HAW HAMBURG

MASTER THESIS
Lena Boeckmann

Integration and Evaluation of a Secure Firmware for Arm Cortex-M Devices in RIOT OS

Faculty of Engineering and Computer Science
Department Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Lena Boeckmann

Integration and Evaluation of a Secure Firmware for Arm Cortex-M Devices in RIOT OS

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

Supervisor: Prof. Dr. Thomas Schmidt
Supervisor: Prof. Dr. Franz Korf

Submitted on: 18. March 2025

Lena Boeckmann

Thema der Arbeit

Integration and Evaluation of a Secure Firmware for Arm Cortex-M Devices in RIOT OS

Stichworte

TrustZone-M, TEE, RIOT OS, Sichere Firmware, Cortex-M, PSA Crypto

Kurzzusammenfassung

Im IoT sammeln Milliarden von heterogenen Geräten Informationen über ihre Umgebung und übertragen sie über lokale Netzwerke und das Internet. Oft sind IoT-Geräte in Bezug auf Speicher, Batterie und Rechenleistung eingeschränkt. Dies macht es schwierig, Sicherheitsmaßnahmen wie Datenschutz, Integrität und Authentifizierung zu implementieren. Hardwaremechanismen wie Arm TrustZone und die RISC-V Physical Memory Protection Unit können verwendet werden, um Trusted Execution Environments (TEE) in Mikrocontrollern zu implementieren, die es ermöglichen, sensible Daten und Codeausführung von nicht vertrauenswürdigen Systemkomponenten zu isolieren. Ein TEE bietet auch sichere Dienste, die von Anwendungen und Betriebssystemen über eine dedizierte Schnittstelle angefordert werden können. Ein IoT-Betriebssystem abstrahiert hardwarespezifische Details von Anwendungen und bietet eine gemeinsame Schnittstelle zum Zugriff auf Hardwarefunktionen. Um sicherzustellen, dass Anwendungen, die auf einem Betriebssystem laufen, portabel sind, sollte die Verwendung von Speicherisolation und Codeausführung in geschützten Umgebungen transparent und austauschbar sein. Diese Masterarbeit befasst sich mit der Integration und Bewertung einer sicheren Firmware für Arm-Mikrocontroller mit der TrustZone-M-Sicherheitserweiterung in RIOT OS. Es implementiert eine gemeinsame Schnittstelle für kryptografische Dienste, die von einem Betriebssystem zur Anforderung von Operationen verwendet werden kann, unabhängig von der zugrunde liegenden plattformspezifischen Firmware-Implementierung. Diese Arbeit stellt das Design und die Implementierungsdetails dieser sicheren Firmware vor und bewertet deren Speicherplatz- und Verarbeitungszeitverbrauch.

Lena Boeckmann

Title of Thesis

Integration and Evaluation of a Secure Firmware for Arm Cortex-M Devices in RIOT OS

Keywords

TrustZone-M, TEE, RIOT OS, Secure Firmware, Cortex-M, PSA Crypto

Abstract

In the IoT, billions of heterogeneous devices collect information about their environment and transmit it through local networks and the internet. Often, IoT devices are constraint in memory, battery and processing power. This makes it challenging to implement security measures such as privacy, integrity and authentication. Hardware mechanism such as Arm TrustZone and the RISC-V Physical Memory Protection Unit can be used to implement Trusted Execution Environments (TEE) in microcontrollers. TEEs allow for isolating sensitive data and code execution from untrusted system components. A TEE also provides secure services that can be requested by applications and operating systems through a dedicated interface. An IoT operating system abstracts hardware specific details from applications and provides a common interface to access hardware features. To ensure that applications running on an OS are portable, the use of memory isolation and code execution in protected environments should be transparent and interchangeable. This master's thesis deals with the integration and evaluation of secure firmware for Arm microcontrollers with the TrustZone-M security extension in RIOT OS. It implements a common interface for cryptographic services, that can be used by an operating system to request operations, regardless of the underlying platform-specific firmware implementation. This thesis presents the design and implementation details of this secure firmware and evaluates its consumption of memory space and processing time.

Contents

List of Figures vii

List of Tables ix

Listings x

Abbreviations xii

1 Introduction 1

2 Background and Related Work 5

2.1 Security in the IoT . . . 5
2.2 Establishing Trust in IoT Systems . . . 7
2.3 Trusted Execution Environments . . . 9
2.4 Hardware Protection Mechanisms for Constrained Devices . . . 11
2.4.1 Memory Protection . . . 11
2.4.2 Arm TrustZone . . . 12
2.4.3 TrustZone-based TEE Implementations . . . 15
2.5 The IoT Operating System RIOT . . . 17
2.5.1 A Cryptographic Subsystem in RIOT . . . 17
2.5.2 Trusted Computing in RIOT . . . 19

3 Software Design 22

3.1 Requirements . . . 22
3.2 The CryptoService API and Library . . . 23
3.3 Integration with the PSA Crypto Module . . . 27
3.4 A New Secure Firmware for Arm TrustZone-M . . . 28

4 Implementation 33

4.1 Target Platform . . . 33

4.2 Required Modifications in RIOT 35
4.2.1 Threading 36
4.2.2 PSA Crypto Integration 36
4.3 Secure Firmware Implementations 40
4.3.1 System Configuration during Start-Up 42
4.3.2 Calling a Secure Function from the Non-Secure Side 47
4.3.3 Using the Root of Trust to Encrypt and Decrypt Keys 52

5 Evaluation **56**
5.1 Runtime Overhead 57
5.1.1 Hashes 58
5.1.2 Cipher 59
5.1.3 ECDSA 59
5.2 Memory Usage 60
5.3 Interpretation of Results 61

6 Conclusion and Outlook **63**

Bibliography **65**

Glossary **70**

Declaration of Authorship **71**

List of Figures

2.1 Example for a secure boot chain with a secure firmware and a non-secure OS. 8
2.2 Basic concept of a system with trusted and untrusted components. 9
2.3 Comparison between TrustZone-A (left) and TrustZone-M (right) according to Pinto and Santos [1]. 13
2.4 PSA Crypto module in RIOT OS. 18
2.5 Schematic example of Trusted Firmware-M combined with RIOT OS. . . . 19
2.6 RAM and ROM usage of RIOT with a secure bootloader (BL) and Trusted Firmware-M built with medium profile as measured in previous work. . . . 20

3.1 Implementation structure of a secure firmware with the generic CryptoService API and library. Purple parts are part of CryptoService, the green parts are platform specific. 26
3.2 Integration of a secure firmware into the PSA Crypto module in RIOT OS with the CryptoService API. 28
3.3 Call flow between the non-secure application, the PSA Crypto module and the secure firmware with the CryptoService API. 29

4.1 RIOT-TEE secure firmware with the CryptoService API and library. Most code from CryptoService is reused. 41
4.2 RIOT-TEE with most CryptoService modules replaced by platform specific code (*e.g.,* hardware drivers). 41
4.3 Program flow of the secure firmware start-up process. 43
4.4 Example of a program flow for encrypting a plaintext with a plain key. . . 47
4.5 Example of a program flow for generating and storing a sealed key in PSA Crypto. 53
4.6 Example of a program flow for generating a hash signature with a sealed key. 55

5.1 Evaluation Configuration. 57
5.2 Comparison of processing times of multi-step hash computation. 58
5.3 Comparison of processing times of multi-step cipher encryption and decryption. 60
5.4 Comparison of processing times of ECDSA key generation, signature and verification. 61
5.5 Comparison of Flash and RAM usage of the example application with different secure firmware variants. 62

List of Tables

2.1 Differences between RISC-V PMP, Arm Cortex-A MPU and Cortex-M MPU. 12

4.1 PSA Crypto key slot location values. 37

5.1 Execution times of SHA-256 computation in numbers. 59
5.2 Execution times of AES-128 CBC encryption and decryption. 60
5.3 Execution times of ECDSA key generation, hash signature and message verification times with a NIST P-256 curve. 61

Listings

3.1 Declarations of the sealed key type and protected key operations. 24
3.2 Declarations of the unprotected operations. 25
3.3 Definitions of I/O structures and `tee_secure_entry` function. 29
3.4 Root of Trust operations. 31
3.5 Signature of a low-level API function in RIOT-TEE. 31
3.6 Signatures of crypto and random methods. 31

4.1 Conditional secure/non-secure register access on the nrf9160. 35
4.2 Conditional secure/non-secure register access on the nrf9160. 35
4.3 Default exception return values for new threads. 36
4.4 PSA sealed key slot structure. 36
4.5 PSA low-level driver glue code for CryptoService API. 38
4.6 Mutex wrapper implementations in RIOT. 39
4.7 CYS function claiming the secure firmware mutex. 39
4.8 PSA Crypto compile time configuration. 40
4.9 Configuration of non-secure callable regions in secure main function. . . . 43
4.10 Configuration of non-secure peripherals. 44
4.11 Configuration of floating point registers and exception priorities. 44
4.12 Initialization of RNG, CryptoCell and device root key. 45
4.13 Transition to non-secure side. 46
4.14 Example of a call to `tee_secure_entry` and I/O parameter packing. . 47
4.15 Implementation of the secure entry function. 48
4.16 Function type definition and jump table for dispatching secure function calls. 49
4.17 Address range check of non-secure input and output pointer with the `cmse_check_address_range` function. 50
4.18 Implementation of the `tee_cipher_aes_128_cbc_encrypt` function with the CryptoCell low level hardware driver. 51

4.19 Call of the OCB mode operation with a custom AES block operation. . . 53

Abbreviations

AEAD Authenticated Encryption with Associated Data.

AES Advanced Encryption Standard.

ARoT Application Root of Trust.

BOF Buffer Overflow.

CMSE Cortex-M Security Extension.

CSPRNG Cryptographically Secure Pseudo Random Number Generator.

DDoS Distributed Denial of Service.

DMA Direct Memory Access.

DRBG Deterministic Random Bit Generator.

ECC Elliptic Curve Cryptography.

ENISA European Union Agency for Cybersecurity.

ETSI European Telecommunications Standards Institute.

HWRNG Hardware Random Number Generator.

IoT Internet of Things.

IPC Inter-Process Communication.

KMU Key Management Unit.

MPU Memory Protection Unit.

NIST National Institute of Standards and Technology.

NSC Non-Secure Callable.

NSPE Non-Secure Processing Environment.

OS Operating System.

PKA Public Key Algorithms.

PMP Physical Memory Protection.

PoC Proof of Concept.

PSA Platform Security Architecture.

PUF Physical Unclonable Function.

RAM Random Access Memory.

RNG Random Number Generator.

ROM Read Only Memory.

RoT Root of Trust.

RTOS Real-Time Operating System.

SG Secure Gateway.

SoC System on a Chip.

SPE Secure Processing Environment.

TEE Trusted Execution Environment.

TF-A Trusted Firmware-A.

TF-M Trusted Firmware-M.

TPM Trusted Platform Module.

TRNG True Random Number Generator.

TZ TrustZone.

VTOR Vector Table Offset Register.

1 Introduction

The Internet of Things comprises a heterogeneous set of devices, that process and transmit sensitive data over the Internet [2]. Their hardware capabilities (memory capacity, power consumption and processing power) are usually constrained, but can differ widely between devices [3]. Hardware constraints also lead to differences in the software that can run on devices. Small sensor nodes with small memory and processing power may only fit a bare-metal application with a very small, specialized set of functions. Other, larger devices are equipped with features such as radio antennas and are able to run an Operating System (OS) with a scheduler, network stack, and complex cryptographic operations. The number of Internet of Things (IoT) devices has been on the rise in the past years, with several billions already deployed worldwide in the industry sector, critical infrastructure, the health and fitness sectors, as well as private homes. Since IoT devices are mass-produced and deployed in large quantities, it is a requirement for them to be as cheap as possible. This leads to restriction of resources, such as memory and processing power. Due to the resource constraints, IoT devices often use weak encryption algorithms, store sensitive key material in unprotected Read Only Memory (ROM) and Random Access Memory (RAM) and forgo security mechanisms such as secure boot and secure firmware updates. Often there is no long term support for devices, with companies deprecating products to replace them with newer ones, or even going bankrupt [2]. This results in large numbers of potentially vulnerable, unpatched devices, which transmit private or sensitive data, and provide entry points to larger company or home networks.

A common vulnerability of systems are keys that are hard-coded in flash or stored in unprotected RAM at runtime, and can be extracted with firmware dumps or read by Buffer Overflow (BOF) attacks. BOF attacks enable attackers to change the program flow and execute an arbitrary function, as well as inject and execute malicious code. Such an attack could be used to extract delicate information from memory, such as cryptographic keys, or interrupt the program flow to circumvent security checks. One

way to prevent unauthorized data extraction, is to store sensitive data and perform security critical operations in a Trusted Execution Environment (TEE). A TEE is an isolated environment within a System on a Chip (SoC), in which Trusted Applications can execute security critical operations, such as secure storage of data or cryptographic key material. A TEE can also be used to authenticate with other devices or attest to the integrity and trustworthiness of the firmware running on a device. Often, TEE implementations are combinations of software and hardware mechanisms for memory isolation and protection.

Today, many IoT devices run special operating systems, such as Zephyr OS, FreeRTOS and RIOT [4, 5, 6]. The benefit of an operating system is that it can provide a wide range of features and services, such as a network stack, a file system, and a scheduler, which can be used to implement complex applications. Thanks to hardware abstraction layers, an operating system can be ported to a wide range of platforms, which makes it easier to develop applications for different devices. E.g. instead of developing a bare-metal application with platform specific code for several devices, an application developer can use an operating system that abstracts the hardware and provides a common interface to the application. This way an application can easily run on different platforms and architectures without requiring modifications.
Those operating systems usually consist of a patchwork of modules, third party libraries and vendor drivers, which are developed and updated continuously. This complex and dynamic environment introduces vulnerabilities and bugs, and cannot be trusted at all times. Due to the constraints and simple memory layout of IoT devices, OS services, drivers and libraries often run in the same memory space. This is a security risk, especially when applications use cryptographic services with keys that are stored in unprotected memory, such as flash or RAM. The lack of isolation and protection makes operating systems susceptible to attacks, such as the aforementioned BOF attacks [7].
It is therefore beneficial to move security critical operations from an OS to an isolated environment. A TEE provides only a small set of features to execute security related tasks. It runs in protected memory regions, which require privileged access to read and write data or execute code. An operating system, on the other hand, has a large attack surface, due it's Internet connection via its network stack, serial interfaces, unrestricted access to peripherals and memory, as well as numerous features and services, whose implementations can introduce bugs and vulnerabilities. If the OS is compromised by an attacker exploiting such a vulnerability, a TEE provides an additional layer of protection

to make it harder for the attacker to extract keys or take over control of the whole device.

Some types of IoT devices are equipped with hardware mechanisms to support the implementation of TEEs based on memory protection and isolation. Examples of this are Arm Cortex-M and RISC-V, which are also two of the most common platforms for IoT systems. Both platforms have very different architectures and implement memory protection in different ways. Therefore, they require different approaches to implement TEEs based on their features. A non-secure operating system running side by side with such TEEs must be able to communicate with both of them. Abstracting platform specific secure firmware implementations with a common interface makes it easier for an operating system to use the secure services provided by the firmware, without knowing the specifics of the underlying hardware.

The vendor-independent IoT operating system RIOT [6] already runs on Arm and RISC-V platforms and provides a wide range of features and services. At the time of writing, it lacks support for privilege management and memory protection. As part of a separate thesis, the CryptoService API was developed. The CryptoService API can be implemented by platform-specific secure firmware implementations to provide cryptographic services to RIOT below a common, platform-agnostic interface. This way the portability of the operating system and applications running on top of it can be guaranteed.

This thesis provides two major contributions: It integrates the CryptoService API as a backend to the existing cryptographic submodule in RIOT, enabling applications and the operating system to request services from a secure firmware without requiring platform-specific code. The second contribution is a secure firmware implementation based on the TrustZone technology on Arm Cortex-M microcontrollers. This secure firmware implements the CryptoService API and can be used as a cryptographic backend by the crypto module in RIOT.

In Chapter 2, we provide background information and an overview of related work on security in the IoT, and existing hardware mechanisms that can support TEE, with a focus on TrustZone on Arm Cortex-M devices. In this chapter, we will also provide an introduction to the RIOT operating system and existing security features. We will describe previous work on improving cryptographic capabilities in RIOT, as well as the integration and evaluation of an existing reference implementation of a secure firmware for Arm TrustZone in RIOT.

In Chapter 3 we compile a list of requirements for a secure firmware in RIOT. We describe the design of the hardware-agnostic *CryptoService API*, and how it can be integrated with the existing cryptographic module in RIOT. We will also describe the design of a secure firmware for Arm TrustZone-M, which implements the *CryptoService API*.

Chapter 4 we present implementation details of the firmware and how it provides the services that are supported by the *CryptoService API*. The section futher describes the core contributions to the cryptographic module in RIOT, introducing a new method of securely storing cryptographic keys and using a secure firmware for cryptographic operations. Furthermore, it explains how the secure firmware complies with Arm guidelines for developing a secure firmware.

In Chapter 5 we will measure the processing time overhead of cryptographic operations with and without the secure firmware, as well as the introduced overhead in memory usage.

Finally, in Chapter 6, we will summarize the results and present directions for future work and improvement.

2 Background and Related Work

This chapter provides an overview of the concepts and technologies that are relevant for this work. It starts with an introduction to security and trust establishment in the IoT. The chapter continues with a section on Trusted Execution Environments in general and more specifically in constrained devices. We describe the existing hardware protection mechanisms in RISC-V and Arm Cortex platforms, that can be used to implement a TEE. The chapter closes with a description of the target operating system RIOT and its cryptographic submodule, as well as previous work on secure computing in RIOT.

2.1 Security in the IoT

The IoT comprises heterogeneous, typically constrained devices, that can form networks with other IoT devices. They can also connect to the Internet through gateways. These devices often collect data about their environment through sensors, process this data and can act on that information, by performing some physical action that has an impact on their environment [3, 2]. Typically they transmit the data they collect to a backend system for further processing and storage [3].

