UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Keystone Availability: Extending Keystone to Guarantee Availability for Critical Enclaves

Keystone Availability: Eine Erweiterung für Keystone zur Gewährleistung der Ausführung kritischer Enklaven

Masterarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Christopher Peredy

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Till Eifert
Claudius Pott

Lübeck, den 11. Februar 2025

Abstract

Our surroundings get flooded with a rapidly increasing number of IoT devices which often handle sensitive private data. Most devices receive over-the-air security updates during their lifetime but the secure and guaranteed application of these is not given in most cases. Compromised systems may reject updates to avoid recovery and the patching of exploited vulnerabilities. In this work, we study the implementation of Keystone, an open-source and software-based TEE for RISC-V, in great detail and develop two extensions to support guaranteed execution for Keystone enclaves on top of the present isolation properties. The first extension is a Firmware Enclave that is managed by the trusted firmware instead of an untrusted OS and operates fully independent of untrusted components. The second extension is a hierarchical scheduler that runs inside the trusted domain and provides availability to Firmware Enclaves. Both extensions combined ensure reliable enclave execution even in the presence of a strong attacker with OS-level privileges. Finally, we implement a prototype that performs an authenticated update of another enclave on a generic RISC-V device emulated in QEMU. It shows that our solutions can be used in realistic environments and guarantee the execution of secure update protocols.

Zusammenfassung

IoT-Geräte verbreiten sich rasant in unserer Umgebung. Einige davon verarbeiten sensitive private Daten und werden bei entdeckten Sicherheitslücken oft automatisch über das Internet aktualisiert. Allerdings ist die sichere und garantierte Ausführung der Aktualisierungen größtenteils nicht gewährleistet. Bereits von Angreifern übernommene Systeme könnten dadurch das Einspielen von Updates verhindern, um mögliche Gegenmaßnahmen auszuhebeln, wie beispielsweise das Beheben einer ausgenutzten Sicherheitslücke. In dieser Arbeit analysieren wir die Implementierung von Keystone, einer open source und softwarebasierten TEE, und erarbeiten zwei Erweiterungen, um neben der schon vorhandenen Isolation von Enklaven, auch deren Ausführung zu garantieren. Solche Enklaven können dann sichere Aktualisierungen ausführen, deren tatsächliche Ausführung auch garantiert werden kann. Die erste Erweiterung ist die Firmware Enklave, welche von der vertrauenswürdigen Firmware verwaltet wird, anstatt vom nicht vertrauenswürdigen Betriebssystem. Dadurch agieren Firmware Enklaven vollständig unabhängig von nicht vertrauenswürdigen Komponenten des Systems. Die zweite Erweiterung, die wir in dieser Arbeit vorstellen, ist ein hierarchisch aufgebauter Scheduler, welcher als Teil der vertrauenswürdigen Domäne läuft. Seine Aufgabe ist es, Enklaven Prozessorzeit zu garantieren, sodass sie garantiert ausgeführt werden. Beide Erweiterungen können zusammen dazu verwendet werden, Aktualisierungen auf einem System zu ermöglichen, selbst wenn ein Angreifer mit Betriebssystemrechten das System kontrolliert. Wir implementieren diese Erweiterungen und einen Prototyp einer Update-Enklave für eine generische RISC-V-Plattform, welche wir in QEMU emulieren. Der Prototyp zeigt, dass unser erweitertes Keystone in realistischer Umgebung die Ausführung von sicheren, authentifizierten Aktualisierungsprozeduren garantieren kann.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 11. Februar 2025

Acknowledgements

I would like to express my gratitude to Till Eifert and Claudius Pott who tirelessly supported me throughout the entire thesis. I would also like to thank Prof. Dr. Thomas Eisenbarth for his guidance, despite his busy schedule, and the ability to work on this interesting topic.
Nevertheless, I would not be able to finish this thesis without the continuous support and love of my family, friends and co-workers who all understood when I needed time for myself. - Thank you!

Contents

1 Introduction

The number of deployed Internet of Things (IoT) devices is rapidly increasing. In 2023 there were 13.8 billion connected IoT devices and by 2030 the number is expected to be around 32 billion [Statista, 2024]. This fast evolving technology forces manufactures to prefer fast releases over software assurance, leaving security concerns to be resolved after deployment through over-the-air (OTA) updates. But the vast majority of IoT devices lacks a secure update mechanism that can guarantee the application of the update. Misbehaving or compromised systems may reject or actively prevent updates leaving the devices compromised until physical intervention.

Trusted Execution Environments (TEEs) can be used to create a secure environment for enclaves that run secure update procedures including authenticity checks [Dhobi et al., 2019] isolated from the vulnerable system. It enables device vendors to ensure that their devices only apply genuine update packages. However, when updating a vulnerable system, it is possible that the system is already compromised and refuses to execute the update procedure. So, delivering secure updates should also include a guarantee that the update is applied, a guarantee current TEEs are not designed to give.

In this work, we extend the Keystone TEE to allow it to give such a guarantee for the use case of system updates and system recovery. Keystone is a TEE implemented for RISC-V systems and only uses general security primitives such as Physical Memory Protection (PMP) instead of relying on custom hardware or instructions like most other TEEs. We extend Keystone with the concept of a Firmware Enclave (FWE), a new type of enclave that is initialized and executed fully independent of all untrusted components. It is fully managed by the firmware, therefore preventing denail-of-service (DoS) attacks from the untrusted domain. At all times, the memory of FWEs is protected by PMP, the same hardware security primitive used to protect original enclaves. To guarantee the execution of FWEs we further extend Keystone with a hierarchical FWE scheduler that manages CPU time between active FWEs and the untrusted system. The scheduler is based on the hardware machine-mode (M-mode) timer present on all RISC-V systems, and it includes sophisticated features to ensure the execution of one or multiple FWEs. All our extensions retain the base confidentiality and integrity guarantees of Keystone and also do not require additional hardware features. Furthermore, our extensions allow the preparation and application of updates while the system runs, significantly reducing downtime compared to reboot-oriented approaches. Finally, we implement our designs into a prototype which consists of an updater FWE that successfully performs an authenticated update of another FWE on the system.

1.1 Contributions

We make the following contributions:

Keystone Deep Dive To design and implement our extensions, we studied the internals of Keystone and present detailed technical information about Keystone in our work. Most of these details go beyond the official documentation of the developers.

Firmware Enclave A new type of Keystone enclave designed to be fully independent of untrusted components, especially the untrusted OS. We also provide a reference implementation that is also used to build our updater prototype. Further, we developed a script to conveniently deploy FWEs into the target system.

Availability Scheduler We designed a scheduler as an extension to Keystone's Security Monitor (SM) that interrupts the untrusted system to allow the execution of one or multiple FWEs. The scheduler allocates CPU time to ensure the timely execution of FWEs, such as the updater, and also reserves some resources to allow progress of untrusted jobs. We provide a reference implementation that is also used to build our updater prototype.

Updater Prototype Finally, we implemented a prototype of a simple updater that updates another FWE. This updater was tested in an emulated RISC-V environment with a generic RISC-V device.

2 Background

2.1 RISC-V

RISC-V is a new evolving and open Instruction Set Architecture (ISA) for high performance general purpose computing. In 2010, it started as a project at UC Berkley [Waterman et al., 2011] and evolved to a large organization involving various industry leading companies and many chip designers such as Raspberry Pi or NXP that participate in the development of RISC-V. The open ISA is highly extensible and aims to combine the needs of a wide spectrum of applications from small IoT devices to high performance server applications. As the name suggests, RISC-V is a reduced instruction set computer (RISC) architecture that aims for simple instructions that simplify the hardware implementation rather than complex instructions that result in significantly more complex hardware implementations.

2.1.1 License

The RISC-V ISA is published under the BSD open source license and free to use. It is the main difference to its competitors Arm and x86 and allows everyone to design and publish RISC-V CPUs without an expensive and potentially restrictive license. In case of Arm, only a few selected chip designers have an architecture license to create Arm compatible core designs while most of the Arm customers are only allowed to integrate pre-designed Arm Cortex cores into their SoCs. Such ISAs are proprietary for business reasons, not for technical reasons [Asanovi and Patterson, 2014].

An open ISA gives hardware manufactures a lot of flexibility in designing their chips. It is especially useful for deeply embedded and highly specialized systems like SSD controllers and IoT applications. It also allows researchers to experiment with the architecture and its features to further improve and extend the ISA and hardware implementations. Some implementations (by researchers, companies and hobbyists alike) are even published under open licenses.

2.1.2 Extensibility

The RISC-V base specification [RISC-V International, 2024a, RISC-V International, 2024b] defines a base ISA and a set of standard extensions. All implementations must implement the base ISA called RV32I. It defines 32 general purpose registers and consists of only 40 instructions (load and store, control flow instructions, and basic 32-bit integer arithmetic). All other base extensions are optional. These define for example 64-bit, multiplication, floating point and vector instructions. The ISA can be extended further by adding own instructions for specific purposes.

2.1.3 Relevant Technical Details

We briefly describe the RISC-V architecture features that are most relevant for our work. Please refer to the RISC-V specification documents to further read about all features and architectural decisions.

Privilege Levels The RISC-V specification defines 3 privilege levels [RISC-V International, 2024a]. The most privileged level is the machine-mode (M-mode) that has full low-level access to the system and is usually inherently trusted for that reason. Code running in M-mode is usually firmware as the CPU starts executing in this mode after reset. The specification also explicitly states that M-mode can be used by TEEs to implement a secure monitor and manage secure execution environments. The less privileged supervisor-mode (S-mode) is intended for operating system usage with access to Control and Status Registers (CSRs) to manage system calls and virtual address spaces. Some hardware features such as PMP which are accessible in M-mode are not accessible for code running with lower privileges like in S-mode. The least privileged level is the user-mode (U-mode) that is intended for user applications. It has access to only a few CSRs and almost no low-level hardware access.

Physical Memory Protection (PMP) PMP is an optional (but often implemented) security feature of RISC-V hardware threads (harts) to limit the memory access of the code running on a hart. It consists of a fixed number of CSRs that define a start and end address for a consecutive region. If the access permission is revoked for a particular region, any S-mode and U-mode memory access to this region will result in a PMP fault. This also includes page-table accesses as the page table is also located in memory. The definition of PMP regions and their access permissions can only be set by M-mode code.

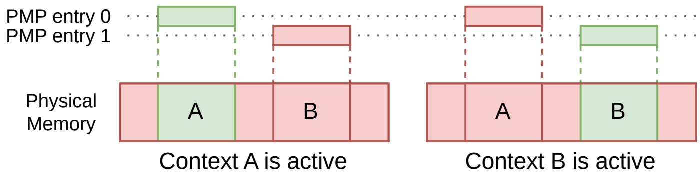

Figure 2.1: Isolation between physical memory regions A and B using PMP. Red areas mark prohibited access; Green areas mark allowed access (read, write, execute).

Deploying PMP to isolate execution environments works by invoking a context switch routine in M-mode. Such a routine changes the PMP access permissions as needed for the environment the system is switching to. All access permissions of the previous environment are revoked. For example, given environments A and B should be fully isolated from each other, then as soon as code from A is executed, the PMP permissions for the memory region associated with B are configured to restrict all accesses. When the execution switches to environment B, a monitor running in M-mode is called to reconfigure the PMP permissions, so that B is accessible, but A is not. This example is visualized in Figure 2.1.

