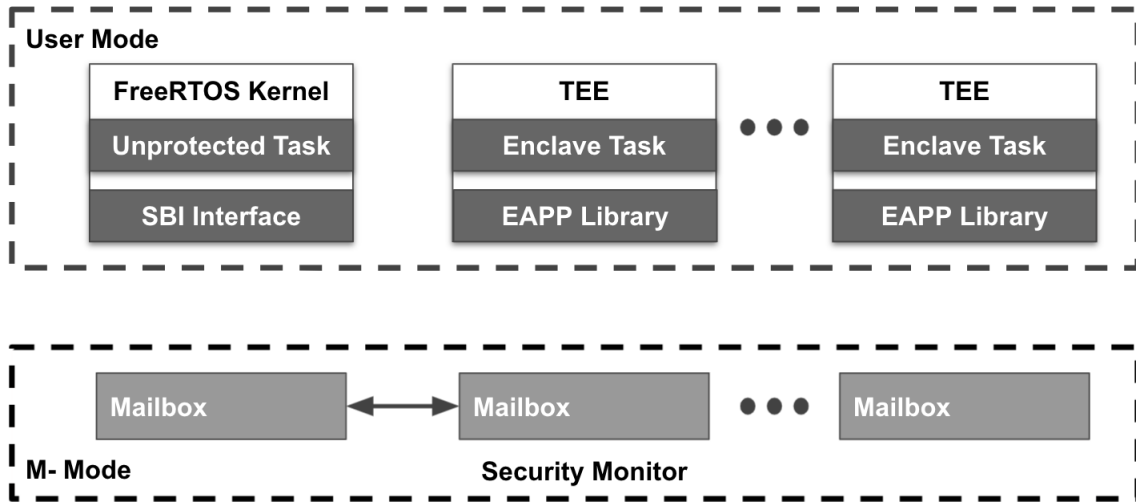


Figure 2.2: Design Overview of ERTOS

compromised, any tasks that are protected by an enclave will continue to be protected. This will virtually stop all privileged escalation attacks for compromising our framework. Furthermore, we minimize our total TCB by not having to trust the RTOS kernel.

I present a framework for creating Enclaves in Real-Time Operating Systems (ERTOS). ERTOS is a module for FreeRTOS that provides an API for creating and running enclaves dynamically, using Keystone’s SM as a backend. A visual overview of our architecture can be seen in Figure 2.2.

We place the FreeRTOS kernel inside an enclave so that an embedded device can be remotely attested by the manufacturer or owner of the device to ensure it is using a correct and unmodified FreeRTOS kernel. We also have two types of tasks, **enclave** or **unprotected** tasks. Unprotected tasks do not have any hardware protection from other unprotected tasks and exist inside the FreeRTOS enclave. Enclave tasks live outside the FreeRTOS enclave and are protected by the SM. Both **enclave** and **unprotected** tasks will be scheduled by the FreeRTOS kernel. We support inter-task enclave communication as each enclave has a 512-byte mailbox that is maintained by the SM. Because message passing is handled by the SM, the SM can verify which enclave is sending a message and ensures the receiving enclave that a message it receives from its mailbox is authentic.

2.3 Bootloading

The bootloading process is visualized in Figure 2.3. We use the **Berkeley Bootloader**, which is a second-stage bootloader that originally booted Linux, but was modified to boot FreeRTOS.

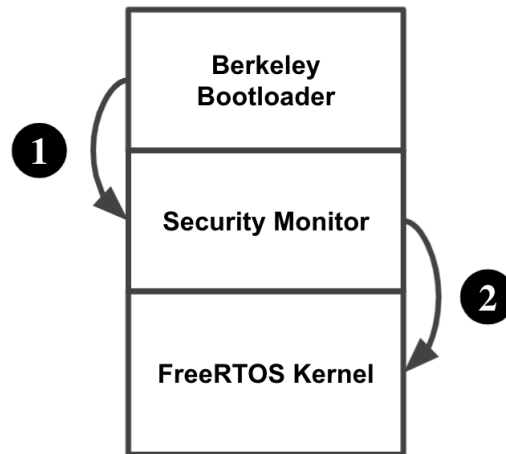


Figure 2.3: Bootloading Process

The only hardware modification Keystone requires is an embedded secret key, H_{sk} . We also assume that the hardware has access to a secure source of randomness. Keystone’s secure boot process involves creating an asymmetric pair of keys, (A_{sk}, A_{pk}) , that is later used for attestation.

❶ Upon CPU boot or reset, the bootloader initializes the security monitor and generates $H(SM)$ (hash of the security monitor), using some cryptographic hash function such as SHA256. SM initialization involves allocating a PMP register and locking access to the SM. Furthermore, the bootloader puts A_{sk} , the attestation secret key and $SIGN(H_{sk}, A_{pk})$, the signature of the public attestation key, inside the SM. With the hardware secret key, the bootloader signs the SM hash and the public attestation key. By doing this, the hardware can later prove the SM was initialized correctly and vouch any future attestation reports.

❷ After the SM is initialized, the SM hashes the FreeRTOS kernel and signs the hash with A_{sk} the secret attestation key. The SM then saves the FreeRTOS kernel hash as well as the signature so that later anyone can verify the SM initialized the FreeRTOS kernel correctly. Furthermore, the SM isolates the FreeRTOS kernel by allocating a PMP register and locking the kernel memory region from external memory access. By doing this, we detect any attacker who may modify the FreeRTOS kernel. Moreover, we prevent any attacker from accessing the FreeRTOS kernel due to the physical memory protection. Once the FreeRTOS kernel is initialized, it is free to create and schedule tasks. The enclave that holds the RTOS kernel is a special enclave that we call the "Enclave Scheduler". The Enclave Scheduler has special privileges over other allocated enclaves. It has the ability to trap into the SM to create enclaves, to switch into other enclaves, and to enable or disable interrupts. The SM keeps track of which enclave is currently running and can verify the origin of any SBI call.