Constrained Device Classes. RFC 7228 [3] provides an overview of the types of constraints devices can have. The authors list constraints in maximum code complexity (impacted by available ROM and flash), size of state and buffers (impacted by RAM size), battery power, processing power, user interface and accessibility in deployment. Based on memory constraints, the authors distinguish between three classes of devices:

- **Class 0:** Devices with less than 10 KB of RAM and less than 100 KB of flash.
- **Class 1:** Devices with $\approx$ 10 KB of RAM and $\approx$ 100 KB of flash.
- **Class 2:** Devices with $\approx$ 50 KB of RAM and $\approx$ 250 KB of flash.

Bellman and van Oorshot [8] extend these classes with **Class 2+** devices, which have more resources than Class 2 devices, but are still constrained compared to traditional computers.

Lifecycle of an IoT Device. According to RFC 8576 [2], IoT devices differ from traditional, larger computer systems not only in resource constraints, but also in terms of their requirements, lifecycle and application areas. Those devices are usually tailored to specific tasks and consist of specialized hardware components, which are provided by multiple manufacturers and need to be interoperable.
The authors of RFC 8576 [2] outline a schematic lifecycle for IoT devices consisting of four phases: Bootstrapping, Operation, Maintenance and Decommissioning. During the bootstrapping phase devices are installed and commissioned in a network. In that phase they are also provisioned with secret keys they will use during normal operation for identification. During the operation phase devices are used in their intended environment and communicate with other devices and servers. In this phase they are controlled by the owner of the system they are installed in. Often they are deployed in public areas and are easily accessible to authorized and potentially unauthorized users. During their lifespans of several years, devices need to be maintained and upgraded. Also, running applications may need to be updated and reconfigured. These maintenance tasks can be performed either locally or remotely from a backend system. It may also be necessary to rebootstrap a device after an upgrade. After their lifespan has expired, devices are removed from the system and replaced by a different one. A decommissioned device may not necessarily be defective. It may be recommissioned in a different context or sold to a different owner. It is also possible that a device is deprecated by the manufacturer and will no longer obtain updates, leaving the device owner responsible for its functionality and security.

Security Requirements. RFC 8576 [2] also provides an overview of the state of the art of IoT security and the challenges introduced by constraints and heterogeneity. The authors describe a number of risks introduced by insecure IoT devices. Since they often can impact their environment though actors, they not only pose a threat to user privacy and data security, but also to physical safety. Cyberphysical systems with constrained sensor and actor nodes can be manipulated in order to cause them to malfunction, which can lead to environmental damage or injury. Additionally, compromised devices can be misused for scaled attacks such as Distributed Denial of Service (DDoS) attacks, or serve as entry points to larger networks, allowing attackers to move laterally and also compromise critical system components and infrastructure. As one of the most important

security threats the authors of RFC 8576 [2] name software bugs and bad software design choices. These can lead to attacks such as buffer overflow attacks and weak or missing authentication mechanisms. The importance of buffer overflow attacks is also shown by Mullen and Meany [7], who analyzed the susceptibility of an IoT operating system to them.
Other threats listed in RFC 8576 are privacy violations due to device location and usage tracking, cloning or substitution of devices to infiltrate a network, replacing firmware with malicious versions and privilege elevation.

Countermeasures need to be taken on different levels to ensure security of the whole system. The Internet draft *A summary of security-enabling technologies for IoT devices* [9] compiles a list of security measures and technologies based on security requirements specified by the National Institute of Standards and Technology (NIST), European Union Agency for Cybersecurity (ENISA) and European Telecommunications Standards Institute (ETSI). The draft overview spans multiple levels, from the hardware itself up to the management level. Listed countermeasures that are relevant for this work include establishing a Root of Trust, secure boot mechanisms, secure storage and handling of key material, state-of-the-art cryptography, data protection through Trusted Execution Environments and software isolation. These measures will be described in more detail in the following sections.

2.2 Establishing Trust in IoT Systems

When IoT devices communicate with each other, it is important to ensure the trustworthiness of all the devices involved. A secure boot chain (see Figure 2.1) ensures that a device was not compromised by an attacker, *e.g.,* through a malicious firmware update. To perform a secure boot, all software components must be signed with a private key by the vendor. The components are started one after another and each one verifies the integrity of the following one, before it is started.

The chain starts with an immutable *first stage bootloader*, which is stored in read-only memory. It is responsible for initializing the hardware and loading a signed second stage bootloader. The immutable bootloader is the first component that is executed after power-on. It is also responsible to write a device root key into a secure storage area. This key can either be provided by the manufacturer during production, or generated on the device with a mechanism such as SRAM PUF [10]. The key storage area can be

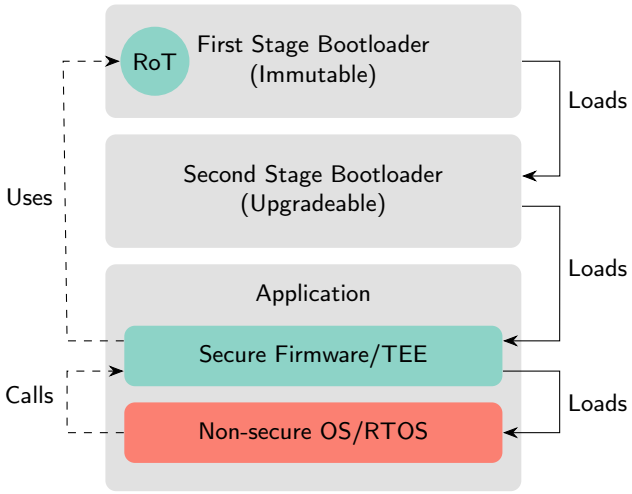

Figure 2.1: Example for a secure boot chain with a secure firmware and a non-secure OS.

a designated area in flash memory, which is locked afterwards, a secure element, or a protected key slot on some platforms. This key can later be used by a secure firmware for encryption and authentication of data, or for deriving new keys.

The *second stage bootloader* is responsible for verifying the integrity of the secure firmware and operating system images and loading the secure firmware. This bootloader is optional and only necessary if the bootloader itself should be upgradeable.
The second stage bootloader then loads the application, which can be either a simple bare-metal application or an operating system. In our example, it is a combination of a secure firmware and an operating system. The secure firmware runs prior to the OS and configures secure and non-secure memory, memory protection and GPIO access. It can access the device root key written by the first stage bootloader and use it to encrypt and authenticate data. Depending on its implementation it can also provide secure services to the OS through an API. After setting up the system it loads the *non-secure operating system*, which provides services to the user. If the secure firmware provides an API for secure services, the OS can call it to perform security critical operations.

Measured Boot and Remote Attestation. A different way to check for system integrity is measured boot. Instead of checking image signatures before starting them, the currently running software computes a hash (also called *measurement*) of the next component in the chain and stores it in a Trusted Platform Module (TPM). Other than secure boot, measured boot does not stop the system from booting when a component is compromised. Instead, it makes it possible to analyze the running parts of a system

later. To perform remote attestation, these measurements can be reported to a third party, which can use them to ensure the trustworthiness of the measured system. Secure boot, measured boot and remote attestation can be combined.

While a secure or measured boot process is necessary to make a device trustworthy, it will not be part of this work. The focus of this thesis is the implementation of a secure firmware that can use a Root of Trust (RoT) and provide secure services to an existing operating system. A secure boot process should be implemented in a separate effort.

2.3 Trusted Execution Environments

A Trusted Execution Environment is an isolated environment within a SoC, in which sensitive key material and data can be stored and processed, without exposing them to the main operating system or other applications. According to the GlobalPlatform *TEE System Architecture* [11] and the IEEE *Standard for Secure Computing Based on Trusted Execution Environment* [12] specifications, a TEE comprises a combination of hardware and software components which can be used to restrict memory access, manage privilege levels, and perform security critical operations. Its purpose is to ensure the integrity of trusted applications, which are executed within the TEE, and protect critical assets from unauthorized access. A TEE usually runs side by side with an untrusted operating system on the same device, as shown in Figure 2.2. An untrusted OS is not able to directly call internal functions of the TEE, but can request secure services through dedicated APIs.

The fundamental layer of a TEE comprises a Root of Trust [12], which is a combination of "reliable hardware, firmware and software components that perform specific, critical

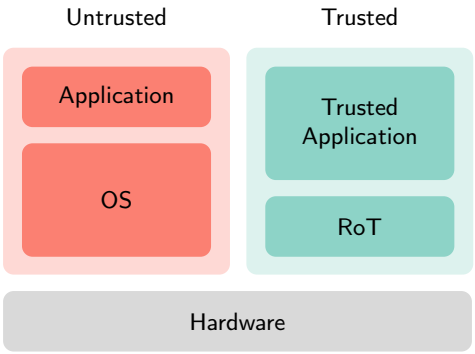

Figure 2.2: Basic concept of a system with trusted and untrusted components.

security functions" [12] and is instantiated in a secure boot process [11]. It is a non-extractable and unchangeable hardware-protected secret (*e.g.,* a device root key), that is the source of trustworthiness of the system, and can be used to derive other keys used for signatures or encryption.

Hardware-based TEEs. Examples of hardware components that can be used to implement TEEs in desktop computers are the *Intel SGX* and its successor *Intel TDX*, and *AMD PSP* for AMD processors. These examples have been developed for large computer systems with lots of resources and cannot be easily transferred to embedded devices.
In Arm Cortex application processors, which are used in mobile phones, tablets, gaming consoles, automotive and the industrial sector, TrustZone provides a base for TEE implementations. While Arm TrustZone already takes some constraints into account, it was originally designed for Cortex-A processors, which still have a lot more resources than microcontrollers used in IoT devices. To bring TrustZone to microcontrollers, Arm introduced the *TrustZone-M* security extension, which is a more lightweight redesign of the original TrustZone. Other memory protection mechanisms that were optimized for microcontrollers are the Arm Memory Protection Unit (MPU) and RISC-V Physical Memory Protection (PMP), which will be described in Section 2.4. All of these mechanisms only provide a base for a TEE and need to be combined with software components to provide secure services to an operating system.

Software-based TEEs. Pure software implementations of TEEs can also provide memory isolation on platforms without the necessary hardware mechanisms. SofTEE [13] deprivileges the kernel and provides a secure monitor for privileged execution. OpenTEE [14] is a software-based TEE that runs Trusted Applications as isolated processes and manages communication between trusted and untrusted applications through a manager process. Zandberg and Baccelli introduce Femto-containers [15] to isolate software modules through containerization and virtualization in IoT devices. While these software implementations are flexible and hardware-independent, they do not make use of available hardware protection mechanisms.

2.4 Hardware Protection Mechanisms for Constrained Devices

Manufacturers of microcontrollers have added several hardware mechanisms to protect memory regions from unprivileged access. Examples are Arm MPU and RISC-V PMP, as well as the TrustZone-M security extension for Cortex-M microcontrollers. They all can be used as a basis for TEE implementations in the IoT.

2.4.1 Memory Protection

The RISC-V PMP and Arm MPU operate in similar ways. They rely on applications and operating systems running on different privilege levels and restricting certain memory regions to only be accessible with certain privilege levels. The main differences are summarized in Table 2.1.

RISC-V. The RISC-V architecure [16, 17] provides three different privilege modes: Machine, Supervisor and User mode. The machine mode has the highest privilege level and is the only one that is required in all implementations, while supervisor mode and user mode are optional. A RISC-V processor boots in machine mode and can configure up to 16 memory regions with sizes ranging from 4 B to 32 GB for unprivileged access [18]. After initializing the system, it switches to user mode to run unprivileged applications. When access to a protected memory region is needed, a software interrupt triggers the switch to machine mode for executing privileged code.

Arm MPU. Devices with the Armv8-A architecture and an MPU provide four exception levels (E0-E3), with E0 being the lowest privilege level reserved for applications and E3 being the highest privilege level for the firmware. Armv8-M devices provide a privileged handler mode and a thread mode, which can be privileged or unprivileged. Similar to the RISC-V PMP, the MPU can configure memory regions to only be accessible on certain exception levels (Cortex-A) or in privileged mode (Cortex-M). Compared to the RISC-V PMP, the MPU is less flexible in terms of configuration: There are 8 regions for privileged access and 8 for unprivileged access available, with sizes ranging from 32 B to 4 GB. Cortex-M0/M3/M4 devices only support 8 different regions in total, with a minimum size of 32 B. Additionally, sizes can only be a power of 2. Those regions can be further divided into eight subregions, which all have equal access rights, but can be individually activated.

	RISC-V PMP	Cortex-A	Cortex-M	
			M0/M3/M4	Others
Privilege Levels	User (Unprivileged) Supervisor (Medium Privilege) Machine (High Privilege)	E0 (Applications) E1 (privileged OS kernel) E2 (Hypervisor) E3 (Secure Monitor)	Thread Mode (unprivileged) Thread Mode (privileged) Handler Mode (privileged)	
Memory Regions	0-16 freely configurable	8 privileged 8 unprivileged	8 regions with 0-8 subregions	8 privileged 8 unprivileged
Region Sizes	4 B - 32 GB	32 B - 4 GB	$2^5 - 2^x$ B memory aligned	32 B - 4 GB

Table 2.1: Differences between RISC-V PMP, Arm Cortex-A MPU and Cortex-M MPU.

2.4.2 Arm TrustZone

In addition to the MPU, Arm has introduced the TrustZone security extension for Cortex-A application processors with the Armv8-A architecture and, most recently, Cortex-M microcontrollers with the Armv8-M architecture. Both add a system separation into secure and non-secure domains. The processor switches between a secure and non-secure mode to manage access to memory and peripherals. While TrustZone-A and TrustZone-M share the same basic principles, they are implemented in different ways to take the difference in hardware constraints into account. Pinto and Santos [1] provide an overview of the differences between the two, which are also illustrated in Figure 2.3.

Arm TrustZone-A. TrustZone-A introduces a system separation into two domains, the secure world and the normal, non-secure world. The processor can only operate in one of those domains at a time, which requires switching between secure and non-secure states, as shown on the left side of Figure 2.3. To preserve state and execute the switch, the processor needs to be in the *secure monitor mode* triggered either by a special instruction, the *secure monitor call*, or by exceptions and interrupts. For additional isolation, some specific registers are banked, meaning they have a separate copy for each domain. Additional, optional security features such as the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA), can be used to configure memory regions to be accessible only from the secure world.

Arm TrustZone-M. TrustZone-M implements a different approach to make context switches between the secure and non-secure world feasible in low-powered microcontrollers. On Cortex-A devices, using the secure monitor for context switches introduces overhead in execution times, interrupt latency and energy consumption. On Cortex-M, the system separation is memory map-based and the switch between secure and non-

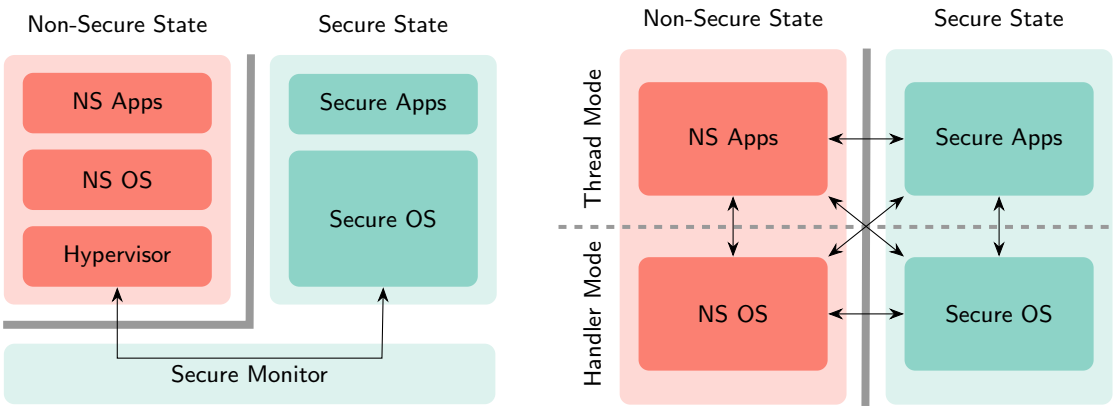

Figure 2.3: Comparison between TrustZone-A (left) and TrustZone-M (right) according to Pinto and Santos [1].

secure worlds is performed by hardware. To use the TrustZone-M technology, software running on the system needs to be split into two applications: A secure firmware needs to run in secure mode and access secure memory addresses, while a bare-metal app or an operating system operates in non-secure mode and non-secure memory regions.
Per default, flash and RAM are configured as secure. When the secure and non-secure binaries are flashed to the device, no memory regions have been configured as non-secure, yet. This means, the non-secure application can only run after the secure firmware has explicitly configured its dedicated flash and RAM regions as non-secure. The same applies to peripherals, which are also separated into secure and non-secure regions. Most peripherals have two addresses, one for each security state. Security critical peripherals, such as the crypto accelerator or the system protection unit are only accessible in the secure state. Non-secure peripheral access must be explicitly configured by the secure firmware during system setup. When the device is booted, the secure firmware runs prior to the non-secure application and can configure the memory regions and peripherals that the non-secure application should be allowed to access.

Between the secure and non-secure flash areas is a special Non-Secure Callable (NSC) area located. The NSC holds Secure Gateway (SG) instructions, which serve as entry points for the non-secure side to call secure functions. This NSC area must also be configured by the secure firmware during startup.

The secure and non-secure images also operate on separate stacks. There are four different stack pointers available, two main stack pointers (`MSP_S` and `MSP_NS`) and two process

stack pointers (`PSP_S` and `PSP_NS`) for the respective security states. The secure main stack pointer is used by the secure firmware, while the non-secure main stack pointer is used by the non-secure application. Process stack pointers are only used, when threading is enabled on either side.
Registers are mostly shared between the states, except for the stack pointers, the Vector Table Offset Register (VTOR) and some special registers such as the Control and Fault Mask Register.

There are several ways to switch between secure and non-secure states after the system is set up and running. The non-secure side can call secure functions provided by the secure side. In this case, the call is directed to the `SG` instruction in the NSC region, which triggers the switch to the secure state and then executes the actual secure function in the secure world. After execution, the secure side can use two special instructions (`BXNS` and `BXLNS`), to either jump back to non-secure after a secure execution or call non-secure functions.

Another way to switch between states is through exceptions and interrupts. These can also be marked as secure or non-secure. In general, it is possible that a non-secure interrupt preempts a secure execution, though this can be prevented by deprioritizing non-secure interrupts or disabling them during secure execution.

Secure and non-secure states both provide a handler mode and a thread mode for additional privilege separation. These are useful when threading is enabled on either side. Switches between all states and privilege levels can be triggered by exceptions and interrupts (shown on the right side of Figure 2.3).