2.2 OpenSBI

The RISC-V Open Source Supervisor Binary Interface (OpenSBI) is the most used reference implementation of the supervisor binary interface on RISC-V systems. The implementation is part of the platform's firmware and runs in M-mode. The interface is basically a hardware abstraction layer allowing S-mode software to be portable across all RISC-V implementations [RISC-V International, 2024c]. For example, OpenSBI provides interfaces to set a hardware timer, suspend the hart, and read performance counters. OpenSBI also provides functionalities for the early boot stage to load an S-mode bootloader like U-Boot.

2.3 Trusted Execution Environment (TEE)

A TEE allows code to be executed in a secure execution environment, commonly referred to as an enclave. These secure environments are usually protected by hardware features and used in scenarios where the OS is not trusted. They provide a secure environment even if the OS is compromised. Most TEEs provide enclaves that are isolated from untrusted components and, provide features for users to verify enclave contents (attestation) and encrypt enclave data (sealing). A typical use case for TEEs is a cloud scenario in which the user wants to protect his data from the cloud provider. With the attestation feature, the user can verify the correct deployment of the enclave on the remote host. Among the most popular TEEs are Arm TrustZone[1], Intel SGX[2], AMD SEV[3], and Intel TDX[4] which are all closed-source, commercial, and hardware based designs and therefore require the corresponding hardware support (e.g., Intel TDX is only available on recent processors of the Xeon series). There are also open TEEs, such as OP-TEE[5] (relies on Arm TrustZone for isolation) and Keystone [Lee et al., 2020], that are more flexible due to their open design and implementation. All these TEEs focus on isolation, that is enclave confidentiality and integrity. For securely applying updates, these isolation properties are needed. Out of the box, current TEEs provide no solution for additional availability guarantees.

2.4 Keystone

Keystone is an open TEE that leverages existing hardware security primitives on RISC-V platforms to achieve strong isolation and is designed to be extensible. Its open design and implementation allows for modifications and adding features that would otherwise be impossible for us to add to commercial TEEs.

In Section 2.1 we already mentioned all the primitives Keystone needs to build a secure and trusted execution environment. It utilizes PMP to isolate enclaves from the rest of system especially from the untrusted environment. The core component of Keystone is the SM that executes in M-mode and manages various aspects of the TEE (Figure 2.2). It implements secure context switches between different environments and provides an interface to create, stop, and destroy enclaves dynamically. In contrast to close-sourced TEEs we mentioned previously, the entire SM is integrated into the device firmware which provides high flexibility to device vendors. The current implementation of Keystone integrates the SM into the OpenSBI firmware. Therefore, deploying modifications to Keystone and the SM is a matter of recompiling the firmware. This open nature and modifiability is the main reason why we decided to build our extensions for Keystone.

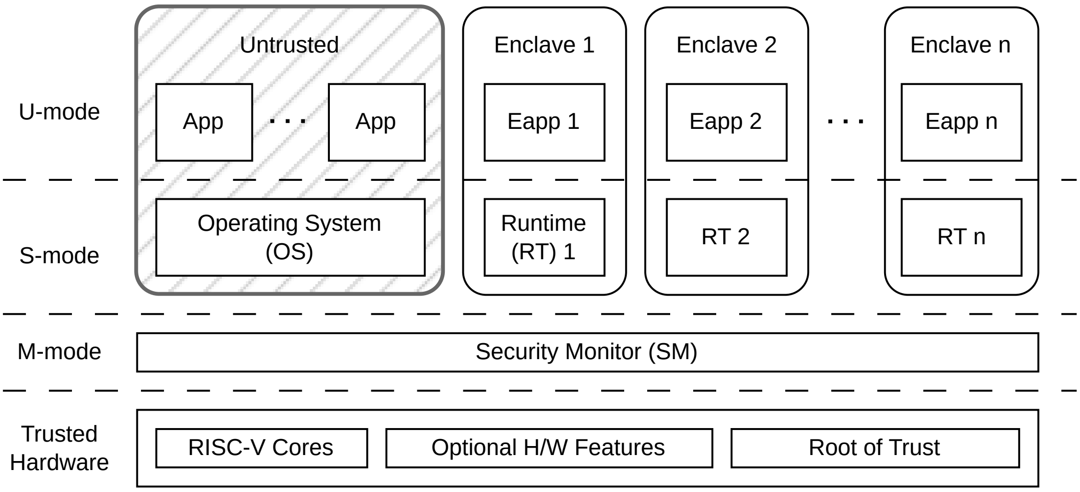

Figure 2.2: Keystone system with untrusted OS, SM, and multiple enclaves (replica of [Lee et al., 2020, Figure 1]).

2.4.1 Keystone Architecture

A Keystone enclave consists of two key components, the runtime and the enclave user-mode application (eapp). The runtime executes in S-mode (Figure 2.2) and provides the eapp with an OS-like environment such as virtual addresses and various available system calls. Keystone offers a minimal modular runtime called Eyrie but also supports other runtimes such as seL4[6]. The second component is the eapp itself which executes in U-mode and must be specifically developed for the use with the chosen runtime. Both components run fully isolated inside the enclave context protected by the SM. Optionally, the runtime can forward OS system calls through the SM to the untrusted OS in order to let the eapp access OS functionality.

Keystone enclaves are managed by their corresponding host application that run inside the untrusted OS. Interactions between host application and SM are realized through a Keystone driver. Calls from the host OS to the SM are implemented by `ECALL` instructions to cause a trap inside M-mode that is then handled by the SM. A host application utilizes this interface to create, run, resume, and destroy enclaves. A similar interface ex-

[1] `https://www.arm.com/technologies/trustzone-for-cortex-a` (accessed 21.06.2024)
[2] `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html` (accessed 21.06.2024)
[3] `https://www.amd.com/en/developer/sev.html` (accessed 21.06.2024)
[4] `https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html` (accessed 07.02.2025)
[5] `https://www.trustedfirmware.org/projects/op-tee/` (accessed 07.02.2025)
[6] `https://sel4.systems` (accessed 06.02.2025)

ists for interactions initiated by the enclave, e.g., to communicate data over the network using the OS's network driver.

2.4.2 Enclave Boot

To provide some context for the following chapters, we briefly explain the boot process of an original Keystone enclave. Later in this thesis (Chapter 4) we introduce a new type of enclave, so we refer to the unmodified Keystone enclaves as *original enclaves*.

First, the enclave needs a block of consecutive physical memory. A part of it is needed to store the executables, the remainder is used as working memory. This working memory is mapped to virtual addresses and used to store data and executable code during enclave execution. The entire enclave memory range is allocated by the Keystone kernel driver and provided to the enclave context through the host application. Then, the host application loads three executables into this memory area:

- *Runtime ELF-Loader:* An ELF-loader binary whose purpose it is to load the runtime executable into virtual address space of the enclave. It is executed in-place, and is the first code executed after switching to the enclave context for the first time. The loader runs in the physical address space.
- *Runtime ELF:* ELF-file of the runtime (Eyrie by default). The runtime provides a system-level abstraction layer for eapps. It can provide kernel-like functionality and runs in S-mode.
- *Eapp ELF:* ELF-file of the eapp. The eapp runs in U-mode and is the main part of the enclave.

The boot process starts with jumping to the location of the ELF-loader binary. It is directly executable, runs in S-mode, and creates the virtual address space for the enclave. Next, the loader loads the runtime binary from its ELF-file into the virtual address space. After loading, the loader transitions to virtual addressing and starts executing the runtime inside the virtual address space. The corresponding page table is located inside the protected enclave memory and is therefore also protected from the untrusted system. Finally, the runtime loads the eapp ELF-file and starts executing it.

Recently, the enclave boot process experienced a comprehensive redesign which fundamentally changed how the enclave is prepared and booted. Before, the ELF loading was done by the host application. This was replaced by a simpler in-enclave ELF loading approach[7] with an in-enclave loader as explained before. Fortunately, that change also simplifies booting our FWE that we will introduce later in Chapter 4.

2.4.3 Enclave Scheduling

In Keystone the execution of original enclaves is scheduled by the untrusted host OS. In contrast to host processes, the host OS cannot interrupt enclaves because the isolated enclave context even isolates it from host OS interrupts. Most operating systems use preemptive schedulers to achieve a certain responsiveness especially on systems with just a few hardware threads. Preemptive scheduling is a scheduling method that allows the scheduler to interrupt (preempt) a running job to schedule other jobs. Enclaves, that cannot be interrupted may block a core for a longer time and reduce the overall responsiveness. To circumvent this issue, keystone enclaves include the feature to interrupt themselves in predefined intervals and wait for the host scheduler to schedule them again.

The untrusted host OS schedules an enclave indirectly through the enclave's host application. The host application runs inside the untrusted OS as a normal application and is responsible for setting up the enclave (see Subsection 2.4.2). Furthermore, it runs a loop to resume the enclave when it stopped. Therefore, if the host OS schedules the host application, it effectively initiates the resumption of the corresponding enclave. Algorithm 1 shows a simplified version of this loop. Note that when an enclave terminates by exiting from the eapp, `resume_enclave()` returns an error code to exit the while loop.

Algorithm 1: Host Application Runs Enclave

```
ret ← run_enclave() // returns when enclave stops
while ret ≠ error do
    ret ← resume_enclave() // returns when enclave stops
destroy_enclave()
```

Enclave Preemption Implementation The current implementation of preemptive enclave scheduling is described as unstable and vulnerable[8] by the authors, hence it is disabled at the time of writing. In detail, on context switch to the enclave they clear all bits in the machine interrupt delegation register `mideleg` which disables the supervisor interrupt enable and pending registers `sie` and `sip` entirely.

7 Keystone commit `58a5151` - In-enclave ELF loading (accessed 13.01.2025);
Keystone commit `874b3d8` - Update to in-enclave load and move communication structs defines into shared headers (accessed 13.01.2025)

8 Keystone commit `2e0bd84` - Undelegate all interrupts from enclave for now (accessed 13.01.2025)

In summary, the original enclave scheduling design requires both parties to cooperate with each-other. The enclave expects to be scheduled by the OS and the OS expects the enclave to return control at some point again. The host OS cannot interrupt an enclave because it runs in an isolated environment with a different Supervisor Trap Vector Base Address (`stvec`), so interrupts from the host OS are not effective. In Keystone's threat model, enclaves explicitly remain vulnerable to DoS attacks from the untrusted host OS. This is fine for general TEE use cases, but it poses a problem for use cases needing an enclave execution guarantee like ours.

3 Threat Model

In this chapter we define our threat model consisting of the definition of the Trusted Computing Base (TCB) and our attacker model. Furthermore, we define the scope of our work in regard to the attack vectors. Due to this work being largely an extension to Keystone, the threat model mostly overlaps with Keystone's threat model [Lee et al., 2020, Section 3.4]. We point out the differences at the end of this chapter.

3.1 Trusted Computing Base (TCB)

Included in our TCB are the RISC-V specification, the hardware implementation, and all M-mode software. M-mode software consists of the platform firmware (e.g., OpenSBI) and the Keystone SM including our modifications to it, namely the FWE's management and the FWE scheduler. Keystone and our extensions (FWE and the scheduler) trust the PMP specification and RISC-V hardware implementation to be correct, because it is the main isolation primitive between the trusted and untrusted system. General FWEs are not part of the TCB but our FWE-based updater enclave is included due to the imposed extensive write permissions to apply updates on other parts of the system such as other FWEs. Generally, we assume that all parts of the TCB are correctly implemented and bug-free. That is a strong assumption but can be partially achieved with formal verification as argued by Lee et al.