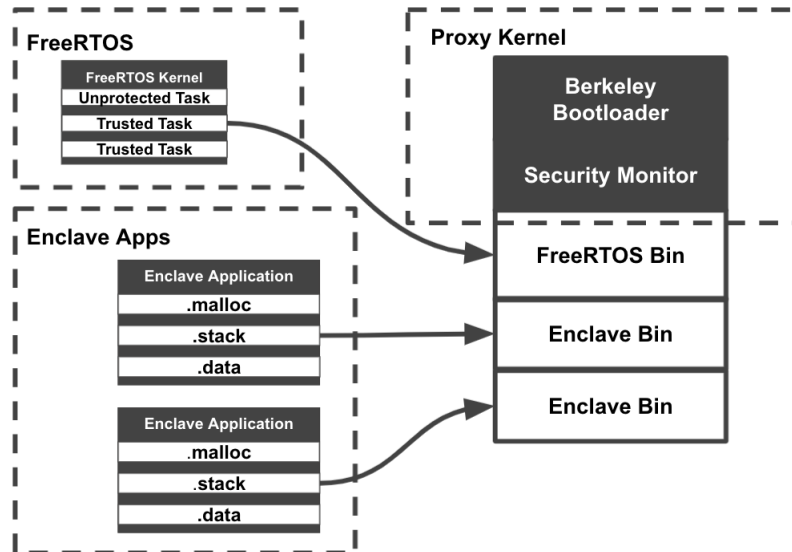


Figure 2.4: Embedded Device Framework Layout

2.4 Enclave Binary Format

Each enclave application binary is compiled separately beforehand with a statically allocated stack and heap segment in ELF format. Any user application must use our provided linker-script to compile their application. The linker script specifies the size and location of the heap and stack. Furthermore, any ELF sections in the binary that are marked 'NOBITS', most notably the .bss section, must be allocated beforehand. Typically, the OS loader will expand any 'NOBITS' section in memory.

Our goal was to have each enclave application be run directly without any OS loader intervention. This minimized memory usage, as an OS loader would have to copy the contents of the ELF file and expand any 'NOBITS' sections to another contiguous memory address. Having a pre-loaded application binary allow us to directly run the ELF file without wasting memory to copy the contents to another space.

We combine the boot loader, FreeRTOS kernel and any subsequent enclave applications into a single binary. Our binary format is visualized in Figure 2.4. We use the RISC-V proxy kernel [23], which contains the Berkeley Boot Loader and our security monitor. We modified the proxy kernel to boot up FreeRTOS instead of Linux. Our FreeRTOS kernel contains all the source files of FreeRTOS, including any unprotected tasks that are not required to be in a TEE. Lastly, we have the enclave applications, which are all tasks that will be stored in a TEE. All three of these components are compiled separately (the enclave applications are further compiled separately) and are appended together to create a single binary that is uploaded to the embedded device.

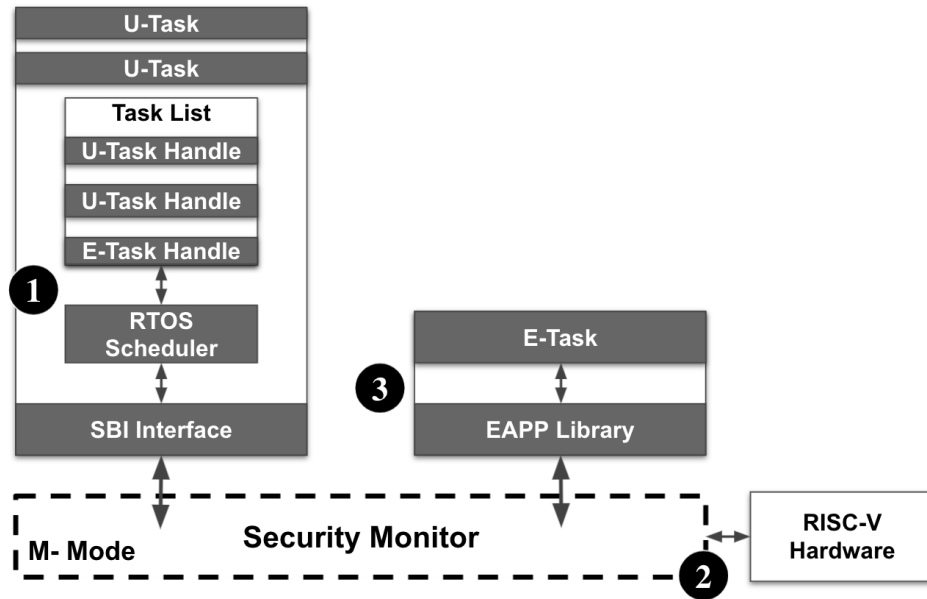


Figure 2.5: Enclave Scheduling Diagram. **E-Task** and **U-Task** refer to Enclave Task and Unprotected Task respectively.

2.5 Task Registration

In order for an enclave task to be scheduled, it must first be registered with the SM. Registering an enclave task to the SM consists of sending the SM a request to create an enclave. The request will include the enclave entry point and the base and size of the enclave application. Using the base and size of the enclave application allows the SM to allocate a PMP register to lock the enclave application. The SM will assign the enclave a unique task ID, which will then be returned to the FreeRTOS kernel as a handle to the enclave task. The task id will be used to schedule the enclave task later and is also used for sending and receiving messages. Once the enclave task is created, the FreeRTOS kernel can only switch into the enclave and cannot observe any internal state of the task, such as the registers. The SM does not handle any scheduling specific information, such as priority values for each task or the task state. We defer all scheduling policies to the FreeRTOS kernel.

2.6 Scheduling

The FreeRTOS kernel is responsible for creating and scheduling enclaves. FreeRTOS uses a task list to keep track of all tasks that can be scheduled. When the kernel wants to switch to another task, it will choose the task with the highest priority from the task list. We

modified the FreeRTOS kernel to also schedule tasks protected by an enclave. All scheduling policies will be completed from the FreeRTOS kernel. By pushing the scheduling decision to the user-mode kernel, we allow embedded system developers greater confidence in modifying the scheduling policy to use without compromising the SM and breaking security guarantees of other applications running on the embedded device. We illustrate the scheduling process in Figure 2.5.

❶ The kernel selects a task and does an SBI call to trap into the SM. The kernel passes the task ID which the SM should switch into.

❷ The SM will then verify that the enclave scheduler sent the SBI request. Any calls to switch enclaves that does not originate from the enclave scheduler will cause the SM to switch back to the enclave scheduler. We chose to only allow the enclave scheduler to switch to another enclave to preserve the semantics of the FreeRTOS kernel. In a traditional RTOS, tasks must relinquish control to the kernel. In similar fashion, non-scheduling enclave tasks can only relinquish control back to the enclave scheduler.

❸ Upon an enclave task switch, the SM will switch the PMP registers to allow access to the memory region specified in the new enclave. The PMP registers that allow access to the SM and the enclave scheduler will be locked to guard against potential memory access by the newly switched in enclave.

We also allow enclave tasks to hint to the SM to allow directly switching to another enclave that it trusts. This is an optimization to avoid switching back to the FreeRTOS kernel if the enclave task expects frequent context switching or message passing to another enclave.