Known Vulnerabilities of TrustZone-M. TrustZone-A has been analyzed extensively and a number of vulnerabilities have been reported [19]. While the research on TrustZone-M is not as extensive, vulnerabilities and attacks have also been reported.

In 2019 Thomas Roth [20] demonstrated fault injection vulnerabilities allowing to bypass secure boot and TrustZone-M features, and recover symmetric keys on Microchip's SAML11 platform.

In 2023, Ma *et al.* [21] published *ret2ns*, a technique that exploits the fast switch between secure and non-secure worlds for arbitrary code execution and privilege elevation. The authors show that an attacker can corrupt a code pointer used by the `BXNS` and `BXLNS` instructions to jump to an attacker-controlled user space program without switching to unprivileged mode. They present several variations of this attack, suitable for operating

systems as well as bare-metal apps with support for privilege separation. The authors propose an address sanitizing mechanism for mitigation.

In 2024, Rodrigues, Oliveira and Pintos presented *BUSted* [22], a time-based side channel attack on Cortex-M CPUs. The authors prove that it is possible to exploit the shared bus on Cortex-M CPUs to extract secrets from code running on the platform. They show that using TrustZone-M does not protect from side channel attacks.

In 2021, Abbott and Altherr [23] demonstrated how to use a hidden ROM patching feature on the NXP LPC55S69 to perform a privilege escalation and enable the non-secure world to execute an arbitrary payload in the secure world. While this is not a vulnerability in TrustZone-M itself, it shows how implementation and configuration errors can introduce flaws to a TrustZone-M based system.

While previous work showed that TrustZone-M is vulnerable to physical attacks, it is notable, that those attacks require high effort, specialized equipment, and prolonged physical access to a device. A TrustZone-M-based secure firmware can still protect from remote attacks and unauthorized access to secure data.

2.4.3 TrustZone-based TEE Implementations

TrustZone-A has been released in 2004 and since then is used as a foundation for a number of TEE implementations, such as TrustICE [24], OP-TEE [25] and the official reference implementation of the Arm Firmware Framework Specification Trusted Firmware-A (TF-A) [26]. Zhao *et al.* [10] have built a Root of Trust based on SRAM Physical Unclonable Function (PUF) for Arm TrustZone. The authors argue that TrustZone only provides an "isolated" environment, which needs to be combined with a RoT to become trustworthy. Current TrustZone-based systems work on the assumption that the RoT consists of a unique device key only accessible by the secure world. Zhao *et al.* argue that such a key is not always available and also that keys should be updateable to protect from side-channel attacks. They therefore implement secure key storage and a PUF-based truly random source as the foundation of a RoT. Yang *et al.* [27] published Trust-E, a trusted embedded operating system architecture, which is based on Arm TrustZone, but also aims to support other processors with similar spatial isolation features. Trust-E implements the GlobalPlatform TEE System Architecture specification [11].

The open-source project μTango [28] is built on the much newer Arm TrustZone-M. μTango is a TEE with a minimal secure scheduler and communication channel, which can run multiple isolated non-secure applications. In this approach, each application runs in the non-secure domain, but has its own dedicated memory partition. The secure firmware does not provide any secure services. It is only responsible for scheduling the non-secure applications and providing secure communication between them. In an example setup, they run an Real-Time Operating System (RTOS), a network stack, a command line and an LED blinker application in separate non-secure partitions. Each of the application only accesses the peripherals and memory areas that it actually needs. Security benefits of μTango are a minimization of privileged execution, a granular enforcement of memory isolation and protection from control-flow hijacking. Their evaluation shows a linear increase in overhead with each additional non-secure world and increased interrupt latency as non-secure worlds need to wait for their turn on being scheduled. While μTango [28] has a good approach to strictly enforce isolation, it has limitations at time of publication. Several Arm microcontrollers with TrustZone-M provide features, that can only be accessed from the secure world (*e.g.,* cryptographic accelerator and hardware key storage). In the current version of μTango, these features are not supported, since the secure firmware does not provide them and the non-secure applications cannot access them. The goal of this work is to make such features available to our target operating system RIOT, which is why a different approach is needed. Additionally, running services such as the network stack isolated from the operating system, would require us to extract this functionality from the OS and compiling it separately, while still ensuring interoperability. This would require a lot of effort and increase overall complexity of the system.

Similar to Trusted Firmware-A, Trusted Firmware-M (TF-M) [29], is the official reference implementation of a secure processing environment for Cortex-M processors, which leverages TrustZone-M and dual-core architectures. In a previous effort, TF-M was run and evaluated as a secure firmware with RIOT [30]. That work concluded that TF-M is not a fitting solution for RIOT, mostly because it is not platform-agnostic and can only be run on Arm devices. It is also a highly complex and configurable implementation, which significantly increases code size.

2.5 The IoT Operating System RIOT

RIOT is an open source operating system for constrained IoT devices, which is designed to be energy efficient with a small memory footprint [6]. It is portable to a wide range of devices and architectures: At the time of writing it supports more than 270 different platforms. The kernel provides basic functionality, such as multi-threading and context switching, synchronization primitives, a tickless scheduler based on fixed priorities and preemption, and Inter-Process Communication (IPC).

Additional system libraries, third-party libraries, drivers and a network stack can be added through modules, depending on the platform and use-case, allowing for fine-grained configuration and small-sized binaries without unnecessary components.

2.5.1 A Cryptographic Subsystem in RIOT

In previous work we introduced an implementation of the Arm Platform Security Architecture Crypto API [31] as a unified, platform-agnostic crypto module for RIOT OS [32, 33].

PSA Crypto is one of four secure service APIs that were specified by Arm as part of the Platform Security Architecture (PSA) Framework [34]. PSA comprises a set of specifications, tools and requirements for hardware and software design with the goal of achieving a *PSA Certified* status. While the focus of PSA lies on Arm Cortex-A and Cortex-M devices, several specifications (*e.g.,* for the secure service APIs) are architecture agnostic and thus applicable to other platforms.

In RIOT, the PSA Crypto module provides a common interface for cryptographic operations and key management, with interchangeable backends. Operations can be executed in hardware or software, depending on the platform. Key material is managed internally by a key management module and can be stored either in local memory or in a secure element. This way RIOT OS can support the whole range of cryptographic software and hardware backends that is available in IoT devices. The implementation architecture is shown in Figure 2.4.

To perform crypto operations, applications can call the PSA Crypto API. The second layer below the PSA Crypto API is the key management module, which is responsible for storing and retrieving keys. Keys have to be generated or imported before they can

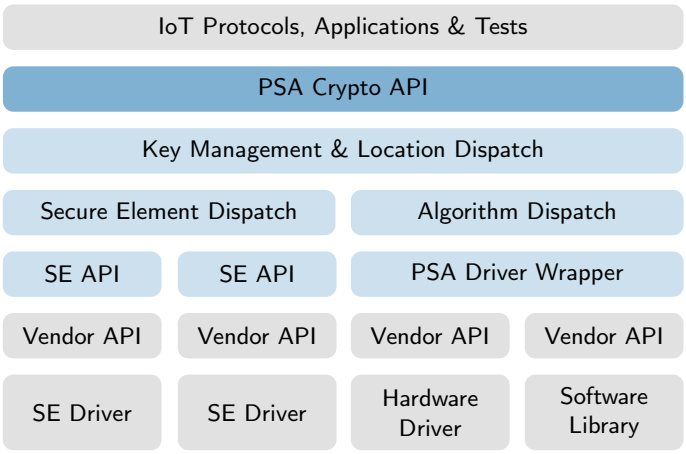

Figure 2.4: PSA Crypto module in RIOT OS.

be used and are then handled internally, without being accessible to the user. Each key is stored with a set of attributes, including the key type, size, storage location, lifetime and usage policy. Upon key creation, each key is assigned a unique identifier, which is used to reference the key in further operations.

After calling a crypto function, the location dispatcher checks whether the key is stored in local memory or in a secure element. If the key is stored in a secure element, the call will be forwarded to the corresponding secure element driver. Through a secure element dispatcher and a dedicated secure element API, the module can support multiple secure elements simultaneously.

If a key is stored in local memory, the call will be forwarded to the algorithm dispatcher. This instance will check the key type and usage policy, forward the call to the corresponding algorithm implementation and pass the key as an argument. Each algorithm can be implemented either by a hardware driver or a software library and can be transparently replaced by another implementation.

At the time of writing, all crypto operations in RIOT are executed in the same memory space as the rest of the operating system. Also, the PSA Crypto module stores key material in RAM or unencrypted in flash (exceptions are keys stored in secure elements, but those are not relevant for this work). To provide memory isolation and secure key storage, this work will add support for hardware-based memory protection and isolation.

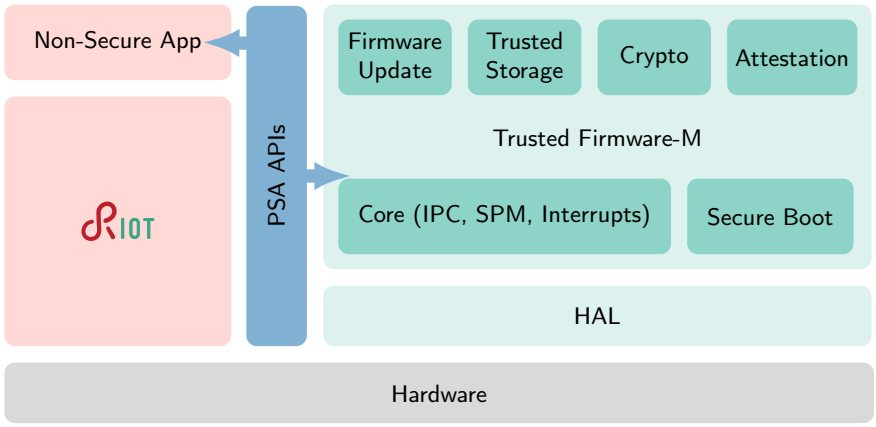

Figure 2.5: Schematic example of Trusted Firmware-M combined with RIOT OS.

2.5.2 Trusted Computing in RIOT

To leverage the security features of TrustZone-M and provide secure services to RIOT applications, a secure firmware is needed. An example of such a firmware is TF-M [29], the reference implementation of the Arm Firmware Framework-M, which is also part of the PSA framework. It provides a secure processing environment on Armv8-M and Armv8.1-M architectures with the TrustZone-M security extension, as well as dual-core platforms.

TF-M implements the four secure service APIs specified by the PSA framework: Crypto, Secure Storage, Attestation and Firmware Updates. It provides multiple levels of memory isolation: level one provides a simple separation into two worlds. Higher isolation levels provide separate partitions for each PSA service, IPC, and, at the highest level, a separate Application Root of Trust (ARoT) for each application running in the non-secure world. Several configuration profiles with varying levels of isolation and feature sets allow for adapting the firmware image to the capabilities of the underlying platform. E.g. the small profile only provides basic crypto operations, such as hashes and symmetric encryption and level one isolation, while the largest profile provides level three isolation, asymmetric cryptography and ARoT.

In previous work we ran RIOT together with TF-M [30], and measured the overhead the firmware introduces to RIOT (the setup is shown in Figure 2.5). For this purpose we built an example application that uses a hash generation, a symmetric cipher operation and an ECDSA key generation, signature and verification. Those crypto operations are provided by the PSA Crypto API.

TF-M uses the PSA Crypto implementation provided by the MbedTLS library [35]. For better comparability we replaced the PSA Crypto implementation of RIOT with MbedTLS, with the same configuration that is used by TF-M. We then built and ran this application with RIOT as a Non-Secure Processing Environment (NSPE) with TF-M (medium profile with ARoT) as the Secure Processing Environment (SPE) and compared the memory usage and processing time to a RIOT-only binary.

Figure 2.6 shows the RAM and flash usage of both binaries as they were measured in [30]. The SPE and NSPE binaries amount to a combined size of 150 KB (around three times as much as RIOT alone) and 50 KB of RAM (around ten times as much as RIOT alone). A secure bootloader with included cryptographic features adds another 50 KB of flash and 25 KB of RAM.

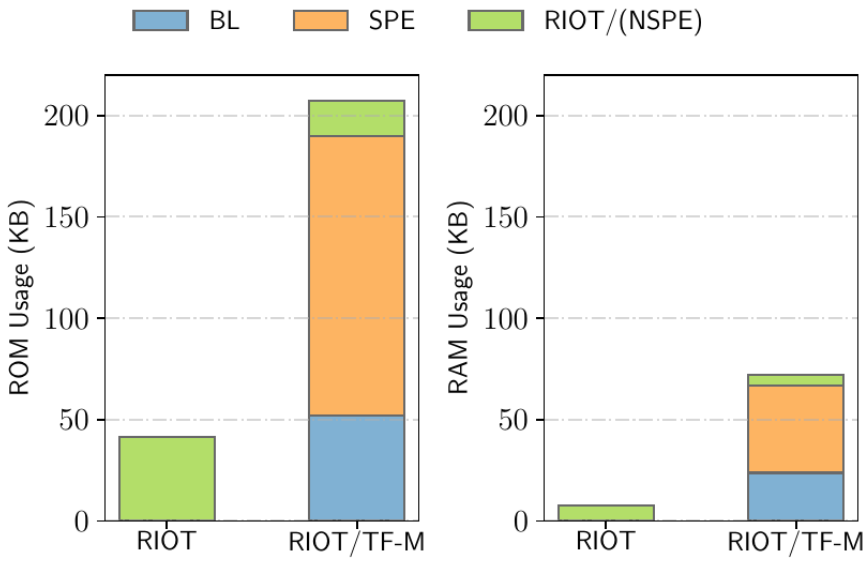

Figure 2.6: RAM and ROM usage of RIOT with a secure bootloader (BL) and Trusted Firmware-M built with medium profile as measured in previous work.

While this is not an issue on devices such as the nRF9160, it is infeasible for smaller devices with less flash and RAM. While it is possible to build a smaller TF-M profile, this will also reduce the available features, such as support for asymmetric cryptography.

When using TF-M as a secure firmware, it is intended that non-secure applications and the operating system directly call the secure service APIs provided by the firmware (see Figure 2.5). Additionally, TF-M only supports Arm devices and no other architectures. On other platforms, TF-M would need to be replaced completely by a different secure

firmware implementation. To ensure portability, other secure firmware implementations used by RIOT would also be required to implement the PSA APIs and provide their own secure service implementations.

The main objective of this work is to isolate cryptographic operations from the operating system and provide a way to securely store keys. It is more beneficial to reuse the existing PSA Crypto implementation in RIOT and just replace the implementation of the cryptographic operations with an isolated secure firmware.

Therefore, we design and implement a new approach to provide cryptographic services to the PSA Crypto module, below a common, platform-agnostic API. This API can be implemented by different secure firmware implementations for different architectures. This way, applications and RIOT itself can still use the PSA Crypto interface, which then transparently dispatches crypto operations to an isolated environment. Our approach ensures that applications in RIOT remain platform-agnostic and portable to a wide range of devices.

3 Software Design

In this chapter we first present a summary of the requirements for a secure firmware in RIOT. In subsections, we describe the design of a common, platform-agnostic API for secure services in RIOT, that can be implemented by different secure firmware backends. We also describe the design of a secure firmware for TrustZone-M-enabled Arm devices, which implements the API as a proof of concept. We also demonstrate how the CryptoService API and the secure backend can be integrated in the PSA Crypto module in RIOT.

3.1 Requirements

The target operating system RIOT supports a wide range of platforms with varying features and capabilities. When adding support for memory protection mechanisms, the solution should take the different architectures into account and aim to support as many platforms as possible. Different platforms provide different technologies to support memory isolation and protection, which require different approaches to implement a secure firmware. Since application development in RIOT should remain platform-agnostic and user-friendly, develpers should be able to use the secure services provided by the firmware without taking into account the specifics of the underlying hardware. These prerequisites lead to the following requirements for a secure firmware solution in RIOT in no specific order:

1. The secure firmware must allow RIOT to leverage memory protection features on different constrained devices.
2. A secure firmware must provide an isolated execution environment for security critical operations.

3. The firmware must provide secure services to the non-secure world in RIOT though a common interface that abstracts the underlying hardware.
4. The secure firmware must provide a way to securely store cryptographic key material and execute cryptographic operations.
5. Access to secure services must be transparent and not require platform-specific code on the application level.
6. A secure firmware should not impact functionality of the operating system. RIOT should be able to provide its services to applications as if there were no secure firmware.
7. The firmware should only provide a minimal set of cryptographic operations to keep the memory footprint and the attack surface small.
8. The firmware should not introduce additional vulnerabilities to the system.
9. The firmware should be able to use a Root of Trust that has been set during system initialization.
10. For usability, integration in RIOT should be transparent and not require developers to have extended knowledge about the underlying technologies.

In the following sections we will describe the design of our secure firmware solution for RIOT and how we fulfilled the listed requirements.

3.2 The CryptoService API and Library

A platform-agnostic API for secure firmware implementation is needed to fulfill the requirement of a common interface for secure services in RIOT. For this purpose of the CryptoService (CYS) API was designed by Lars Pfau. The API provides a basic set of cryptographic operations, which can be implemented by different secure firmware backends for various architectures, such as Armv8-M and RISC-V. It also provides a default

software implementation for cryptographic operations, which can be replaced by platform specific implementations, *e.g.,* drivers for hardware accelerators.

Key Encryption. While some platforms provide hardware protected key slots to store keys securely, others do not. To support secure key storage on all platforms, the CryptoService API supports the encryption of key material. For this purpose the API supports a protected key type `CYS_PROT_ecc_p256_key_t`. A secure firmware implementing the API must support the encryption of a plain key with Authenticated Encryption with Associated Data (AEAD). The encrypted key is stored with a corresponding nonce and tag as shown in lines 1-4 of Listing 3.1.

