HAW HAMBURG

MASTER THESIS
Lars Pfau

Protecting Cryptographic Secrets with RIOT OS on RISC-V Secure Embedded Systems

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Lars Pfau

Protecting Cryptographic Secrets with RIOT OS on RISC-V Secure Embedded Systems

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Thomas C. Schmidt
Supervisor: Prof. Dr. Martin Becke

Submitted on: July 15th, 2026

Lars Pfau

Thema der Arbeit

Protecting Cryptographic Secrets with RIOT OS on RISC-V Secure Embedded Systems

Stichworte

Internet of Things, IoT Security, RISC-V, Trusted Execution Environments

Kurzzusammenfassung

Diese Arbeit untersucht die Absicherung von kryptographischen Operationen auf RISC-V Mikrocontrollern in IoT-Anwendungen. Ziel der Arbeit ist es, Angreifer daran zu hindern, Schwachstellen in dem Echtzeitbetriebssystem RIOT auszunutzen, um gespeicherte private Schlüssel aus der Ferne auszulesen. Zu diesem Zweck wird RIOT OS so portiert, dass es im User-Mode auf RISC-V Mikrocontrollern läuft. Dafür wird eine minimale Firmware implementiert, die im Machine-Mode läuft. Diese Machine-Mode Firmware enthält einen Crypto Service, der alle kryptographischen Operationen durchführt. RISC-V Physical Memory Protection wird verwendet, um den Zugriff von RIOT auf den Firmware-Speicher zu beschränken. Dies verhindert, dass RIOT die vom Crypto Service gespeicherten Schlüssel auslesen kann. Der Crypto Service ist in die PSA Crypto API von RIOT integriert. Der entwickelte Prototyp wird durch eine Reihe funktionaler Tests und Benchmarks evaluiert. Die Ergebnisse zeigen, dass der Ansatz praktikabel ist und zusätzliche Sicherheit bietet. Weitere Untersuchungen zur Sicherheit des Ansatzes sind notwendig. Außerdem müssen die Echtzeitfähigkeit und die Performance des Ansatzes verbessert werden.

Lars Pfau

Title of Thesis

Protecting Cryptographic Secrets with RIOT OS on RISC-V Secure Embedded Systems

Keywords

Internet of Things, IoT Security, RISC-V, Trusted Execution Environments

Abstract

This work investigates the potential to secure cryptographic operations on constrained RISC-V microcontrollers in IoT applications. The goal of this work is to prevent remote attackers from exploiting vulnerabilities in the real-time operating system RIOT by reading private keys stored in memory. This is achieved by porting RIOT to run in user-mode on RISC-V microcontrollers and to implement a minimal firmware that runs in machine-mode. This machine-mode firmware contains a crypto service that performs all crypto operations. RISC-V physical memory protection is used to restrict access of RIOT to the firmware memory. This prevents RIOT from reading the keys stored by the crypto service. The crypto service is integrated with the PSA Crypto API. The prototype is evaluated through a series of functional tests and performance benchmarks. The results show that the approach is feasible and provides an additional layer of security. Further research on the security of the approach is needed to improve real-time capabilities and performance.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The security of constrained Internet of Things devices is an area of growing concern [49]. These devices are used in applications such as wireless sensor networks and building automation, in which power consumption and cost-effectiveness are key factors of hardware design. As a result, they are often built around microcontrollers with low-power wireless radios. The trend to connect microcontrollers to the Internet raises security concerns.

Microcontrollers, which have traditionally been used in embedded control systems, generally do not have virtual memory capabilities. This means that software has full access to the physical memory of the system, which includes all RAM, ROM, and peripherals. A common vulnerability found in many softwares is an out-of-bounds read [30]. These vulnerabilities are the result of a programming error where the developer does not properly validate user inputs before performing memory access. Such vulnerabilities can allow a remote attacker to gain access to arbitrary locations of memory. This could include private keys and other long-term secrets used to uniquely identify and authenticate the IoT node on the network, which can enable the attacker to impersonate the IoT node and gain deeper access to the system [20].

A solution proposed to address these security issues are trusted execution environments. The idea behind this technology is to separate the application into two or more execution environments using a hardware-based isolation mechanism so that each environment can only access its own memory [22]. Thus, an attack on one environment does not automatically have an effect on the others. The RISC-V instruction set architecture supports implementing trusted execution environments in 32-bit microcontrollers through a feature called physical memory protection [44]. This feature allows privileged software to efficiently restrict access to physical memory by unprivileged code unlike a full virtual memory implementation.

In this work, we address this security problem using the phyiscal memory protection feature of the RISC-V platform and protect security sensitive information such as cryptographic long term secrets. This thesis is organized as follows: Section 2 provides the

background knowledge necessary to understand this work. Section 3 presents related work. Section 4 states a specific attack scenario to be addressed in this work. Sections 5, 6, and 7 describe the implementation and its evaluation based on a set of benchmarks. This is followed by a discussion in Section 8 and a final conclusion in Section 9.

2 Background

2.1 RISC-V

RISC-V is a free and open instruction set architecture developed and standardized by the members of RISC-V International [38]. The instruction set supports a wide range of applications through a modular design using extensions and profiles. They allow the implementation of small, low-power 32-bit microcontroller CPUs up to high-performance 64-bit application processors [31].

The main RISC-V specification is divided into two volumes. An unprivileged specification [42] includes the basic integer, bit manipulation, floating point, and vector instructions, among others. A privileged specification [44] includes privileged modes, interrupt processing, and virtual memory.

2.1.1 Privilege Modes

The RISC-V privilege specification defines several privilege modes in which software can run. These modes include machine (M-mode), hypervisor (HS-mode), supervisor (S-mode), and user (U-mode). Of these four, machine-mode has the highest privileges with full access to the hardware. It is the only mode that all RISC-V compliant CPUs must support. It can be used to run bare-metal firmware or the secure monitor of a trusted execution environment. User-mode is the lowest privilege mode and can be used to run application code in user threads.

The specification recommends that RISC-V simple embedded systems implement only machine mode. RISC-V secure embedded systems should implement both machine- and user-mode. RISC-V systems with support for Unix-like operating systems should implement machine-, user-, and supervisor-mode. At the time of writing, currently available

RISC-V microcontroller platforms such as the SiFive E2 series [46], Espressif ESP32-C3 [16], and Raspberry Pi RP2350 [34] implement M+U-mode. Therefore, from now on, this work will assume RISC-V secure embedded systems as the target platform.

2.1.2 Physical Memory Protection

A feature that has made the RISC-V instruction set architecture (ISA) interesting for security research is physical memory protection (PMP). PMP is an isolation mechanism that can be efficiently implemented in microcontrollers without an MMU. It allows machine-mode software to define memory regions with a granularity as small as 4 bytes. Each region can set read, write, and execute permissions for lower privileged user-mode software to access physical memory. Implementations may have up to 64 PMP regions. However, some implementations, such as the SiFive FE310 and RP2350, support only 8 regions.

By default, user-mode software does not have access to any physical memory. PMP regions must be explicitly defined to grant access permissions. Each of these regions can be of one three types:

TOR Top of range regions. These allow for the definition of arbitrary areas of memory that extend from a start address set by the previous region to an end address.

NA4 Naturally aligned 4-byte regions. These regions are 4 bytes in size and start at a 4-byte aligned address.

NAPOT Naturally aligned power-of-two region. These regions are 2^N bytes in size, but at least 8 bytes. They start at a 2^N-byte aligned address.

2.1.3 Control and Status Registers

The RISC-V privileged architecture uses control and status registers (CSRs) defined in the Zicsr-extension [44]. These CSRs reside in their own address space. They are read and modified using dedicated atomic read-modify-write instructions. CSRs are used by a number of other RISC-V extensions to support use cases like the configuration of PMP regions and the retrieval of status flags from the floating-point unit. They are also used for interrupt handling. Some of the CSR addresses, such as those used for the floating-point `fcsr`, are unprivileged, which means that they can be accessed from any of the

privilege modes. Other CSRs can only be accessed by machine-mode software. This includes the CSRs that are used for interrupt processing, which can only be accessed from machine-mode on M+U-mode secure embedded systems.

2.1.4 Interrupt Processing

There are two types of traps defined by the RISC-V privileged specification [42]:

1. Exceptions. These are synchronous traps caused by memory access faults, invocation of system calls, and illegal instructions. They are non-maskable.
2. Interrupts. These are asynchronous traps. Examples are timer interrupts, software interrupts, or external interrupts.

Interrupt configuration and processing on RISC-V microcontrollers varies from implementation to implementation. It can be done using a core-level interrupt controller [41] and a platform-level interrupt controller [39] defined in their own specifications. At the ISA level, however, the process of handling traps is strictly defined by the RISC-V privileged specification.

Trap handlers on RISC-V secure embedded systems is always executed in machine-mode and require access to M-mode CSRs:

1. The `mtvec`-CSR is used to configure the address of the trap handler.
2. The `mcause`-CSR is used by the trap handler to determine the type of trap to handle.
3. The `mepc`-CSR contains the program counter of the instruction that was interrupted by the trap.
4. The `mie`-CSR can be used to mask interrupt causes.
5. The `mstatus`-CSR is used to globally enable or disable interrupts and to configure the privilege mode of the CPU.

A RISC-V CPU generally performs the following sequence of events when a trap is requested:

1. Interrupts are disabled globally and the global interrupt enable status at the time of the trap is saved in the `mstatus.MPIE` bit.
2. The privilege mode at the time of the trap is saved in `mstatus.MPP`.
3. The privilege mode switches to machine-mode.
4. The `mcause`-CSR is set to indicate the exception or type of interrupt that has caused the CPU to trap.
5. The CPU jumps to the address of the trap handler, configured in the `mtvec`-CSR.

Now, control flow has been transferred to the machine-mode trap handler, which can then handle the trap depending on the value of the `mcause`-CSR. When the trap handler is done, it terminates using the trap-return-instruction `MRET`. This instruction causes the CPU to:

6. Set the privilege mode to the mode previously saved in `mstatus.MPP`.
7. Set the global interrupt enable status to `mstatus.MPIE`.
8. Jump to the address in the `mepc`-CSR.

If the trap handler did not change any of the values of `mstatus.MPIE`, `mstatus.MPP`, or `mepc`, this means that the CPU continues at the instruction where it was interrupted.

2.1.5 Privileged Instructions

There are some instructions in the RISC-V ISA that can only be executed in the machine-mode. These include the trap return instruction `MRET`, the wait for interrupt instruction `WFI` and some CSR-instructions. CSR-instructions that access M-mode CSRs (indicated by the prefix 'm'), such as those used for trap handling, are also considered privileged instructions. This is because CSR-instructions have the address of the CSR encoded in the opcode itself. Therefore, there is no direct way for machine-mode software to delegate control of M-mode CSRs to user-mode software. The RISC-V specification recommends using trap-and-emulate to overcome this limitation [42].

2.2 RIOT OS

RIOT [7] is a real-time operating system for the constrained Internet of Things. It supports a wide range of low-end IoT platforms from 8-bit to 32-bit microcontrollers with different architectures. Advantage of using this RTOS is that it is built with a high degree of modularity. It also comes with many protocol stacks and device drivers already implemented. This makes it particularly well suited for research and rapid prototyping purposes.

Modularity is achieved using modules [36] and packages [37]. Dependencies between modules are modelled by include relationships.

Modules Modules are collections of code located within the RIOT source tree. Examples of modules are CPU targets and device drivers.

Packages Packages are a set of source code that is pulled from an external source. An example of a package would be an external library.

Pseudomodules Pseudomodules are modules that do not contain any code. They are used for dependency modeling and during the pre-processing of source files.

As of writing, RIOT supports only a limited number of RISC-V microcontrollers, such as the ESP32-C3 [15] and the SiFive FE310 [47]. Both are RISC-V secure embedded systems implementing machine-mode, user-mode and the physical memory protection feature. However, RIOT currently does not make use of the ability to run software in user-mode. Instead, all code expects to run in machine-mode and makes frequent use of privileged instructions. If the user chooses to enable it, PMP is used for data execution prevention.

2.3 PSA Crypto API

The PSA Crypto API [4] is part of the Arm Platform Secure Architecture. It represents a larger effort by Arm to unify the interaction of software with a variety of different crypto hardware accelerators and libraries found in embedded microcontrollers. An important design feature of the PSA Crypto API is the concept of key identifiers. The API provides generic interfaces for symmetric ciphers, digital signatures, cryptographic hashes, message authentication codes, and authenticated encryption. All of these operations are

performed on handles of the key material, not on the keys themselves. They are managed by the implementation of the PSA Crypto API and its drivers, which makes them opaque to the user. When importing or generating the key, the user specifies the location and persistence of the keys. The API will then dispatch all API requests transparently to the appropriate device driver. As a result, application software can easily take advantage of secure elements or other forms of protected key storage without having to rewrite the application for a specific hardware configuration.