2.7 Interrupt Handling

FreeRTOS uses queues to service interrupts. Specifically for the FreeRTOS port in RISC-V, interrupts are registered via a vector table and the pointer to the vector table is stored in register `mtevec`. Upon an asynchronous interrupt, the `mcause` register is analyzed and used to decode how to handle the interrupt. The appropriate entry in the vector table is then chosen to branch to (given by `mtvec`). Since the FreeRTOS kernel is delegated to user mode, the SM handles interrupts and exceptions in similar fashion to the RTOS kernel. Currently, to mitigate DoS attacks, we setup a machine timer interrupt upon FreeRTOS kernel scheduler initialization. Any user-mode enclave must handle the timer interrupt and it cannot be ignored. If a user-mode enclave is running and uses up a lot of resources, the SM can step in and kill the enclave once the enclave’s time quanta completes. Currently, the way the SM handles a machine-mode timer interrupt is to always switch into the enclave scheduler.

Interrupts and exceptions are by default handled in machine mode in RISC-V; however, interrupts and exceptions can be delegated to lower privileged levels by the `mideleg` and `medeleg` registers respectively [27]. RISC-V also has the proposed N extension, which allows for user-mode interrupts. With the N extension, since the FreeRTOS kernel runs in user-

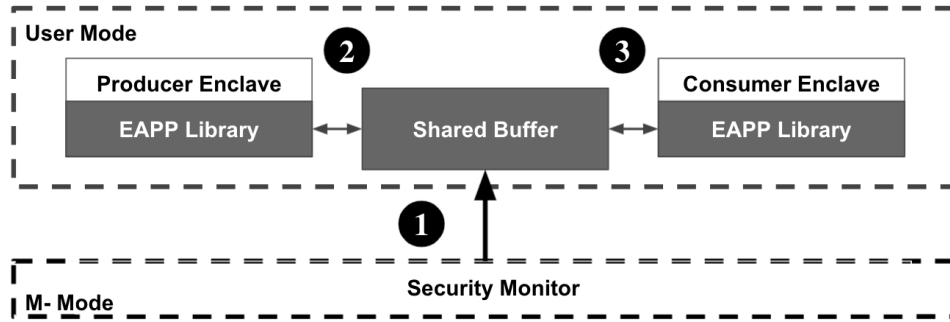


Figure 2.6: Shared memory for enclave to enclave communication

mode, we may delegate interrupts to the FreeRTOS kernel. Currently, our QEMU build did not support the N-extension, so we were not able to develop support for user-mode interrupts. With N-extensions, we may receive a favorable performance gain because the SM will no longer be directly involved in handling specific interrupts and exceptions. At the same time, by allowing user-mode enclaves to handle timer interrupts, our system is more vulnerable to DoS attacks. We leave user-level interrupt handling and its security implications for future work.

2.8 Message Passing

Small Message Passing

FreeRTOS supports inter-task communication via queues. In order to support inter-enclave communication, we implemented per-enclave mailboxes. Each enclave has a statically allocated mailbox buffer that is managed by the SM. We must have the SM involved when sending these messages as the SM intervention ensures that if an enclave receives a message from enclave A, then enclave A actually sent the message. The SM guarantees that the message is sent from the purported enclave and the message contents are not seen from anyone other than the sender, recipient, and the SM. The mailbox interface was only meant for small (≤ 512 bytes) message passing between enclaves.

Bulk Message Passing

For larger messages, the SM can allocate an enclave buffer that is only accessible by the sender and receiver enclave. If there is no harm in leaving the shared buffer unprotected, the shared buffer can also be allocated outside of the enclave. Enclaves will be able to send messages directly without SM intervention. This is visualized in Figure 2.6 if Enclave 1 were the producer and Enclave 2 the consumer.

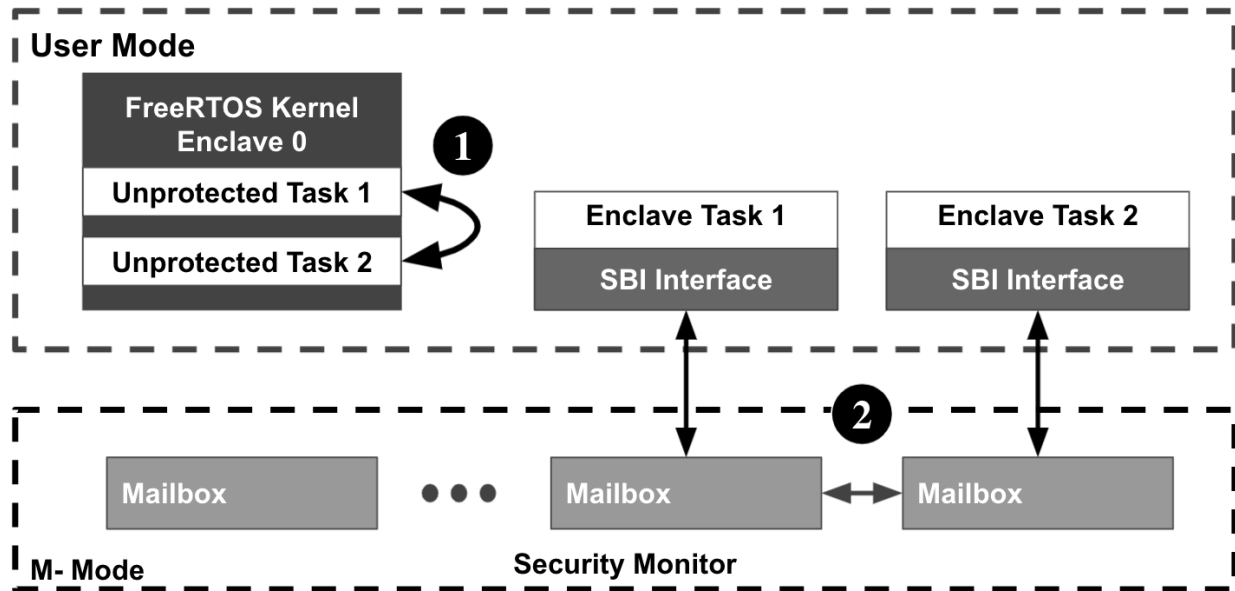


Figure 2.7: Message Passing Visualization

❶ The SM creates a shared enclave buffer that is only accessible by Enclave 1 and Enclave 2.

❷ Enclave 1 would write to the shared memory buffer

❸ Enclave 2 will read from the buffer.

Other than switching enclave contexts, bulk message passing does not require further intervention from the SM.

2.9 Unprotected Tasks

Not all tasks will require strong isolation guarantees. For example, multiple tasks that communicate frequently with each other do not require strong isolation guarantees between them. Message passing between enclaves require SM intervention, which requires trapping into machine-mode, flipping the PMP registers, and switching in the new enclave registers. This causes unnecessary overhead if isolation is not required. This is further visualized in Figure 2.7. For this reason, we also add the notion of **unprotected tasks**. This allows embedded device developers greater flexibility in choosing which tasks are necessary to run in a TEE and which tasks do not.

❶ Unprotected tasks run in the same enclave as the enclave scheduler, so switching or sending messages between unprotected tasks do not require communication with the SM.