Listing 3.1: Declarations of the sealed key type and protected key operations.

```
1  typedef struct {
2      uint8_t nonce[CYS_PROT_SEAL_NONCE_SIZE];
3      uint8_t private_key[CYS_PROT_ECC_P256_KEY_SIZE + CYS_PROT_SEAL_TAG_SIZE];
4  } CYS_PROT_ecc_p256_key_t;
5
6  CYS_error_t CYS_PROT_ecc_p256_generate(CYS_PROT_ecc_p256_key_t *sealed_key,
7                                         uint8_t *public_key);
8
9  CYS_error_t CYS_PROT_ecc_p256_seal(const uint8_t *unsealed_key,
10                                    CYS_PROT_ecc_p256_key_t *sealed_key);
11
12 CYS_error_t CYS_PROT_ecc_p256_derive(
13                                   const CYS_PROT_ecc_p256_key_t *sealed_key,
14                                   uint8_t *public_key);
15
16 CYS_error_t CYS_PROT_ecc_p256_sign(const CYS_PROT_ecc_p256_key_t *key,
17                                    const uint8_t *hash,
18                                    size_t hash_len,
19                                    uint8_t *signature);
```

A set of operations is provided to work with protected keys (see Listing 3.1). At the time of writing, only the encryption of Elliptic Curve Cryptography (ECC) keys is supported, but the API can be extended to support other key types. The four existing functions provide several features. A key generation function generates a key pair, encrypts the private key and then returns the encrypted private key and the public key. In this case, the unencrypted key is never exposed to a non-secure application. It can only ever be decrypted and used within the context of the secure firmware. A sealing function encrypts a plain key that is passed to the secure firmware by the application and then returns the

encrypted key back to the non-secure world. A key derivation function can be used to derive a public key from an encrypted private key, while a signature function can be used to sign a hashed message with an encrypted private key. The keys should be encrypted with a device specific root key. It should not be possible to decrypt the key on a different device.

Cryptographic Operations. Unprotected functions operate on plain keys that are passed by the application as arguments (see Listing 3.2). At the time of writing, the unprotected API supports random number generation, multi-step hashing, cipher operations, as well as asymmetric key generation, derivation, signature and verification.

Listing 3.2: Declarations of the unprotected operations.

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

// Hash Context and Operations
typedef struct {
    uint32_t data[CYS_HASH_SHA256_STATE_SIZE];
} CYS_hash_sha256_ctx_t;

CYS_error_t CYS_hash_sha256_init(CYS_hash_sha256_ctx_t *ctx);

CYS_error_t CYS_hash_sha256_update(CYS_hash_sha256_ctx_t *ctx,
                                   const uint8_t *data,
                                   size_t len);

CYS_error_t CYS_hash_sha256_finalize(CYS_hash_sha256_ctx_t *ctx,
                                     uint8_t *digest);

// Cipher operations
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,
```

```
31                                      uint8_t *message);
32
33 // Asymmetric Operations
34 CYS_error_t CYS_ecc_p256_generate(uint8_t *private_key,
35                                   uint8_t *public_key);
36
37 CYS_error_t CYS_ecc_p256_derive(const uint8_t *private_key,
38                                 uint8_t *public_key);
39
40 CYS_error_t CYS_ecc_p256_sign(const uint8_t *private_key,
41                               const uint8_t *hash,
42                               size_t hash_len,
43                               uint8_t *signature);
44
45 CYS_error_t CYS_ecc_p256_verify(const uint8_t *public_key,
46                                 const uint8_t *hash,
47                                 size_t hash_len,
48                                 const uint8_t *signature);
```

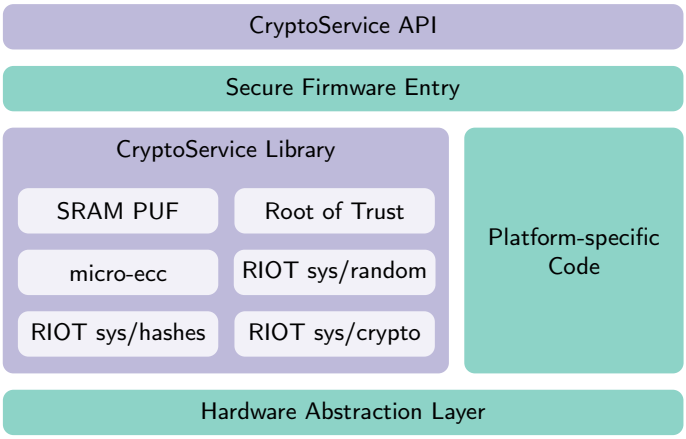

Figure 3.1: Implementation structure of a secure firmware with the generic CryptoService API and library. Purple parts are part of CryptoService, the green parts are platform specific.

CryptoService Library. The CryptoService API provides default software implementations of cryptographic operations for devices that do not have cryptographic hardware acceleration. For these implementations, existing code from the RIOT operating system is reused. The CryptoService library includes Random Number Generator (RNG), hash and cipher operations from the RIOT `sys` module, as well as the RIOT SRAM PUF implementation for entropy generation. For ECC operations it uses the third-party library micro-ecc, which provides a lightweight implementation of ECC operations and is

also supported by RIOT. A secure firmware that implements the CryptoService API can either reuse the provided software or replace it with platform specific implementations.

Figure 3.1 shows how the CryptoService API and library are intended to be used by a secure firmware. The common components are the northbound interface and the default software library (marked in purple). Both can be reused by all platforms. Green parts need to be provided by a platform specific firmware. This way, hardware-specific code for the secure firmware entry and the memory configuration of the RISC-V PMP and Arm TrustZone-M can be implemented separately. Some platforms may provide additional security features, such as hardware key storage and a Hardware Random Number Generator (HWRNG), which need to be provided as additional platform-specific code and hardware abstraction.

3.3 Integration with the PSA Crypto Module

In RIOT, PSA Crypto is integrated into the operating system as the official crypto module. It provides a common interface for cryptographic operations, which already allows for the transparent use of different cryptographic libraries and drivers. It also implements internal key management, allowing for ID-based indirect access to stored keys. This way applications can pass identifiers of the keys they want to use for cryptographic operations, while PSA Crypto transparently handles the storage and management.

It is easy and straightforward to integrate a secure firmware that implements the CryptoService API as an alternative crypto backend into the PSA Crypto module (see Figure 3.2).

An application or any code running in RIOT on the non-secure side can call the PSA Crypto APIs as usual. All calls to cryptographic operations are automatically passed to the secure firmware through the CryptoService API. Keys that are generated and encrypted by the secure firmware can be securely stored by the key management module of PSA Crypto. When an encrypted key should be used for a cryptographic operation, the PSA Crypto module passes the encrypted key to the secure firmware, which can decrypt it internally and use it for the operation.

If a platform provides hardware key storage, keys can transparently be stored in hardware key slots, with only a reference to it being returned to the PSA Crypto module for storage and later access.

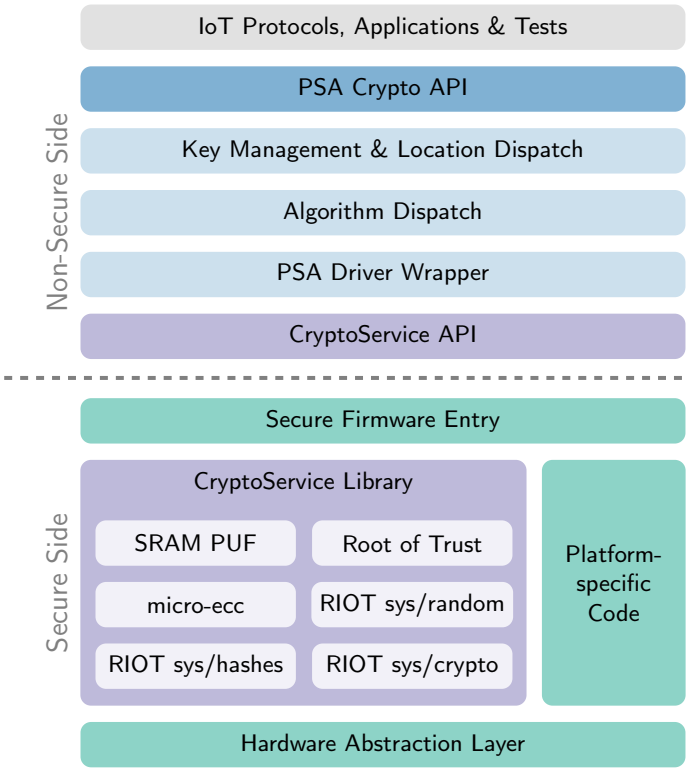

Figure 3.2: Integration of a secure firmware into the PSA Crypto module in RIOT OS with the CryptoService API.

Figure 3.3 shows a simplified call flow between the non-secure application, the PSA Crypto module and the secure firmware with the CryptoService API. When comparing it to the setup of RIOT with Trusted Firmware-M shown in Figure 2.5, the difference is clear. The interface between RIOT and TF-M are the PSA APIs, which could be called directly by applications. In our design, the interface is provided by the CryptoService API, which cannot be called directly by applications, but is only called by the PSA Crypto module. This way applications can us the PSA Crypto APIs as usual, while the PSA Crypto module calls the secure firmware through the CryptoService API.

3.4 A New Secure Firmware for Arm TrustZone-M

The product of this thesis is RIOT-TEE, a secure firmware for Arm Cortex-M devices with the Armv8-M architecture and the TrustZone-M security extension. RIOT-TEE

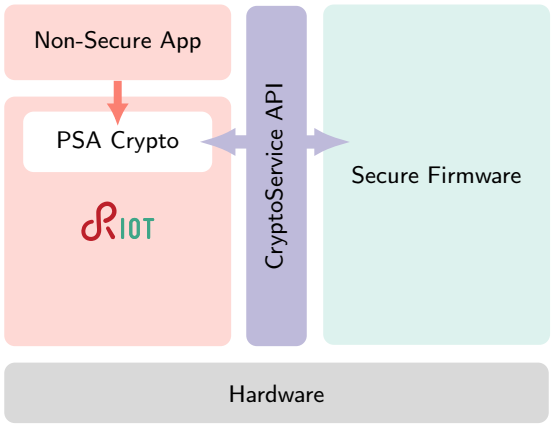

Figure 3.3: Call flow between the non-secure application, the PSA Crypto module and the secure firmware with the CryptoService API.

implements the CryptoService API described in Section 3.2 and provides all the specified crypto operations.

TrustZone-M provides a memory map-based separation of secure and non-secure memory regions. This means a secure firmware must be contained in its own binary that is executed in secure and non-secure callable flash regions. Secure functions can not be called directly from the non-secure side. To request cryptographic services from RIOT-TEE, all calls must be passed through a secure entry function.

Listing 3.3: Definitions of I/O structures and `tee_secure_entry` function.

```
typedef struct {
  const void* data;
  size_t len;
} io_pack_in_t;

typedef struct {
  void* data;
  size_t len;
} io_pack_out_t;

typedef struct {
  const int32_t operation;
  const size_t in_len;
  const size_t out_len;
} io_operation_info_t;

__attribute__((cmse_nonsecure_entry))
```

```
18 CYS_error_t tee_secure_entry(io_operation_info_t *op_info,
19                               io_pack_in_t *in,
20                               io_pack_out_t *out);
```

A secure firmware for TrustZone-M can define as many secure entry functions as needed, as long as they fit into the non-secure callable memory region. To keep the interface simple, RIOT-TEE provides only one generic non-secure entry function, through which all secure operations must be called (Listing 3.3). This function must be declared with `__attribute__((cmse_nonsecure_entry))`. When compiling and linking the firmware, the compiler will automatically generate secure gateway instructions for the functions marked with the `cmse_nonsecure_entry` attribute and place them in the non-secure callable memory region.

TrustZone-M does not support passing arguments to non-secure entry functions through the stack. This means that only a maximum of four arguments using registers R0-R3 of the Arm architecture is allowed. Since many operations require more than four arguments, they are packed into `structs` (definitions shown in Listing 3.3), which are then passed as a list to the secure entry function. For input pointers, the read-only `io_pack_in_t` structure is defined, for output pointers the writable `io_pack_out_t` structure. The `io_operation_info_t` structure contains meta information about the call, such as which operation should be executed and how many input and output structures are passed. This way the executing operation can check, if the correct number of arguments has been passed and if all the input and output addresses point to a valid memory range.

The `tee_secure_entry` function dispatches the call to an internal function interface (Listing 3.6), which can then call the underlying crypto implementation. This can be either the CryptoService Library or a platform-specific driver. The platform-specific code can directly use secure hardware features, such as hardware key slots and the crypto hardware accelerator.

Root of Trust. The RoT interface (Listing 3.4) is not a part of the CryptoService API and thus not directly callable from the non-secure side. It can only be used by the secure firmware itself. It provides the `tee_rot_try_generate_aes_key` function, which generates a new platform Advanced Encryption Standard (AES) key during start-up and writes it to a hardware key slot as the device root key. This function is a temporary workaround for this thesis. Such a key should be set by an immutable bootloader and this function should be removed in the future. In this work, this key will be used to encrpt

and decrypt key material, which will then be stored by the PSA Crypto module. Keys can be encrypted and decrypted with the `tee_rot_encrypt_key_ocb` and `tee_rot_decrypt_key_ocb` functions.

Listing 3.4: Root of Trust operations.

```
/* RoT operations */
CYS_error_t tee_rot_try_generate_aes_key(void);

CYS_error_t tee_rot_encrypt_key_ocb(uint8_t *key_in,
                                    CYS_PROT_ecc_p256_key_t *sealed_key);

CYS_error_t tee_rot_decrypt_key_ocb(CYS_PROT_ecc_p256_key_t *sealed_key,
                                    uint8_t *key_out);
```

Crypto Operations and Random Number Generation. When the non-secure side requests a cryptographic operation, the `tee_secure_entry` function dispatches the call to the corresponding internal function in the crypto module. The internal function signatures follow the pattern in Listing 3.5.

Listing 3.5: Signature of a low-level API function in RIOT-TEE.

```
CYS_error_t tee_<operation>_<algorithm>_<step>(io_pack_in_t *in,
                                               size_t in_len,
                                               io_pack_out_t *out,
                                               size_t out_len);
```

They mirror the operations defined in the CryptoService API, so they can be easily mapped during execution. They all expect pointers to the previously defined input and output structures which they will unpack and use internally.

Listing 3.6: Signatures of crypto and random methods.

```
/* Hash operations */
CYS_error_t tee_hash_sha256_setup(io_pack_in_t *in, size_t in_len,
                                  io_pack_out_t *out, size_t out_len);

CYS_error_t tee_hash_sha256_update(io_pack_in_t *in, size_t in_len,
                                   io_pack_out_t *out, size_t out_len);

CYS_error_t tee_hash_sha256_finish(io_pack_in_t *in, size_t in_len,
                                   io_pack_out_t *out, size_t out_len);

/* Cipher operations */
```

```
12 CYS_error_t tee_cipher_aes_128_cbc_encrypt(io_pack_in_t *in,
13                                             size_t in_len,
14                                             io_pack_out_t *out,
15                                             size_t out_len);
16
17 CYS_error_t tee_cipher_aes_128_cbc_decrypt(io_pack_in_t *in,
18                                             size_t in_len,
19                                             io_pack_out_t *out,
20                                             size_t out_len);
21
22 /* Protected ECC Operations */
23 CYS_error_t tee_prot_ecc_p256_generate(io_pack_in_t *in, size_t in_len,
24                                        io_pack_out_t *out, size_t out_len);
25
26 CYS_error_t tee_prot_ecc_p256_seal(io_pack_in_t *in, size_t in_len,
27                                    io_pack_out_t *out, size_t out_len);
28
29 CYS_error_t tee_prot_ecc_p256_derive(io_pack_in_t *in, size_t in_len,
30                                      io_pack_out_t *out, size_t out_len);
31
32 CYS_error_t tee_prot_ecc_p256_sign(io_pack_in_t *in, size_t in_len,
33                                    io_pack_out_t *out, size_t out_len);
34
35 /* Unprotected ECC operations */
36 CYS_error_t tee_ecc_p256_generate(io_pack_in_t *in, size_t in_len,
37                                   io_pack_out_t *out, size_t out_len);
38
39 CYS_error_t tee_ecc_p256_derive(io_pack_in_t *in, size_t in_len,
40                                 io_pack_out_t *out, size_t out_len);
41
42 CYS_error_t tee_ecc_p256_sign_hash(io_pack_in_t *in, size_t in_len,
43                                    io_pack_out_t *out, size_t out_len);
44
45 CYS_error_t tee_ecc_p256_verify_hash(io_pack_in_t *in, size_t in_len,
46                                      io_pack_out_t *out, size_t out_len);
47
48 /* Random operation */
49 CYS_error_t tee_generate_random_bytes(io_pack_in_t *in, size_t in_len,
50                                       io_pack_out_t *out, size_t out_len);
```

4 Implementation

In this chapter we describe the implementation of the secure firmware and its integration with RIOT. It starts with a description of the target platform and the required modifications to RIOT to run in non-secure mode. Then it will describe the modifications made to the PSA Crypto module to support the secure firmware as a backend. Finally, we will describe the implementation of RIOT-TEE, a secure firmware for RIOT on Arm devices with the TrustZone-M extension. We will provide example program flows to illustrate how the secure firmware is started and how it interacts with RIOT in the non-secure world.

4.1 Target Platform

The target board for this implementation is the Nordic nRF9160dk. It is less constrained compared to other microcontrollers, with 1 MB of flash memory and 256 KB of RAM and runs at a clock speed of 64 MHz. It has a Cortex-M33 CPU, which implements the Armv8.0-M architecture with the TrustZone-M security extension. An LTE modem provides networking capabilities and the board has a variety of peripherals, including an Arm CryptoCell 310 (CC310) cryptographic accelerator.

Arm CryptoCell 310. The CryptoCell 310 is a cryptographic accelerator, which provides hardware acceleration for symmetric and asymmetric cryptographic operations. It has AES, CHACHA, HASH, PKA and RNG hardware engines to support numerous algorithms, including but not limited to:

- AES 128 (ECB, CBC, CTR, CMAC/CBC-MAC, CCM)
- ChaCha20 stream cipher
- RSA

- ECC (NIST FIPS 186-4, SEC 2, Koblitz, Brainpool, Edwards/Montgomery)
- SHA-1, SHA-224, SHA-256 hashes

It therefore supports all the algorithms necessary for a CryptoServic API implementation.

The RNG engine provides a True Random Number Generator (TRNG) for entropy collection, which is compliant with the FIPS 140-2 [36], BSI AIS-31 [37] and NIST SP 800-90B [38] standards. Additionally, it provides a NIST SP 800-90A [39] compliant AES-based Cryptographically Secure Pseudo Random Number Generator (CSPRNG), consisting of three Deterministic Random Bit Generator (DRBG)s. A Direct Memory Access (DMA) engine can be used to transfer data between the CryptoCell and the main memory without involving the CPU.

A Key Management Unit (KMU) provides hardware key slots for up to 128 session keys, which can be pushed to the AES and CHACHA engines over a secure bus without granting key access to the CPU.

Additionally, the platform supports two types of hardware unique keys. The first one is the RTL key (K_{PRTL}), which is hard-coded into the device during production and can not be changed. This key value is the same for all devices with the same part code and can be used for cryptographic operations without a bootloader or application getting access to the value. This key being the same across multiple devices is a potential security risk. If an attacker obtains this key, they can compromise all other devices of the same type. Therefore, this work will refrain from using it.

The second type of hardware unique key is the device root key (K_{DK}), which is a 128 bit AES key and should be set by an immutable bootloader during the boot sequence, to establish a chain of trust. This means it can be unique for each device and rotated regularly. Since the software developed for this thesis does not use a bootloader, this key will be set by the secure firmware itself during start-up. It will then be used to encrypt and decrypt keys for the PSA Crypto module.

An open-source low-level driver for CryptoCell accelerators is available in the Trusted Firmware-M project. This firmware includes that driver code as an external dependency, to be able to use hardware acceleration and to access the hardware key slots.

4.2 Required Modifications in RIOT

Per default, RIOT is the only operating system running on a device and therefore runs in secure mode. Running RIOT as a non-secure application on the nRF9160dk, required changing the access to memory addresses and peripherals from secure to non-secure. As a first step, all secure register accesses in the CPU and board specific code had to be changed to their non-secure counterparts whenever RIOT runs in non-secure mode (see Listing 4.1).

Listing 4.1: Conditional secure/non-secure register access on the nrf9160.