Arm provides a reference implementation of the PSA Crypto API in its MbedTLS [3] library. However, RIOT does not use the MbedTLS implementation and instead provides its own implementation of the API [9].

3 Related Work

Similar work to this, but for ARMv8-M based microcontrollers, can be found in Arm Trusted Firmware-M and the TF-M crypto service[5]. TF-M is a solution from Arm Limited that provides Trusted Execution Environments based on the TrustZone-M feature in ARMv8-M Cortex-M based microcontrollers. The firmware implements common services specified in the Arm Platform Secure Architecture. These include services for cryptography, attestation, sealed storage and firmware updates. The crypto service is based on the PSA Crypto API implementation in MbedTLS. Normal applications can interact with the crypto service through the PSA Crypto API and use it to securely store and manage cryptographic keys. By isolating the crypto service from normal applications, secrets are protected from being extracted by an attacker.

Trusted Execution Environments on RISC-V secure embedded systems can be found in [23] and [21]. OpenMZ is a TEE solution for the SiFive FE310 microcontroller. It was developed by Henrik Karlsson as part of his master thesis. It allows the creation of multiple zones that run in user-mode and are isolated from each other. A small machine-mode firmware handles the scheduling of zones, interrupt delegation to zones, and IPC mechanisms for communication between zones. OpenMZ itself is based on the MultiZone API. MultiZone is a commercial and patented solution for Trusted Execution Environments targeting RISC-V processors. It was developed by HexFive Security and can support RISC-V secure embedded systems. Unlike Arm Trusted Firmware-M, both solutions only provide a framework for creating secure enclaves. They do not implement ready-to-use services. These must be implemented by the users themselves.

Previous master projects investigated the feasibility of this work. The first project [33] investigated the performance impact of running an RTOS like RIOT OS in RISC-V user-mode. It showed that while there was a significant impact on IPC mechanisms, it was generally feasible to run RIOT in a lower privilege mode. The second project [32] investigated code sharing and code size issues when applications are distributed across

multiple isolated environments. It showed that duplication of crypto libraries can be effectively eliminated using the PSA Crypto API.

Finally, in the master thesis in [8], Lena Boeckmann ported the approach developed in this work to the Arm Cortex-M platform. In her work, she integrated the crypto service API developed in this work with the PSA Crypto API implementation of RIOT OS. Additionally, she made improvements to the crypto service API implementation.

4 Problem Statement

The goal of this work is to explore the use of physical memory protection in RISC-V microcontrollers as a mechanism to protect cryptographic secrets against Heartbleed-like attacks [13] in constrained Internet of Things devices. A remote-attacker with access to the device via a network connection should no longer be able to abuse such vulnerabilities to extract long-term secrets from the device.

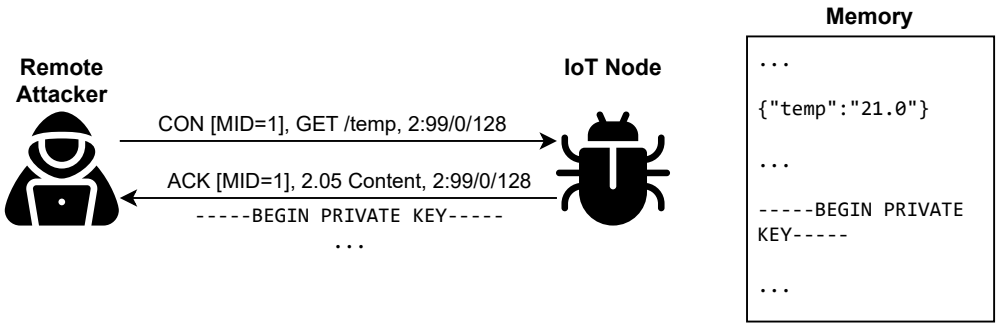

Figure 4.1: Example of an Attack Scenario in which a Remote Attacker Extracts a Private Key by Exploiting an Out-of-Bounds Read Vulnerability.

In this context, an example of an attack could look like the scenario shown in Figure 4.1. A remote attacker may be able to target security vulnerabilities in the protocol stack of the device. These vulnerabilities can be simple out-of-bounds read vulnerabilities [30] that are common in software written in the C programming language. They can result from improper input validation, such as was the case with the Heartbleed vulnerability in the OpenSSL library. In this scenario, the attacker can exploit these vulnerabilities by sending malformed packets to the device. As a result of the input validation failure, the device responds to the request with additional memory contents of the device. These additional memory contents may contain secret data, potentially giving the attacker access to sensitive information such as private keys.

As an alternative to the scenario in Figure 4.1, the attacker may also use remote code execution through buffer overflow vulnerabilities [29]. In this case, the attacker is able

to run arbitrary code on the device, which gives him the ability to extract the memory contents of the device over the network. In this way, the attacker can gain access to long-term secrets, such as the private keys used to identify the device and for remote attestation. These secrets allow the attacker to impersonate the device and gain further access to other parts of the IoT application.

The scope of this work will be limited in three ways. First, only remote attackers are considered as threat actors. While protection against physical attackers is also an important topic to investigate, it is not covered in this work. The reason for this is that it requires tamper-resistant hardware [26], which is not available at the time of writing. Second, it is assumed that the target platform is cost sensitive. Therefore, the use of secure elements or other types of trusted platform modules or hardware-protected key storage is not an option. Instead, all cryptographic secrets are stored in either ROM or RAM. Finally, it is assumed that current microcontrollers used in constrained IoT applications do not make use of memory isolation features such as RISC-V PMP for other security features like user-level threads. The solution developed in this thesis is later compared to a platform where all code has access to all physical memory and all peripherals.

5 Concept

To mitigate the scenario of a remote-attacker extracting long-term secrets from the device, this work proposes the solution of a crypto service running inside a secure firmware. The idea is to use the RISC-V physical memory protection mechanism to restrict the access of the real-time operating system and the application to cryptographic secrets. Private keys are only accessible from within the crypto service. The application can call the crypto service to perform operations with keys, such as signing a message. However, the application does not have access to the keys themselves. Therefore, a remote attacker cannot extract keys from memory.

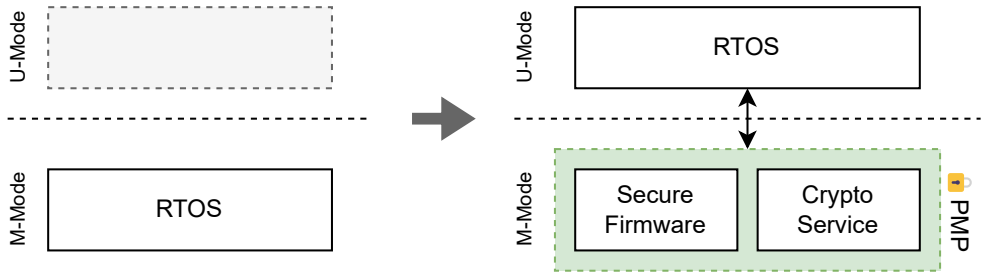

Figure 5.1: Transformation of a Machine-Mode RTOS to a User-Mode RTOS with Secure Firmware and Crypto Service

Figure 5.1 illustrates the change in execution model that must be implemented to achieve this. Previously, the operating system on RISC-V microcontrollers would run in machine-mode with full access to the hardware. In order to restrict memory access from the application, this work modifies the operating system to run in user-mode. With this, a minimal secure firmware which contains the crypto service can run in machine-mode. The application can then interact with the crypto service through system calls. However, the PMP feature prevents user-mode code from accessing the memory of the crypto service.

5.1 Requirements

In accordance with the previously stated goals and scope, the following basic requirements for the secure firmware and crypto service are defined:

1. **Suitable for constrained devices**
   The solution should run on constrained IoT nodes in IETF terminology [10]. This means it should fit into a memory envelope of 50 KiB of RAM and 250 KiB of ROM.

2. **Software-only implementation**
   The solution should not require any additional hardware besides a RISC-V CPU core with user-mode and PMP.

3. **Protect long-term cryptographic secrets**
   The solution should protect long-term cryptographic secrets from being read by the application from memory.

   a) **Protect the applications private keys**
   The solution should provide a way to generate, import, persist, and use private keys to the application. The application should not be able to read the value of these keys.

   b) **Protect hardware-unique private keys**
   The solution should provide a mechanism to use hardware-unique private keys for platform identification and remote attestation. The application should not be able to read these long-term keys.

   c) **Protect the state of the random number generator**
   To enable the generation of long-term secrets by the application, the solution should provide a mechanism to generate cryptographically secure random numbers. The application should not be able to read the state of the random number generator.

4. **Integration with RIOT OS**
   The solution should support the RIOT real-time operating system in such a way that the solution can be used on other platforms in the future as well.

5. **Ease of use**
   It should be easy for the application developer to integrate the crypto service and its features into existing applications.

5.2 Sealed Keys

To meet the third requirement, a concept must be developed for how keys are secured by the crypto service. If keys should only be accessible from within the crypto service and not from user-mode applications, then this creates limitations on how keys can be stored and persisted. The obvious way to solve this is to have protected keys never leave the protected memory space. In TF-M, keys can be stored in RAM by the MbedTLS PSA Crypto API [28] within the protected memory area. Keys that are persistent and should survive a reboot are additionally stored using the TF-M Internal Trusted Storage Service [6]. This ITS service has exclusive access to a tamper-resistant flash memory.

The problem with the Arm Trusted Firmware-M approach is twofold. First, it potentially wastes scarce memory on constrained IoT devices because the trusted firmware must allocate enough key slots of the right type and size. This can lead to developers over-allocating key slots, as it can be difficult to predict the maximum number of keys used by all applications and enclaves at the same time. Second, it can also lead to security problems, as insecure applications can use up all of the key slots in the firmware and thus perform a denial of service attack against secure enclaves. Third, the use of a dedicated tamper-resistant flash memory goes against our second requirement, that a software-only implementation should be achieved.

There exists alternative methods of storing keys similar to the concept of Hardware-wrapped keys implemented in Android [18] that can avoid both of these problems. This method can be conceptualized as sealing a key by encrypting it and then letting the user-mode applications handle storing and persisting the keys. Figure 5.2 illustrates this concept using the example of generating a private key and using it to sign a message.

When the user-mode asks the crypto service to generate a key pair, the crypto service will use its internal random number generator to generate a private key and derive its public key. Next, the private key is encrypted and authenticated using a key encryption key (KEK) that acts as a master secret for all sealed keys. The sealed key can then be returned to the user-mode application and stored in insecure memory. Because the

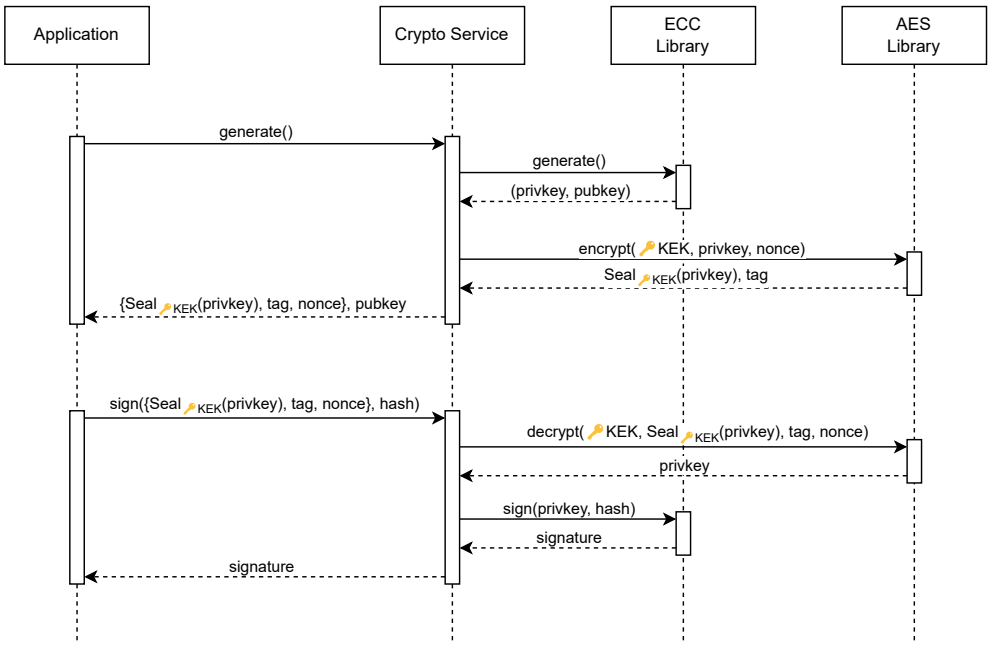

Figure 5.2: Example Sequence of Generating a Sealed Key and using it to Sign a Hash

private key can only be decrypted using the key encryption key, the sealed key alone is useless to potential attackers. If the user-mode application wants to use a sealed key for a crypto operation like signing a hash, it calls the firmware with the sealed key and the hash as arguments. The crypto service then uses the key encryption key to decrypt the key and verify its authenticity. If the authentication succedes, the crypto service performs the signing of the hash with the decrypted key and returns the signature.