❷ Enclave tasks context switching or sending messages between each other must signal to the SM. Message passing requires copying the message from the enclave’s buffer to its

mailbox stored in the SM.

Even though unprotected tasks run in the same TEE as the FreeRTOS kernel, unprotected tasks are still prevented from gaining any confidential information in an enclave. A unprotected task may overwrite or modify the initial state of an enclave (i.e. enclave entry point), but this would be detectable because an enclave’s initial state (enclave entry point, start address, and size) is hashed as well as each page of the enclave. Furthermore, an unprotected task cannot access another enclave task’s memory because of the strong isolation guaranteed by the PMP registers. At worst, the unprotected task may be able to cause the FreeRTOS kernel to crash and prevent any enclaves from running, but preventing DoS attacks is beyond the scope of our trust model.

Unprotected tasks use a messaging interface provided by the FreeRTOS kernel. Each unprotected task is compiled within the FreeRTOS kernel and can be scheduled similar to enclaves. We focused on providing the same interface to both unprotected tasks and enclave tasks. We wanted to provide embedded developers an easy-to-use API that allowed them flexibility in creating TEEs only if they deem it necessary and are not weighed down by a separate API for enclave tasks and unprotected tasks.

2.10 Implementation

Our implementation added around 1000+ LOC to the FreeRTOS Kernel and over 500+ LOC to Keystone’s Security Monitor. The modifications included porting FreeRTOS to run in user-mode, creating APIs to create and interact with other enclaves, and modifying the existing FreeRTOS API to integrate tasks and enclaves. ERTOS can be compiled for RISC-V 32-bit (RV32GC) or 64-bit (RV64GC) machines.

Base Enclave API

We wanted to ensure that our API was familiar to use for embedded application developers, so we did our best to stick to any naming conventions defined by FreeRTOS. We list out below FreeRTOS’ function header for creating a task and our function header for creating an enclave.

Listing 2.1: Enclave Task Creation Function

```

BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask);

uintptr_t xTaskCreateEnclave(uintptr_t start, uintptr_t size,

```

```

const char *const enclaveName ,
  UBaseType_t uxPriority ,
void * const pvParameters ,
  TaskHandle_t *const pxCreatedTask );

```

Furthermore, we wanted to ensure that an enclave task will be treated like any other task in FreeRTOS. We use the RTOS kernel's task list, which keeps track of all tasks, to also track enclaves. By doing this, we can reuse the RTOS scheduler to schedule enclaves. We had to modify several FreeRTOS functions that dealt with switching task contexts.

We wanted to modify the existing FreeRTOS functions to handle tasks or enclaves differently, but still have the developers use the same API. For the implementation, I modified FreeRTOS to create general wrapper functions for both enclaves and tasks that required switching task contexts. For example, I created a wrapper function `yield_general()`, which can be called by either an enclave or task. If the current task is an enclave, it must signal to the SM to switch contexts back to the FreeRTOS kernel. If it is a regular task, it doesn't require signalling to the SM as all unprotected tasks run in the same TEE as the RTOS kernel.

Message Passing API

The only functions exposed to the user that were different was message passing. FreeRTOS has their own message queue data structure that they provide to users. For enclave message passing, we use a per-enclave mailbox. Because all untrusted tasks run in the same enclave as FreeRTOS, all untrusted tasks share a single mailbox. There is performance overhead when sending a message because it requires copying the message and signaling to the SM. Because of the performance difference between message passing of enclaves versus tasks, we decided to keep the functions separate to avoid any confusion to developers.

Listing 2.2: Enclave Message Passing API

```

//Provided by FreeRTOS
void xQueueSend( xQueue , pItemToQueue , xTicksToWait );
BaseType_t xQueueReceive( QueueHandle_t xQueue ,
    void * const pvBuffer ,
    TickType_t xTicksToWait );

//Provided by ERTOS
int sbi_send(int task_id , void *msg, int size ,
    unsigned long xCyclesToWait , uintptr_t yield );
int sbi_recv(int task_id , void *msg, int size ,
    unsigned long xCyclesToWait , uintptr_t yield );

```

There are some differences between RTOS's message passing and ours. First, their queue only passes the pointer of the message they want to send. They can do this because all tasks share the same address space. Second, we introduce a `yield` parameter that signals to the SM to switch out of the enclave if the message isn't in the mailbox for `sbi_recv()` or to always switch upon sending a message for `sbi_send()`. This could be useful if we must immediately wait for a message from whom we send the message to.

Chapter 3

Evaluation

For all benchmarks, I simulate a single RV64GC core, out-of-order machine (Berkeley’s Out of Order Machine [2]) with DDR3 using FireSim [14]. The specific configuration in FireSim used is `firesim-boom-singlecore-no-nic-12-11c4mb-ddr3`.

3.1 MicroBenchmarks

Message Passing Cost

For message passing between two non-enclave tasks, we can simply pass a pointer between the tasks as the two tasks are not memory isolated. Passing a pointer between enclaves is not possible due to the strong memory isolation guarantees. Because of this, the message passing performance linearly scales with message size due to message copying.

Because of the strict memory isolation between enclaves, message passing requires copying the message multiple times per message. Furthermore, we run on a single-core machine and FreeRTOS currently only supports a single-core system. For this reason, we cannot take advantage of switch-less message passing, where two threads share memory through a buffer. Currently, we use asynchronous message passing, where the users must poll the mailbox if they are waiting on a message.

A study found that **inter-task** communication accounted for the most frequent type of task communication (over forward and backward intra-task communication) [15]. Because of the importance of inter-task communication, we measured the total cycles required to send a small message between two tasks, using different configurations. We define our message passing test as a task sending a message to another task, then waiting for a reply. We will measure the number of cycles it takes for a full round trip.

For the **baseline**, both tasks will run within the same enclave domain. We will compare it to the performance of both tasks in separate enclaves. We get an average roundtrip cost of 4324 cycles for the **baseline**. Because varying the size of the message doesn’t matter for the **baseline**, we use this average to calculate the slowdown for enclave message passing.

Message Size (bytes)	Cycles	Slowdown
32	10244	2.37x
64	10818	2.50x
128	11393	2.63x
256	12689	2.93x
512	15412	3.56x

Table 3.1: Asynchronous Message Passing Benchmarks

In table 3.1, we observe that there is significant overhead and that the overhead increases as the message size is increased. The performance overhead is due to the fact that we copy the message twice. Furthermore, upon each `send` and `receive` of a message, we must signal to the SM via an SBI ECALL to retrieve or put a message in the mailbox. Next, we discuss ways to improve message passing latency.

Message Optimization

Currently, our scheme requires 2 memory copy operations. First, the enclave task must copy the message to the receiver’s mailbox. Later on in time, the receiver enclave task must receive the message from its mailbox by performing an SBI ECALL to copy the message from its mailbox in the SM to its buffer in its own memory space. Another limitation is that if an enclave is waiting for a message, it must loop and poll its mailbox until a message is received. This could cause more unnecessary signalling to the SM.