```
#ifdef BOARD_NRF9160DK_NS
#define NRF_P0 NRF_P0_NS
#else
#define NRF_P0 NRF_P0_S
#endif
```

When flashing the operating system to the device, it must be written to a flash address that will be configured as non-secure during runtime. To ensure this, an offset was added to the RIOT ROM start address. Similarly, the available non-secure RAM region was defined in the `cpu/nrf9160/Makefile.include` file (Listing 4.2). These numbers are hardcoded and must be congruent with the addresses and sizes defined in the secure firmware.

Listing 4.2: Conditional secure/non-secure register access on the nrf9160.

```
ifneq (,$(filter nrf9160dk-ns,$(BOARD)))

export SECURE_FLASH_SIZE = 0x20000

RAM_LEN = 0x2a000
ROM_LEN = 0xF0000
RAM_START_ADDR ?= 0x20016000
ROM_OFFSET = $(SECURE_FLASH_SIZE)
endif
```

Both the secure firmware and RIOT are built as separate binaries. The RIOT build system can only flash one binary at a time, which means we need to merge both binaries into one. For this purpose, a new flash target called `tee-flash` was added to RIOT, which needs to be called when RIOT is built as a non-secure image. When building for this flash target, the secure and non-secure binaries are merged and converted to a hex

file. The merged hex file is then flashed to the device at the ROM start address. In the future this flash target can be extended to also flash a bootloader before the secure firmware.

4.2.1 Threading

RIOT runs two threads, a `main` thread and an `idle` thread. Those are created and started during kernel initialization. New threads are created with an initial exception return value, which describes the required system state to run the thread and will be used later to restore the correct thread context after an exception has been handled.

RIOT usually runs in secure mode, which is why the default exception return value set by RIOT starts the thread in a secure context and with the secure main stack pointer. When RIOT runs in non-secure mode, the exception return value must be changed to start the thread in non-secure mode and with a non-secure stack pointer. This is done in `cpu/cortexm_common/thread_arch.c` (see Listing 4.3).

Listing 4.3: Default exception return values for new threads.

```
#ifdef BOARD_NRF9160DK_NS
#define EXCEPT_RET_TASK_MODE        (0xffffffbc)
#else
#define EXCEPT_RET_TASK_MODE        (0xfffffffd)
#endif
```

4.2.2 PSA Crypto Integration

The CryptoService API supports the encryption of cryptographic keys to securely store them in memory. For this it introduces the `CYS_PROT_ecc_p256_key_t` type (described in Section 3.2). To use this type with the PSA Crypto module, it needs to be supported by the PSA key slot management module. The key slot management already supports storing three different types of keys: plain keys, asymmetric key pairs and hardware protected keys. We added a new slot type, which can store a sealed key with attributes and, optionally, a corresponding public key. (Listing 4.4).

Listing 4.4: PSA sealed key slot structure.

```
typedef struct {
    clist_node_t node;
```

```
    size_t lock_count;
    psa_key_attributes_t attr;
    struct sealed_key_data {
        CYS_PROT_ecc_p256_key_t sealed_key;
        uint8_t pubkey_data[PSA_EXPORT_PUBLIC_KEY_MAX_SIZE];
        size_t pubkey_data_len;
    } key;
} psa_sealed_key_slot_t;
```

Describing a Sealed Key. When generating or importing a new key, it must be specified that the key should be sealed. PSA Crypto already defines a number of attributes to describe keys, which can be extended by implementations. The best option to describe that a key should be sealed, is the key lifetime attribute, which is used to specify how a key is stored.

The `psa_key_lifetime_t` type is a compound type consisting of the `psa_key_location_t` and the `psa_key_persistence_t` types. The `psa_key_location_t` type is used to describe where a key is stored, *e.g.,* `PSA_KEY_LOCATION_LOCAL_STORAGE` or `PSA_KEY_LOCATION_PRIMARY_SECURE_ELEMENT`. The `psa_key_persistence_t` type is used to describe how long a key should be stored, *e.g.,* `PSA_KEY_PERSISTENCE_VOLATILE`, `PSA_KEY_PERSISTENCE_READ_ONLY`. Location and persistence values can be extended by vendors to describe additional storage locations and persistence levels.

Table 4.1: PSA Crypto key slot location values.

Value Range	Usage
`0x0` to `0x7fffff`	Specification-defined locations
`0x800000` to `0xffffff`	Implementation-defined locations
`0x800000` to `0x8000ff`	Secure element locations
`0x800100` to `0x8001ff`	Sealed key locations

The PSA Crypto specification reserves values from `0x0` to `0x7fffff` for specification-defined locations and `0x800000` to `0xffffff` for implementation-defined locations (overview shown in Table 4.1). `0x800000` up to `0x8000ff` are already used by RIOT for secure element locations, so this implementation defines `0x800100` to `0x8001ff` as sealed key locations, with `0x800100` as the default `PSA_KEY_LOCATION_LOCAL_SEALED`. This allows for storing sealed keys in local memory.

The key location only needs to be set once when the key is generated or imported, and will be stored within the key attributes. After key creation, the user does not need to specify the location again and can select the key by passing a key identifier to the PSA Crypto functions, just like with every other type of key.

Listing 4.5: PSA low-level driver glue code for CryptoService API.

```
psa_status_t psa_generate_ecc_p256r1_key_pair(
                                    const psa_key_attributes_t *attributes,
                                    uint8_t *priv_key_buffer,
                                    uint8_t *pub_key_buffer,
                                    size_t *priv_key_buffer_length,
                                    size_t *pub_key_buffer_length)
{
    psa_status_t status;
    psa_key_location_t location =
                    PSA_KEY_LIFETIME_GET_LOCATION(attributes->lifetime);

    switch(location) {
      case PSA_KEY_LOCATION_LOCAL_STORAGE:
        status = CYS_ecc_p256_generate(priv_key_buffer, pub_key_buffer);
        break;
      case PSA_KEY_LOCATION_LOCAL_SEALED:
        status = CYS_PROT_ecc_p256_generate((CYS_PROT_ecc_p256_key_t *)
                                            priv_key_buffer,
                                            pub_key_buffer);
        break;
      default:
        return PSA_ERROR_NOT_SUPPORTED;
    }

    if (status != PSA_SUCCESS) {
      return status;
    }

    *pub_key_buffer_length = CYS_ECC_P256_PUB_SIZE;

    (void) priv_key_buffer_length;
    return PSA_SUCCESS;
}
```

CryptoService API as a PSA Crypto Backend. The CryptoService library is included in RIOT as an external package, that is downloaded, compiled and linked by the RIOT build system. To use it as a cryptographic backend for PSA Crypto, glue

code needs to be added. The wrapper function will be called by the PSA Crypto module and then invoke the corresponding secure firmware function. An internal switch case dispatches the call depending on whether the key is sealed or not (Listing 4.5).

Since RIOT supports threading, it is possible that multiple threads try to access the secure firmware. To prevent this, it needs to be protected with a mutex. For this, three wrapper functions for initializating, claiming and releasing a mutex were added to the CryptoService API. Those are implemented with calls to the RIOT mutex implementation (Listing 4.6).

Listing 4.6: Mutex wrapper implementations in RIOT.

```
#include "mutex.h"
#include "CYS/os_mutex.h"

mutex_t cryptoservice_mutex;

void os_init_mutex(void)
{
    mutex_init(&cryptoservice_mutex);
}

CYS_error_t os_get_mutex(void)
{
    return mutex_trylock(&cryptoservice_mutex) ?
                         CYS_SUCCESS : CYS_ERROR_BAD_STATE;
}

void os_release_mutex(void)
{
    mutex_unlock(&cryptoservice_mutex);
}
```

Listing 4.7: CYS function claiming the secure firmware mutex.

```
CYS_error_t CYS_hash_sha256_init(CYS_hash_sha256_ctx_t *ctx)
{
    io_pack_out_t out[1] = {
        { .data = ctx, .len = sizeof(CYS_hash_sha256_ctx_t) }
    };

    io_operation_info_t info = {
        .operation = TEE_HASH_SHA256_SETUP,
        .in_len = 0,
```

```
10          .out_len = sizeof(out)/sizeof(io_pack_out_t)
11      };
12
13      while (os_get_mutex() != CYS_SUCCESS) {};
14      CYS_error_t status = tee_secure_entry(&info, NULL, out);
15      os_release_mutex();
16
17      return status;
18  }
```

This way it is possible to use the RIOT mutex implementation from within the CryptoService API implementation. The CryptoService API will call the `os_get_mutex()` and `os_release_mutex()` functions before and after each call to the secure services (Listing 4.7, lines 13 and 15).

Compile Time Configuration. To use the CryptoService as a backend for PSA Crypto, the corresponding module must be selected explicitly when configuring the PSA Crypto module in the application makefile. And example of this is shown in Listing 4.8.

Listing 4.8: PSA Crypto compile time configuration.

```
1 USEMODULE += psa_crypto
2 USEMODULE += psa_asymmetric
3 USEMODULE += psa_asymmetric_ecc_p256r1
4 USEMODULE += psa_asymmetric_ecc_p256r1_custom_backend
5 USEMODULE += psa_asymmetric_ecc_p256r1_backend_cryptoservice
```

When `psa_asymmetric_ecc_p256r1_backend_cryptoservice` is enabled, the PSA Crypto module will add the CryptoService package to the dependencies and use it to execute ECC operations. The actual secure firmware implementation behind the CryptoService API depends on the platform and will be selected automatically at compile time. If, for example, the application is built for the nRF9160dk, RIOT-TEE will be included as a package.

4.3 Secure Firmware Implementations

This work comprises two implementation variants for the RIOT-TEE firmware for Armv8-M devices with the TrustZone-M security extension. Both implement the CryptoService

API described in Section 3.2. Figures 4.1 and 4.2 show a comparison of the two implementations. The first variant (Figure 4.1) uses the default software implementation supplied by the cryptoservice library. On the non-secure side the CryptoService API serves as an interface between the PSA Crypto module and the secure firmware. On the secure side the secure entry function manages the transition between the two domains and dispatches operations to the internal RIOT-TEE APIs. Calls to cryptographic functions are passed to CYS library functions. Only the key encryption and decryption functions are modified to use an AES key stored in a key slot on the device. For random number generation, a software CSPRNG seeded with entropy collected from an SRAM PUF is used.

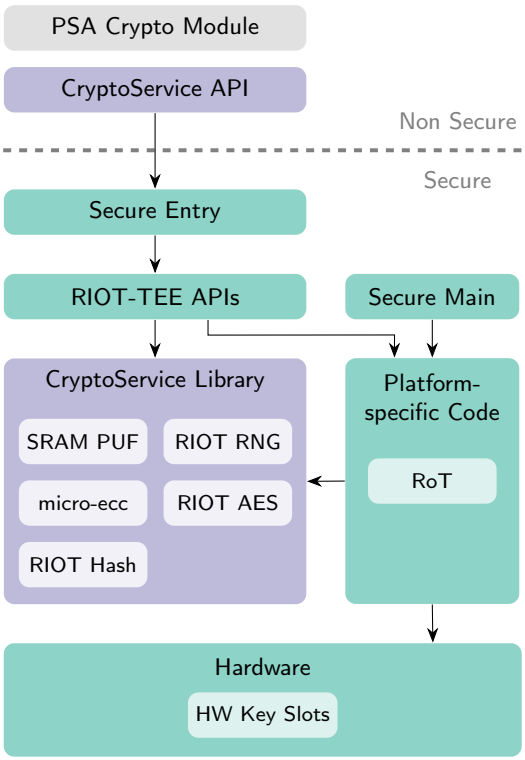

Figure 4.1: RIOT-TEE secure firmware with the CryptoService API and library. Most code from CryptoService is reused.

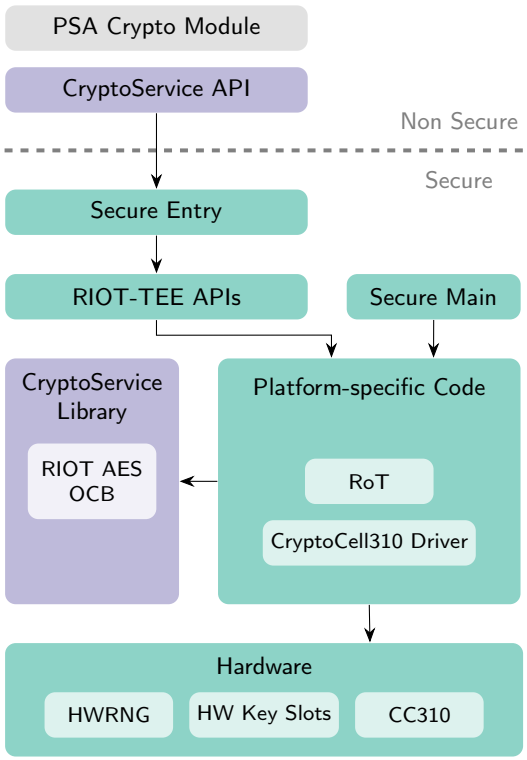

Figure 4.2: RIOT-TEE with most CryptoService modules replaced by platform specific code (*e.g.,* hardware drivers).

The second variant (Figure 4.2) replaces almost all software implementations with driver code for the crypto accelerator of the target platform as well as code to access hardware key slots. The CryptoCell 310 hardware random number generator is used to seed a deterministic random bit generator during start-up. Keys are generated, and crypto-

graphic operations are performed by the CryptoCell 310 accelerator. For key encryption and decryption, the device root key is used. This key can be pushed directly to the AES engine over a secure bus of the CryptoCell 310. Since keys are encrypted with AES OCB and CryptoCell 310 does not support this mode, the implementation reuses the OCB implementation from the CryptoService library, but replaces the AES block function with the CC310 hardware driver.

The purpose of these different variants is to demonstrate the flexibility of the software design and to show how it can be adapted to different hardware platforms. The pure software implementation is also used for evaluation and comparing results to a RIOT-only version in Chapter 5. It can also be used to compare this TEE implementation to other implementations, which use the same software libraries. The implementation using the hardware driver is used as a Proof of Concept (PoC) to demonstrate how the CryptoService API can be implemented with platform specific code.

4.3.1 System Configuration during Start-Up

The secure firmware has its own main function, which is run after the secure firmware has been booted. This function is responsible for configuring secure and non-secure memory regions, non-secure callable regions, peripherals, as well as starting the non-secure firmware. Additionally, some security related configurations are required by the *Secure Software Guidelines for Armv8-M* [40]. Figure 4.3 shows the program flow of the secure firmware start-up process. The following sections describe the configuration steps in the order of execution.

Configuration of Memory Regions ❶. Flash is separated into 32 regions of 32 KB size, which are configured as secure per default. To allow an application or operating system to run, parts of the flash and RAM need to be configured as non-secure before the non-secure image can be started. The first four regions keep the secure configuration, to fit the secure firmware and the non-secure callable region. The rest will be configured as non-secure by setting a permission bit for each region. Similarly, RAM is separated into 32 regions of 8 KB size, of which the first eleven blocks keep the secure configuration. The number of secure and non-secure regions depends on the size requirements of the secure and non-secure binaries and can be adjusted accordingly. Regions can only be configured as a whole.

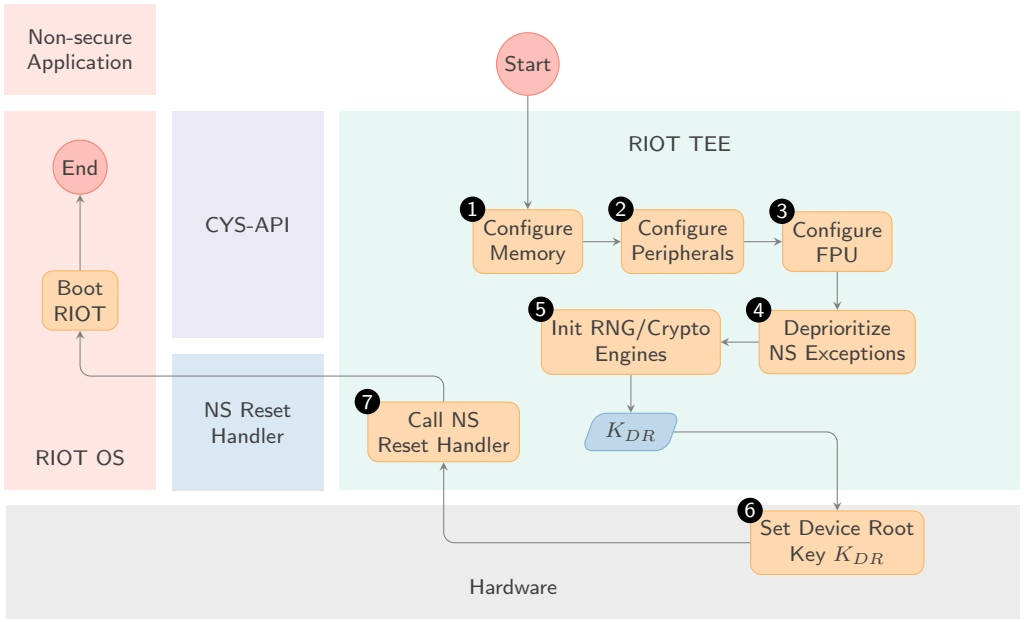

Figure 4.3: Program flow of the secure firmware start-up process.

The non-secure callable regions are placed end of the last of the four secure flash regions. It is possible to configure two NSC regions, but for our purposes we only need one.

According to the *Secure Software Guidelines for Armv8-M* [40], there is a risk of inadvertently placing a secure gateway instruction in the non-secure callable region. This can happen if there is uninitialized memory, or if general data, such as jump tables, is placed in the region and certain bit patterns are interpreted as secure gateway calls. To reduce the risk of such an occurrence, the non-secure callable region should be as small as possible and only contain secure gateway veneers. The smallest possible non-secure callable size supported by Nordic platforms is 32B, which is enough space to fit the secure entry function in this implementation(Listing 4.9).

Listing 4.9: Configuration of non-secure callable regions in secure main function.