3.2 Attacker Model

We assume an attacker without physical access to the system, because such an attacker could threaten the availability of the entire system by simply cutting the power or destroying the hardware altogether. Instead, we assume a remote attacker that can control the untrusted OS, launch new arbitrary Keystone enclaves, and has access to all memory not protected by Keystone. Additionally, we assume the attacker intents to remain in control of the system which can include actively preventing legitimate system updates and remote maintenance by the device owner.

Speculative execution attacks, cache or timing side-channels as well as side-channel attacks with off-chip components are out-of-scope because they are also not covered by Keystone. Please refer to the Keystone paper [Lee et al., 2020, Section 3.4] for more information about the excluded attacks.

3.3 Comparison to Vanilla Keystone

Keystone introduces a physical attacker A_{Phy}, a software attacker A_{SW}, a side-channel attacker A_{SC}, and a DoS attacker A_{DoS} [Lee et al., 2020, Section 3.4]. Compared to Keystone's threat model, our attacker model fits the description of the attackers A_{Phy}, A_{SW}, A_{SC} and A_{DoS} with the difference, that our work additionally aims to prevent A_{DoS} from being successful against FWEs. Original Keystone enclaves, which are deployed and partially controlled by the untrusted OS, remain vulnerable to A_{DoS}. Furthermore, we include the updater FWE in our TCB because of its updating purposes. In summary, our threat model is a stronger version with added availability guarantees for special enclaves.

4 Firmware Enclave (FWE)

We divide our design into two significant contributions. The FWE is one of the two main contributions, and it is the necessary base for independent enclave execution. We designed it as an extension to Keystone, so that the original Keystone enclaves remain available and can be used in conjunction with our new FWEs. In this chapter, we present the concept of an FWE in full detail and explain the challenges faced.

4.1 Why a Firmware Enclave?

The lifecycle of an original Keystone enclave starts with the execution of its host application inside the untrusted OS, which lives in the untrusted context. The untrusted OS takes care of allocating physical memory for the enclave and provides the executable binaries, namely the loader, the runtime (e.g. Eyrie), and the actual eapp. To perform these tasks, it leverages the Keystone kernel driver. When everything is copied in place, the host application calls the SM to “create” the enclave which isolates the enclave context by protecting its memory with PMP. At this point, right before the execution, the SM measures and verifies the initial state of the enclave (the verification is not yet implemented by the Keystone developers). If the verification fails, the enclave is destroyed, meaning the allocated memory is freed and the enclave state is reverted. On successful verification, the host application schedules the execution of the enclave. For a more in-depth explanation of the original Keystone enclave boot process, please refer to Subsection 2.4.2.

With this enclave boot approach, original enclaves achieve their confidentiality and integrity goals, but the availability can be compromised by the untrusted OS, through these three main attack vectors:

1. Manipulating enclaves files during deployment causing the integrity check to fail. This is possible because the enclave memory is not protected until the enclave is finalized.

2. Not scheduling the host application that is supposed to initiate the enclave construction and its execution.

3. Blocking or not correctly invoking the Keystone kernel driver. This attack effects all

communication between the user-mode host application and the SM. Therefore, it can be used perform a wide range of attacks such as (1.), (2.), and not creating the enclave in the first place.

To mitigate DoS attacks from the untrusted context, an enclave must be deployed by a routine inside the TCB, i.e., inside the firmware. Our FWE fulfills this requirement. It is a Keystone enclave entirely deployed and hosted by the trusted device firmware, in this case, the SM itself, as opposed to the untrusted OS. The FWE's required memory is allocated at compile time, so the executable binaries can be written into memory at compile time as well. Although FWEs ship as part of the firmware, they are executed in an environment isolated from the firmware and other enclaves. Like with original Keystone enclaves, this isolation is realized by PMP and context switches managed by the SM. Code executed in such enclaves also only runs with S-mode and U-mode privileges. Combining all these properties, our approach allows an FWE to run code inside an enclave with all of Keystone's security guarantees, additional availability guarantees, and entirely independent of the host OS.

libc Compatibility Because we are aiming for OS independence, FWEs cannot make use of wrapped system calls to the OS, which is an optional feature of the Eyrie runtime. This feature allows Eyrie to call certain Linux system calls, which are then transported through various context switches ("edge-calls") to the untrusted OS and executed there. With these wrappers the runtime can create a libc compatible environment, so eapps can integrate libc functions. The point of FWEs is to be independent of untrusted components, so we suggest using a runtime which can provide a libc compatible environment natively, if needed. At the time of writing, we have not tested any other runtimes than Eyrie with FWEs.

4.2 Firmware Enclave Memory Layout

As described in the previous section, the main advantage of the FWE is its independence from the untrusted OS. We achieve this by initializing the FWE's memory at compile time. For an original enclave this would be done by the host application, where the host application would leverage the Keystone kernel driver to allocate physical memory and copy the binaries into it. By allocating and copying the binaries at compile time however, we trade some flexibility with the guarantee, that the memory range is always available and initialized with the required binaries for enclave execution. This memory range is also secured by PMP against access from the untrusted OS or other enclaves. Additionally, the

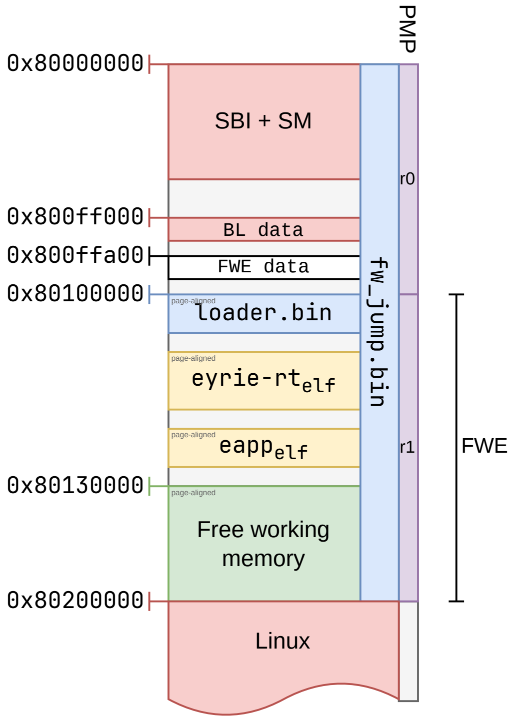

Figure 4.1: System memory map with a "Hello world" example FWE.

static initialization of the memory provides the possibility to execute the FWE at startup before the kernel boots and even without a kernel installed at all.

Figure 4.1 details the memory map of a small example FWE and its surrounding memory blocks on a generic RISC-V system running in QEMU. The allocated enclave memory starts at physical address `0x80100000` and ends at address `0x80200000` (addresses are specific to this example). This memory range is dedicated to this FWE only and consists of the enclave loader (`loader.bin`), the runtime (`eyrie-rt`), the eapp, and free working memory. All these components are part of the system's firmware binary image `fw_jump.bin` and therefore already initialized at the moment of firmware initialization. Apart from the placement of the components in memory at compile time, the FWE's inner components are functionally identical to their counterparts found in original enclaves. Please refer to Subsection 2.4.2 for more information about the individual components.

In order to create the firmware binary image containing the FWE, we inject the files in the form of new ELF sections into the corresponding firmware ELF file `fw_jump.elf`. Originally, it only contains OpenSBI and the Keystone SM. At address `0x800ffa00`, we place metadata about any included FWEs such as the memory range and the individual component offsets. We need this because the addresses are calculated depending on the

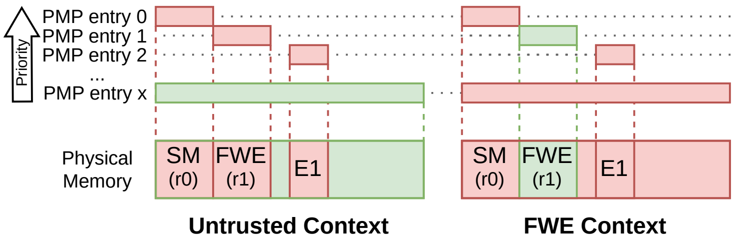

Figure 4.2: PMP permissions for an FWE in untrusted context (left) and FWE context (right). E1 is an original Keystone enclave. The PMP region IDs $r0$ and $r1$ correspond to the regions in Figure 4.1. Red areas mark prohibited access; Green areas mark allowed access (read, write, execute).

file sizes which are known only after compilation during injection. This block of metadata can be seen as a header for the following FWEs. It is also needed to support multiple FWEs, which we will introduce later in Section 4.3. After injection, the ELF file is re-exported to obtain the final firmware binary.

For injecting FWEs into the firmware ELF file we developed an easy-to-use script which contains a JSON-like configuration. It calculates all addresses and offsets automatically (page-aligned if required) and populates the metadata block (header) accordingly.

Injected ELF Sections Our script greatly reduces the manual work needed to deploy FWEs to a system, but it is admittedly not the ideal solution because arbitrarily added ELF sections, that are not part of any segment, are not supported by every ELF loader. For example, the ELF loader used by QEMU does not load such sections outside of ELF segments. Therefore, we export the modified ELF file to a raw binary with GNU's `objcopy` that also exports our injected sections. When deploying to actual hardware, the firmware is always exported to a binary, so our approach is sufficient. The build process can be improved by customizing linker scripts and build receipts to directly include the enclave files in the `fw_jump.elf` at compile time. The linker will then do the work of putting all additional FWEs into appropriate ELF segments.

4.2.1 Memory Protection

Equal to original enclaves in Keystone, an FWE is also hardware protected by PMP and therefore provides the same security guarantees. In Figure 4.1, the PMP regions are depicted by purple boxes and labeled $r0$ and $r1$. The latter is the PMP region of the example FWE and isolates this enclave from untrusted components such as other enclaves (original enclaves and FWEs alike) and the untrusted OS. It also protects the firmware and SM in region $r0$ from unauthorized access by the FWE. A more detailed visualization of the PMP configuration is given in Figure 4.2. During execution of untrusted code (untrusted context), memory accesses to the SM, FWE, and other enclaves are prohibited. When the FWE is invoked, the SM reconfigures the PMP, so that only accesses to the FWE region are allowed. The FWE's PMP region is configured during firmware boot before any untrusted code is executed, which guarantees that untrusted code, such as the OS, never has the chance to access (read/write) the FWE contents.

4.2.2 Firmware Enclave Boot

In contrast to an original Keystone enclave, an FWE has no host application that would allocate memory and copy binaries. As explained previously in Section 4.2, the enclave memory is already prepared at compile time.

At device startup, immediately after the initialization of OpenSBI and the SM, the FWE is created, which includes the configuration of the PMP region specific to the enclave. Such an early creation of the enclave context is unique to FWEs and important to ensure that no S-mode code can access the enclave's memory, including any untrusted OS that may be booted afterwards. After successful creation, the enclave is ready to be booted and executed at any time. The enclave-internal boot process is identical to that of an original enclave, which is described in Subsection 2.4.2.