Asynchronous Message Passing

To mitigate both issues, I implemented synchronous message passing. The sender of a message only sends to an enclave who has previously requested a message already. This would only require a single message copy as the SM can simply copy the message directly to the receiver’s specified buffer. Furthermore, the receiver no longer has to poll for a message. Upon calling `sbi_recv`, the SM will set a flag to indicate that the enclave is waiting for a message then immediately switches to the RTOS enclave. When the sender enclave sends the message to the receiver, the sender will copy the contents directly to the receiver’s buffer, then the receiver will be able to return from its initial call to `sbi_recv` without having to poll the SM any further.

Shared Buffer

We also introduce another form of message passing, which allocates a shared buffer between both enclaves. Both enclaves can directly read/write from the buffer. When enclave A wants to send a message to Enclave B, Enclave A will store the message in the shared

Size (bytes)	SYNC Cycles	SYNC Slowdown	SHARED Cycles	SHARED Slowdown
32	9310	2.15x	6526	1.51x
64	9457	2.19x	6581	1.52x
128	9946	2.30x	6819	1.58x
256	10517	2.43x	7129	1.65x
512	12616	2.92x	8038	1.86x

Table 3.2: Message Passing Optimization Results

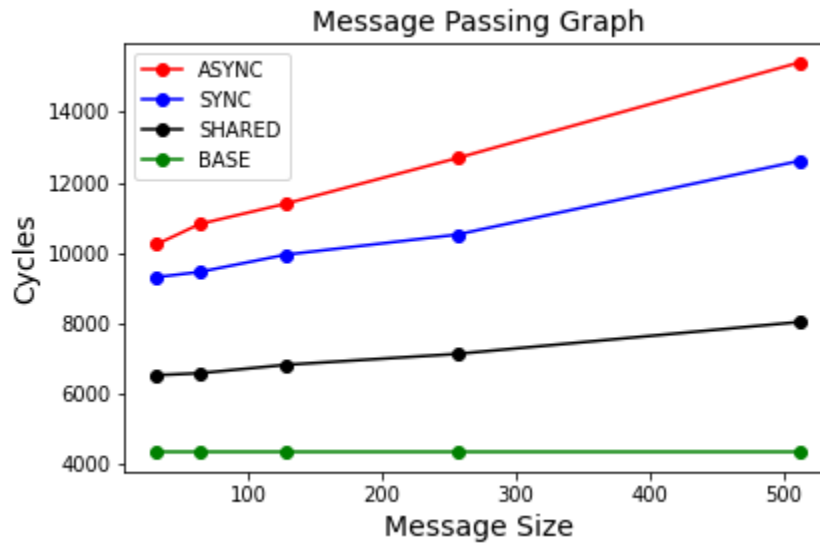


Figure 3.1: Message Passing Results Graph

buffer and signal to the SM to switch to Enclave B. If needed, the buffer can be protected by allocating another PMP region and securing the buffer in an enclave, but this will consume a PMP register. To get around the PMP register cost, one solution is to grant a temporary PMP register to lock a memory space for the sender and receiver enclave. Once message passing is completed, the SM can revoke access to the buffer and relinquish the PMP register. If it isn't necessary to protect the data, the buffer can be allocated in unprotected memory. For the test, we allocated the buffer in unprotected memory.

Message Passing Results

With synchronous message passing, we lower our average overhead to 2.40x and 1.62x for the synchronous and shared buffer message passing respectively. We observe that both asyn-

Test	Baseline Time (ms)	ERTOS Time (ms)
aes	625	600 (1.04x Speedup)
dhrystone	634376	577084 (1.10x Speedup)
norx	326	332 (0.98x Slowdown)
qsort	97	95 (1.02x Speedup)
sha512	373283	373283 (1.00x Speedup)
primes	3166	2735 (1.16x Speedup)

Table 3.3: RV8 Benchmark Results

chronous and shared message passing approach provide a fair improvement to synchronous message passing. Message passing through the shared buffer had the best performance as the context switching time is minimized. This is due to the SM only having to context switch to the receiver enclave, whereas for synchronous message passing, the SM has to do additional processing (i.e. find the correct mailbox) and then copy the message to the corresponding mailbox. The trade off of using a shared buffer is that the buffer must be either allocated in a shared enclave, which consumes a PMP register, or the data on the buffer must be encrypted to avoid leaking secrets to other tasks.

3.2 General Computation Benchmarks

To observe any computational overhead, we ported the RV8 benchmarks, which are compute-bound workloads, into our RTOS framework [6]. We measure the total time it takes to run the benchmarks inside an enclave versus running the benchmarks as an unprotected task in FreeRTOS. I exclude `miniz` and `bigint` for our evaluation because `miniz` had a large memory requirement and `bigint` relied on the C++ run-time, which our framework does not yet support.

Observing table 3.2, we found that the results using ERTOS had slightly better performance compared to the baseline. On average, we get a 1.05x speedup. For enclaves, we pre-allocate a heap and each enclave runs our version of `malloc`, so that the enclave does not have to rely on FreeRTOS for `malloc`. The `malloc` inside the enclave has a lower memory footprint and does not support coalescing heap blocks. The unprotected tasks use `malloc` provided by FreeRTOS which does support coalescing, which may cause memory de-allocation to be slower. This was the main attribute to the slight difference in performance. Furthermore, the heap for the baseline is shared with the FreeRTOS kernel, whereas the enclave application has its own private heap. This could make it more likely that the baseline could have poor cache locality as the heap might be fragmented due to sharing with the FreeRTOS kernel. We observe that there are no significant compute slowdowns when running the RV8 benchmarks on ERTOS.