```
1 int flash_nsc_id = 0;
2 int flash_region = 3;
3
4 NRF_SPU_S->FLASHNSC[flash_nsc_id].REGION = flash_region;
5 NRF_SPU_S->FLASHNSC[flash_nsc_id].SIZE = NRF_SPU_NSC_SIZE_32B;
```

Configuration of Peripherals for Non-Secure Use ❷. To make sure RIOT has access to the peripherals it needs, the secure firmware must configure the System Pro-

tection Unit (SPU) to allow non-secure access to them. In this case we set non-secure permissions for the GPIO, UART and TIMER peripherals as shown in Listing 4.10. Others can be added as needed.

Listing 4.10: Configuration of non-secure peripherals.

```
/* Configure GPIO P0 for NS access */
NRF_SPU_S->PERIPHID[NRFX_PERIPHERAL_ID_GET(NRF_P0_NS)].PERM &= \
                                          ~(SPU_FLASHREGION_PERM_SECATTR_Msk);

/* Configure UARTE0 for NS access */
NRF_SPU_S->PERIPHID[NRFX_PERIPHERAL_ID_GET(NRF_UARTE0_NS)].PERM &= \
                                          ~(SPU_FLASHREGION_PERM_SECATTR_Msk);

/* Configure TIMER0 for NS access */
NRF_SPU_S->PERIPHID[NRFX_PERIPHERAL_ID_GET(NRF_TIMER0_NS)].PERM &= \
                                          ~(SPU_FLASHREGION_PERM_SECATTR_Msk);

/* Configure TIMER1 for NS access */
NRF_SPU_S->PERIPHID[NRFX_PERIPHERAL_ID_GET(NRF_TIMER1_NS)].PERM &= \
                                          ~(SPU_FLASHREGION_PERM_SECATTR_Msk);

/* Set GPIO P0 pin attributes to 0 (= non-secure) */
NRF_SPU_S->GPIOPORT[0].PERM = 0x00000000ul;
```

Clearing Floating Point Registers ❸. The Armv8-M architecture allows for some optimizations to decrease interrupt latency when using floating point registers. It is possible to configure the processor to not save and clear floating point registers after handling a secure exception. According to the *Secure Software Guidelines for Armv8-M* [40], this can cause information leakage, if the secure software uses floating point registers for data storage. It is therefore recommended to always clear the registers after handling a secure exception. This is done by setting the `FPCCR` register as shown in Listing 4.11.

Listing 4.11: Configuration of floating point registers and exception priorities.

```
/* Make sure floating point registers are cleared
   when returning to non-secure world */
FPU->FPCCR |= FPU_FPCCR_TS_Msk | \
              FPU_FPCCR_CLRONRET_Msk |
              FPU_FPCCR_CLRONRETS_Msk;

/* Raise NS exception priority to 0x80 to prevent
```

```
8     preemption of secure fault exceptions */
9  SCB->AIRCR |= SCB_AIRCR_PRIS_Msk;
```

Preventing Preemption of Secure Fault Exceptions ❹. Another weakness according to *Secure Software Guidelines for Armv8-M* [40] is that the secure stack could be corrupted, if a secure fault exception is preempted by a non-secure exception, which then triggers another operation on the secure context associated with the fault. To prevent this, it is recommended to ensure that all secure fault exceptions have a higher priority than non-secure exceptions. This is achieved by setting the *Prioritize Secure exceptions (PRIS)* bit in the *Application Interrupt and Reset Control Register (AIRCR)* as shown in line 9 in Listing 4.11. An alternative would be to disable all faults except for the HardFault on the secure side. This way all secure exceptions would escalate to a HardFault, which always has the highest priority and cannot be preempted.

Initialization of RNG and Crypto Modules ❺. A cryptographically secure random number generator is required for a number of cryptographic operations, such as key generation and signatures. The target platform provides a hardware random number generator, which uses a ring oscillator to generate entropy. Generating random numbers in hardware can be slow and consumes a lot of energy [41], which is why it is more efficient to use a software pseudo-random number generator seeded with entropy from a hardware random number generator. This way the hardware generator only has to be used for the initial entropy generation and, optionally, for occasional reseeding.

This implementation uses an HMAC deterministic random bit generator (DRBG) from the CryptoCell 310 as a hardware random number generator. The initialization of this DRBG with an initial seed happens during system setup in the main function by calling the `tee_init_random` function as shown in Listing 4.12. After that the low-level driver of the CryptoCell hardware accelerator needs to be initialized. To reduce power consumption, the CryptoCell reference manual recommends disabling the peripheral when it is not used and only enable it if needed. This is done by setting and clearing the `ENABLE` bit in the `NRF_CRYPTOCELL` register before and after each operation.

After initializing the RNG and CryptoCell, the device root key is set ❻. Usually this should be done by an immutable bootloader. As a temporary workaround, this is done by the secure firmware during start-up. The key can be set only once per reset, so we check if it already exists before generating a new one.

Listing 4.12: Initialization of RNG, CryptoCell and device root key.

```
/* Initialize the random number generator */
CYS_error_t status = tee_init_random();
if (status != CYS_SUCCESS) {
    return -1;
}

/* Initialize the CryptoCell */
NRF_CRYPTOCELL->ENABLE = 1;
cc3xx_lowlevel_init();
NRF_CRYPTOCELL->ENABLE = 0;

/* Generate the device root key */
status = tee_rot_try_generate_aes_key();
if (status != CYS_SUCCESS && status != CYS_ERROR_ALREADY_EXISTS) {
    return -1;
}
```

Loading and Starting the Non-Secure Image ❼. Just as an entry function from the non-secure side to the secure side, the entry from the secure side to the non-secure side must also be declared with a specific attribute (`cmse_nonsecure_call`, shown in line 2 in Listing 4.13). The secure firmware then writes the address to the non-secure vector table to the non-secure vector table offset register (VTOR) and sets the non-secure main stack pointer to the beginning of the non-secure stack. Afterwards it loads the address of the non-secure reset handler and calls it to trigger the transition to the non-secure side. After the transition, RIOT will start executing in non-secure mode. It can now access the secure firmware through the CryptoService API for cryptographic services.

Listing 4.13: Transition to non-secure side.

```
/* Define the function pointer type for the non-secure reset handler */
typedef int __attribute__((cmse_nonsecure_call)) nsfunc(void);

SCB_NS->VTOR = TZ_START_NS; /* TZ_START_NS = 0x20000*/
uint32_t* vtor = (uint32_t*)TZ_START_NS;

/* Set the non-secure main stack pointer */
__TZ_set_MSP_NS(vtor[0]);

/* Load and call the non-secure reset handler */
nsfunc *ns_reset_handler = (nsfunc*)(vtor[1]);
ns_reset_handler();
```

4.3.2 Calling a Secure Function from the Non-Secure Side

During system operation, the non-secure side can call secure functions through the CryptoService API. The following sections describe the implementation of the firmware with the help of an example program flow for cipher encryption (Figure 4.4).

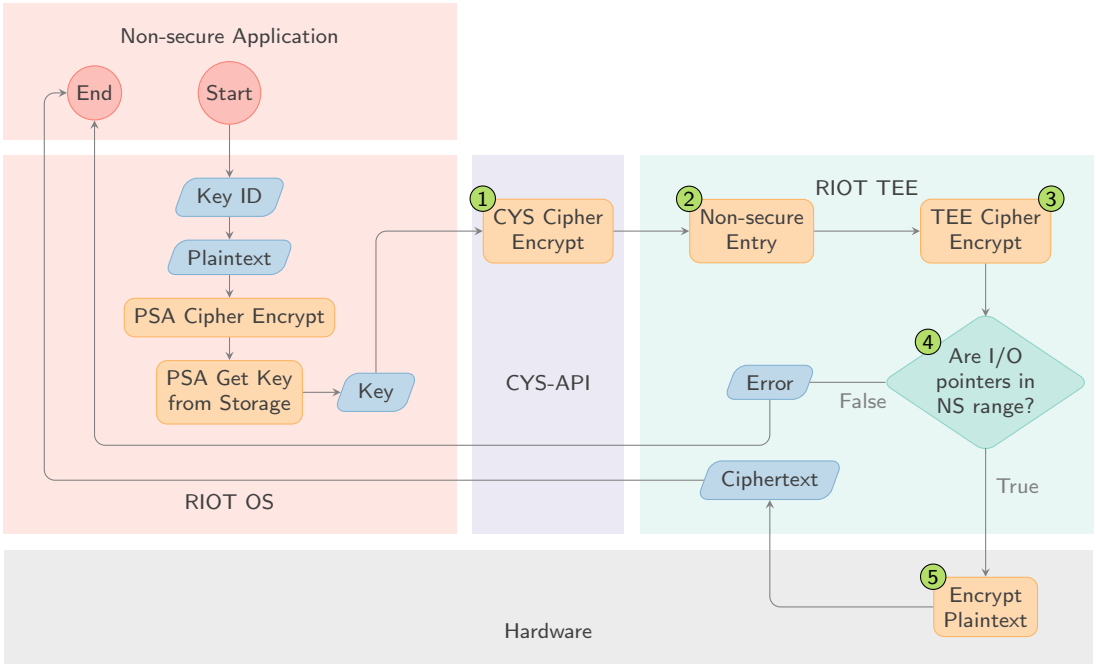

Figure 4.4: Example of a program flow for encrypting a plaintext with a plain key.

CryptoService API (1). When a cryptographic operation is called from the non-secure side, the PSA Crypto module will call the corresponding function in the CryptoService API. A representative implementation example of such a function is shown in Listing 4.14. Here, the `CYS_aes_128_cbc_encrypt` operation needs to pass five arguments to the secure entry function. The input pointers (`key, nonce, message`) and their lengths are packed into read-only `io_pack_in_t` structures, while the output pointer (`ciphertext`) is packed into a writable `io_pack_out_t` structure. Those structs are then stored in two arrays (input and output). The pointers to those arrays are passed to the secure entry function. The `io_operation_info_t` structure contains information about which operation should be executed (in this example `TEE_CIPHER_AES_128_CBC_ENCRYPT`) and how many input and output structures are passed.

Listing 4.14: Example of a call to `tee_secure_entry` and I/O parameter packing.

```
1 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)
{
    io_pack_in_t in[3] = {
        { .data = key, .len = CYS_AES_128_KEY_SIZE },
        { .data = nonce, .len = CYS_AES_128_NONCE_SIZE },
        { .data = message, .len = message_len }
    };

    io_pack_out_t out[1] = {
        { .data = ciphertext, .len = message_len }
    };

    io_operation_info_t info = {
        .operation = TEE_CIPHER_AES_128_CBC_ENCRYPT,
        .in_len = sizeof(in)/sizeof(io_pack_in_t),
        .out_len = sizeof(out)/sizeof(io_pack_out_t),
    };

    while (os_get_mutex() != CYS_SUCCESS) {};
    CYS_error_t status = tee_secure_entry(&info, in, out);
    os_release_mutex();

    return status;
}
```

Secure Entry Function (2). All calls to the secure firmware are made through the secure entry function shown in Listing 4.15. The operation that needs to be called is encoded in the `io_operation_info_t` struct that is passed as a parameter to the secure entry function. The corresponding function is then looked up in a jump table (Listing 4.16). The pointers to the input and output structures are passed on to the internal crypto API.

Listing 4.15: Implementation of the secure entry function.

```
__attribute__((cmse_nonsecure_entry))
CYS_error_t tee_secure_entry(io_operation_info_t *op_info,
                             io_pack_in_t *in,
                             io_pack_out_t *out)
{
    tee_operation_t function = tee_operation_table[op_info->operation];
    if (function == NULL) {
```

```
8         return CYS_ERROR_NOT_SUPPORTED;
9     }
10
11    return function(in, op_info->in_len, out, op_info->out_len);
12 }
```

Listing 4.16: Function type definition and jump table for dispatching secure function calls.

```
1  typedef CYS_error_t (*tee_operation_t)(io_pack_in_t *in,
2                                         size_t in_len,
3                                         io_pack_out_t *out,
4                                         size_t out_len);
5
6  static const tee_operation_t tee_operation_table[] = {
7      [TEE_RANDOM_GENERATE]               = tee_generate_random_bytes,
8      [TEE_HASH_SHA256_SETUP]             = tee_hash_sha256_setup,
9      [TEE_HASH_SHA256_UPDATE]            = tee_hash_sha256_update,
10     [TEE_HASH_SHA256_FINISH]            = tee_hash_sha256_finish,
11     [TEE_CIPHER_AES_128_CBC_ENCRYPT]    = tee_cipher_aes_128_cbc_encrypt,
12     ...,
13     [TEE_PROT_ECC_P256_DERIVE]          = tee_prot_ecc_p256_derive
14 };
```

Internal Crypto API (3). This API is implemented as a set of internal functions, which are called by the secure entry function (Listing 3.6 in Chapter 3). Listing 4.17 shows an example implementation of an internal function.

Since the non-secure side can pass pointers as input and output parameters to the secure services, the secure firmware must validate them before using them. Otherwise, the non-secure side could pass a pointer to a secure memory region and read or write data from or to it. For this we perform address range checks (4), for which the Cortex-M Security Extension (CMSE) for compilers provides a set of functions.

All input pointers and their length must be passed to the `cmse_check_address_range` function. This function checks whether the pointers start and end address are within the non-secure memory region. If the address range is valid, the function returns a pointer, otherwise it returns `NULL`. All future accesses to the pointer must be done through the returned pointer to make sure that they have been validated (example shown in Listing 4.17). After checking the pointers, they are passed to the actual crypto implementation. In Listing 4.17 the default software implementation provided by

the CryptoService library is used. Alternatively this could be replaced by a hardware driver.

Listing 4.17: Address range check of non-secure input and output pointer with the `cmse_check_address_range` function.

```
1  CYS_error_t tee_cipher_aes_128_cbc_encrypt(io_pack_in_t *in,
2                                             size_t in_len,
3                                             io_pack_out_t *out,
4                                             size_t out_len)
5  {
6      if (in_len != 3 || out_len != 1) {
7          return CYS_ERROR_INVALID_ARGUMENT;
8      }
9
10     uint8_t *key = cmse_check_address_range(
11                                 in[0].data, in[0].len, CMSE_NONSECURE);
12
13     uint8_t *iv = cmse_check_address_range(
14                                 in[1].data, in[1].len, CMSE_NONSECURE);
15
16     uint8_t *plain = cmse_check_address_range(
17                                 in[2].data, in[2].len, CMSE_NONSECURE);
18
19     uint8_t *cipher = cmse_check_address_range(
20                                 out[0].data, out[0].len, CMSE_NONSECURE);
21
22     if (plain == NULL || cipher == NULL || key == NULL || iv == NULL) {
23         return CYS_ERROR_CORRUPTION_DETECTED;
24     }
25
26     size_t plain_len = in[2].len;
27
28     return CYS_aes_128_cbc_encrypt(key, iv, plain, plain_len, cipher);
29 }
```

Using the Hardware Accelerator (5). To use the hardware accelerator for cipher encryption, a driver is needed. The open source project Trusted Firmware-M provides a low-level driver implementation for CryptoCell accelerators. Listing 4.18 shows how the first part of the function containing the address range checks remains the same. Only in line 28 the driver code is called to perform the actual encryption. First, the CryptoCell peripheral must be enabled and the AES operation is initialized. This example sets the AES engine up for an AES encryption in CBC mode (lines 29 and 30). Line 31 specifies

that the operation should use a user key that is passed as an input argument. If a key stored in protected hardware on the device were used, this could be specified here. In line 39 the output buffer is explicitly set. Afterwards, the encryption operation is executed, and the cipher text is written to the output buffer. When exiting the function, the CryptoCell peripheral is disabled again to save power (line 53).

Listing 4.18: Implementation of the `tee_cipher_aes_128_cbc_encrypt` function with the CryptoCell low level hardware driver.