4.3 Multiple Firmware Enclaves

A firmware image can contain multiple FWEs which are expected to be placed consecutively in memory as shown in Figure 4.3. In our implementation, we currently limit the maximum number of enclaves to eight, due to the reason that most RISC-V cores support no more than 8 PMP regions [Microchip Technology Inc., 2024, StarFive Technology Co., Ltd., 2023]. However, we see no other technical limitations other than the number of PMP registers and memory. Our post-compilation script we mentioned before, writes the metadata needed to execute the FWEs to `0x800ffa00` in memory. The FWE initialization function in the SM can read this metadata and set up the given FWEs for execution. Every individual FWE is assigned a unique PMP region, so they are also isolated amongst each other.

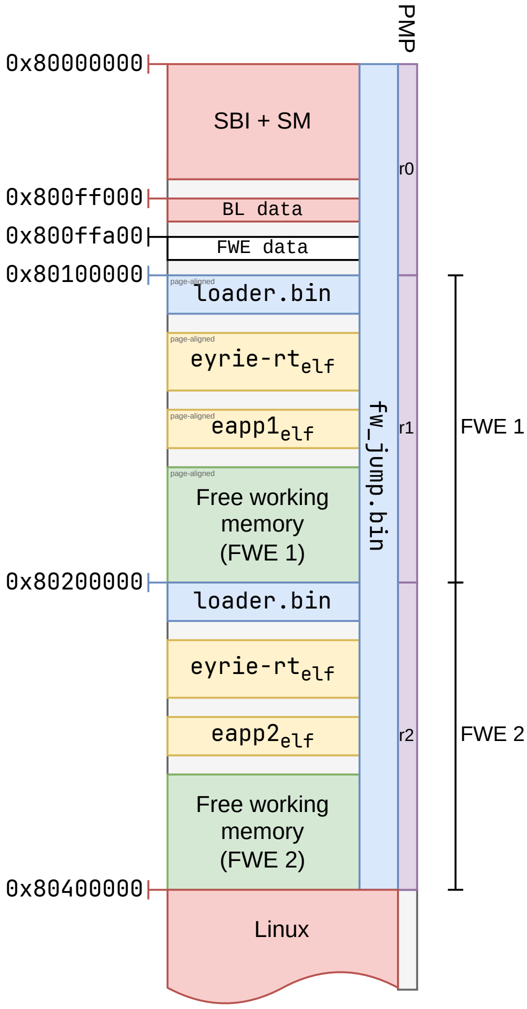

Figure 4.3: System memory map with two FWEs placed consecutively in memory.

4.4 PMP Limitations

During our work, it came to our attention that Keystone is not fully secure on current RISC-V hardware. The reason for this is that PMP only protects memory accesses issued by a core (via load and store instructions) but not memory accesses coming from other I/O agents with access to the internal bus. The Direct Memory Access (DMA) is one of these devices. The point of a DMA device is to copy data without involving a core to do so. Such a device has direct access to the physical memory without PMP protection, so it can be commanded to read and write to arbitrary locations. We do not protect FWEs against this kind of attacks. However, there is already a new extension IOPMP [RISC-V International, 2025] in development that plans to bring a new hardware component to the bus to mitigate the aforementioned issue.

5 Availability Scheduler

In this chapter we present the design of our scheduler for Keystone to guarantee availability for FWEs. We define availability as the ability to always interrupt and take precedence over any untrusted component of the system. To realize this guarantee, we introduce a new scheduling level with machine-mode privileges to Keystone. Our result is a privileged scheduler that can reliably execute FWEs even if the untrusted OS is not cooperative.

In this work, we use the same terminology for "exception", "interrupt" and "trap" as defined the RISC-V ISA:

> *We use the term **exception** to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart. We use the term **interrupt** to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term **trap** to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.* [RISC-V International, 2024a, Section 1.6]

We present our scheduler design in the following sections as a base scheduler with two extensions that add more capabilities while keeping the base's availability guarantees. The *Timer Interrupt Scheduler* is the base scheduler providing periodic execution of one single FWE. We extend it by adding *preemptive scheduling* to avoid untrusted code and future concurrent FWEs getting blocked for unknown durations. The second extension to the base is the feature to schedule and run multiple concurrent FWEs. The entire scheduler is able to provide its availability guarantees on single core and multicore systems, albeit we schedule all FWEs onto one hart in our prototype implementation.

Compared to previous work implementing a secure scheduler [Wang et al., 2022] [Alder et al., 2021], our availability scheduler is also based on a secure timer primitive, but it offers fair preemptive scheduling for multiple strongly isolated enclaves to guarantee availability. RT-TEE [Wang et al., 2022] proposed a more real-time oriented approach based on Arm TrustZone with weaker isolation properties than Keystone and less flexibility. Aion's [Alder et al., 2021] scheduler is influenced by the TI MSP430 architecture, and one of its core challenges is to handle critical (atomic) sections which disable interrupts globally and therefore threaten availability. Such a challenge does not exist for secure schedulers on RISC-V, because atomic sections do not disable interrupts on RISC-V.

5.1 Timer Interrupt Scheduler

As explained in Subsection 2.4.3, original enclaves are scheduled by the untrusted OS. To achieve our availability goal, we cannot rely on untrusted components to schedule FWEs in-time, so we develop this new FWE scheduler as a plugin to the Keystone SM.

The timer interrupt scheduler builds the foundation of our FWE scheduler and serves as a base for further additions we present later in this chapter. Our FWE scheduler requires a high privileged timer that is able to interrupt the system reliably to force the invocation of the scheduler. We formulate the following minimal requirements for a timer that we can use as the base primitive for the FWE scheduler:

1. The timer must be restricted to a higher privileged mode than the highest privilege of any untrusted code.
2. The timer's clock speed must be independent of external factors, such as core clock, bus clocks, etc.
3. The timer should be implemented on most platforms related to our IoT use case (ideally part of the base RISC-V specification).

5.1.1 Machine-mode Timer

RISC-V cores are equipped with a real-time counter which exposes memory-mapped M-mode registers. By specification, this counter is guaranteed to always increment at a constant frequency [RISC-V International, 2024b]. If the counter contains a value higher than the value in the corresponding compare register, it throws an interrupt which essentially makes it a timer. This M-mode accessible compare register `mtimecmp` ensures, that only M-mode firmware is eligible to schedule and reschedule timer interrupts.

We use this M-mode timer for our scheduler to reliably control FWE execution even if the OS tries to actively avoid FWE progress. We found it to be the only timer present on our target platforms which fulfills our previously mentioned requirements:

1. The timer can only be scheduled or manipulated by code running with M-mode privileges, as the involved registers are only accessible from M-mode. Even core periphery such as a DMA device can not access these registers.

2. It increments at a constant frequency that is independent of the core clock in contrast to a cycle count driven timer.
3. It is part of the privileged RISC-V specification, so every specification conform implementation provides this timer.

Reason (1) and (2) are crucial for availability, otherwise untrusted code might reconfigure and delay or disable the timer.

RTCs We also considered the use of dedicated external timers such as a Real-Time Clock (RTC) (also known as Real-Time Counter) but discovered multiple disadvantages with that approach. Most notably, RTCs are not part of the RISC-V specification but rather additional core external peripherals. The communication with such peripherals is highly dependent on the specific hardware implementation and may not isolate RTC registers to M-mode accesses exclusively. All the SoCs we have looked at (i.e., Microchip PolarFire SoC [Microchip Technology Inc., 2024], StarFive JH7110 [StarFive Technology Co., Ltd., 2023], Allwinner D1 [Allwinner Technology Co., Ltd., 2022]) implement RTCs as a (core) external device with memory mapped registers. Their reference manuals do not clearly state whether the RTC registers are M-mode exclusive, so we expect them to be accessible with S-mode permissions for performance reasons. Another disadvantage is that despite most Systems-on-Chips (SoCs) incorporating an RTC, a system without one would need to fall back to the core local M-mode timer anyway.

Use Case The use case for our availability scheduler is to guarantee the execution of an updater FWE. The updater should periodically check for available updates and apply them. For this use case, it is desirable that the FWE is scheduled and executed periodically and retains its runtime data between the periodic activations. The interval length is expected to be quite long with a range between a second and multiple hours.

5.1.2 Periodic Execution with Explicit Stop

A resumable FWE consists of a main loop that explicitly stops the enclave after every iteration. So when the enclave is resumed, it executes exactly one iteration of the internal loop. Keystone does not provide the possibility for an eapp to stop its execution without fully exiting, so we added a new runtime system call named `stop_fwencl()`. This new system call stops the enclave and initiates a context switch back to the previously interrupted context. The enclave switches into paused state and can be resumed later. We

```c
1  int main() {
2    // Main loop
3    while (1) {
4      // Do important task, i.e., check for updates, etc.
5      important_task();
6
7      // Explicitly stop firmware enclave
8      stop_fwencl();
9    }
10 }
```

Listing 5.1: Firmware Enclave Main Loop (implemented in C).

Figure 5.1: FWE periodic scheduling with interval T.

refer to this form of stopping as an *explicit stop*. It tells the scheduler to not schedule the stopped FWE anymore for the current interval.

Listing 5.1 shows an example of a periodically running FWE. Once it is started, it enters an infinite loop (line 3) consisting of "important tasks" (line 5) and an explicit stop (line 8). The function call to the important tasks (line 5) is a placeholder referring to the FWE's core tasks like checking for updates and downloading them or measuring components of the system for anomaly detection. The while loop and the explicit stop compose the periodic execution of the important tasks, as the FWE continues in that loop as soon as it is scheduled again after the stop. From the perspective of the FWE, the runtime system call `stop_fwencl()` behaves similar to a delay timer. The resulting schedule is depicted in Figure 5.1. In this figure, the timestamps a_1 and a_2 mark the M-mode timer interrupts and thus, the beginning of the execution time of the FWE. The explicit stop is marked with a red vertical line after which the M-mode timer is started with the remaining interval time of T. Immediately after this timer elapses (e.g., at a_2), the FWE is scheduled and resumes the infinite loop until the next explicit stop.

5.1.3 Implementation

We implemented the FWE scheduler as an (optional) plugin for the Keystone SM. We also modified the firmware's default M-mode timer interrupt handler by adding a call to the scheduler which runs or resumes pending FWEs. The scheduler makes sure to always reschedule the timer after the previous timer elapsed to guarantee periodic execution of the scheduler. When the timer elapsed and the scheduler is invoked for the first time, all pending FWEs are in the "fresh" state, meaning their internals are not initialized, yet. Initialization happens during the first run of the enclave, so the scheduler calls `sbi_run_enclave()` to boot all FWEs and execute them until they stop. All subsequent invocations encounter the FWEs in the "stopped" state, meaning they are already initialized and only need to be resumed with `sbi_resume_enclave()`.

Later in Chapter 6 we introduce an updater for FWEs that implements a FWE context reset. It resets updated FWEs to the "fresh" state which are initialized with `sbi_run_enclave()` once scheduled again.

Timer Interference In order to leverage the M-mode timer to guarantee availability for the FWE scheduler, the scheduler must have exclusive access to the timer. Being an M-mode timer means all components running in M-mode have access to the timer, i.e., OpenSBI and the SM (both are part of the TCB). Since we cannot restrict these M-mode components from accessing the timer, we analyzed the current implementation of Keystone and OpenSBI and found no internal usage of the M-mode timer.