5.3 Algorithms

It must be determined which cryptographic algorithms should be supported and used by the crypto service. This includes algorithms for symmetric encryption, hashing, and public key cryptography. There are several factors to consider. First and foremost, the security of the selected algorithms should meet minimum regulatory standards. This work will use BSI TR-02102 [11] as a baseline. Second, the algorithms must meet the first requirement of being suitable for use in constrained devices. Third, the algorithms must be supported by RIOT OS and its PSA Crypto API. Finally, they must be supported by protocols used in IoT applications such as DTLS [35] and OSCORE [45].

With these requirements, the choice is fairly straightforward. The BSI TR-02102 recommends algorithms with at least 120 bits of security. More precisely, the symmetric cipher AES-128, the hash function SHA-256 and ECDSA with at least 250 bits are recommended. For ECDSA, the smallest curve that meets this requirement is the NIST P-256 curve. This results in the following algorithms being chosen:

- AES-128
- SHA-256
- NIST P-256

All of these algorithms are supported by the PSA Crypto API in RIOT and the IoT protocols.

6 Implementation

The implementation can be thought of in two parts. In the first part, RIOT OS is modified to run in user-mode. This includes the implementation of a basic secure firmware that provides emulation of the privileged functionality necessary for the real-time operating system to function. The second part includes the implementation of the crypto service, which provides RIOT with system calls to use sealed keys. Finally, the crypto service is integrated with the PSA Crypto API in RIOT.

The prototype described here targets the SiFive FE310 microcontroller [48] on a SparkFun RED-V Thing Plus board. Details on other platforms may differ.

6.1 RISC-V Secure Firmware

Porting RIOT OS to run in user-mode has a few challenges that need to be addressed. First, user-mode software in the RISC-V architecture does not have access to privileged instructions like the CSR instructions required for interrupt configuration and processing, or the wait-for-interrupt instruction used for power saving during idle. Also, interrupts and system startup are always handled in machine-mode on the SiFive FE310 microcontroller.

To overcome this, a minimal machine-mode firmware must be developed that provides the following functionality:

1. Boot the microcontroller and transfer control flow to RIOT OS in user-mode
2. Delegate interrupts to RIOT OS in user-mode
3. Emulate access to privileged control and status registers
4. Emulate the wait-for-interrupt instruction

Each of these tasks is described in one of the following sections.

6.1.1 System Startup

The boot process on the SiFive FE310 microcontroller follows a multi-step approach. On the SparkFun RED-V Thing Plus board, this process can be summarized as follows: From an application developer's perspective, all software is programmed into an external QSPI flash module. This module is memory mapped into the physical address space starting at address `0x20000000`. When the microcontroller boots, it will ultimately jump to the fixed address `0x20010000` in the QSPI flash in machine-mode. This is where the secure firmware takes control of the microcontroller.

When the secure firmware starts, it runs assembly code that initializes RAM by loading the data section from ROM and clearing the BSS section. Then constructors are called and libc is initialized by the `__libc_init_array` function. After this step, the rest of the initialization can be done in C-code. This includes the initialization of all firmware components. Once initialization is done, the secure firmware configures PMP to protect all memory and peripherals that should only be accessible from machine-mode. A detailed description of the PMP configuration can be found in Section 7.2. Finally, the secure firmware performs a mode switch into user-mode and jumps to the start address of RIOT OS.

6.1.2 CSR Emulation

To overcome the limitation of privileged CSRs not being accessible from user-mode, some CSRs can be mapped into shared memory. They are written by the firmware at startup and updated before and after an interrupt request. This allows RIOT to access them with minimal performance overhead and without calling the firmware. A list of all memory mapped CSRs and their function is shown in Table 6.1.

6.1.3 Interrupt Enable

While some CSRs are read-only and can be easily emulated by writing their contents to shared memory, others require a more complex approach. The most important of these is interrupt enable and disable. Real-time operating systems such as RIOT make frequent

Table 6.1: List of memory mapped control and status registers

Memory location	**M-mode CSR equivalent**	**Description**
Interrupt processing		
`uapi_csr.uie`	`mie`	Interrupt enable mask register. Only used during interrupt processing. See section 6.1.4
`uapi_csr.utvec`	`mtvec`	Trap vector. Contains address of the trap handler
`uapi_csr.uepc`	`mepc`	Program counter of interrupted instruction. Also used to set the return address when returning from a trap
`uapi_csr.ucause`	`mcause`	Cause of the trap
CPU info		
`uapi_csr.uvendorid`	`mvendorid`	CPU vendor id
`uapi_csr.uarchid`	`marchid`	CPU architecture id
`uapi_csr.uimpid`	`mimpid`	CPU implementation id
`uapi_csr.uisa`	`misa`	CPU ISA support
`uapi_csr.uhartid`	`mhartid`	CPU thread id

use of interrupt disable to achieve mutual exclusion for inter-process communication and to mask interrupts in device drivers. RIOT uses the API shown in Listing 6.1 for global interrupt control.

Listing 6.1: Summary of the RIOT OS IRQ API

```
unsigned int irq_enable(void);
unsigned int irq_disable(void);
void irq_restore(unsigned int);
bool irq_is_in(void);
bool irq_is_enabled(void);
```

These functions are provided by the `riscv_common` CPU-target by reading, setting, or clearing the machine interrupt enable bit in the `mstatus`-CSR. In addition, peripheral drivers for GPIO, UART, and timers mask external and timer interrupts by setting and clearing the interrupt enable bits in the `mie`-CSR using the functions shown in Listing 6.2.

Listing 6.2: Example of RISC-V Encoding Macros used by RIOT OS for Interrupt Enable

```
write_csr(mie, MXIP);
```

```
set_csr(mie, MXIP);
clear_csr(mie, MXIP);
```

The ability to mask interrupts in the `mie`-CSR can be easily provided by the secure firmware using a single system call. The prototype of this system call is shown in Listing 6.3. It allows the user-mode application to perform an atomic read-modify-write of the `mie`-CSR by providing the value to be written as an argument and returns the value of the `mie`-CSR before writing.

Listing 6.3: Prototype of the System Call used for Interrupt Control

```
uint32_t csrrw_uie(uint32_t);
```

More complex operations such as setting or clearing bits can be accomplished by two consecutive calls, as shown in Listing 6.4.

Listing 6.4: Example of How the Interrupt Control System Call can be used to Set and Clear Bits in the mie-CSR

```
uint32_t tmp = csrrw_uie(0);
tmp |= MEIP;
tmp &= ~MTIP;
csrrw_uie(tmp);
```

The global enabling and disabling of interrupts can be achieved in a similar way by clearing all bits in the `mie`-CSR.

Listing 6.5: Implementation of the RIOT IRQ API using the Interrupt Control System Call

```
unsigned int irq_disable(void) { return csrrw_uie(0); }
void irq_restore(unsigned int state) { (void)csrrw_uie(state); }
bool irq_is_enabled(void) {uint32_t state = csrrw_uie(0); (void)csrrw_uie(state);
    return state; }
```

6.1.4 Interrupt Processing

Another challenge is interrupt handling. As explained in Section 2.1.4, interrupts are always handled in machine-mode on the SiFive FE310. This means that the secure firmware must delegate them to user-mode for processing by RIOT OS. The problem with

this is that there exists an errata in the SiFive FE310, where the machine-mode firmware cannot mode-switch to user-mode with interrupts globally disabled. If an interrupt is pending, the CPU will trap at the time of a mode switch even if interrupts are disabled. This means that when jumping into user-mode, interrupts must be disabled using the `mie`-CSR. This is achieved with the procedure shown in Figure 6.1.

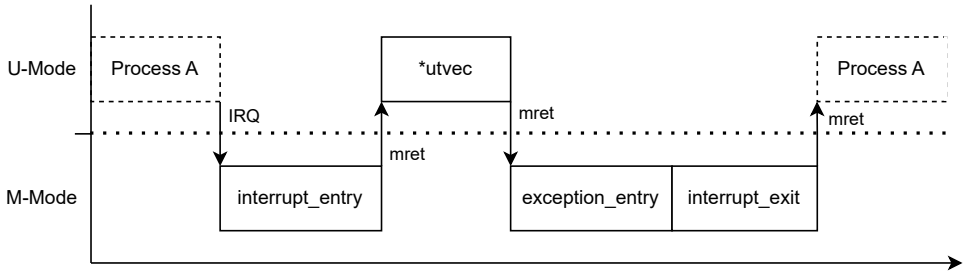

Figure 6.1: Process of Handling an Interrupt Request in User-Mode

At the first transition, an interrupt request is triggered and the RISC-V CPU traps by disabling interrupts and calling the exception handler of the secure firmware in machine-mode. This interrupt handler then delegates the interrupt request to RIOT into user-mode, which includes writing the values of the `mepc`, `mie`, and `mcause`-CSRs into shared memory. Next, the `mie`-CSR is cleared, effectively disabling interrupts on the FE310. And finally, an `MRET`-instruction is run that jumps to the address specified in the `uapi_csr.utvec` register in user-mode.

Then, after the second transition, the control flow has been successfully transferred to the RIOT OS interrupt handler. RIOT will then process the interrupt request almost as if it runs in machine-mode. However, two changes to the default behavior are required. First, writing to the `mie`-CSR must be done using the `uapi_csr.uie` register. Second, changes to `mepc` must be made in `uapi_csr.uepc`. Finally, the interrupt cause must be read from the `uapi_csr.ucause` register. Once the interrupt handler has completed in user-mode, it terminates using the privileged `MRET` trap-return instruction. This instruction will cause an illegal instruction exception to be triggered.

At the third transition, the exception handler of the secure firmware is called and detects that the user-mode interrupt handler has terminated. It then calls the interrupt exit function. This interrupt exit function simply restores the values of the `mepc` and `mie` CSRs from the emulated user-mode CSRs in shared memory and runs an mret instruction that jumps to the address previously specified in `uapi_csr.uepc` in user-mode. This

causes the previously interrupted process to resume at the exact instruction where it was previously interrupted.

6.1.5 Software Interrupts

Another component is the ability to trigger software interrupts from user-mode. This functionality is used by RIOT when performing a context switch. When a thread yields, it executes an `ECALL`-instruction that causes the interrupt handler to run. At the end of the interrupt handler, the scheduler is invoked and the CPU is given to the next process. The problem with the RIOT approach is that it assumes that all ECALLs are handled by the RIOT trap handler. To overcome this, a new system call is defined. This system call causes the secure firmware to invoke the user-mode interrupt handler. A new interrupt cause `CAUSE_USER_ECALL` is defined so that the real time operating system can detect that it has been called via the `ucause`-CSR.

6.1.6 WFI Emulation

Another instruction that needs to be dealt with is the wait-for-interrupt instruction. This instruction is used by RIOT in the idle thread to put the CPU to sleep and lower the power consumption. The RISC-V specification allows the `WFI`-instruction to be implemented as a simple no-operation [44]. The SiFive FE310 does have a full implementation of this instruction. To emulate the `WFI`-instruction from user-mode, trap-and-emulate is used. This way, no changes to the RIOT idle thread behavior are required.

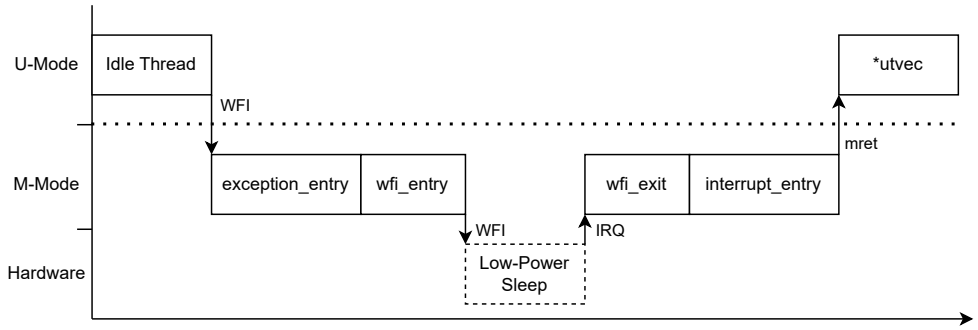

Figure 6.2: Process for Emulating the Wait-for-Interrupt Instruction

Figure 6.2 shows the execution flow of the `WFI` emulation. When RIOT executes a `WFI` -instruction, an illegal instruction exception is triggered and the secure firmware trap

handler is run and the exception is sent to the `WFI` entry handler. This entry handler temporarily modifies the address of the trap handler in the `mtvec`-CSR with the address of the `WFI` exit handler. Then interrupts are enabled and the `WFI`-instruction is executed from machine-mode, which puts the CPU into a sleep state. When the next interrupt is requested, the `WFI` exit handler is called. This exit handler restores the interrupt state and the `mtvec`-CSR. Then the interrupt request is dispatched to user-mode as described in section 6.1.4.