```
1  CYS_error_t tee_cipher_aes_128_ecb_encrypt(io_pack_in_t *in,
2                                             size_t in_len,
3                                             io_pack_out_t *out,
4                                             size_t out_len)
5  {
6      if (in_len != 2 || out_len != 1) {
7          return CYS_ERROR_INVALID_ARGUMENT;
8      }
9
10     uint32_t *key = cmse_check_address_range((void *)in[0].data,
11                                              in[0].len,
12                                              CMSE_NONSECURE);
13     uint8_t *plain = cmse_check_address_range((void *)in[1].data,
14                                               in[1].len,
15                                               CMSE_NONSECURE);
16     uint8_t *cipher = cmse_check_address_range(out[0].data,
17                                                out[0].len,
18                                                CMSE_NONSECURE);
19
20     if (plain == NULL || cipher == NULL || key == NULL) {
21         return CYS_ERROR_CORRUPTION_DETECTED;
22     }
23
24     size_t plain_len = in[1].len;
25     size_t cipher_len = out[0].len;
26     size_t output_bytes = 0;
27
28     NRF_CRYPTOCELL->ENABLE = 1;
29     cc3xx_err_t status = cc3xx_lowlevel_aes_init(CC3XX_AES_DIRECTION_ENCRYPT
30                                                  CC3XX_AES_MODE_CBC,
31                                                  CC3XX_AES_KEY_ID_USER_KEY,
32                                                  key,
33                                                  CC3XX_AES_KEYSIZE_128,
34                                                  NULL, 0);
35     if (status != CC3XX_ERR_SUCCESS) {
```

```
36          goto exit;
37      }
38
39      cc3xx_lowlevel_aes_set_output_buffer(cipher, cipher_len);
40
41      status = cc3xx_lowlevel_aes_update(plain, plain_len);
42      if (status != CC3XX_ERR_SUCCESS) {
43          goto exit;
44      }
45
46      status = cc3xx_lowlevel_aes_finish(NULL, &output_bytes);
47      if (status != CC3XX_ERR_SUCCESS) {
48          goto exit;
49      }
50
51  exit:
52      cc3xx_lowlevel_aes_uninit();
53      NRF_CRYPTOCELL->ENABLE = 0;
54      return tee_map_error_values(status);
55  }
```

4.3.3 Using the Root of Trust to Encrypt and Decrypt Keys

When generating asymmetric keys, it is possible to encrypt the private key for secure storage on the non-secure side. To encrypt and decrypt a key with the device root key that has been set during system setup, the RIOT-TEE firmware provides a Root of Trust API. This API is for internal use only and cannot be called from the non-secure side. Figure 4.5 shows an example program flow for generating and storing a sealed key in PSA Crypto.

The application can normally call the `psa_crypto_generate_key` function from the non-secure side to generate a new key. To generate a sealed key, it needs to specify the key location as `PSA_KEY_LOCATION_LOCAL_SEALED`. Now the call will be automatically dispatched to the correct operation in the CryptoService API. Internally, the key is generated by the CryptoCell 310 hardware accelerator (1) and then passed to the Root of Trust API to encrypt it with the device root key (K_{DR}) (2). The firmware now passes the encrypted key back to the non-secure side, where PSA Crypto stored it in the internal key management module and returns the key ID to the application. The application can

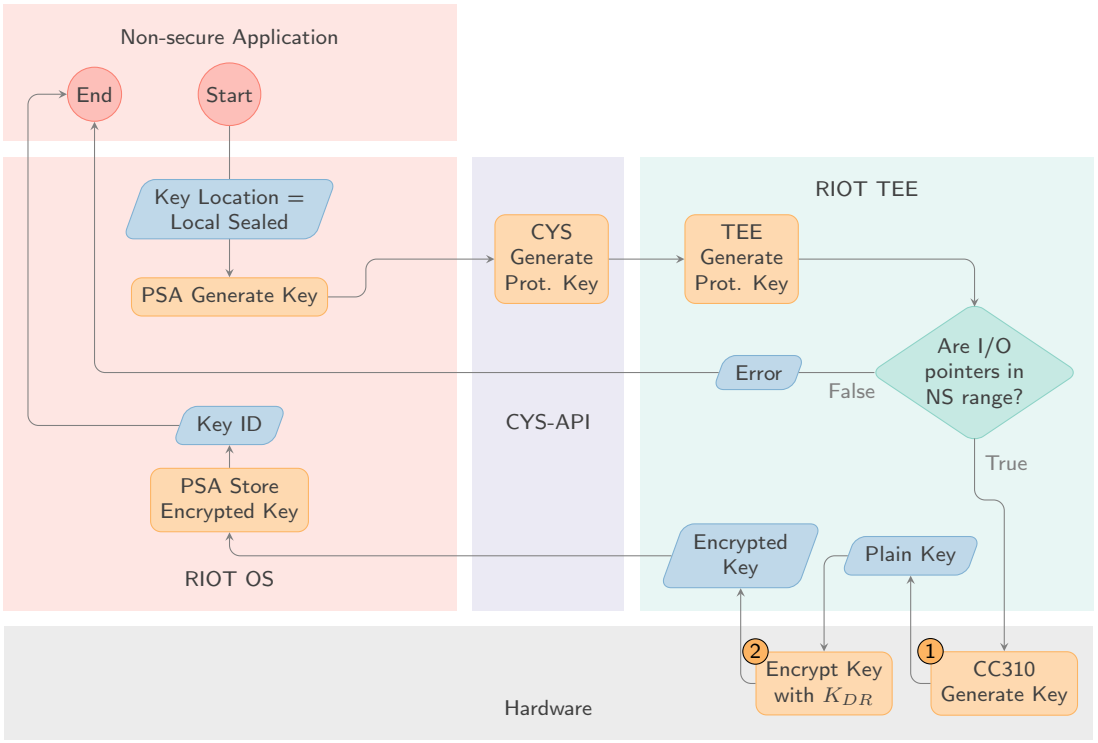

Figure 4.5: Example of a program flow for generating and storing a sealed key in PSA Crypto.

now use the encrypted key with the identifier, just like any other key stored by PSA Crypto.

Key Encryption Function. The CryptoService API stipulates the use of the AES OCB mode for key encryption. Since the CryptoCell 310 hardware accelerator does not support AES OCB mode, this implementation uses the software implementation of the OCB mode provided by the CryptoService library. This implementation provides the OCB mode operations, but allows for replacing the AES block operation with a custom one. To be able to use the device root key for encryption and decryption, the secure firmware defines its own AES block function, which uses the CryptoCell 310 hardware accelerator with direct access to the device root key. This way, when encrypting a key, the `tee_rot_encrypt_key_ocb` function will call the `cipher_encrypt_ocb` operation provided by the CryptoService library, but pass a function pointer to the custom AES block function, to perform the AES block operation in hardware, as shown in Listing 4.19.

Listing 4.19: Call of the OCB mode operation with a custom AES block operation.

```
1 /**
```

```
 * Interface to the aes cipher
 */
static const cipher_interface_t cc310_aes_interface = {
    AES_BLOCK_SIZE,
    cc310_aes_init,
    cc310_aes_encrypt_block,
    cc310_aes_decrypt_block
};

CYS_error_t tee_rot_encrypt_key_ocb(uint8_t *key_in,
                                    CYS_PROT_ecc_p256_key_t *sealed_key)
{
    cipher_t cipher = { 0 };
    cipher.interface = &cc310_aes_interface;

    int32_t result = cipher_encrypt_ocb(&cipher,
                                        NULL, 0,
                                        CYS_PROT_SEAL_TAG_SIZE,
                                        sealed_key->nonce,
                                        CYS_PROT_SEAL_NONCE_SIZE, key_in,
                                        CYS_PROT_ECC_P256_KEY_SIZE,
                                        sealed_key->private_key);

    if(result == CYS_PROT_ECC_P256_KEY_SIZE + CYS_PROT_SEAL_TAG_SIZE) {
        return CYS_SUCCESS;
    }