S-mode software such as an OS often uses the S-mode timer provided by the Supervisor-mode Timer Interrupts (Sstc) extension implemented on most modern RISC-V processors. The Sstc extension introduces a supervisor-level compare register `stimecmp` which is similar to `mtimecmp` but is accessible from S-mode and triggers the corresponding interrupt directly in S-mode as well. If the S-mode software does not leverage the `stimecmp` register (i.e., lagacy code), it calls OpenSBI to set the `mtimecmp` register. OpenSBI provides an interface that allows the manipulation of the M-mode timer as a fallback only if the Sstc extension is not available on the processor[9]. Because Sstc is available on most modern RISC-V processors that also implement S-mode, we removed the fallback to the M-mode timer. This change ensures that our FWE scheduler is using the M-mode timer exclusively, but it results in our modified OpenSBI not being compatible with non-Sstc processors anymore. To summarize, we enforce the use of the S-mode timer through Sstc to protect the M-mode timer from untrusted access. For future work, the compatibility

[9]OpenSBI GitHub: `sbi_timer.c:L141` at commit `bd613dd` (accessed 07.01.2025)

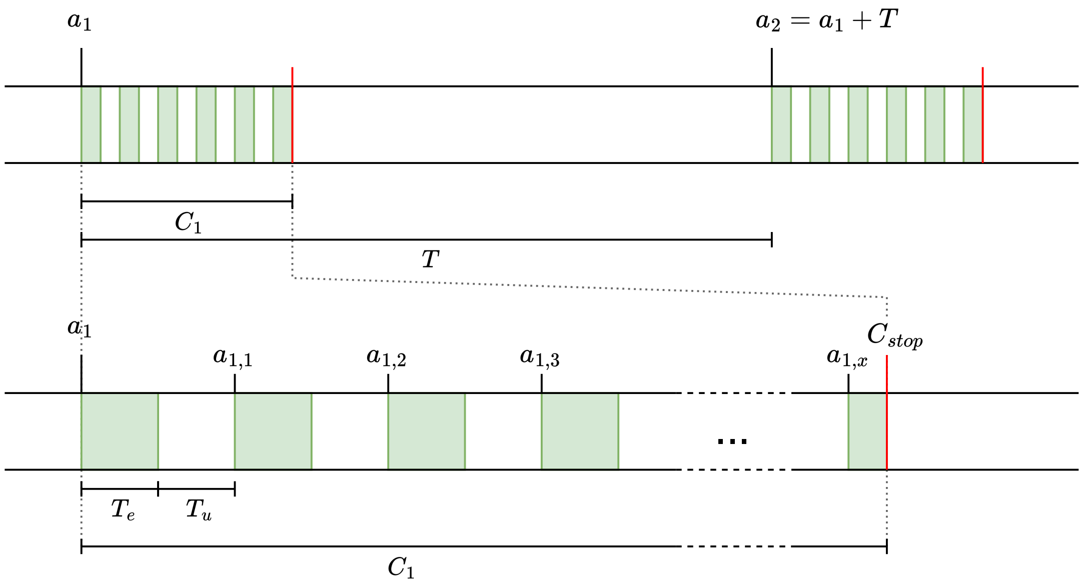

Figure 5.2: FWE with preemptive scheduling. The green area marks the execution of the FWE.

with non-Sstc processors can be restored by implementing virtual timers as suggested by the RISC-V specification [RISC-V International, 2024b, Section 3.2.1].

5.2 Preemptive Scheduling

Our first extension to the base scheduler is preemptive scheduling. This feature can be configured to split the execution of an FWE into short discrete slices. Using this, we want to allow other jobs to make progress on the same core during FWE execution, so the system remains responsive. When executing multiple FWEs concurrently (introduced in Section 5.3), this feature is required to avoid one FWE blocking other FWEs, as the execution time of an FWE may be unpredictable. We use the same M-mode timer as the base scheduler to divide the untrusted components and the FWE into slices by preempting their execution.

A visualization of the preemptive scheduling feature is given in Figure 5.2. At time a_1 the M-mode timer triggers the execution of the scheduler that immediately schedules the first slice of the FWE for the duration T_e. This is implemented by reprogramming the M-mode timer to elapse after time T_e. When the timer elapses, it is reprogrammed again but this time with the interval T_u during which untrusted code is executed. At $a_{1,1} = a_1 + T_e + T_u$ the FWE scheduler is queried again and starts a new cycle by setting the timer and switching to the enclave. These cycles repeat until the enclave is done with its work for the current iteration and explicitly stops itself (C_{stop}) as described in Subsection 5.1.2. After this explicit stop, the M-mode timer is started with the remaining time of T, which is usually a longer period, where the untrusted code continues its execution uninterrupted until the timer elapses again at a_2. In our prototype implementation T is five seconds, T_e is two milliseconds, and T_u is one millisecond, but all values are customizable. Our values for T_e and T_u proved in our tests to be a good balance between FWE and the rest of the system. In summary, the scheduler divides the FWE execution into short slices and alternates the execution between the FWE and untrusted code at a high frequency.

The preemptive scheduling aims to increase the responsiveness of the system and also interrupts FWEs for other FWEs. The execution of an FWE without preemption blocks the entire core for a possibly unknown duration. This might be undesirable especially on single-core systems but also on multicore systems because we do not usually know what other OS scheduled jobs currently run on that affected core. Furthermore, it allows fair scheduling of concurrent FWEs which is detailed in Section 5.3.

To conclude this extension, blocking potentially critical processes for an unspecified time period is dangerous and should be avoided. The preemptive scheduler provides a predictable scheduling approach by using discrete slices for running FWEs. It ensures the execution of FWEs and provides reliable timings to allow predictable progress in the trusted and also untrusted domain.

5.2.1 No OS Dependencies

With our approach using preemptive scheduling, we avoid needing to rely on the cooperation of the untrusted OS. One alternative without preemptive scheduling would be to notify the untrusted OS of an upcoming context switch to the FWE. In that case, the OS would be given some time to react and move all processes away from the affected core. However, if the OS is not cooperative, such a notification could have no effect, and we would still block the core with FWEs. Our preemptive scheduler on the other hand, always keeps the affected core usable for other tasks including other enclaves (original enclaves and FWEs alike). Furthermore, this approach does not require any modification to the untrusted OS and therefore also supports other operating systems than Linux.

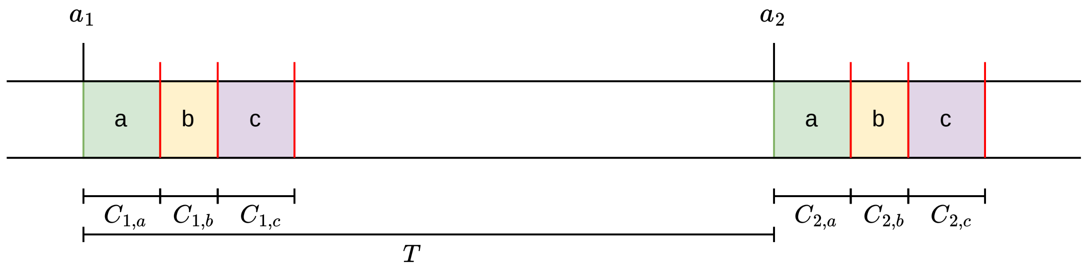

Figure 5.3: Concurrent FWE scheduling with three FWEs a, b, and c.

5.3 Concurrent Firmware Enclave Execution

Our second extension enables our FWE scheduler to manage and schedule multiple concurrent FWEs. We achieve this with a simplistic but robust approach that still fulfills the availability requirements of our use case.

As introduced in Section 5.1, our scheduling is realized by periodic hardware timer interrupts which trigger context switches between the FWE and the untrusted system. The scheduler basically divides the processor time into a block for FWEs and a block for all the remaining, less critical tasks (e.g., untrusted OS and normal enclaves). To add support for concurrent execution of multiple FWEs, we keep this concept and extended the FWE block to fit the execution of all FWEs. In the current case of non-preemptive scheduling, it is as simple as executing all FWEs one after the other and return control to the untrusted system once all these FWEs stopped.

Example An example of our technique for non-preemptive concurrent scheduling is illustrated in Figure 5.3 with three concurrent and independent FWEs a, b, and c. As soon as the interval timer elapses at a_1, the scheduler selects enclave a and transfers control to it. When this enclave is done, it explicitly stops itself (marked by the red vertical marker) and returns control to the scheduler. The scheduler then schedules enclave b next and so on. Only after all three enclaves finished their execution for the first period, the scheduler returns control to the untrusted system. All following periods are scheduled identically. This concept is robust, as it guarantees the execution of all FWEs in every period based on the M-mode timer primitive explained in Section 5.1. Furthermore, the previously mentioned simplicity of this design comes from the low configuration required, as FWEs do not need to be adjusted. The periodic interval length T remains the only configurable setting.

Limitations We argue, that having such a static and shared period length T is an adequate base for any FWEs, as they can simulate dynamic intervals on their own with a granularity equal to the static periodic interval T. For example, FWE x needs to be executed exactly every two minutes, and FWE y exactly every five minutes, then the static interval length would be set to one minute. FWE x executes on every second activation while FWE y executes on every fifth activation and ignoring all other activations. Note, that ignoring means directly stopping with `stop_fwencl()` without doing any work aside from the context switches. Therefore, we consider implementing native support for dynamic intervals (e.g. by using virtual timers) a performance optimization and future work.

As well as with single non-preemptive FWE scheduling, also in non-preemptive concurrent FWE scheduling one FWE can perform a DoS attack on that core by never returning. This is not an issue when executing one single FWE, because we do not guarantee availability for untrusted code in our use case, but it is an issue if multiple FWEs want to be executed. A solution to this vulnerability is presented in the following subsection.

5.3.1 Concurrent Preemptive Firmware Enclave Execution

Running multiple FWEs concurrently also works with preemptive scheduling. The concept is almost identical to running a single FWE preemptively, but we have to consider the fact that the execution time of FWEs is variable, so one FWE might need fewer execution slices than others. We solved this by keeping track of which FWEs stopped explicitly, so only slices of unfinished enclaves get scheduled. The result is a scheduling similar to the round-robin algorithm that schedules slices of multiple FWEs in circular order.

Example In Figure 5.4 we show an example of this concept. The example consists of the same three FWEs a (green), b (yellow), and c (purple) as in Figure 5.3. When the interval timer elapses at a_1, the scheduler schedules one slice of FWE a. As soon as a gets interrupted (after T_e, see Figure 5.2), a slice of FWE b is scheduled which is also interrupted after T_e, and so on. Finally, after FWE c gets interrupted, the execution returns to the untrusted system for the length of also one slice. This pattern repeats until FWE b stops, marked by the first red vertical marker. Since the preemption scheduler does not schedule stopped enclaves, the last group of slices does not include a slice from FWE b anymore.

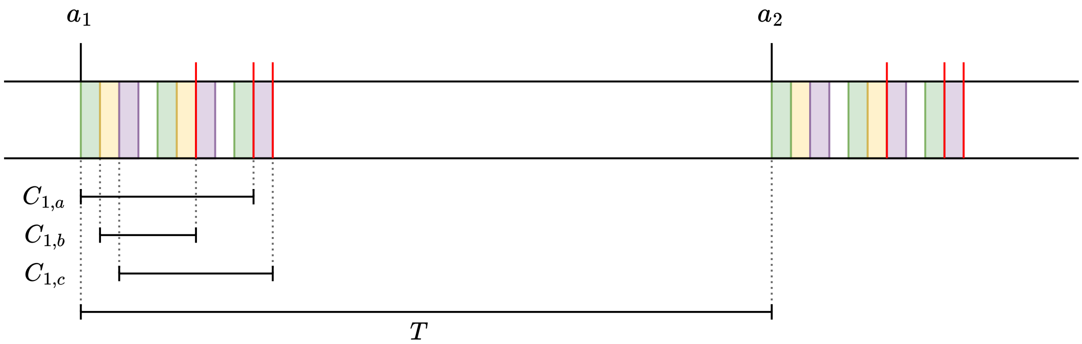

Figure 5.4: Preemptive and concurrent FWE scheduling with three FWEs a (green), b (yellow), and c (purple).