6.1.7 Build Process

The memory layout for the combined binary of secure firmware and application is designed so that the secure firmware is placed at lower addresses and RIOT is placed at higher addresses. Thus, the secure firmware can be built independently of the application and optionally distributed in binary format.

The build process of the secure firmware uses the XPack RISC-V GCC toolchain [1] for reproducible builds. The artifacts that are generated are as follows:

1. **The secure firmware binary in ELF format.**
   This can be used to flash the secure firmware into the lower address range of the microcontroller. The ELF file also includes debug symbols.
2. **The user-mode API headers.**
   These contain the emulated user-mode CSR struct specification as well as the ECALL numbers and macros to invoke the ECALLs.
3. **Linker script.**
   This linker script contains the address configuration for the user-mode RTOS. It includes ROM and RAM start and end addresses, as well as the address of the user-mode CSRs in shared memory.

These artifacts are then packaged and can be imported by the user-mode application.

6.1.8 RIOT Integration

The secure firmware repository will be made available in RIOT as a package called `riscv_secure_firmware`. This package can be enabled by the application developer

using the USEPKG make variable. When the riscv_secure_firmware package is included, a pseudo module riscv_umode is enabled, which is used to conditionally enable code paths inside the RIOT riscv_common CPU target that are necessary for it to be able to run in user-mode on top of the secure firmware. These conditional changes alter the behavior of some RIOT API functions.

Changes to the IRQ API are made to functions:

- irq_enable
- irq_disable
- irq_restore
- irq_is_enabled

They are changed to use the csrrw_uie system call instead of accessing the privileged mstatus-CSR. The RISC-V encodings macros for reading and modifying CSRs are overwritten to transparently translate to either emulated user-mode CSRs or system calls depending on the execution context:

- read_csr
- write_csr
- set_csr
- clear_csr

The thread_yield_higher function is modified to use the UCALL system call instead of a plain ECALL. And finally, the trap_handler function is modified to use the emulated user-mode CSRs instead of reading and writing the mcause and mepc-CSR directly. From a maintainability perspective, this approach is beneficial because it does not require a new CPU target, but only uses a total of 12 conditional changes in the existing riscv_common target, without the need to change microcontroller or board configurations. From an application developers perspective, using the secure firmware is as simple as adding USEPKG=riscv_secure_firmware to the make command, which is an optimal solution.

6.2 Crypto Service

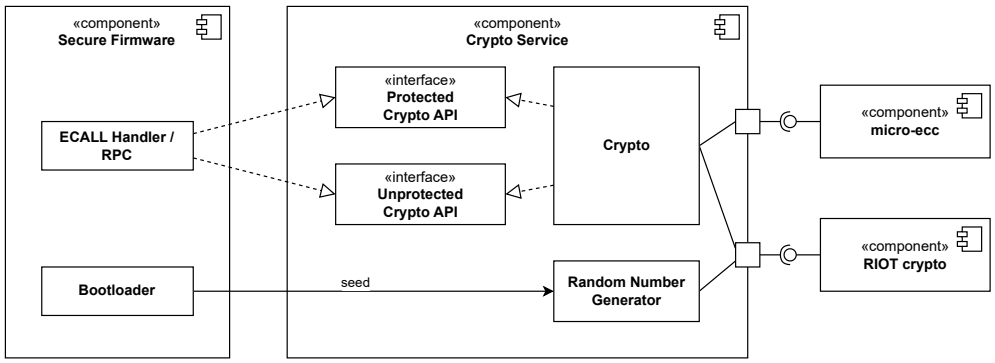

Figure 6.3: Component Diagram of the Secure Firmware and the Crypto Service

The crypto service is a software library that runs within the secure firmware. Figure 6.3 shows an overview of all the components that make up the crypto service. A protected key module provides the implementation of the cryptographic key API, which allows the application to perform operations using cryptographic keys. In addition, an unprotected module provides standard crypto operations using plain keys. This is an effort to save code space which is based on previous work. The library also contains a module for random number generation as well as the crypto libraries used to perform crypto operations. Besides this, an RPC-like mechanism is developed that allows a user-mode application to transparently call functions of the crypto service in machine-mode.

6.2.1 Crypto Service API

Listing 6.6: Definition of the Sealed Key Format

```
#define CYS_PROT_P256_KEY_SIZE (32)
#define CYS_PROT_SEAL_NONCE_SIZE (12)
#define CYS_PROT_SEAL_TAG_SIZE (16)

typedef struct {
    uint8_t nonce[CYS_PROT_SEAL_NONCE_SIZE];
    uint8_t private_key[CYS_PROT_P256_KEY_SIZE];
    uint8_t tag[CYS_PROT_SEAL_TAG_SIZE];
} CYS_PROT_p256_key_t;
```

Listing 6.6 shows the sealed key format used by the Crypto Service for P-256 private keys. This format combines a 12-byte nonce with the 256-bit private key and a 16-byte

authentication code. As a result, the protected private keys are 28 bytes larger than a plain, unencrypted P-256 private key. The nonce and tag sizes are chosen because AES-128 in OCB mode [25] was chosen as the encryption algorithm. A 12-byte nonce and 16-byte tag are the maximum parameters for this algorithm. The use of AES in this case is optimal in the sense that it works with a 16-byte block size. This means that a 256-bit private key can be encrypted in exactly 2 blocks. OCB mode allows for potentially fewer AES block calculations when moving to larger key sizes.

Listing 6.7: Crypto Service Protected Key API

```
CYS_error_t CYS_PROT_p256_generate(CYS_PROT_p256_key_t *sealed_key, uint8_t *public_key
    );

CYS_error_t CYS_PROT_p256_seal(const uint8_t *unsealed_key, CYS_PROT_p256_key_t *
    sealed_key);

CYS_error_t CYS_PROT_p256_derive(const CYS_PROT_p256_key_t *sealed_key, uint8_t *
    public_key);

CYS_error_t CYS_PROT_p256_sign(const CYS_PROT_p256_key_t *key, const uint8_t *hash,
    size_t hash_len, uint8_t *signature);
```

The P-256 sealed private key format is used in the Protected Key API shown in Listing 6.7. This API provides four essential functions that are necessary to use protected keys within a user-mode application. The first two, generate and seal, are used to create sealed keys. The generate function uses the internal random number generator of the crypto service to create a new key pair and returns the private key in sealed format to the caller and the public key in unencrypted SEC-1 [12] format. The seal function is a way for the application to convert an existing private key into a sealed key, effectively importing a private key into the crypto service. The derive function takes a sealed P-256 private key, derives the corresponding public key, and returns the derived P-256 public key in unencrypted format. The sign function provides the most important feature for using a P-256 private key without knowing its value. It takes a sealed P-256 private key and a hash value to sign. The crypto service then signs the hash using the sealed private key and returns the generated signature to the caller. Public keys do not need to be encrypted, so there is no need to provide a function to verify signatures using sealed keys.

Listing 6.8: Crypto Service Plain Key API

```
CYS_error_t CYS_random_generate(uint8_t *buffer, size_t size, size_t *len);
```

```
CYS_error_t CYS_p256_generate(uint8_t *private_key, uint8_t *public_key);
CYS_error_t CYS_p256_derive(const uint8_t *private_key, uint8_t *public_key);
CYS_error_t CYS_p256_sign(const uint8_t *private_key, const uint8_t *hash, size_t
    hash_len, uint8_t *signature);
CYS_error_t CYS_p256_verify(const uint8_t *public_key, const uint8_t *hash, size_t
    hash_len, const uint8_t *signature);

CYS_error_t CYS_sha256_init(CYS_sha256_ctx_t *ctx);
CYS_error_t CYS_sha256_update(CYS_sha256_ctx_t *ctx, const uint8_t *data, size_t len);
CYS_error_t CYS_sha256_finalize(CYS_sha256_ctx_t *ctx, uint8_t *digest);

CYS_error_t CYS_aes_128_cbc_encrypt(const uint8_t *key, const uint8_t *nonce, const
    uint8_t *message, size_t message_len, uint8_t *ciphertext);
CYS_error_t CYS_aes_128_cbc_decrypt(const uint8_t *key, const uint8_t *nonce, const
    uint8_t *ciphertext, size_t ciphertext_len, uint8_t *message);
```

Listing 6.8 shows the API for standard unencrypted keys. These functions are designed to be similar to the PSA Crypto Driver Interface for ease of integration. The API includes functions for random number generation, P-256 public key cryptography, SHA256 hashing, and AES-128 encryption in CBC mode.

6.2.2 Random Number Generation

A critical component of the crypto service is random number generation. Besides the obvious `random_generate` function, the crypto service needs secure random numbers to generate unpredictable keys. They are also needed to protect some of the crypto functions against timing-based side-channel attacks. The SiFive FE310 microcontroller does not have a hardware-based random number generator and does not have peripherals such as analog-to-digital converters that could be used as an entropy source. This is a known problem in constrained IoT devices that can be addressed using SRAM PUF [24] The idea of a PUF is to use the initial state of SRAM cells as an entropy source to seed a random number generator. Due to microchip manufacturing tolerances, some SRAM cells are initialized to a random state after a power-off reset. These cells can be used as an entropy source by hashing the uninitialized SRAM using a cryptographically secure hash function. The crypto service uses a modified version of the PUF SRAM implementation by the authors of [24]. Their implementation performs a SHA256 hash on the uninitialized SRAM before loading the `.data` and `.bss` sections. The hash is used as seed for a SHA256 based pseudo random number generator.

6.2.3 Crypto Libraries

To perform the operations specified in the API, the Crypto Service needs access to an implementation of the crypto algorithms. Since the SiFive FE310 has no hardware accelerators, only software libraries can be considered according to the second requirement. In accordance with the algorithms selected in Section 5.3 and as required by the API, the Crypto Service depends on libraries for the algorithms P-256, AES-128, OCB and CBC mode, SHA-256, and pseudo random number generator.

To make the benchmarks included in the evaluation comparable, it was decided to use the same libraries that are used by default with the PSA Crypto API in RIOT OS. These include the RIOT `sys/hashes`, `sys/crypto`, and `sys/random` subsystems, as well as the `micro-ecc` library [27]. Additional ROM and RAM savings could be achieved by using other libraries. However, code size comparisons and measurement of performance overhead would be impossible to do, as it would merely be benchmarking crypto libraries against each other.

6.2.4 Stubs and System Calls

Another design aspect of the crypto service is the invocation of functions from user-mode. In the RISC-V architecture, calling a machine-mode function from user-mode is done using environment calls. These ECALLs are a way for user-mode software to trigger an exception that traps the CPU into machine-mode and is then handled by the secure firmware. A convention commonly used with ECALLs is to place an ECALL id in registers `a7` and `a6`, while the arguments of the ECALL are placed in registers `a0 - a5` [43]. Thus, each API function is assigned an ECALL id according to Table 6.2.

While this answers how cross-level functions are invoked, an important question that still needs to be answered is how arguments are passed between user-mode and machine-mode. There are multiple mechanisms for passing parameters from one privilege mode to another. While the RISC-V application binary interface (ABI) makes it trivial to pass simple data types like integers, the crypto service API also uses pointers. Since some functions, such as SHA-256 hashing could take the entire user-mode memory as an argument, copying the input data into a shared memory area may require splitting operations into smaller chunks. To avoid such extra steps and reduce copying of data, which could negatively impact performance, we are instead going to use a model where

Table 6.2: System Call IDs for the Secure Firmware and Crypto Service

ECALL Name	**ID**
Secure Firmware API	
CSRRW_UIE	0x00
UCALL	0x01
Crypto Service Plain Key API	
CYS_RANDOM_GENERATE	0x10
CYS_P256_GENERATE	0x11
CYS_P256_DERIVE	0x12
CYS_P256_SIGN	0x13
CYS_P256_VERIFY	0x14
CYS_SHA256_INIT	0x15
CYS_SHA256_UPDATE	0x16
CYS_SHA256_FINALIZE	0x17
CYS_AES128_CBC_ENCRYPT	0x18
CYS_AES128_CBC_DECRYPT	0x19
Crypto Service Sealed Key API	
CYS_PROT_P256_SIGN	0x20
CYS_PROT_P256_GENERATE	0x21
CYS_PROT_P256_SEAL	0x22
CYS_PROT_P256_DERIVE	0x23

the crypto service reads and writes to user-mode memory directly, and checks pointer values before accessing that memory.