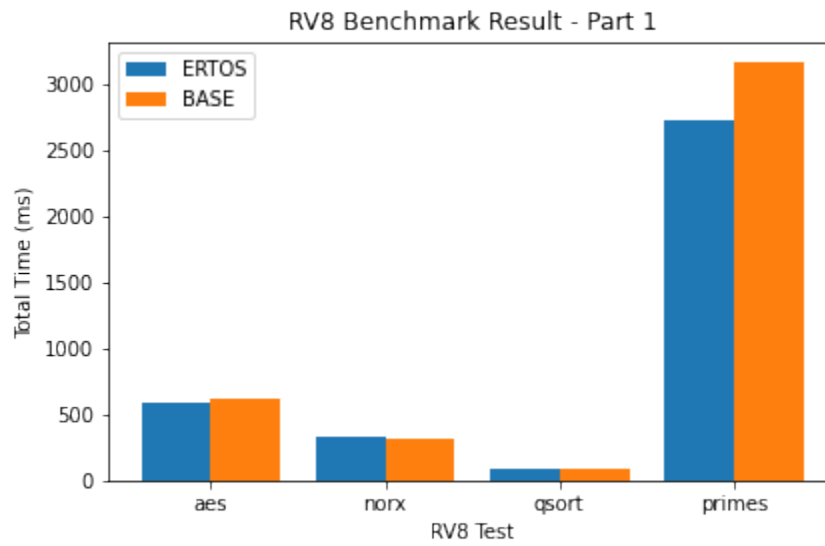


Figure 3.2: RV8 Benchmark Graph - Part I

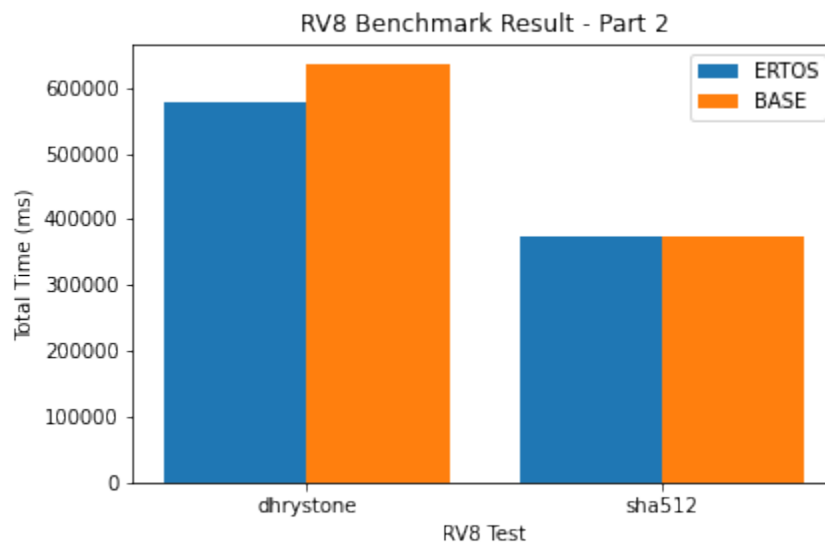


Figure 3.3: RV8 Benchmark Graph - Part II

3.3 Synthetic Benchmarks

Robotics and Edge Computing

The trend for robotics has been to push several of the computationally expensive tasks onto the cloud as robotics have to perform real-time tasks. Typically, the on-chip processor of a robot is not sufficient to run expensive motion planning problems. The problem with this current trend of pushing computation onto the cloud is the assumed trust that we place onto the cloud provider in dealing with privacy-sensitive information retrieved from the robot's sensors.

Sending privacy-sensitive information to the cloud introduces an expanded TCB as we now must implicitly trust the cloud provider. One solution is to use TEEs inside these cloud providers, but there remains the round trip time to first attest the TEE and then subsequently receive actions from the TEE.

Case Study: Motion Planning

One robotics paper, **Motion Planning Lambda** (MPLambda), utilized server-less computing to offload the computation cost of running expensive 3D motion planners onto the cloud [11]. In MPLambda, the workload involves a robot sending a snapshot of its environment, a starting state, and an end state to the cloud provider [11]. The motion planning is distributed to several Amazon Lambda hosts that coordinate to find the best action for the robot to take. The cloud provider will then return an action for the robot to execute. This process is repeated until the robot reaches its final goal. The 3D motion planning problem is distributed to several planners and are sent to Amazon's Lambda [4] service to run the distributed motion planning computations. Once a planner receives a better path than their current best path, they will broadcast this path to other planners as well as the robot.

Some of these planners can be scheduled and run as a task inside of a robot. Every time the robot receive a valid solution path, it broadcast the solution to the other planners to improve their own efficiency. By having a hybrid approach, where some planners run directly on the robot and some on the cloud, we could reduce the average latency to find our first solution if the solution is found on a local planner running on the robot. I visualize this hybrid approach in Figure 3.4.

Why would we need to run our local planners in a TEE? In a model where the robot sends its data over to the cloud provider, we mainly wanted to protect the privacy sensitive information from the cloud provider hosts or any administrators in charge of the hosts. The local planners run directly on the robot and do not generally have to worry about an adversary host; however, another set of issues arise. The planners for MPLambda relied on several third-party libraries like the **Open Asset Import Library** (Assimp), **Eigen**, and **libccd**. With a large code base and a lack of isolation between tasks on embedded devices, running a planner locally introduces a new set of problems. We want to ensure that the planners do not modify the RTOS kernel or any other critical tasks.

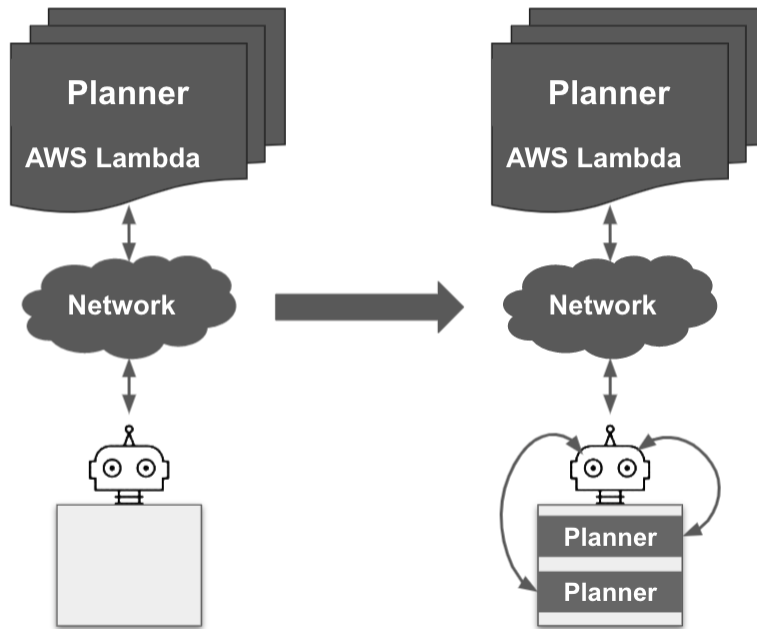


Figure 3.4: Illustration of using a hybrid computing approach, where some planners are executed directly on the robot.

One can use ERTOS to run the planner inside an enclave task. Currently, the libraries that MPLambda relies on are incompatible with FreeRTOS, but I simulate a similar scenario, where a robot has a third party application that runs reinforcement learning to calculate the next action for the robot.

Background: Reinforcement Learning

Reinforcement learning involves an agent and the environment. The agent chooses to do an action to maximize a reward. Once an action is chosen, the agent observes from the environment a possible reward. For our evaluation, we had an agent run the Q-Learning algorithm [12] on a simulated environment. Q-learning is a model-free reinforcement learning algorithm that will help the agent choose the best action to do given the current environment. It is considered model-free because the agent does not require transition probability information or know what actions lead to a specific reward, as the agent will learn this after taking an action.