5.3.2 Implementation

We propose implementing the preemptive scheduler using the M-mode timer. The challenge in the implementation is to manage two parallel timers, one for the periodic execution with interval T and the other for preemption, with only one hardware timer. We decided to let the scheduler manage two virtual timers which are based on the hardware timer. When the scheduler is invoked at the beginning of a new period (e.g., in Figure 5.4 at time a_1), the timer for the next periodic interrupt is calculated ($a_2 = a_1 + T$) and stored. Next, the scheduler starts with the preemptive execution of FWEs using the M-mode timer for preemption but every time the scheduler is invoked, it checks whether the time counter surpassed a_2. If true, it activates the other FWEs again, so they are scheduled alongside the still running FWE. When the last FWE finally stopped, the M-mode timer is free and is scheduled to interrupt at a_2 again. In Algorithm 2 we show a simplified version of a possible implementation. The variable $active\_enclaves$ stores a bitmap of active FWEs which have not explicitly stopped themselves yet. At invocation of the scheduler after the periodic interrupt, all FWEs are activated (line 1). The scheduler then executes slices in round-robin fashion. In each iteration, it checks whether the current time already surpassed $next\_interval$ (line 4). If true, all FWEs are activated and the time for the next periodic interrupt is calculated again. As soon as an FWE stops itself, it is removed from $active\_enclaves$ (line 11). After all FWEs stopped, the scheduler sets the M-mode timer to the next periodic interrupt time stored before (line 12).

DoS Mitigation The management of the two parallel timers is crucial for the availability guarantee of the scheduler. In a previous design draft, we found a DoS vulnerability because it did not manage both timers in parallel. We want to describe the vulnerable

Algorithm 2: Scheduler with virtual timer management

```
active_enclaves ← all
next_interval ← current_time() + T
while active_enclaves > 0 do
    /* If next interval elapsed, activate all FWEs        */
    if current_time() > next_interval then
        active_enclaves ← all
        next_interval ← current_time() + T

    is_explicit_stop ← execute_next_active_slice()

    /* If FWE stopped explicitly, deactivate it           */
    if is_explicit_stop = true then
        active_enclaves ← deactivate_executed_slice(active_enclaves)
set_mtimer(next_interval)
```

design draft here for two main reasons. First, it shows how our final design protects the availability of FWEs against malicious components and second, the implementation of this final design could not be finished before the submission of this thesis. Therefore, we explain the vulnerability of the previous draft (corresponding to current prototype implementation) in detail and discuss the mitigation we developed leading to the final design.

The key difference between our current vulnerable prototype implementation and the final design is the time at which the periodic interval timer T starts. In the vulnerable implementation the interval timer started only after the FWE stopped its execution. In case of multiple concurrent FWEs, it started after *all* FWEs stopped. This design decision was previously made to allow the implementation to work with one single timer, as there is only one M-mode timer available per core. During preemptive execution of FWEs, this M-mode timer is used to time the slices, so it cannot be used for the periodic interval timer in parallel. Therefore, we let the interval timer start as soon as the timer was not used for FWE preemption anymore. Although this implementation worked great for single FWE execution, it turned out to allow concurrent FWEs to DoS other FWEs by simply not stopping, causing the interval timer T to be delayed indefinitely.

Our scheduler aims to protect FWEs from DoS attacks by other (possibly faulty or malicious) FWEs, as FWEs are generally not part of our TCB (see Chapter 3). Therefore, a resilient design must not start the interval timer T after the last enclave stopped, because this stop may never happen. Our final design starts the interval timer at the same time the

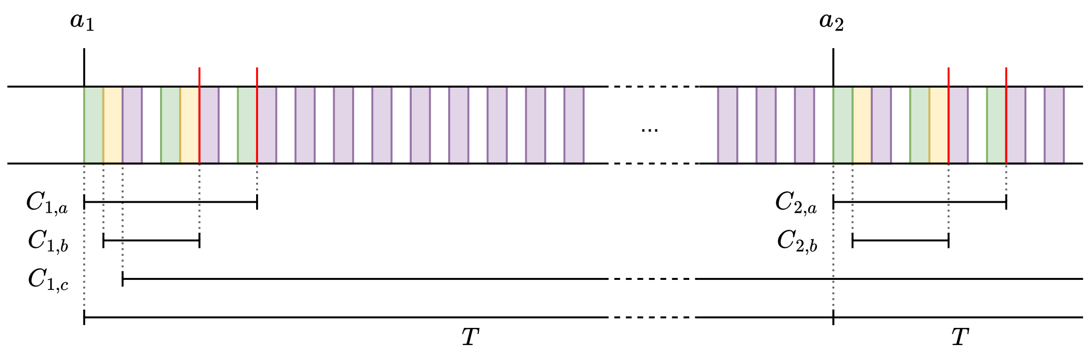

Figure 5.5: Preemptive scheduling attempted DoS attack from FWE c (purple).

first FWE starts executing (see Subsection 5.3.2). If an FWE takes longer than T to stop, the scheduler schedules the other FWEs alongside the still running FWE again, making a DoS attack ineffective. An example is illustrated in Figure 5.5, where the FWE c (purple) takes longer to finish than the interval length T. At time a_2 the scheduler recognizes that a new period started, so the other FWEs are scheduled again.

5.4 Support for "Invisible Traps"

The scheduling of FWEs is implemented using M-mode timer interrupts. Such timer events can interrupt the system at any point in execution, so the resulting trap should be a so-called invisible trap that fully restores the interrupted environment. Keystone does not natively provide this property in its context switches, so we implemented them for our scheduler interrupt.

5.4.1 Original Context Switches are not Invisible

An interrupt is an asynchronous event that causes an unexpected trap. In case of the FWE scheduler interrupts, the untrusted OS does not expect these and may be interrupted in the middle of execution. This requires the traps to be handled transparently, so that the untrusted system can resume its execution normally after the trap returned. In other words, the untrusted system is not aware of this trap, which is called an invisible trap and is common for timer interrupt trap handlers.

In the original implementation of Keystone, context switches between the untrusted host and an original enclave are implemented as *exception* trap handlers that are called by executing the `ECALL` instruction. The host application of the original enclave executes

`ECALL` to initiate a trap that handles the context switch into the enclave. Such exception trap handlers exchange input parameters and return values by modifying registers and are therefore not invisible traps. We cannot use these for handling the M-mode timer interrupt trap, because they modify the state of the interrupted execution environment without fully restoring it.

In detail, after executing the SM function `sbi_sm_stop_enclave()` to return from an enclave to the previous execution environment, the SM restores the previous register values except for the registers `a0` and `mepc`.

- `a0` contains an error code for the stopping action. After switching back to the host environment, the host application can check the reason for stopping and react accordingly.
- `mepc` contains the address where the `mret` instruction returns to. It is incremented by 4 because the trap handler expects to return from an `ECALL` instruction. When an ECALL trap is taken, `mepc` contains the address of the `ECALL` instruction, so the handler must increment `mepc` to point to the instruction following the `ECALL` before returning from the trap.

From the perspective of the untrusted system, such an unexpected but non-transparent timer interrupt would mean that `a0` is suddenly altered, and it would skip up to two instructions, possibly resulting in an illegal instruction.

5.4.2 Invisible Traps for Firmware Enclave

In order to achieve reliable context switches to an FWE via M-mode timer interrupts, we implemented invisible traps to the Keystone trap handler. The SM can distinguish between original enclaves and FWE. If the SM handles context switches of an FWE, the involved trap handler performs a fully transparent context switch by restoring the environment state precisely as it was before the trap. This allows the FWE scheduler to safely interrupt the system at any point in time. Our implementation is also able to interrupt other enclaves (original enclaves and FWEs).

5.5 Availability Offerings

In this chapter, we presented multiple scheduler configurations but not all of them provide the same availability guarantees. This section clarifies which exact feature configurations provide the availability we seek for our use case.

Single FWE & non-Preemptive Scheduling This configuration provides availability for a single FWE guaranteed by the periodic M-mode timer interrupt. Since this execution model blocks the rest of the system until the FWE stops, we can only recommend this configuration for predictable and very short running FWEs. Due to a minimum number of context switches between the FWE and untrusted context, this configuration incurs only a small amount of overhead.

Single FWE & Preemptive Scheduling Scheduling a single FWE preemptively allows the rest of the system to also make progress, resulting in a more responsive system over all. Availability is still given with this configuration, because the FWE scheduler is always able to interrupt the untrusted system after its slice to switch to the FWE again. The preemptive execution model comes at the cost of creating potentially many more context switches inducing a higher scheduling overhead in total. Furthermore, interrupting the FWE increases its total computation time. We strongly recommend this configuration, if the execution time of the FWE is either long or unknown. This is likely to be the case when relying on external factors, such as network communication or reading from external data drives.

Multiple FWEs & non-Preemptive Scheduling The combination of non-preemptive scheduling and multiple concurrent FWEs does not guarantee the execution of specific FWEs, because an indefinitely running FWE can block the other FWEs. Although the scheduler is guaranteed to schedule one enclave, it is not specified in our design which one is executed first. Only the execution of the first scheduled FWE in every period is guaranteed by the periodic M-mode interrupt. This first FWE could block all following FWEs. However, our implementation always schedules the FWEs in ascending order according to their ID, so the enclave with ID 0 is scheduled first and gets the availability guarantee. Due to the unspecified scheduling order in our design, we do not recommend this configuration for any purpose involving execution guarantees. Instead, we suggest using the preemptive scheduling configuration to schedule multiple FWEs. This configuration was only described in this chapter as a less complicated introduction to concurrent FWE scheduling. Furthermore, executing multiple FWEs consecutively and uninterrupted may result in a long break for the rest of the system.

Multiple FWEs & Preemptive Scheduling This configuration includes all of our features and guarantees periodic execution of all FWEs while also assuring progress for the rest of the system. This is the default configuration that works in all scenarios. The division of the execution time into constant slices ensures every FWE is scheduled in predictable intervals and makes progress. The availability comes at the cost of potentially many added context switches which increases the totaled scheduling overhead.

6 Prototype: Updater Firmware Enclave

In this chapter we demonstrate the feasibility of our contributions with an FWE that guarantees the secure update of another FWE. It demonstrates our main use case, that is the guaranteed execution of a secure updating procedure.

We have built a prototype updater FWE that securely updates another FWE to a newer version. We refer to the outdated enclave that gets updated as the *target FWE*, and we refer to the new enclave as the updater's *payload*. Both are statically compiled into the firmware equal to normal FWEs as described in Chapter 4 and depicted for this prototype in Figure 6.1. The target is a normal FWE which has no special permissions and is not part of our TCB. It automatically starts at boot together with the updater enclave. The payload is also a normal FWE with no special permissions and not part of our TCB, but it is not started at boot because it is only in memory to get copied during update.