The resulting process used to call the crypto service from a user-mode application is shown in Figure 6.4. The user-mode application is compiled against the stubs of the crypto service API. These stubs simply place the ECALL id into register `a7`. The ECALL instruction is then executed. This triggers an exception and runs the trap handler of the secure firmware. The ECALL handler checks the ECALL number and sends the ECALL request to the appropriate crypto service skeleton. This skeleton performs a permission check on all arguments that are pointer values. If the user-mode application has read and write permissions to access the specified memory, the function call is forwarded to the crypto service. Once the crypto service function call has completed, the return value is placed into register `a0` of the user-mode application as specified by the RISC-V ABI. Control flow is then returned to the stub and the user-mode application.

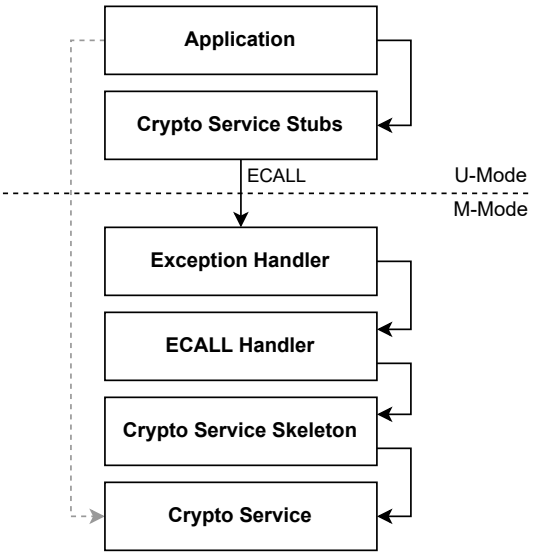

Figure 6.4: Call Hirarchy of a User-Mode Application Invoking the Crypto Service

6.2.5 PSA Crypto API Integration

While the user-mode application could use the crypto service API directly, drivers for the PSA Crypto API are provided to better match the fifth requirement. For an overview of the interaction between the PSA Crypto API and the crypto service, see Figure 6.5.

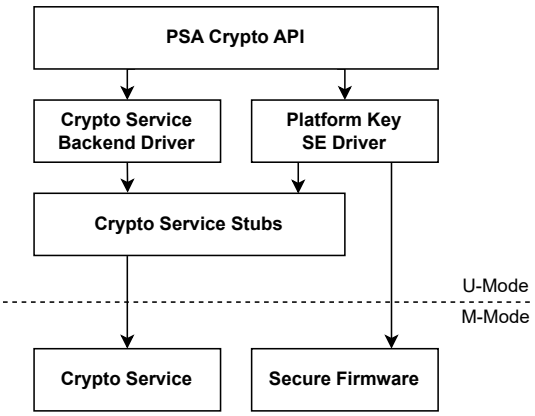

Figure 6.5: Overview of the Integration of the Crypto Service with the PSA Crypto API in RIOT

A PSA Crypto API backend driver, developed by Lena Boeckmann in [8], dispatches crypto operations to the crypto service. Keys with the `LOCAL_SEALED` location indicator

are dispatched to the sealed key API. Other keys are sent to the unencrypted API. Both sealed and unsealed keys are stored in key slots allocated by the PSA Crypto API. An example of how sealed keys are used with the PSA Crypto API is shown in Listing 6.9. The use of a sealed key is as simple as the specification of the key location at the time of key creation.

Listing 6.9: Example Program Generating a P-256 Sealed Key using the PSA Crypto API

```
psa_key_id_t key_id;
psa_key_attributes_t attr = psa_key_attributes_init();
psa_set_key_algorithm(&attr, PSA_ALG_ECDSA(PSA_ALG_SHA_256));
psa_set_key_usage_flags(&attr, PSA_KEY_USAGE_SIGN_HASH);
psa_set_key_type(&attr, PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&attr, 256);
psa_key_lifetime_t lifetime = PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
PSA_KEY_LIFETIME_VOLATILE, PSA_KEY_LOCATION_LOCAL_SEALED);
psa_set_key_lifetime(&attr, lifetime);
psa_generate_key(&attr, &key_id);
```

In addition to the backend driver, an SE driver is used to provide access to a platform-specific private key. The SE driver provides a key ID that can be used to sign messages. An example of how this key can be used is shown in Listing 6.10.

Listing 6.10: Minimal Program using the Platform Key to Sign a Hash

```
extern psa_key_id_t platform_key_id;
psa_sign_hash(platform_key_id, PSA_ALG_ECDSA(PSA_ALG_SHA_256), hash, sizeof(hash),
    signature, sizeof(signature), &signature_length);
```

6.2.6 RIOT Build System Integration

This brings the total number of packages that have been added to RIOT to two:

1. A `cryptoservice` package:
   This package provides:
   - The crypto service API header files
   - The PSA Crypto API backend driver for the crypto service.

2. A `riscv_secure_firmware` package:
   This package contains:
   - The code of the secure firmware
   - The code of the crypto service library
   - Scripts to build the secure firmware and the crypto service
   - The crypto service API stubs
   - The PSA Crypto SE driver for the platform key

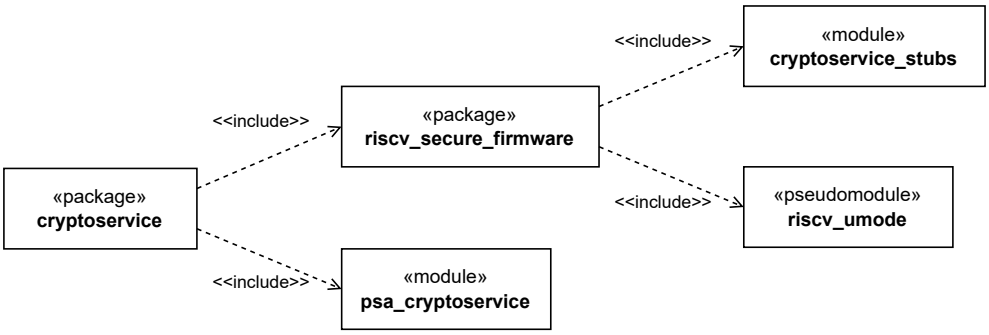

Figure 6.6: Overview of all the Modules and their Dependencies

An overview of all packages and modules and their dependencies is shown in Figure 6.6. From the application developer perspective, building an application with the crypto service and secure firmware enabled is as simple as including the crypto service module. All dependent modules are automatically selected. Listing 6.11 shows how to build an application with the crypto service enabled.

Listing 6.11: Minimal Command of Building a RIOT Application with the Crypto Service and Secure Firmware Enabled

```
USEMODULE=cryptoservice make all
```

6.2.7 Key Provisioning

When the secure firmware is flashed onto the microcontroller, the provisioning of the key encryption key and the platform unique key must be considered. Since there is no way for the FE310 microcontroller to generate and store the key encryption key or the

platform unique key, both must be generated on the host computer and flashed to the device with the secure firmware. This is accomplished with the help of a Python script that is run prior to flashing the firmware.

The script takes a PEM-encoded P-256 private key as an optional argument. If no key file is provided, a new platform unique key is generated. The script also generates the 128-bit encryption key. Both keys are then written to the ELF file. This is possible because at the time the secure firmware is built, it already contains placeholder keys that are allocated and initialized in the binary. By looking up the symbol addresses in the symbol table of the ELF file, the script can determine the file offset of both the platform key and the key encryption key. It then overwrites the values of these keys in a temporary file. The temporary file is then flashed to the microcontroller and immediately deleted afterwards to prevent the keys from leaking. The secure firmware can be flashed with a custom key using the command shown in Listing 6.12.

Listing 6.12: Example of a Flash Command Specifying a Platform Key

```
KEYFILE=./key.pem USEMODULE=cryptoservice make flash
```

7 Evaluation

7.1 Functional Testing

The first step of the evaluation is to perform functional testing of the secure firmware, crypto service and modifications to RIOT OS. This is done using the RIOT OS test suite. Some of the tests verify that the test case is built without errors, while others also execute the test case on the target microcontroller and verify that the test has passed successfully. The RIOT test suite includes several functional tests that check the correct operation of core functions such as the IRQ and threading APIs that were modified as part of this work. It also includes test cases for the PSA Crypto API. These tests have been modified to include keys stored by the crypto service. No regressions were found.

7.2 Memory Isolation

Next, the memory isolation using the RISC-V physical memory protection feature were tested. As a first step, it must be determined which parts of the memory a user-mode application is allowed to access and which parts should be restricted.

Table 7.1 shows an overview of the memory map taken from the SiFive FE310 manual. There are some memory areas such as the upper part of RAM and ROM as well as peripherals that should be accessible from user-mode. The lower parts of RAM and ROM should only be accessible from machine-mode as those contain the firmware code and sensitive data like the key encryption key and platform key. There are also some peripherals that could be used to leak information from secure memory or capture control flow. These include the debug memory, mode select, and OTP control peripherals. Because they are not strictly required by the RIOT operating system, they are made to be inaccessible from user-mode.

Table 7.1: SiFive FE310 Memory Regions and PMP Configuration

Address Range	**Type**	**Description**	**Permissions**		
			R	W	X
0x00000000 – 0x00001000	I/O	Debug	-	-	-
0x00001000 – 0x00002000	I/O	Mode Select	-	-	-
0x00003000 – 0x00004000	I/O	Error Device	-	-	-
0x00010000 – 0x00012000	Mem	Mask ROM	-	-	-
0x00020000 – 0x00022000	Mem	OTP ROM	-	-	-
0x02000000 – 0x02010000	I/O	CLINT	1	1	-
0x08000000 – 0x08002000	Mem	ITIM	-	-	-
0x0C000000 – 0x10000000	I/O	PLIC	1	1	-
0x10000000 – 0x10001000	I/O	AON	1	1	-
0x10008000 – 0x10009000	I/O	PRCI	1	1	-
0x10010000 – 0x10011000	I/O	OTP Control	-	-	-
0x10012000 – 0x10013000	I/O	GPIO	1	1	-
0x10013000 – 0x10014000	I/O	UART 0	1	1	-
0x10014000 – 0x10015000	I/O	QSPI 0	1	1	-
0x10015000 – 0x10016000	I/O	PWM 0	1	1	-
0x10016000 – 0x10017000	I/O	I2C 0	1	1	-
0x10023000 – 0x10024000	I/O	UART 1	1	1	-
0x10024000 – 0x10025000	I/O	SPI 1	1	1	-
0x10025000 – 0x10026000	I/O	PWM 1	1	1	-
0x10034000 – 0x10035000	I/O	SPI 2	1	1	-
0x10035000 – 0x10036000	I/O	PWM 2	1	1	-
0x20000000 – `__flash_secure_end`	Mem	ROM (M-mode)	-	-	-
`__flash_normal_start` – 0x20400000	Mem	ROM (U-mode)	1	-	1
0x80000000 – `__ram_secure_end`	Mem	RAM (M-mode)	-	-	-
`__ram_normal_start` – 0x80004000	Mem	RAM (U-mode)	1	1	-

The difficulty in configuring PMP is that the FE310 microcontroller has only 8 regions available. This presents the challenge of how to most effectively use the regions to block all security-critical memory areas [19]. Fortunately, most of the 4 GiB address space is empty, and the peripherals are grouped in a way that allows for efficient overlapping of regions.

A test program has been written to verify that the selected PMP configuration meets the access constraints shown in Table 1. This program iterates over each word, half-word, and byte in each memory area and performs a read. If the area does not have read privileges, the test succeeds if a memory access failure exception is generated. Otherwise, the test passes if the load instruction is executed without generating an exception. For memory areas with write permissions, the test application also checks whether exceptions are generated on store instructions. The test application confirms that the PMP configuration meets the constraints set by the analysis in Table 7.1.

7.3 Benchmarks

With functional testing complete, the next step is to consider the performance impact of the secure firmware and crypto service. This is done from several perspectives. First, the performance impact of the secure firmware on RIOT caused by changing the IRQ and the threading API is measured. This is done using synthetic microbenchmarks and a real-world network benchmark. The performance impact of the crypto service must also be considered. The performance overhead of using the crypto service is also measured. All benchmarks compare the performance of a default build of RIOT OS to a build that has both the secure firmware and the crypto service enabled.