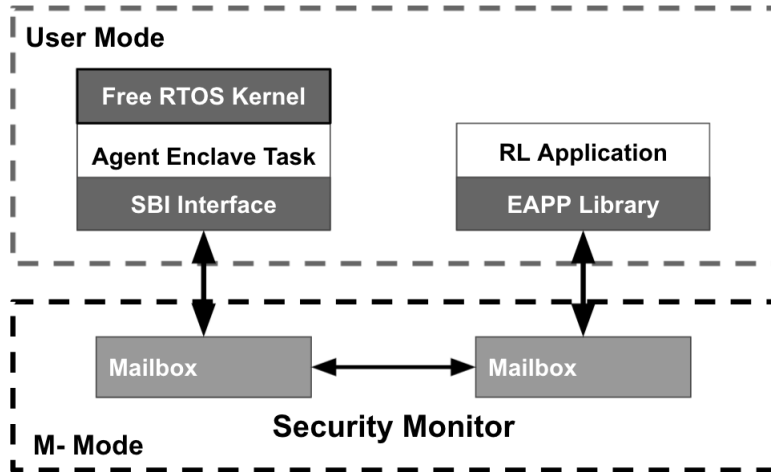


Figure 3.5: Visualization of the Synthetic Benchmark

Episodes	Baseline (ns)	ASYNC (ns)	SYNC (ns)	SHARED (ns)
1000	3721	27295	18974	13302
10,000	34776	261157	181617	127806
25,000	86281	651178	452566	318592
50,000	171594	1300583	903482	636099
75,000	256923	1950098	1354477	953661
100,000	342320	2599744	1805575	1271291

Table 3.4: Synthetic Benchmark Results

ASYNC : Asynchronous Message Passing

SYNC : Synchronous Message Passing

SHARED : Message Passing via Shared Buffer

Results

For the analysis, I used an existing environment provided by OpenAI Gym [5], where an agent must navigate a 2D grid to reach a goal state. For this experiment, we have a task running in the RTOS enclave, sending state information (i.e. x and y coordinates) to the reinforcement learning (RL) application. The RL application will receive information from the task, run Q-learning, and generate an optimal action to feedback to the task. The goal of the agent is to navigate across a *Frozen Lake* represented by a 2D grid. I provide a visualization of the communication in Figure 3.5. This benchmark relies heavily on message passing and context switching between tasks. All messages passed are 12 bytes or less.

The results can be found in Table 3.3. I ran the experiment with varying number of

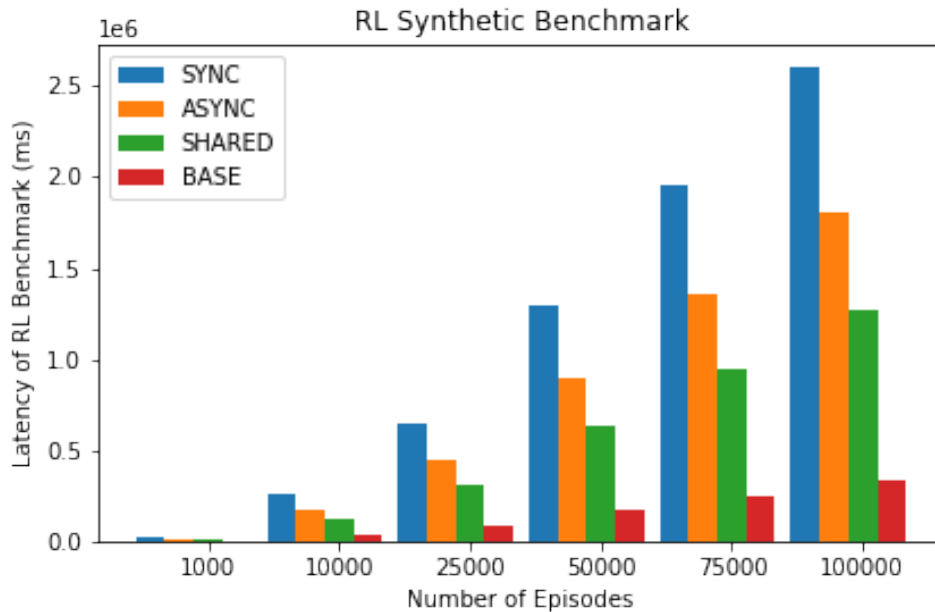


Figure 3.6: RL Benchmark Graph

episodes and get around a 7.53x slowdown on average for asynchronous message passing. For synchronous message passing the overhead drops down to 5.23x. When sending messages via an allocated, shared buffer, we see the overhead drop even more to an average of 3.68x.

The better performance from the shared buffer was because of its quicker context switch time. For both **ASYNC** and **SYNC** message passing, the SM must do a linear scan to find the corresponding task with a specific **task id** and copy the message to its mailbox **ASYNC** or directly to the task’s buffer **SYNC**. In **SHARED**, the enclave application copies the data over to the shared buffer and signals to the SM to switch to the receiving enclave. Because of the small message exchanged (≤ 12 bytes), context switching became the most critical component in this benchmark.

While we have improved the performance of message passing, we observe that the overhead due to message passing is still an obstacle for achieving the baseline performance due to copying the message and signaling to the SM.

Further Optimizations

Zero Copying: For medium-sized messages, message copying is necessary, but for smaller messages, one can further improve performance by using zero copying. This is a technique used in the L4 micro-kernel, which fits a small message into hardware registers and perform a context switch without modifying the registers [8]. This achieves message

passing without any copying, but can only fit smaller messages.

Switch-less OCALLs: In a multi-core system, multiple threads may interact without having to perform a context switch. This could save in penalty cost when sending messages, as messages could be sent in a shared buffer between the sender and receiver. There has been previous work, which shows a clear advantage into using switch-less call that receive a 13x-17x speedup using SGX [29]. Because FreeRTOS assumed a single-core machine, we could not take advantage of using switch-less calls to handle message passing.

Chapter 4

Conclusion

As we face a growing popularity of edge computing and IOT devices, there is an increased need for strong hardware isolation on embedded devices. We create ERTOS, a module in FreeRTOS that allows the creation of dynamic, secure tasks that can be attested and isolated from other tasks using Keystone as an enclave backend. Because of the strong isolation guarantees between enclaves, the overhead of message passing between enclaves is significant; however, we achieve negligible performance overhead when running compute intensive workloads. As we continue to develop ERTOS, we will actively investigate how to increase the performance of inter-enclave communication.