With this updater prototype, we want to show that the FWE and the availability scheduler can be used to update parts of the system securely and in a guaranteed way without an attacker blocking it. We choose to update an FWE running concurrent to the updater, because it is one of the more challenging update scenarios, due to its PMP protection and privileged scheduling. We are confident, that our framework can also be used to build an updater targeting other components of the system, such as OpenSBI and the untrusted OS, but we leave its implementation to future work. Seamlessly updating OpenSBI would require an A/B system [Google LCC, 2024]. Updating the untrusted OS would require the updater FWE to incorporate a filesystem driver and to reboot the OS. For OTA updates with identical availability guarantees to this prototype updater, an in-FWE network stack would be needed to obtain the payload. This is also left for future work.

Since our updater's payload is compiled into the firmware, it is protected by its own PMP region against accesses from untrusted code. However, the updater performs an additional signature verification on the payload to verify its origin. The verification has no real practical use in our specific prototype, because the signature and public key of the signing party are part of the firmware image, just like the payload. This means, the same party that adds the payload to the image, also writes the signature and the public key into the updater enclave, so the verification adds no security. Verifying the payload is more useful in scenarios, where the payload comes from external untrusted sources such as external

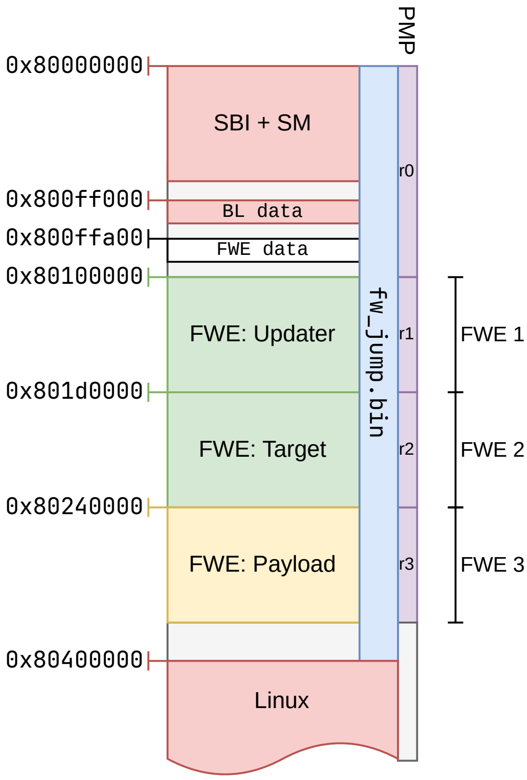

Figure 6.1: System memory map for updater prototype.

storage devices or the internet. Nevertheless, we included it in our prototype to prove the possibility to implement sophisticated secure update mechanisms into an updater FWE.

6.1 Updater Enclave

The updater enclave is a special FWE with permissions that normal FWEs do not have. Due to its purpose to update another FWE, the SM grants the updater FWE the following special permissions:

1. Read access to the payload.
2. Read and write access to the target FWE's memory range to write the payload into it.
3. Enable and disable scheduling of FWEs, so the target is not executing during the update.
4. Reset the SM internal state of FWEs, so the updated target can boot into a clean state.

Permissions (1.) and (2.) are necessary to allow the updater FWE writing the update to the target. By specifically giving these permissions to the updater instead of removing the PMP protection from the target FWE, we ensure that untrusted code cannot interfere with the update procedure. Permission (3.) is required, so that the updater can disable the scheduling of the target FWE in order to update its memory contents without the FWE trying to run at the same time. If an FWE is marked as disabled, the scheduler will not schedule any further execution of it. The SM holds internal information about the target FWE such as its last register contents before context switch, so the updater needs permission (4.) to reset this context allowing the SM to reboot the target FWE with its updated contents.

The updater FWE requests all these permissions from the SM, and the SM decides whether it grants all, none, or a subset of the requested permissions. With this approach, the SM remains in control of the updater FWE and protects the remaining system against unintended modifications. In our current implementation, the SM only grants the updater permissions to modify the state of the one specific target FWE and to read the payload from memory. All other requests are rejected, regardless of whether they come from the updater or other FWEs.

Updater in TCB The answer to the question, whether the updater enclave needs to be part of the TCB, depends highly on the updater's purpose and permissions it can acquire from the SM. An updater that should be able to update OpenSBI, is definitely part of the TCB, as it needs permission to write to M-mode code ranges. On the other hand, an updater that only updates a component of the untrusted system, must not necessarily be part of the TCB. In our updater scenario, we consider the updater FWE a part of our TCB, because the target FWE could be a critical part of the system and is fully exposed to the updater.

6.1.1 Update Procedure

The basic concept of the updater is to overwrite an active but presumably outdated FWE with a new FWE. In the following, we describe the steps the updater performs in order to update the target.

1. Disable the target FWE.
2. Acquire read and write access to the PMP region of the target FWE.
3. Acquire read and write access to the PMP region of the payload.

4. Map physical addresses of target FWE and payload to user-accessible virtual addresses, so the updater eapp can operate on these ranges.
5. Verify the signature of the payload.
6. Copy the payload contents into the target FWE memory (using `memcpy`).
7. Drop the memory access permissions to the target FWE and payload.
8. Reset the target's enclave context, so it can be rebooted.
9. Enable the target FWE again, so it can be scheduled and executed again.

Using this procedure, our updater can securely update running FWEs. Our scheduler ensures that the updater FWE is executed, therefore, the update is guaranteed. In step (7.) the updater drops its access permissions to the target FWE and payload to reduce its attack surface and probability of accidental modifications.

Testing Platform We tested our prototype on a generic RISC-V system running inside of QEMU. We emulated both a multicore system with four cores and a single core system. The FWEs and the payload were injected into the firmware image by our script as described in Section 4.2. The emulated system booted into a Linux environment alongside executing the FWEs and all runs successfully performed the update. To test our prototype on actual hardware, a lot of work has gone into the attempt to port Keystone to the *Microchip PolarFire SoC Icicle Kit*. Unfortunately, we halted our efforts after working on it for a significant time because we could not estimate how much additional time would be needed to finish the port. We also dropped planned experiments on the *StarFive VisionFive 2* board for the same reason. The main issue we had with porting was that we overestimated the compatibility of the (recently overhauled) Keystone build system with these boards.

6.2 Results

With our FWE updater we show the feasibility and practicability of our contributions. We are able to run a secure updater inside an FWE, and we can guarantee its execution. Our prototype can be extended to update other parts of the system as well as obtaining the payload from external sources. The full prototype containing the updater FWE, the target FWE, and the payload is available in our source code.

7 Related Work

There are two categories in the field of secure IoT updates that are most related to our work. The reboot oriented approach, that isolates trusted from untrusted code in a temporal way by running trusted code before booting the untrusted OS, and the secure scheduler approach that uses more complex runtime isolation but provides greater flexibility.

7.1 Reboot Oriented Updates

Required security primitives for TEEs are often missing on low-power microcontrollers, so it is hard to achieve proper runtime isolation on such devices. Thus, multiple approaches [Suzaki et al., 2020, Xu et al., 2019, Huber et al., 2020] deploy a secure early boot environment that can verify the current firmware and perform updates before booting the into the untrusted OS.

CIDER [Xu et al., 2019] proposes an Authenticated Watchdog Timer (AWDT) that enables a remote party to issue reboots, so updates and system recovery can be initiated remotely. The AWDT is a newly proposed hardware component specifically designed for CIDER and not available on current RISC-V hardware. AWDT is similar to a traditional watchdog timer but can only be served by authenticated tickets (signed by the remote party in this instance). If it is not served correctly or not in time, it will reset the device, eventually entering the secure early boot environment. This way, a remote party can stop serving tickets and force a reboot. Alternatively to baking AWDT into hardware, it can also be implemented by an external coprocessor or by software using a TEE [Huber et al., 2020, Suzaki et al., 2020]. In CIDER, updates and system recovery are performed by "gated boot", a program that executes inside the early boot environment before any other firmware starts. At this time, the device is considered to be in a secure state because it solely executed trusted code. The secure boot environment protects its own integrity by read-write-latches that can only be set once per power cycle. CIDER does not require automatic periodic reboots, because reboots are issued remotely on demand.

Lazarus [Huber et al., 2020] is based on CIDER and combines a TEE (specifically Arm

TrustZone-M[10]) with the reboot oriented approach. It leverages the TEE to implement the distinctive hardware features, i.e., read-write-latches, the AWDT, and to manage the access to critical peripherals such as the flash controller. Those peripherals are protected by a trusted peripheral handler inside the secure environment. They aim to mitigate attacks on the hardware such as wearing out the flash. Availability is forced by the AWDT implemented in Arm TrustZone that resets the device if it is not serviced.

Another publication that is based on CIDER is Reboot-Oriented IoT [Suzaki et al., 2020]. It extends CIDER similar to Lazarus by implementing CIDER primitives in a TEE. Additionally, Reboot-Oriented IoT provides hash-based runtime memory integrity checks and periodic autonomous reboots that ensure periodic execution of the trusted early boot environment. In contrast to CIDER that is a push-based approach, Reboot-Oriented IoT periodically pulls for updates. Pull-based approaches are beneficial for devices, that are only sporadic connected to the internet [Bellissimo et al., 2006].

A significant downside of all three reboot oriented solutions is that they require a reboot to enter a secure environment to check for updates or verify the currently running firmware. In other words, the context switch overhead is an entire system reboot. Additionally, Arm TrustZone based implementations only have one single trusted environment called "Secure World". Isolation between trusted applications in this world is managed by a secure OS which drastically increases the size of the TCB and the therefore attack surface [Cerdeira et al., 2022].

7.2 Secure Scheduler

Schedulers designed to provide availability are a well known concept. They play a significant role in real-time systems where availability is one of a real-time scheduler's most important traits. RT-TEE [Wang et al., 2022] and Aion [Alder et al., 2021] are approaches to schedule enclaves or applications inside the trusted environment using real-time enabled schedulers.

RT-TEE is a TEE design based on Arm TrustZone for real-time cyber-physical systems. It leverages a secure hardware timer that traps into a trusted environment to guarantee the availability of trusted components. Such a timer is available on all Arm Cortex-A cores. The included real-time scheduler is an event-based hierarchical scheduler with two layers, a trusted and an untrusted layer. Compared to a single level scheduler that would include the OS scheduler, the hierarchical design improves on minimizing the TCB by moving the

[10] `https://www.arm.com/technologies/trustzone-for-cortex-m` (accessed 07.02.2025)

scheduling of untrusted OS applications back to the OS. RT-TEE's threat model includes a strong adversary, who can execute arbitrary code in the untrusted OS and may try to block secure tasks from executing. While RT-TEE's design may work on other platforms like RISC-V as well, currently its only implemented for Arm using TrustZone.

Aion is another approach on adding availability guarantees to a TEE, by introducing an enclaved real-time scheduler. In contrast to RT-TEE, Aion extends the hardware with an additional instruction added specifically to manage atomic sections. The same instruction can also be found in [Masti et al., 2012]. It is needed due to insufficient security primitives (privilege modes) on their target platform, which are 8 or 16-bit microcontrollers (e.g., TI MSP430). Relying on customized hardware is impractical for wide adoption and only reasonable in very specialized scenarios. Most platforms will integrate general purpose hardware and off-the-shelf processors due to fast and profit oriented development. RISC-V, on the other hand, provides sufficient security primitives in its base Privileged Specification [RISC-V International, 2024b].