7.3.1 Validation Benchmarks

Before running meaningful benchmarks, the experiment setup must be validated. This is to ensure that the numbers generated by the benchmarks are comparable between the default version of RIOT running in machine-mode and the modified version of RIOT running in user-mode. The CoreMark [14] benchmark was used to validate that the privilege mode does not affect the raw instruction throughput of the CPU.

The results of the CoreMark benchmark are shown in Figure 7.1. The privilege mode, in which the CoreMark benchmark is run, has no effect on the benchmark score. Both runs show the same score of 101 iterations/s. This confirms that the user-mode context and the enforcing of PMP regions have no effect on integer, load, and store instructions.

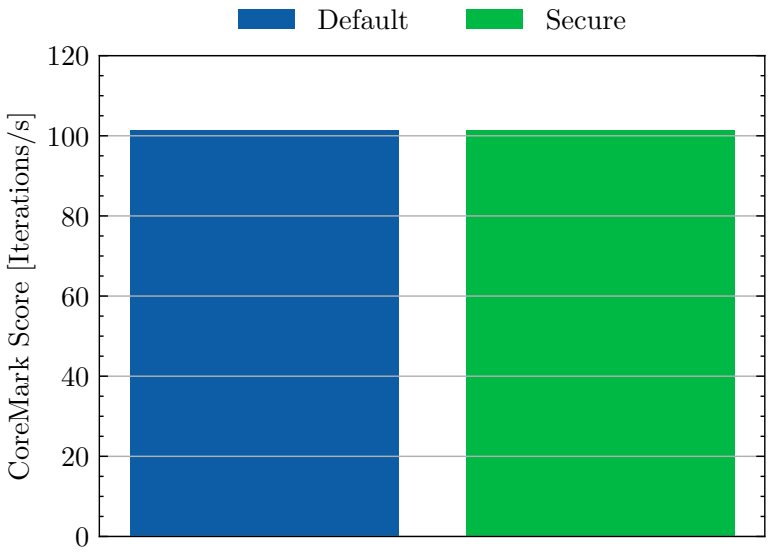

Figure 7.1: Comparison of CoreMark Benchmark Score

Next, the accuracy of the timer is checked. Some of the benchmarks included in RIOT use the `ztimer` subsystem to stop the benchmark after a fixed time. Because of the expected overhead from the secure firmware dispatching interrupts into user-mode, it must be validated that this overhead does not meaningfully change the benchmark scores. This is done using the `ztimer` accuracy test included with RIOT. The test puts a thread to sleep for a wide range of time intervals and compares how time has passed sleeping to the sleep time it requested.

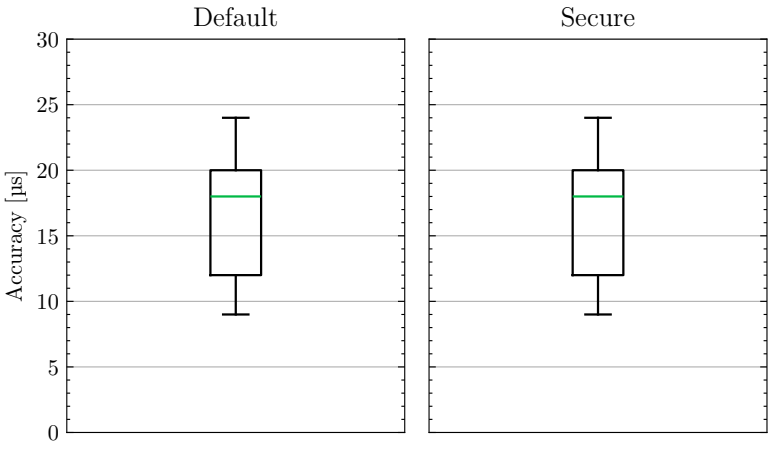

Figure 7.2: Comparison of Timer Accuracy Benchmark Results

Figure 7.2 shows the results of the accuracy test. The timer accuracy is the same. The median, which is indicated by the green bar of the box plot, is identical at 18 µs. The mean decreased by an insignificant 30 ns. The accuracy across all tests appears to be independent of the total sleep time. Since many benchmarks run for more than 1 s to get a significant result, this decrease is not expected to affect the upcoming results.

7.3.2 IPC-Microbenchmarks

When RIOT runs in user-mode with the secure firmware, the behavior of the IRQ API is changed. This is expected to affect not only the IRQ API itself, but also its consumers such as the IPC mechanisms. RIOT already provides microbenchmarks for the IRQ API and IPC mechanisms.

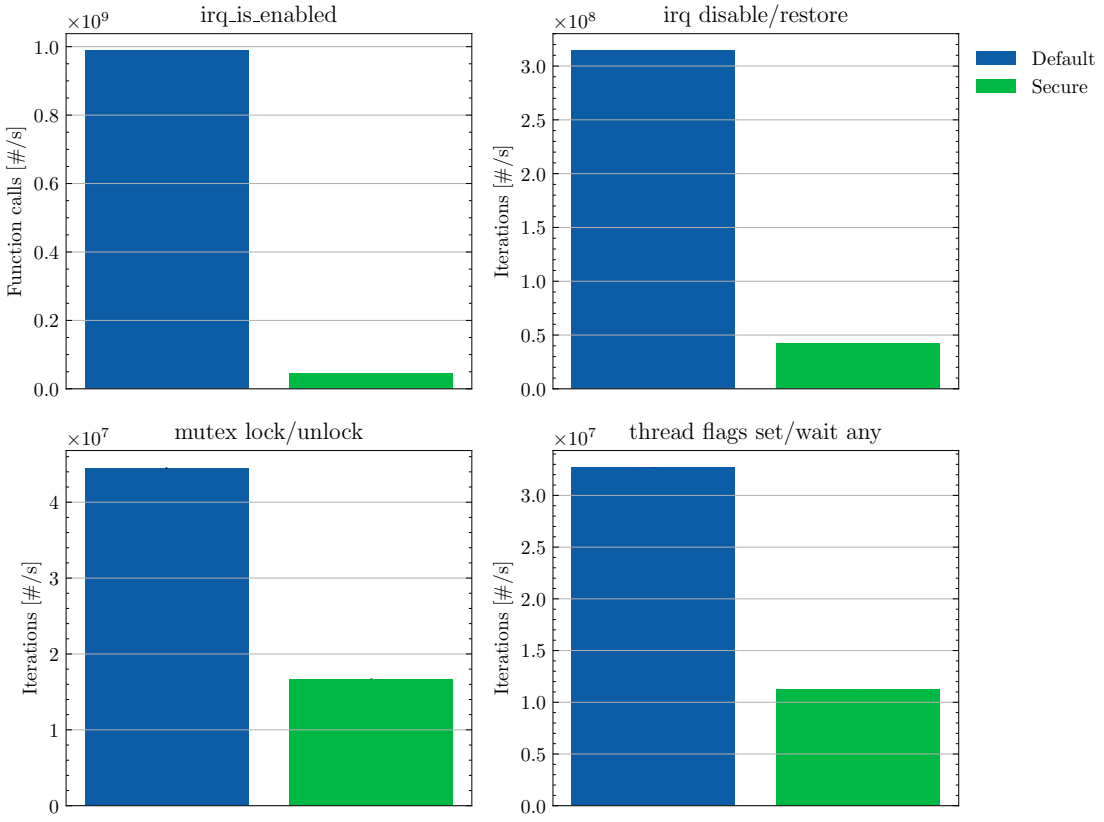

Figure 7.3: Comparison of IPC-Microbenchmark Performance

Figure 7.3 shows the results of four microbenchmarks. The benchmark on the top left shows that the `irq_is_enabled()` function has slowed down by more than an order of

magnitude. The benchmark on the top right shows that a pair of `irq_disable()` and `irq_restore()` function calls slowed down by almost an order of magnitude. Both results are expected because these functions are implemented as a single CSR instruction in default RIOT. With user-mode enabled, they consist of a system call that requires mode switching and dispatching by the secure firmware.

The benchmark on the bottom left measures how fast the platform can lock and unlock a mutex, while the one on the bottom right measures how fast thread flags are processed. These are more realistic benchmarks than the previous ones because application code rarely interacts directly with the IRQ API, instead using the IPC mechanisms provided by RIOT. The mutex IPC mechanism sees a performance loss of 62.5% less locks per second when running in user-mode. The thread flags mechanism suffers a 65.4% decrease in flags sent per second compared to RIOT running in machine-mode.

7.3.3 Context Switching

Since this work also made changes to the `thread_yield_higher` function, context switching performance is expected to be lower. To test this assumption, the RIOT `thread_yield_pingpong` benchmark was run. This benchmark creates two threads that yield in a loop for 1 s. The benchmark measures the number of context switches performed during this interval.

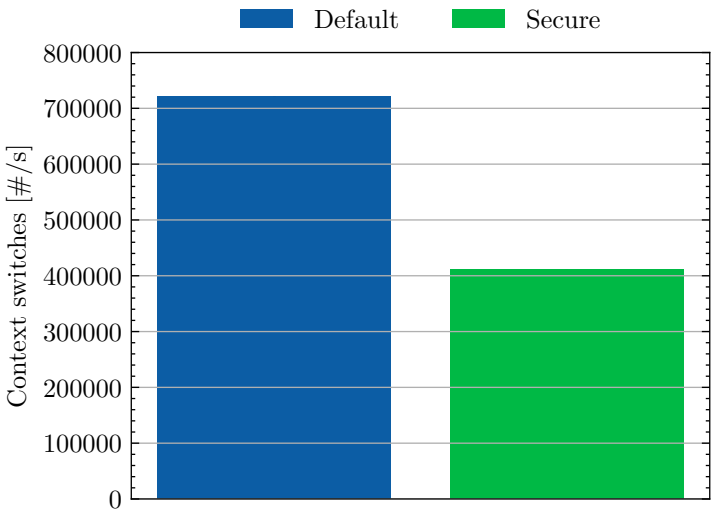

Figure 7.4: Comparison of Context Switch Performance

Figure 7.4 shows the benchmark results. The number of context switches that can be performed per second has decreased by almost 43.0%. This means that context switches take about twice as long with the secure firmware enabled. One reason for this is simply the additional overhead when calling the secure firmware to issue a software interrupt and dispatching it to user-mode instead of just issuing an `ECALL` instruction.

7.3.4 Networking Benchmark

The benchmarks presented so far have been synthetic microbenchmarks designed to expose the weaknesses of the implementation. A normal application is usually a mix of raw compute, which saw no performance impact, and IPC, which saw significant performance loss. To better evaluate the performance impact on a real-world application, a network latency benchmark was performed. For this benchmark, a W5500 Ethernet controller [50] was connected to the SiFive FE310 microcontroller via SPI. Due to RAM limitations, only a minimal RIOT application with IPv6 and ICMP enabled could be loaded onto the microcontroller. The Ethernet was then connected to a host computer and the round-trip time between the host and the microcontroller was measured using ping.

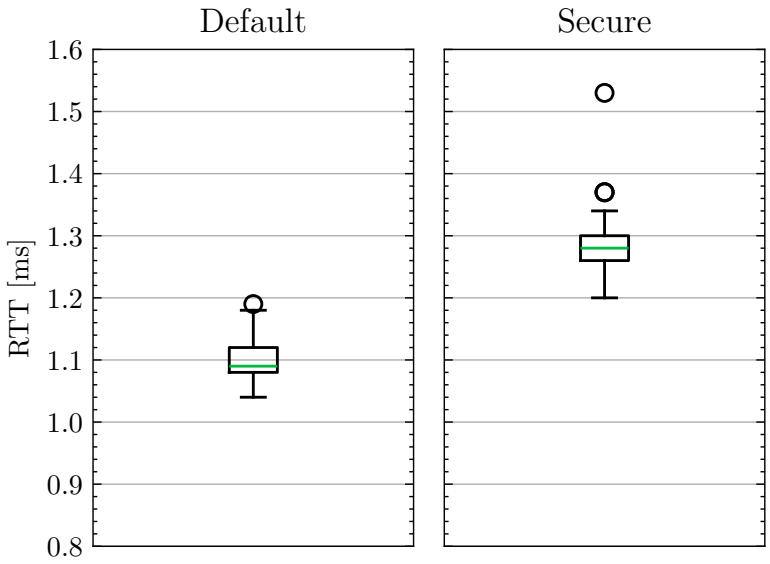

Figure 7.5: Comparison of Average Round-Trip Time over Ethernet

As shown in Figure 7.5, the median and mean round-trip time between the microcontroller and the host computer increased by 16.9% from 1.10 ms to 1.28 ms. Jitter in-

creased from 29 µs to 41 µs. This shows that there is considerable impact on real-world performance, but it is not as extreme as the microbenchmarks would suggest.