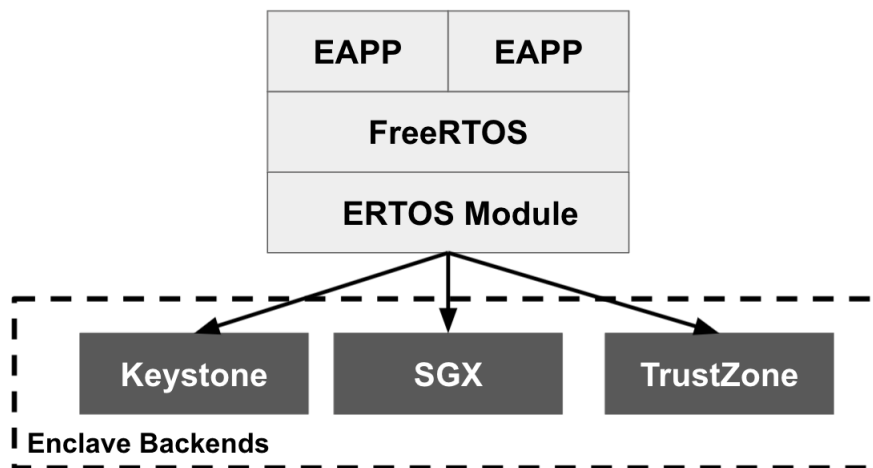


Figure 4.1: Hardware Agnostic ERTOS

4.1 Future Work

Open Framework

In our vision, we wanted to create a framework that can easily support enclave creation on almost all embedded devices that support TEE creation like SGX or TrustZone. Effectively, the vision for this project is to be able to provide a generalized API in FreeRTOS to create secure TEEs similar to OpenEnclave [20] or Asylo [3] that is agnostic of the hardware backend. Some of the challenges of this is that different hardware architectures provide somewhat different security guarantees as covered in Chapter 1. For example, creating a TEE in ARM TrustZone will not allow for strict isolation between enclaves [22]. We must be careful to annotate the different security guarantees, while also providing a generic API for different hardware back ends.

We illustrate our vision in Figure 4.1. Our FreeRTOS module will provide a layer between developers and different hardware backends for creating TEEs. We will provide a set of APIs that can create, run, and attest enclaves that can work with several hardware back-ends.

Bibliography

- [1] *Amazon Skills Marketplace*. <https://www.amazon.com/b?ie=UTF8&node=13727921011&tag=dt-incontent-btn-20&tag=dt-incontent-btn-20>, note = Accessed: 2021-05-10.
- [2] K. Asanović, D. Patterson, and Christopher Celio. “The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor”. In: 2015.
- [3] *Asylo*. <https://asylo.dev/>, note = Accessed: 2021-05-1.
- [4] *AWS Lambda*. <https://aws.amazon.com/lambda/>, note = Accessed: 2021-05-11.
- [5] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].
- [6] M. Clark. “rv 8 : a high performance RISC-V to x 86 binary translator”. In: 2017.
- [7] Victor Costan and S. Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 86.
- [8] Kevin Elphinstone and Gernot Heiser. “From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels?” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farmington, Pennsylvania: Association for Computing Machinery, 2013, pp. 133–150. ISBN: 9781450323888. DOI: 10.1145/2517349.2522720. URL: <https://doi.org/10.1145/2517349.2522720>.
- [9] *Google Actions*. <https://developers.google.com/assistant>, note = Accessed: 2021-05-11.
- [10] Hang Hu et al. *Security Vetting Process of Smart-home Assistant Applications: A First Look and Case Studies*. Jan. 2020.
- [11] Jeffrey Ichnowski et al. “Fog Robotics Algorithms for Distributed Motion Planning Using Lambda Serverless Computing”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 4232–4238. DOI: 10.1109/ICRA40945.2020.9196651.
- [12] B. Jang et al. “Q-Learning Algorithms: A Comprehensive Classification and Applications”. In: *IEEE Access* 7 (2019), pp. 133653–133667. DOI: 10.1109/ACCESS.2019.2941229.

- [13] Xingbin Jiang, Michele Lora, and Sudipta Chattopadhyay. “An Experimental Analysis of Security Vulnerabilities in Industrial IoT Devices”. In: *ACM Trans. Internet Technol.* 20.2 (May 2020). ISSN: 1533-5399. DOI: 10.1145/3379542. URL: <https://doi.org/10.1145/3379542>.
- [14] Sagar Karandikar et al. “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014.
- [15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. “Real world automotive benchmarks for free”. In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2015.
- [16] Dayeol Lee et al. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. <https://arxiv.org/pdf/1907.10119.pdf>. 2020.
- [17] Real Time Engineers ltd. *The FreeRTOS™ Reference Manual*. <https://www.freertos.org/implementation>
- [18] Knud Lasse Lueth. *State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating*. <https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/>. Aug. 2018.
- [19] G. Mullen and L. Meany. “Assessment of Buffer Overflow Based Attacks On an IoT Operating System”. In: *2019 Global IoT Summit (GIoTS)*. 2019, pp. 1–6. DOI: 10.1109/GIOTS.2019.8766434.
- [20] *OpenEnclave Switchless*. <https://openenclave.io/sdk/>, note = Accessed: 2021-05-10.
- [21] Sandro Pinto and José Martins. “The industry-first secure IoT stack for RISC-V: a research project”. In: June 2019.
- [22] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Comput. Surv.* 51.6 (Jan. 2019). ISSN: 0360-0300. DOI: 10.1145/3291047. URL: <https://doi.org/10.1145/3291047>.
- [23] *RISC-V Proxy Kernel*. <https://github.com/keystone-enclave/riscv-pk>, note = Accessed: 2021-05-11.
- [24] *Samsung SmartThings Developer Overview*. <https://docs.smartthings.com/en/latest/getting-started/overview.html>, note = Accessed: 2021-05-11.
- [25] Amazon Web Services. *FreeRTOS: Real-time operating system for microcontrollers*. <https://aws.amazon.com/freertos/>.
- [26] *TBONE – A zero-click exploit for Tesla MCUs*. <https://kunnamon.io/tbone/tbone-v1.0-redacted.pdf>, note = Accessed: 2021-05-10.

- [27] Andrew Waterman and Krste Asanovi´c. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Document Version 1.12-draft. RISC-V Foundation. June 2019.
- [28] Andrew Waterman et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9*. Tech. rep. UCB/EECS-2016-129. EECS Department, University of California, Berkeley, July 2016. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html>.
- [29] Ofir Weisse, Valeria Bertacco, and Todd Austin. “Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves”. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 81–93. ISSN: 0163-5964. DOI: 10.1145/3140659.3080208. URL: <https://doi.org/10.1145/3140659.3080208>.
- [30] wolfSSL. <https://www.wolfssl.com/>.
- [31] Bennet Yee et al. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *IEEE Symposium on Security and Privacy (Oakland’09)*. IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009. URL: http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf.
- [32] Wei Zhou et al. *Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems*. Aug. 2019.