8 Conclusion

In this final chapter, we summarize and discuss the security properties of our results. We also discuss some ideas for future work.

8.1 Summary

The vast majority of IoT devices lacks a secure update mechanism that can provide availability. A secure update mechanism, that can guarantee the application of security updates is very important, because security updates for vulnerable systems may arrive when an attacker already took over control of parts of the system. Prior works in this field mostly focused on the Arm platform, and therefore designed availability solutions around Arm TrustZone. Arm TrustZone however provides only two hardware isolated security domains, so as a consequence, a critical updater would live in the same security domain as some other non-critical applications. Keystone is a TEE that offers stronger isolation between enclaves but lacks a solution to guarantee the execution of enclaves. In this thesis, we presented FWEs and a secure FWE scheduler, that combined are able to guarantee the execution of an updater enclave, even with an OS-level attacker trying to prevent it.

We developed the FWE as a new type of Keystone enclave, which runs in its own environment fully isolated from all untrusted components on the system. All the memory and data an FWE needs is provided by the PMP-protected trusted firmware image instead of the untrusted OS, so it even runs without any functional OS present, making it impossible for an OS-level attacker to hinder the FWE's execution. An FWE allows for updates to be applied securely, as it provides confidentiality and integrity through Keystone like original Keystone enclaves. In addition to that, an FWE can only be (externally) interrupted by the trusted firmware, leaving no additional attack surface for OS-level attackers apart from the firmware interfaces.

To guarantee availability for FWEs, we developed a secure FWE scheduler that we implemented into the SM. This scheduler operates in the trusted and most privileged M-mode above the untrusted OS scheduler. It adds a new scheduling level by scheduling FWEs and the untrusted OS. The OS then has its own scheduler for untrusted applications and original enclaves. We achieve the availability guarantee by invoking the scheduler with a privileged M-mode timer that always takes precedence over code running in the lower privileged S-mode and U-mode. Since the scheduler runs with M-mode privileges, it also cannot be interrupted by any lower privileged events. Our implementation allows for scheduling multiple FWEs that are all guaranteed to be executed in pre-defined intervals, while still allowing for the untrusted components to make some limited progress to keep the rest of the system available as well.

In our updater example, we showed our concepts of the FWE and the scheduler to provide an enclave the ability to securely and reliably update system components while keeping the rest of the system responsive. Therefore, we can guarantee secure system component updates with minimum downtime and minimal TCB size.

All the discussed features and examples are implemented on top of the newest publicly available version of Keystone with an even newer Buildroot version than currently available in the original Keystone repository. We plan to make our implementation publicly available, and we will offer our additions to the upstream project some time in the future.

8.2 Future Work

In this section we discuss some open challenges and ideas in regard to our design and implementation.

8.2.1 Firmware Enclave on Monitoring Core

Our current design is built to run FWEs on normal application cores which we have to share with the untrusted system. It would be an interesting experiment to run the updater FWE on a monitor core that is available on some RISC-V processors, such as the StarFive JH7110 and the Microchip PolarFire SoC. These cores are trimmed-down RISC-V cores but should be technically capable to run something like an updater FWE. A key challenge would be, that such monitor cores only offer M-mode and U-mode but no S-mode, meaning that a new type of Keystone enclave is required that runs eapps without a runtime (bare-metal).

8.2.2 System Recovery using Firmware Enclave

It would also be interesting to see, how FWEs can be used for secure system recovery. Update and recovery are quite similar tasks. In fact, our updater prototype (Chapter 6) could be viewed as an FWE recovery tool, as it uses a read-only payload and overwrites an FWE with it contents. If we assume that the payload is a recovery image for the target FWE, the updater is effectively recovering the FWE. But for future research, it would be interesting to find a solution for full system recovery, where the recovery image does not fit into the main memory of the device.

8.2.3 OS Scheduling for Firmware Enclave

In Chapter 5 we described our FWE scheduler based on M-mode hardware timer interrupts. In this approach we interrupt the system and need to guarantee "invisible" context switches. We think that this approach creates an excellent prototype for our scheduler, as it does not require any communication nor modifications of the untrusted OS. Despite, being OS independent is an advantage of our implementation, cooperating with the untrusted OS might result in better performance both for the FWE and for the untrusted part of the system.

OS-Based Scheduling and Watchdog The idea is to extend the Keystone kernel driver (currently implemented only for Linux) which then spawns a job that is similar to the host application for original enclaves but for FWEs. The main difference would be that the host application for FWEs is not responsible for providing executables and creating the enclave but only for allocating processor time to the already initialized FWE. If the untrusted host OS is benign, it will schedule the FWE eventually.

Since we cannot trust the host OS to be cooperative and actually schedule the enclave, this approach must include a fallback mechanism to ensure FWE execution. The fallback could be a privileged watchdog timer that executes the FWE in the same way as described in Chapter 5. In order to only trigger the timer when needed due to an uncooperative OS, the idea is to let the FWE regularly report its progress to the SM which resets the watchdog timer. If the enclave's next report takes too long, the watchdog timer expires and triggers a forced context switch to the FWE. We found a similar approach in CIDER [Xu et al., 2019] with the *authenticated watchdog timer*.

The advantage of that idea is, that the host OS is indirectly aware of the execution of the FWE and schedules other processes accordingly. On multicore systems, this means the FWE can be fully blocking without disrupting the rest of the system, because all other tasks will be scheduled to the remaining cores. Therefore, less context switches are needed which greatly reduces the total overhead of FWEs.

A disadvantage is a significantly increased complexity resulting in a larger TCB due to added OS communication and the need to maintain specific OS/kernel version support. Furthermore, our already implemented timer based implementation basically remains as a fallback. The advantages we see only apply on multicore systems, because on single-core systems there is no other core where other tasks could be scheduled to. In summary, we consider it an implementation optimization specific to multicore systems.

Announce Firmware Enclave Execution An alternative to the OS-based scheduling is announcing the FWE execution to the host OS. This idea is a compromise between our current M-mode timer based implementation and the previously discussed OS-based scheduling with a watchdog timer. In this alternative approach, the scheduling of FWEs would still be done by the M-mode timer but instead unexpectedly interrupting the untrusted system to execute the FWE, the scheduler first announces the upcoming execution of the FWE to the host OS. The Keystone kernel driver inside the host OS would be extended to be able to reserve a core for the FWE, so all other tasks are scheduled on the remaining cores. The new scheduler can then assume, that the core for the FWE is free, and it would execute the FWE in one continuous block. If the host OS is compromised and ignores the announcement, the FWE would still be guaranteed to execute, but the remaining system may be blocked.

References

[Alder et al., 2021] Alder, F., Bulck, J. V., Piessens, F., and Mühlberg, J. T. (2021). Aion: Enabling open systems through strong availability guarantees for enclaves. In Kim, Y., Kim, J., Vigna, G., and Shi, E., editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1357–1372. ACM.

[Allwinner Technology Co., Ltd., 2022] Allwinner Technology Co., Ltd. (2022). D1-H user manual. `https://github.com/DongshanPI/Awesome_RISCV-AllwinnerD1/blob/master/Tina-SDK/Hardware%E7%A1%AC%E4%BB%B6%E7%B1%BB%E6%96%87%E6%A1%A3/%E8%8A%AF%E7%89%87%E6%89%8B%E5%86%8C/D1-H_User%20Manual_V1.0.pdf`.

[Asanovi and Patterson, 2014] Asanovi, K. and Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley.

[Bellissimo et al., 2006] Bellissimo, A., Burgess, J., and Fu, K. (2006). Secure software updates: Disappointments and new challenges. In *1st USENIX Workshop on Hot Topics in Security, HotSec'06, Vancouver, BC, Canada, July 31, 2006*. USENIX Association.

[Cerdeira et al., 2022] Cerdeira, D., Martins, J., Santos, N., and Pinto, S. (2022). Rezone: Disarming TrustZone with TEE privilege reduction. In Butler, K. R. B. and Thomas, K., editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2261–2279. USENIX Association.

[Dhobi et al., 2019] Dhobi, R., Gajjar, S., Parmar, D., and Vaghela, T. (2019). Secure firmware update over the air using TrustZone. In *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*, volume 1, pages 1–4.

[Google LCC, 2024] Google LCC (2024). A/b (seamless) system updates. `https://source.android.com/docs/core/ota/ab` (accessed 02.02.2025).

[Huber et al., 2020] Huber, M., Hristozov, S., Ott, S., Sarafov, V., and Peinado, M. (2020). The lazarus effect: Healing compromised devices in the internet of small things. In Sun, H., Shieh, S., Gu, G., and Ateniese, G., editors, *ASIA CCS '20: The 15th ACM Asia*

Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020, pages 6–19. ACM.

[Lee et al., 2020] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., and Song, D. (2020). Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys20.

[Masti et al., 2012] Masti, R. J., Marforio, C., Ranganathan, A., Francillon, A., and Capkun, S. (2012). Enabling trusted scheduling in embedded systems. In Zakon, R. H., editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 61–70. ACM.

[Microchip Technology Inc., 2024] Microchip Technology Inc. (2024). PolarFire SoC MSS technical reference manual. `https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/ReferenceManuals/PolarFire_SoC_FPGA_MSS_Technical_Reference_Manual_VC.pdf`.

[RISC-V International, 2024a] RISC-V International (2024a). The RISC-V instruction set manual: Volume I. `https://github.com/riscv/riscv-isa-manual/releases/tag/20240411`.

[RISC-V International, 2024b] RISC-V International (2024b). The RISC-V instruction set manual: Volume II. `https://github.com/riscv/riscv-isa-manual/releases/tag/20240411`.

[RISC-V International, 2024c] RISC-V International (2024c). RISC-V supervisor binary interface specification. `https://github.com/riscv-non-isa/riscv-sbi-doc/releases/tag/vv3.0-rc1`.

[RISC-V International, 2025] RISC-V International (2025). RISC-V IOPMP architecture specification. `https://github.com/riscv-non-isa/iopmp-spec/releases/tag/v0.9.2-RC3`.

[StarFive Technology Co., Ltd., 2023] StarFive Technology Co., Ltd. (2023). StarFive JH-7110 technical reference manual. `https://doc-en.rvspace.org/JH7110/PDF/JH7110_TRM_StarFive_Preliminary_V2.pdf`.

[Statista, 2024] Statista, T. I. (2024). Number of internet of things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033. `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/` (accessed 18.06.2024).

[Suzaki et al., 2020] Suzaki, K., Tsukamoto, A., Green, A., and Mannan, M. (2020). Reboot-oriented IoT: Life cycle management in trusted execution environment for disposable IoT devices. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 428–441. ACM.

[Wang et al., 2022] Wang, J., Li, A., Li, H., Lu, C., and Zhang, N. (2022). RT-TEE: real-time system availability for cyber-physical systems using ARM trustzone. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 352–369. IEEE.

[Waterman et al., 2011] Waterman, A., Lee, Y., Patterson, D. A., and Asanovi, K. (2011). The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley.

[Xu et al., 2019] Xu, M., Huber, M., Sun, Z., England, P., Peinado, M., Lee, S., Marochko, A., Mattoon, D., Spiger, R., and Thom, S. (2019). Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1415–1430. IEEE.