7.3.5 Crypto Benchmarks

Finally, the performance of the PSA Crypto API is compared. Various cryptographic operations such as random number generation, hashing, symmetric ciphers, and digital signatures were run and the average runtime was measured. Some overhead is to be expected when running these functions, since system calls have to be processed in addition to the crypto algorithms themselves.

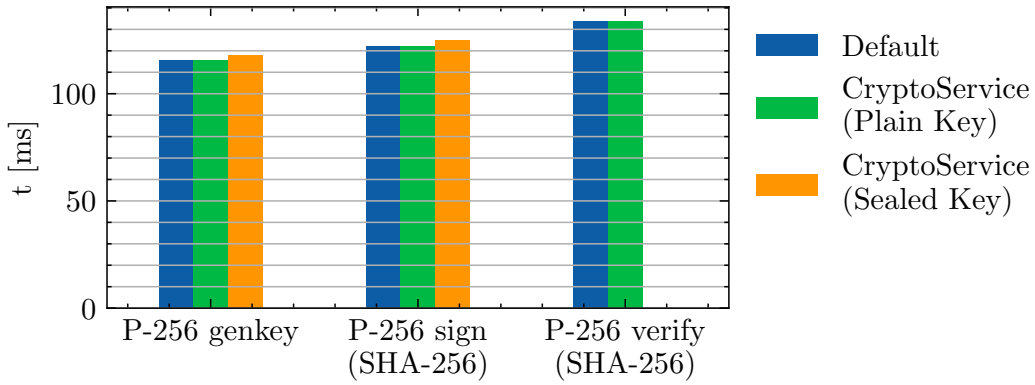

Figure 7.6: Comparison of Time to Generate Key Pairs, Sign Hashes, and Verifiy Signatures using P-256

Figure 7.6 shows the average runtime for the P-256 operations for generating key pairs, signing hashes, and verifying signatures. The overhead of generating a P-256 key pair using the `psa_generate_key` function is 1.92% for sealed keys and only 0.15% for plain keys when comparing to a default configuration of RIOT. The overhead for the plain key comes from system calls only. The overhead for sealed keys also includes decryption and authentication of the sealed key. Signing a SHA-256 hash using the `psa_sign_hash` function shows an overhead of 2.48% for sealed keys and 0.07% for plain keys. The runtime for verifying signatures using plain keys has increased by 0.02% on average compared to RIOT without the crypto service.

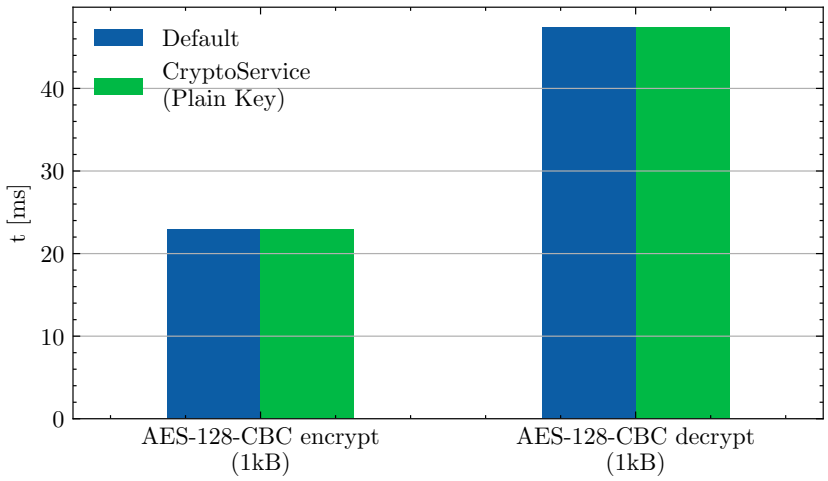

Figure 7.7: Comparison of Time to Encrypt and Decrypt Data using AES-128-CBC

Figure 7.7 shows the performance for encrypting and decrypting 1 KB of data using the `psa_cipher_cbc_aes_128_encrypt` and `psa_cipher_cbc_aes_128_decrypt` functions. The results show a negligible increase in average runtime of 0.03% for encryption and 0.01% for decryption.

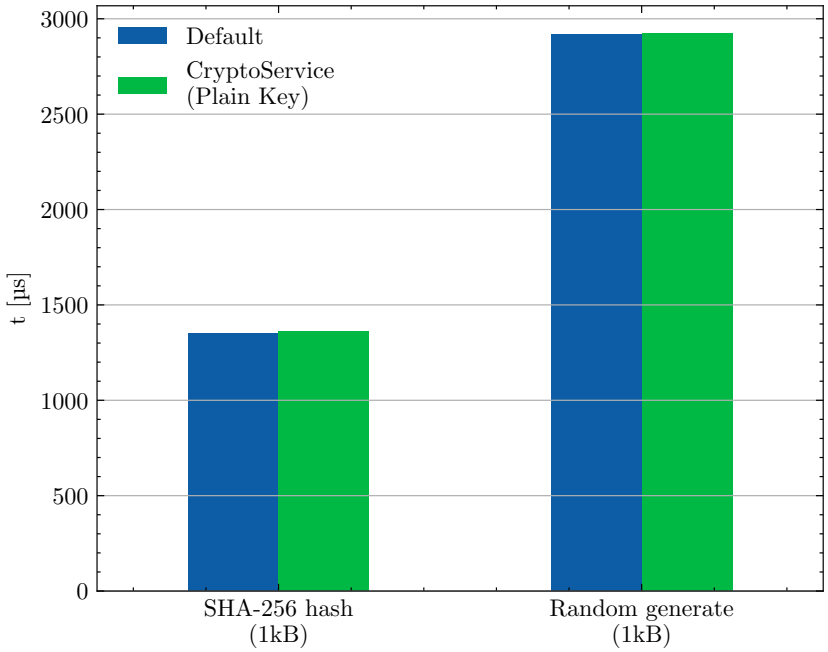

Figure 7.8: Comparison of Time to Hash a Message and Generate Random Bits

Figure 7.8 shows the average runtime for hashing 1 kB of data using the SHA-256 hash function and generating 1 kB of pseudo-random data. As can be seen from the results, the average runtime of the `psa_hash_compute` functions has increased by 0.74%. This is the largest relative increase observed for a function of the plain key API. However, the runtime of this benchmark is also very short. It is an increase of only 10 µs, which is due to the fact that computing a hash requires at least 3 system calls for `sha256_init`, one or more `sha256_update`, and one `sha256_finish`. The average runtime of `psa_generate_random` has increased by 0.10%.

7.3.6 Memory Usage

In addition to performance, the memory footprint of the binary must be analyzed. This analysis needs to be done from two perspectives. The first is the memory footprint of the secure firmware. The second is a RIOT application compiled with the crypto service and secure firmware enabled.

Table 7.2: ELF Section Sizes of the Secure Firmware and Crypto Service

Section	**Size [Bytes]**
ROM	20882
.init	118
.text	17696
.rodata	3012
secrets	48
.data	8
RAM	1744
.preserve	40
.data	8
.bss	296
.stack	1400

Table 7.2 shows the total memory usage of the secure firmware and how much each individual section uses. In total, the firmware uses 20.9 kB of ROM and 1744 bytes of RAM. Most of the RAM is used by the stack, which must be sized to accommodate SHA256 hash and P-256 elliptic curve computations. To analyze the size of a RIOT application, the PSA example application was compiled once with the default RIOT configuration and once with the crypto service enabled.

Table 7.3: Comparison of ELF Section Sizes of the RIOT PSA Example Application

Section	**Size [Bytes]**	
	Default	Secure
ROM	31156	15410
.init	116	112
.text	26352	13486
.rodata	4592	1728
.data	96	84
RAM	3648	3468
.data	96	84
.bss	3296	3128
.stack	256	256

Table 7.3 shows the size of the ROM and RAM sections. The ROM size has significantly reduced from 31.2 kB to 15.4 kB. This reduction is due to the crypto libraries no longer being included in RIOT OS because RIOT uses the crypto libraries included in the secure firmware. RAM usage decreased slightly by 180 bytes from 3648 bytes to 3468 bytes. An additional 64 bytes of RAM are reserved for shared memory between RIOT and the secure firmware, reducing the total savings to 116 bytes.

However, further saving could be achieved by adjusting the stack size. Since all crypto operations are performed by the crypto service, the secure firmware stack is used for these calculations. Thereby, the call stack of RIOT applications could be reduced. When stack usage was measured during the PSA example application of RIOT, the call stack usage was significantly reduced compared to default RIOT.

Table 7.4: Comparison of Maximum Stack Usage of the RIOT PSA Example Application

	Maximum Stack Usage [Bytes]
Default	1888 Bytes
Secure	664 Bytes

Table 7.4 shows that there is a significant reduciton in stack usage of 1224 bytes when using the crypto service. In some scenarios, this saving could be used to reduce the stack size of the main thread in RIOT, making the `.bss` section even smaller. Note that these savings are on a per-thread basis. That is, each thread using the PSA Crypto API could save the amount shown in Table 7.4. It has the side effect that only one application can use the crypto service at a time.

8 Discussion

The results show that the approach of running a real-time operating system on RISC-V microcontrollers in user-mode and securing cryptographic operations using machine-mode firmware is working quite well. The initial requirements were met. The RAM and ROM consumption of the solution make it feasible to run on constrained IoT nodes. In addition, integrating the crypto service with the PSA crypto API results in significant stack and code size savings by eliminating the duplication of all crypto libraries and the stack they use. However, the solution shows some shortcomings with respect to performance. The user-mode CSR emulation and interrupt dispatching has caused significant slowdowns of critical code paths of RIOT OS. These slowdowns have been shown to result in double-digit performance losses in real-world applications.

In terms of security, the integration of sealed keys secured by the crypto service and integrated with the PSA Crypto API in RIOT can be considered a success. Encryption of private keys provides an additional layer of security that mitigates the security threat outlined in the problem statements of this work. A remote attacker can no longer use out-of-bounds read vulnerabilities in RIOT to extract private keys from memory. Since private keys are encrypted, they are useless without the key encryption key, which is only accessible to the crypto service. Even though this does not provide strong guarantees of private key security, it is arguably more secure than was previously the case. This is while the performance overhead of encrypting and decrypting a private key is minimal compared to the time it takes to sign or generate an elliptic curve key pair.

However, there are areas where the security of the secure firmware and crypto service need more scrutiny. The functional correctness of the machine-mode software should ideally be formally verified to ensure that the encryption key cannot be extracted. The implementation also needs to be hardened against side-channel attacks such as timing-based attacks. There is, however, no effective way to protect against weaknesses in the hardware itself.

This work revealed several weaknesses in the hardware used. First, the inability for a user-mode RTOS to efficiently disable and enable interrupts has caused significant performance loss. This use case would benefit significantly from the implementation of user-mode interrupts, which other microcontrollers such as the Espressif ESP32-H2 [17] do provide. However, this so-called N-extension [2] was removed from the RISC-V specification in 2021. It was replaced by a bare supervisor mode, which as of the time of writing has not yet made it into commercially available RISC-V microcontrollers.

There is also a lack of secure hardware-based random number generation in the microcontroller used. A RISC-V extension for hardware-based secure random number generation has been ratified [40]. However, any other form of cryptographically secure entropy source accessible as a peripheral would also be acceptable.

Finally, the impact of the crypto service on real-time applications remains unclear. Currently, no interrupts are dispatched to the user-mode RTOS when the crypto service is used. This can result in interrupt processing delays of up to 120 ms. Further development is needed to make crypto operations interruptible to reduce the impact on the real-time capabilities of the platform.

9 Conclusion and Outlook

This work identified Heartbleed-like attacks on constrained IoT-nodes as a potential security threat in Internet of Things applications. Remote attackers could abuse vulnerabilities in low-cost embedded devices without memory isolation to extract long-term secrets like private keys. To address this threat on RISC-V microcontrollers, this work investigated the use of the RISC-V physical memory protection feature to run a secure firmware and a crypto service inside a secured execution environment. The IoT application and real-time operating system run in a separate execution environment without access to the crypto service resources. The crypto service performs all crypto operations and provides a mechanism for sealed keys – encrypted keys that can be stored in insecure memory but can only be read by the crypto service. The RIOT operating system was modified to run in user-mode and was integrated with the secure firmware and crypto service.

The prototype developed in this work showed that this approach is feasible and provides an additional layer of security. Further work is needed on the security of the approach, however. Future research should investigate the side-channel resistance of the solution developed in this work to ensure keys material is not leaked by the time it takes to complete crypto operations. Additionally, future work should improve on the approach in two ways. First, the real-time capability of the firmware should be improved. This is to ensure time intensive crypto operations can be interrupted. And second, more algorithms and key types should be supported. The end goal should be to full support Internet of Things protocol suites like DTLS, secured by the crypto service.