    return CYS_ERROR_GENERIC_ERROR;
}
```

Using the Sealed Key for a Signature. Figure 4.6 shows an example program flow of a hash generation with a sealed key. The non-secure application passes a message hash and the key ID of a sealed key stored by the PSA Crypto key management module to the PSA Crypto API. The sealed key is fetched from storage and passed to the CryptoService API, which then calls the actual secure firmware implementation. The sealed key is decrypted with the device root key and then used to sign the hash. The generated signature is returned to the non-secure application. This example shows how keys are still transparently handled by the PSA Crypto module and the secure firmware. From the application perspective, there is no difference between using a sealed key and a plain key.

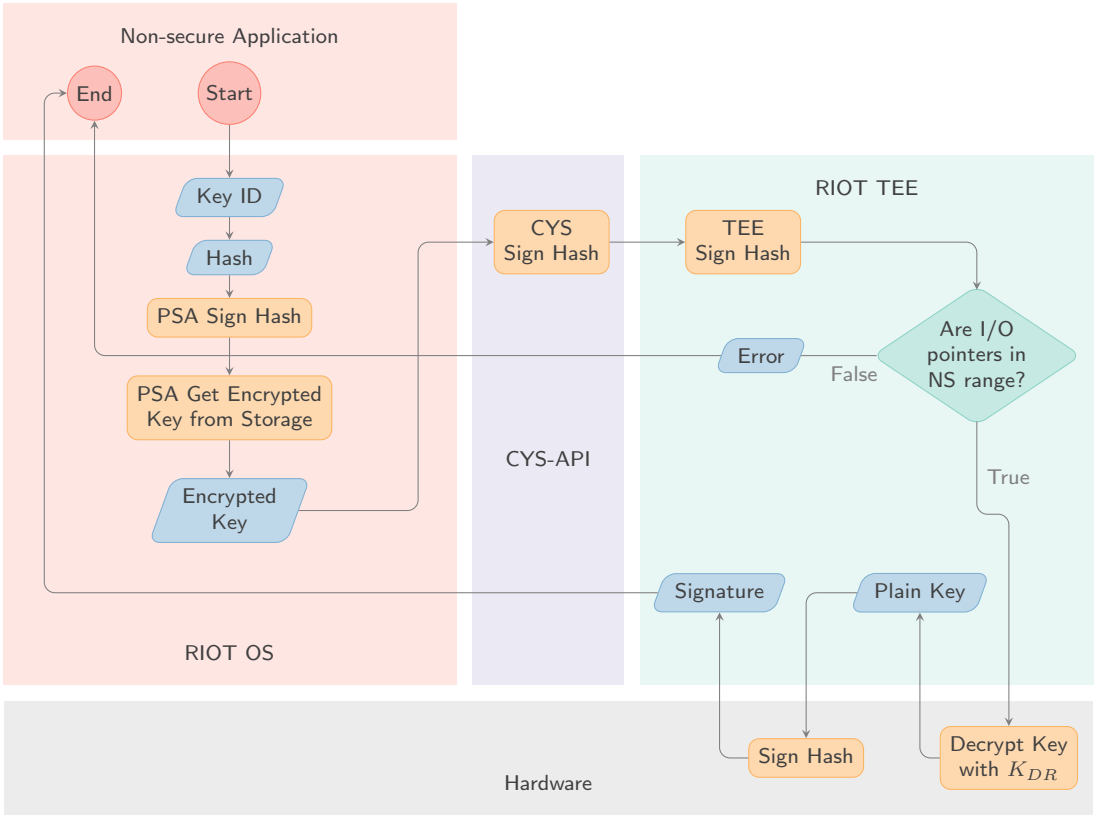

Figure 4.6: Example of a program flow for generating a hash signature with a sealed key.

5 Evaluation

In the constrained IoT, it is required that operating systems and applications have a small memory footprint and low processing overhead. It is therefore necessary to measure the impact of a secure firmware on the performance and memory usage of an application.

The following sections evaluate the measurements of RAM and ROM usage, as well as execution times of an application performing cryptographic operations with different backends.

The compared backend configurations are shown in Figure 5.1. An application runs on RIOT and calls the PSA Crypto API. PSA Crypto is built with three different configurations.

1. The first configuration builds RIOT on its own and uses software implementations that are already provided by the operating system. These are the RIOT Crypto and Hash modules, as well as the μECC library, which is included as a third-party package.

2. The second configuration builds RIOT for the non-secure world and the secure firmware the secure world. In this variant, the CryptoService library is used as a software backend. The CryptoService library uses the same software implementations as RIOT, making both versions comparable.

3. The third configuration also builds the secure firmware for the secure world, but replaces the CryptoService library with the hardware driver for the CryptoCell 310 (CC310).

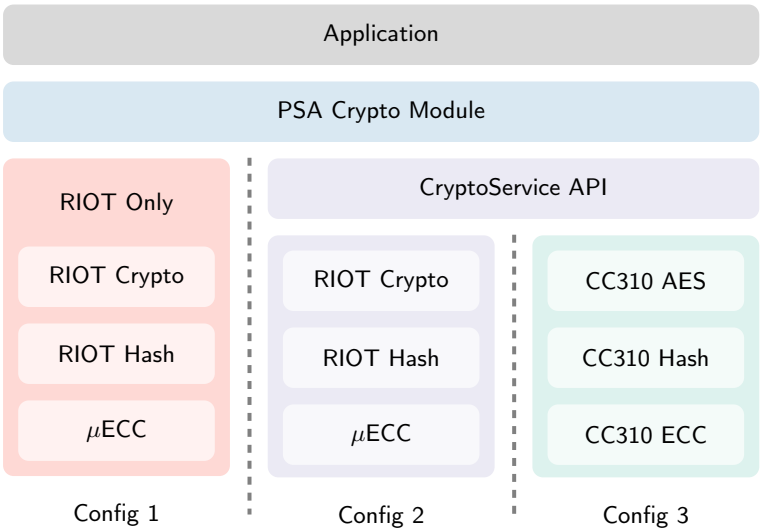

Figure 5.1: Evaluation Configuration.

5.1 Runtime Overhead

To measure performance overhead, we execute one operation for 1000 iterations. Before and after each operation we toggle a GPIO and measure the elapsed time with a logic analyzer sampling at 12 MS/s. All operations are measured from start to finish and include the overhead added by the PSA Crypto module and the firmware. The measurements include time for input checks, function dispatching, as well as the key management (storing and looking up keys) from PSA Crypto. The firmware overhead includes the transition between secure and non-secure mode, as well as the execution of glue code.

The CryptoService library included in the RIOT-TEE software variant uses the same implementation as the RIOT operating system. This makes both versions comparable, since the execution times of the actual crypto operations are the same. The difference between the RIOT only measurements and the RIOT-TEE software variants thus define the additional overhead introduced by the secure firmware.

The RIOT-TEE version with hardware drivers uses the hardware accelerator of the target platform, which has very different execution times from the software implementations and is used more as a proof of concept than a real comparison.

All applications have been compiled with the `-Os` compiler flag, optimizing for size and speed. The target CPU provides an instruction cache, which can be enabled to reduce

the number of flash accesses and wait cycles. During these measurements we discovered that caching significantly reduced the execution time of the crypto operations (*e.g.,* hash computations could be reduced to half the speed). Therefore, caching is enabled during these experiments. We measure inputs of three different sizes, to show how input length impacts execution times.

5.1.1 Hashes

We measure a multi-step SHA-256 execution. It consists of a setup part, which initializes a context with initialization values defined by the FIPS 180-4 standard, an update part which processes the current input and can be called multiple times, and a final part, which performs the last hashing step and copies the computed digest into an output buffer. We perform all these steps for inputs of sizes 32 bytes, 262 bytes and 1024 bytes.

Figure 5.2 shows the different execution times, while the exact values are listed in Table 5.1. The execution times for the software implementations are similar, showing only a small overhead caused by the secure firmware. The execution time of the update operation increases proportionally with the input size when using the software implementations, while the finish operation remains constant. The length of the input has only a small impact on the execution time of the hardware driver. For setup and update operations, the transition between secure and non-secure mode including glue code execution and input checks add up to $\approx 6\mu s$, while the finish operation adds $\approx 15\mu s$ of overhead.

The complete hash computation with a hardware driver takes $\approx 16\mu s$ longer on small inputs, but stays almost constant when increasing the input sizes.

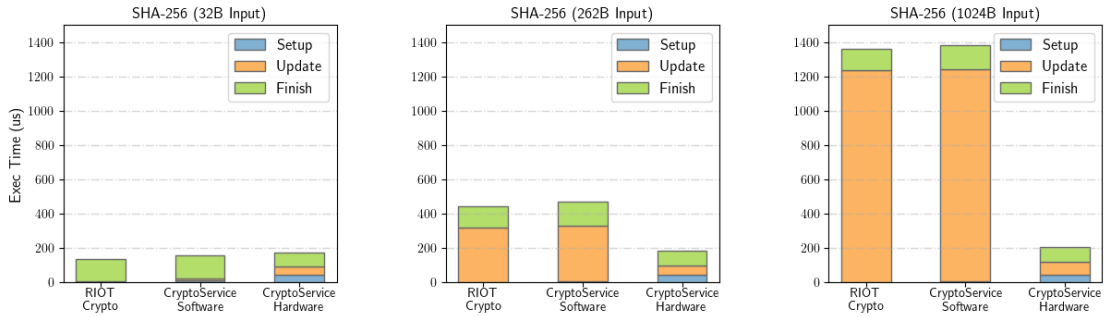

Figure 5.2: Comparison of processing times of multi-step hash computation.

Version	Operation	Execution Times [μs]		
		32 Bytes	262 Bytes	1024 Bytes
PSA + RIOT Crypto	Setup	3	3	3
	Update	5.84	316.94	1232.23
	Finish	125.00	126.43	127.05
PSA + CryptoService with RIOT Crypto	Setup	9.35	9.34	9.34
	Update	11.82	321.03	1231.14
	Finish	139.03	141.01	141.55
PSA + CryptoService with CC310 AES Engine	Setup	46.02	45.99	45.97
	Update	47.17	52.83	72.71
	Finish	83.37	83.92	85.21

Table 5.1: Execution times of SHA-256 computation in numbers.

5.1.2 Cipher

For the cipher we measure an AES operation in cipher block chaining mode with a key size of 128 bit. Here we measure single-step functions consisting of an encrypt and a decrypt operation. The encrypt function includes the generation of a random initialization vector (IV), which is included in these measurements. We encrypt and decrypt inputs of sizes 32 bytes, 256 bytes and 1024 bytes.

Figure 5.3 and Table 5.2 show the comparison between the different implementations. Again, the execution times of the software implementations increase proportionally to the input sizes. The overhead added by the secure firmware remains small.

The hardware implementation of the encrypt operation takes significantly longer than the software encryption with the 32 byte input. This is caused by the random number generator used for the IV generation. Here we use the HMAC DRBG provided by the CC310 driver, which adds $650mus$ to each encryption operation. The encryption operation itself is faster than the software implementation.

As with the hash generation, the execution times of the hardware accelerator remain almost constant for different input sizes.

5.1.3 ECDSA

We measure an ECDSA operation with a NIST-P256 curve. The operation consists of an asymmetric key pair generation, the signing of a 32 byte input hash with a private key and the verification of the hash signature with a public key. As with the hash and

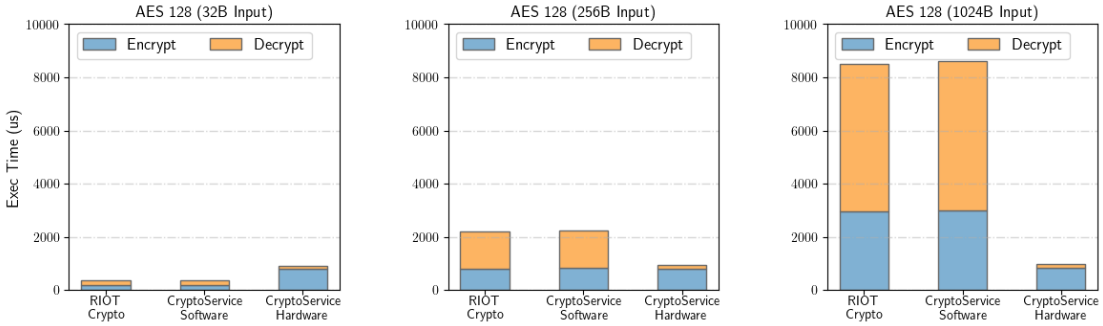

Figure 5.3: Comparison of processing times of multi-step cipher encryption and decryption.

Version	Operation	Execution Times [μs]		
		32 Bytes	256 Bytes	1024 Bytes
PSA + RIOT Hashes	Encrypt	182.41	810.94	2949.29
	Decrypt	198.35	1410.8	5565.76
PSA + CryptoService with RIOT Hashes	Encrypt	181.56	819.95	3006.65
	Decrypt	201.08	1425.82	5619.51
PSA + CryptoService with CC310 Hash Engine	Encrypt	785.01	794.38	825.34
	Decrypt	122.83	132.68	163.62

Table 5.2: Execution times of AES-128 CBC encryption and decryption.

cipher operations, the execution times of the software implementations are similar, with only a small overhead caused by the secure firmware (Figure 5.4).

The hardware driver implementation of the key generation and signature operations is slower than the software implementation. This is probably due to the Public Key Algorithms (PKA) and DRBG implementations of the CC310 driver.

Compiling the PKA driver code with `-Ofast` optimization instead of `-Os`, only lead to negligible improvements, while significantly increasing the code size. Optimizing the hardware driver further is out of scope of this thesis.

5.2 Memory Usage

To compare RAM and ROM usage, we build an example application, which performs hash, cipher and ECDSA operations, and analyze the resulting ELF files. As with the performance measurements, the code was optimized for size with the `-Os` flag.

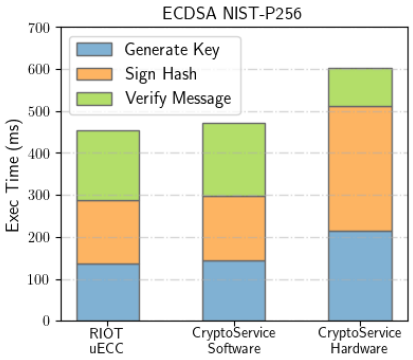

Figure 5.4: Comparison of processing times of ECDSA key generation, signature and verification.

Version	Operation	Execution Times [μs]
		32 Bytes
PSA + μECC	Key Generation	137.82
	Sign Hash	149.22
	Verify Message	167.27
PSA + CryptoService with μECC	Key Generation	143.34
	Sign Hash	155.21
	Verify Message	172.94
PSA + CryptoService with CC310 PKA Engine	Key Generation	213.69
	Sign Hash	299.70
	Verify Message	88.38

Table 5.3: Execution times of ECDSA key generation, hash signature and message verification times with a NIST P-256 curve.

As shown in Figure 5.5, the application built with only RIOT uses $\approx 26kB$ of ROM and $\approx 14kB$ of RAM. When building the application with RIOT and the secure firmware with the CryptoService software library, the RIOT binary gets smaller. This is because implementations of the cryptographic operations are not compiled as part of the RIOT binary anymore, but are now part of the secure firmware code. Combined, the firmware and CryptoService library add $\approx 14kB$ of ROM, while RAM usage remains almost the same.

Using the CC310 hardware driver as a crypto backend, adds another $\approx 15kB$ of ROM usage, mostly due to the code size of the PKA sources. The driver also uses $\approx 2.5kB$ of RAM.

5.3 Interpretation of Results

The measurements of runtime overhead show, that the secure firmware has a constant, but insignificant impact on the execution time of cryptographic operations. When performing symmetric encryption/decryption and hash operations in software, the size of the input has a significant impact on the processing time. Using the hardware accelerator for these operations reduces the processing time significantly.

These results do not translate to the ECDSA operations. Here, the key generation and signature generation take significantly longer in hardware. Only the verification operation is faster with the hardware accelerator.

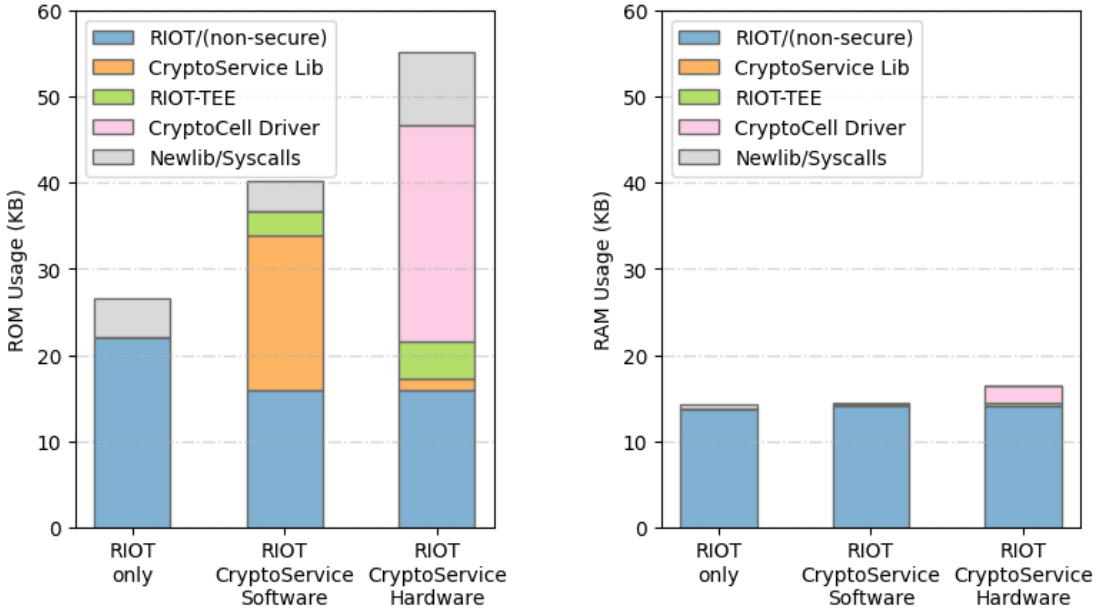

Figure 5.5: Comparison of Flash and RAM usage of the example application with different secure firmware variants.

Regarding the usage of flash and RAM, the results are mixed. The secure firmware variant with the software implementation does not increase the RAM usage. When running the variant with the hardware accelerator, the CryptoCell driver increases RAM usage by merely $\approx 1kB$, which is negligible. Both secure firmware variant have a noticeable impact on the flash usage. The software variant adds $\approx 14kB$, the CryptoCell driver in the hardware variant even doubles the size of the application binary.

Most platforms with TrustZone-M support provide sufficient flash and RAM for these configurations. The biggest platforms (*e.g.,* Nordic nRF91XX, nRF5340, Nuvoton M2354) provide 1 MB of flash memory and at least 256 KB of RAM. Medium platforms such as the Microchip SAML11, STM32L5, STM32U5 and Nuvoton M2351 range from 256 KB to 640 KB of flash with a minimum of 64 KB RAM. The lower end of the STM32H5 series provided the smallest amount, with 128 KB of flash memory and 32 KB RAM (larger options are available). Most of these MCUs will be able to run a secure firmware, even with added features on the non-secure side, such as a large network stack. It can therefore be concluded that, while this secure firmware uses additional resources, it is still feasible to run it, especially considering the security benefits it provides.

6 Conclusion and Outlook

IoT devices with constraints in memory, battery and processing power are vulnerable to a range of attacks. One of the most common threats in IoT devices are buffer overflow attacks, which can be used to extract sensitive data or execute arbitrary code. Devices can be hardened against those attacks by implementing memory protection mechanisms, and isolating security critical data and code execution from untrusted applications or operating systems. Common microcontrollers used for IoT devices, such as RISC-V and Arm Cortex-M33 devices, provide hardware-based protection mechanisms, which can be used to implement memory isolation in constrained devices. A special firmware is needed, to leverage such technologies and provide a so called Trusted Execution Environment, in which trusted code can execute without exposing any sensitive data to unauthorized system components.

Due to the heterogenity of device architectures in the IoT, platforms require different secure firmware implementations to leverage their respective memory protection features. To make sure, RIOT remains portable, the CryptoService API was designed as a common, platform-agnostic interface for secure firmware implementations. A secure firmware can implement this API to provide cryptographic services to the operating system. This way, RIOT can request crypto operations from the interface, without knowledge about the underlying implementations. Applications can therefore be ported to other devices without requiring any platform-specific code.

The contribution of this thesis comprises two parts. First, the CryptoService API was integrated as a backend for the existing crypto subsystem in RIOT. This allows applications to use the regular PSA Crypto interface provided by the operating system, while the actual execution is transparently dispatched to a secure firmware running in protected memory.

In the second step we implemented a secure firmware for Arm Cortex-M microcontrollers with the TrustZone-M security extension for the IoT operating system RIOT.

This firmware implements the CryptoService API and can thus be used by the PSA Crypto module as a backend for cryptographic operations. It provides a set of common cryptographic operations and allows for encryption of asymmetric private keys with a device root key, so they can be stored securely by the PSA Crypto module.

The evaluation of the secure firmware shows that switching states to execute operations in the secure world does not increase the processing time of cryptographic operations significantly compared to the execution in a RIOT-only environment. Also, the increase in RAM usage is small. While the amount of flash memory increases significantly, running the secure firmware is still feasible on all devices with TrustZone-M support. Even on the smallest available platforms the secure and non-secure binaries combined use less than 50% of the available flash, leaving plenty of space for additional operating system modules or a bootloader. Considering the benefits of the secure firmware, such as secure key storage and memory isolation, the increase in memory usage is justified.

Outlook. At the time of writing, a secure boot process is not implemented, which means that the firmware is not protected from unauthorized modifications. The device root key that is used as a Root of Trust, is set by the firmware itself during startup. This key should be provisioned securely, *e.g.,* by an immutable bootloader, to ensure that it is not compromised. For a complete secure boot flow, it is crucial that support for a secure boot process is added to RIOT.

At the time of writing, the secure firmware supports only the nRF9160 microcontroller and only a subset of its cryptographic hardware. In future work this firmware should be extended to support other platforms with TrustZone-M support, as well as other platform features, such as hardware key storage.

New types of affordable IoT devices, such as the Raspberry Pi Pico 2, combine two Arm Cortex cores with two RISC-V Hazard3 cores. The Arm Cortex-M33 core has a TrustZone-M extension, a SHA-256 hardware accelerator and a True Random Number Generator. The RISC-V core provides a Physical Memory Protection Unit. This architecture is a good candidate for a secure firmware implementation that supports both TrustZone-M and RISC-V PMP. Future work should explore the possibilities of integrating such a platform with the secure firmware concept presented in this thesis.

Bibliography

[1] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, jan 2019. [Online]. Available: https://doi.org/10.1145/3291047

[2] O. Garcia-Morchon, S. Kumar, and M. Sethi, "Internet of Things (IoT) Security: State of the Art and Challenges," IETF, RFC 8576, April 2019. [Online]. Available: https://doi.org/10.17487/RFC8576

[3] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," IETF, RFC 7228, May 2014. [Online]. Available: https://doi.org/10.17487/RFC7228

[4] Zephyr Project, "Zephyr," https://www.zephyrproject.org, last accessed 07-17-2020, 2020.

[5] Amazon Web Services, "FreeRTOS Real-time operating system for microcontrollers," https://www.freertos.org/, last accessed 30-11-2020, 2020.

[6] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://doi.org/10.1109/JIOT.2018.2815038

[7] G. Mullen and L. Meany, "Assessment of buffer overflow based attacks on an iot operating system," in *2019 Global IoT Summit (GIoTS)*, 2019, pp. 1–6.

[8] C. Bellman and P. C. van Oorschot, "Analysis, Implications, and Challenges of an Evolving Consumer IoT Security Landscape," in *17th International Conference on Privacy, Security and Trust (PST)*. IEEE, August 2019, pp. 1–7.

[9] B. Moran, "A summary of security-enabling technologies for IoT devices," IETF, Internet-Draft – work in progress 03, October 2024. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-iotops-security-summary-03

[10] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng, "Providing Root of Trust for ARM TrustZone using On-Chip SRAM," in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, ser. TrustED '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 25–36. [Online]. Available: https://doi.org/10.1145/2666141.2666145

[11] GlobalPlatform, "TEE System Architecture v1.3." [Online]. Available: https://globalplatform.org/specs-library/tee-system-architecture/

[12] "IEEE Standard for Secure Computing Based on Trusted Execution Environment," *IEEE Std 2952-2023*, pp. 1–29, 2023.

[13] U. Lee and C. Park, "SofTEE: Software-Based Trusted Execution Environment for User Applications," *IEEE Access*, vol. 8, pp. 121 874–121 888, 2020.

[14] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-TEE – An Open Virtual Trusted Execution Environment," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 400–407.

[15] K. Zandberg and E. Baccelli, "Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules," *CoRR*, vol. abs/2106.12553, 2021. [Online]. Available: https://arxiv.org/abs/2106.12553

[16] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[17] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual volume ii: Privileged architecture version 1.9.1," Tech. Rep. UCB/EECS-2016-161, Nov 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html

[18] T. Lu, "A Survey on RISC-V Security: Hardware and Architecture," *CoRR*, vol. abs/2107.04175, 2021. [Online]. Available: https://arxiv.org/abs/2107.04175

[19] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1416–1432.

[20] T. Roth, "Trustzone-m: Hardware attacks on armv8-m security features," Chaos Computer Club e.V., 2019, https://doi.org/10.5446/53149. [Online]. Available: https://doi.org/10.5446/53149

[21] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu, and Z. Zhao, "Return-to-Non-Secure Vulnerabilitieson ARM Cortex-M TrustZone: Attack and Defense," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, Jul. 2023, pp. 1–6.

[22] C. Rodrigues, D. Oliveira, and S. Pinto, "BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 3679–3696. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00062

[23] L. Abbott and R. Altherr, "Breaking TrustZone-M: Privilege Escalation on LPC55S69." DEF CON 29, 2021. [Online]. Available: https://infocondb.org/con/def-con/def-con-29/breaking-trustzone-m-privilege-escalation-on-lpc55s69

[24] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 367–378.

[25] Linaro Limited, "OP-TEE," 2019-2023. [Online]. Available: https://optee.readthedocs.io/en/latest/general/about.html

[26] ARM Ltd., "ARM Trusted Firmware A," https://trustedfirmware-a.readthedocs.io/en/latest/, last accessed 02-12-2025, 2021.

[27] X. Yang, P. Shi, B. Tian, B. Zeng, and W. Xiao, "Trust-E: A Trusted Embedded Operating System Based on the ARM TrustZone," in *2014 IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing and 2014 IEEE 11th Intl Conf on Autonomic and Trusted Computing and 2014 IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops*, 2014, pp. 495–501.

[28] D. Oliveira, T. Gomes, and S. Pinto, "uTango: An Open-Source TEE for IoT Devices," *IEEE Access*, vol. 10, pp. 23 913–23 930, 2022.

[29] ARM Ltd., "ARM Trusted Firmware M," https://tf-m-user-guide.trustedfirmware.org, last accessed 02-12-2025, 2021.

[30] L. Boeckmann, T. C. Schmidt, and M. Wählisch, "Poster: Integrating a Secure Processing Environment in an IoT Operating System," July 2024, European Symposium on Security and Privacy (Euro S&P). [Online]. Available: https://doi.org/10.5281/zenodo.12635931

[31] ARM Ltd., "PSA Cryptography API 1.1," https://arm-software.github.io/psa-api/crypto/1.1/index.html, last accessed 04-30-2024, 2023.

[32] L. Boeckmann, P. Kietzmann, L. Lanzieri, T. C. Schmidt, and M. Wählisch, "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. [Online]. Available: https://dl.acm.org/doi/10.5555/3578948.3578956

[33] L. Boeckmann, P. Kietzmann, T. C. Schmidt, and M. Wählisch, "Poster Abstract: Offloading Crypto Processing with RIOT," in *Proc. of ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN '22), Poster Session*. Piscataway, NJ, USA: IEEE, May 2022, pp. 535–536. [Online]. Available: https://doi.org/10.1109/IPSN54338.2022.00068

[34] ARM Ltd., "ARM Platform Security Architecture," https://developer.arm.com/architectures/architecture-security-features/platform-security, last accessed 09-28-2021, 2021.

[35] ——, "Mbed TLS," https://tls.mbed.org, last accessed 02-15-2025, 2020.

[36] N. I. of Standards, T. (NIST), A. Lee, M. Smid, and S. Snouffer, "Security Requirements for Cryptographic Modules," 2001-05-25 00:05:00 2001.

[37] "Funktionalitätsklassen und Evaluationsmethodologie für physikalische Zufallszahlengeneratoren," 2013. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_pdf.pdf

[38] M. Sonmez, E. Barker, J. Kelsey, K. McKay, M. Baish, and M. Boyle, "Recommendation for the Entropy Sources Used for Random Bit Generation," 2018-01-10 2018.

[39] E. Barker, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," 2012-01-23 2012.

[40] ARM Ltd., "Secure Software Guidelines for Armv8-M," 2020. [Online]. Available: https://developer.arm.com/documentation/100720/latest/

[41] P. Kietzmann, T. C. Schmidt, and M. Wählisch, "A Guideline on Pseudorandom Number Generation (PRNG) in the IoT," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 112:1–112:38, July 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3453159

Glossary

Application Programming Interface A set of functions that define an interface between computer programs..

Arm Platform Security Architecture A framework developed by Arm for developing secure IoT systems, that defines security requirements and provides implementation guidelines..

Internet of Things The network of physical devices, vehicles, home appliances and other items embedded with electronics, software, sensors, actuators, and connectivity which enables these objects to connect and exchange data..

Root of Trust Unextractable and unchangeable secret on a device, that can be used for device authentication or attestation (*e.g.,* a device root key)..

Trusted Execution Environment An isolated environment within a SoC, in which data and code execution are protected from unauthorized access..

TrustZone A hardware-based security extension for Arm Cortex processors, that provides memory isolation and can be used as a base to build Trusted Execution Environments..

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