Bibliography

[1] *The xPack GNU RISC-V Embedded GCC*. – URL https://xpack.github.io/dev-tools/riscv-none-elf-gcc/. – Accessed 2026-07-01

[2] Andrew Waterman: *Proposed deprecation of N extension*. May 2021. – URL https://lists.riscv.org/g/tech-privileged/topic/proposed_deprecation_of_n/83320504. – Accessed 2026-07-01

[3] Arm Limited: *MbedTLS*. – URL https://github.com/Mbed-TLS/mbedtls. – Accessed 2026-07-01

[4] Arm Limited: *PSA Crypto API 1.1*. February 2022. – URL https://developer.arm.com/documentation/ihi0086/latest/. – Accessed 2026-07-01

[5] ARM Limited, Antonio de A.: *Crypto Service design*. – URL https://tf-m.docs.trustedfirmware.org/en/latest/design_docs/services/tfm_crypto_design.html. – Accessed 2026-07-01

[6] Arm Limited, Jamie F.: *Internal Trusted Storage (ITS) Service*. – URL https://tf-m.docs.trustedfirmware.org/en/latest/design_docs/services/tfm_its_service.html. – Accessed 2026-07-01

[7] Baccelli, Emmanuel ; Gündoğan, Cenk ; Hahm, Oliver ; Kietzmann, Peter ; Lenders, Martine S. ; Petersen, Hauke ; Schleiser, Kaspar ; Schmidt, Thomas C. ; Wählisch, Matthias: RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. In: *IEEE Internet of Things Journal* 5 (2018), Nr. 6, pp. 4428–4440. – URL https://doi.org/10.1109/JIOT.2018.2815038

[8] Boeckmann, Lena: *Integration and Evaluation of a Secure Firmware for ArmCortex-M Devices in RIOT OS*, Master's thesis, March 2025. – URL https://www.inet.haw-hamburg.de/thesis/completed/ma_lena_boeckmann.pdf/@@download/file/MA_Lena_Boeckmann.pdf. – Accessed 2026-07-01

[9] Boeckmann, Lena ; Kietzmann, Peter ; Lanzieri, Leandro ; Schmidt, Thomas C. ; Wählisch, Matthias: Usable Security for an IoT OS: Integrating the Zoo of Embedded Crypto Components Below a Common API. In: *Proc. of Embedded Wireless Systems and Networks (EWSN'22)*. New York, USA : ACM, October 2022, pp. 84–95. – URL https://dl.acm.org/doi/10.5555/3578948.3578956

[10] Bormann, Carsten ; Ersue, Mehmet ; Keränen, Ari: *Terminology for Constrained-Node Networks*. RFC 7228. May 2014 (Request for Comments). – URL https://www.rfc-editor.org/info/rfc7228. – Accessed 2026-07-01

[11] BSI: *BSI TR-02102-1, Cryptographic Mechanisms: Recommendations and Key Lengths*. January 2024. – URL https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=7. – Accessed 2026-07-01

[12] Certicom Research: *SEC 1: Elliptic Curve Cryptography, Version 2.0*. May 2009. – URL https://www.secg.org/sec1-v2.pdf. – Accessed 2026-07-01

[13] Durumeric, Zakir ; Li, Frank ; Kasten, James ; Amann, Johanna ; Beekman, Jethro ; Payer, Mathias ; Weaver, Nicolas ; Adrian, David ; Paxson, Vern ; Bailey, Michael ; Halderman, J. A.: The Matter of Heartbleed. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. New York, NY, USA : Association for Computing Machinery, 2014 (IMC '14), pp. 475–488. – URL https://doi.org/10.1145/2663716.2663755. – ISBN 9781450332132

[14] EEMBC: *CoreMark*. – URL https://www.eembc.org/coremark/. – Accessed 2026-07-01

[15] Espressif Systems: *ESP32-C3 Series Datasheet v1.5*. 2023. – URL https://documentation.espressif.com/esp32-c3_datasheet_en.pdf. – Accessed 2026-07-01

[16] Espressif Systems: *ESP32-C3 Technical Reference Manual, Version 1.1*. January 2024. – URL https://documentation.espressif.com/esp32-c3_technical_reference_manual_en.pdf. – Accessed 2026-07-01

[17] Espressif Systems: *ESP32-H2 Series Datasheet, Version 1.0*. September 2024. – URL https://documentation.espressif.com/esp32-h2_datasheet_en.pdf. – Accessed 2026-07-01

[18] Google LLC: *Android Open Source Project Documentation - Hardware-wrapped keys*. – URL https://source.android.com/docs/security/features/encryption/hw-wrapped-keys. – Accessed 2026-07-01

[19] Hardin, Taylor ; Scott, Ryan ; Proctor, Patrick ; Hester, Josiah ; Sorber, Jacob ; Kotz, David: Application Memory Isolation on Ultra-Low-Power MCUs. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA : USENIX Association, July 2018, pp. 127–132. – URL https://www.usenix.org/conference/atc18/presentation/hardin. – ISBN 978-1-939133-01-4

[20] Hassija, Vikas ; Chamola, Vinay ; Saxena, Vikas ; Jain, Divyansh ; Goyal, Pranav ; Sikdar, Biplab: A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. In: *IEEE Access* 7 (2019), pp. 82721–82743. – URL https://doi.org/10.1109/ACCESS.2019.2924045

[21] Hex Five Security, Inc.: *Hex Five MultiZone Security Datasheet*. 2020. – URL https://hex-five.com/wp-content/uploads/2020/01/multizone-datasheet-20200109.pdf. – Accessed 2026-07-01

[22] Jauernig, Patrick ; Sadeghi, Ahmad-Reza ; Stapf, Emmanuel: Trusted Execution Environments: Properties, Applications, and Challenges. In: *IEEE Security & Privacy* 18 (2020), March, Nr. 2, pp. 56–60. – URL https://doi.org/10.1109/MSEC.2019.2947124

[23] Karlsson, Henrik: *OpenMZ: a C implementation of the MultiZone API*, KTH, School of Electrical Engineering and Computer Science (EECS), Master's thesis, 2020. – 66 p.. – URL https://www.diva-portal.org/smash/get/diva2:1466978/FULLTEXT01.pdf. – Accessed 2026-07-01

[24] Kietzmann, Peter ; Schmidt, Thomas C. ; Wählisch, Matthias: PUF for the Commons: Enhancing Embedded Security on the OS Level. In: *IEEE Transactions on Dependable and Secure Computing* 21 (2024), Nr. 4, pp. 2194–2210. – URL https://doi.org/10.1109/TDSC.2023.3300368

[25] Krovetz, Ted ; Rogaway, Phillip: *The OCB Authenticated-Encryption Algorithm*. RFC 7253. May 2014 (Request for Comments). – URL https://www.rfc-editor.org/info/rfc7253. – Accessed 2026-07-01

[26] Lemke, Kerstin: Embedded Security: Physical Protection against Tampering Attacks. In: *Embedded Security in Cars*. Springer-Verlag, pp. 207–217. – URL https://doi.org/10.1007/3-540-28428-1_12

[27] MacKay, Ken: *micro-ecc*. – URL https://github.com/kmackay/micro-ecc. – Accessed 2026-07-01

[28] Mbed-TLS Documentation: *PSA key store design*. August 2024. – URL https://github.com/Mbed-TLS/TF-PSA-Crypto/blob/development/docs/architecture/psa-keystore-design.md. – Accessed 2026-07-01

[29] MITRE: *CWE-121: Stack-based Buffer Overflow*. July 2006. – URL https://cwe.mitre.org/data/definitions/121.html. – Accessed 2026-07-01

[30] MITRE: *CWE-125: Out-of-bounds Read*. July 2006. – URL https://cwe.mitre.org/data/definitions/125.html. – Accessed 2026-07-01

[31] Patterson, David ; Waterman, Andrew: *The RISC-V Reader: An Open Architecture Atlas*. 1st. Strawberry Canyon, 2017. – ISBN 0999249118

[32] Pfau, Lars: Integrating a RISC-V Secure Firmware into RIOT. (2024). – URL https://inet.haw-hamburg.de/teaching/ss-2024/project-class/pr2_lars_pfau.pdf. – Accessed 2026-07-01

[33] Pfau, Lars: Measuring the Performance Overhead of RIOT Running in RISC-V User-Mode. (2024), February. – URL https://inet.haw-hamburg.de/teaching/ws-2023-24/project-class/pr1_lars_pfau.pdf. – Accessed 2026-07-01

[34] Raspberry Pi Ltd: *RP2350 Datasheet*. October 2024. – URL https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf. – Accessed 2026-07-01

[35] Rescorla, Eric ; Tschofenig, Hannes ; Modadugu, Nagendra: *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. April 2022. – URL https://www.rfc-editor.org/info/rfc9147

[36] RIOT-OS Documentation: *Creating Modules*. December 2015. – URL https://doc.riot-os.org/creating-modules.html. – Accessed 2026-07-01

[37] RIOT-OS Documentation: *Packages*. December 2015. – URL https://doc.riot-os.org/group__pkg.html. – Accessed 2026-07-01

[38] RISC-V International: *About RISC-V*. – URL https://riscv.org/about/. – Accessed 2026-07-01

[39] RISC-V International: *RISC-V Platform-Level Interrupt Controller Specification*. – URL https://github.com/riscv/riscv-plic-spec/releases/download/1.0.0/riscv-plic-1.0.0.pdf. – Accessed 2026-07-01

[40] RISC-V International: *RISC-V Cryptography Extensions Volume I, Scalar & Entropy Source Instructions, Version 1.0.0*. December 2021. – URL https://github.com/riscv/riscv-crypto/releases/download/v1.0.0-scalar/riscv-crypto-spec-scalar-v1.0.0.pdf. – Accessed 2026-07-01

[41] RISC-V International: *RISC-V Advanced Core Local Interruptor Specification, Version 1.0-rc4*. January 2022. – URL https://github.com/riscv/riscv-aclint/releases/download/v1.0-rc4/riscv-aclint-1.0-rc4.pdf. – Accessed 2026-07-01

[42] RISC-V International: *The RISC-V Instruction Set Manual: Volume I, Privileged Architecture, Version 20240411*. April 2024. – URL https://github.com/riscv/riscv-isa-manual/releases/tag/20240411. – Accessed 2026-07-01

[43] RISC-V International: *RISC-V Supervisor Binary Interface Specification, Version 2.0*. January 2024. – URL https://github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v2.0/riscv-sbi.pdf. – Accessed 2026-07-01

[44] RISC-V International: *The RISC-V Instruction Set Manual: Volume II, Privileged Architecture, Document Version 20240411*. April 2024. – URL https://github.com/riscv/riscv-isa-manual/releases/tag/20240411. – Accessed 2026-07-01

[45] Selander, Göran ; Mattsson, John P. ; Palombini, Francesca ; Seitz, Ludwig: *Object Security for Constrained RESTful Environments (OSCORE)*. RFC 8613. July 2019 (Request for Comments). – URL https://www.rfc-editor.org/info/rfc8613. – Accessed 2026-07-01

[46] SiFive Inc.: *SiFive Unveils E2 Core IP Series for Smallest, Lowest Power RISC-V Designs*. June 2018. – URL https://www.sifive.com/press/sifive-unveils-smallest-lowest-power-risc-v-designs. – Accessed 2026-07-01

[47] SiFive Inc.: *SiFive FE310-G002 Datasheet v1p2*. March 2021. – URL https://sifive.cdn.prismic.io/sifive/4999db8a-432f-45e4-bab2-57007eed0a43_fe310-g002-datasheet-v1p2.pdf. – Accessed 2026-07-01

[48] SiFive Inc.: *SiFive FE310-G002 Manual v1p5*. September 2022. – URL https://sifive.cdn.prismic.io/sifive/034760b5-ac6a-4b1c-911c-f4148bb2c4a5_fe310-g002-v1p5.pdf. – Accessed 2026-07-01

[49] Valadares, Dalton Cezane G. ; Will, Newton C. ; Caminha, Jean ; Perkusich, Mirko B. ; Perkusich, Angelo ; Gorgonio, Kyller C.: Systematic Literature Review on the Use of Trusted Execution Environments to Protect Cloud/Fog-Based Internet of Things Applications. In: *IEEE Access* 9 (2021), pp. 80953–80969. – URL https://doi.org/10.1109/ACCESS.2021.3085524

[50] WIZnet: *W5500 Datasheet, Version 1.1.0*. November 2022. – URL https://docs.wiznet.io/img/products/w5500/W5500_ds_v110e.pdf. – Accessed 2026-07-01

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original