HAW HAMBURG

PROJECT REPORT
Lena Boeckmann

Evaluation of a Secure Processing Environment in RIOT OS

Faculty of Engineering and Computer Science
Department Computer Science

Supervision: Prof. Dr. Thomas Schmidt
Submitted: June 18, 2024

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Contents

Abstract: Trusted Execution Environments (TEE) and secure enclaves are promising concepts for increasing security in constrained environments. They provide protected processing areas within a SOC, in which security critical applications can be executed, while preventing unauthorized access to sensitive data and program code. Some Arm Cortex-M devices offer hardware support for TEE. This comes in the shape of a Memory Protection Unit (MPU) and, for the newest class of devices with Armv8-M architecture, TrustZone-M (TZ-M) technology. Arm also provides an open source reference implementation of a Secure Processing Environment (SPE). The so-called Trusted Firmware (TF-M) can be used to leverage the MPU and TZ-M capabilities and use memory isolation to implement a TEE. This report describes how we integrated TF-M with the IoT operating system RIOT OS, evaluates the overhead compared to running RIOT without TF-M and gives an outlook on future work regarding TEE in RIOT.

Keywords: TFM, Arm, TrustZone, Trusted Execution Environments, IoT Security

1 Introduction

Billions of devices connected to the Internet of Things (IoT) store, process and transmit sensitive data, while often being insecure and easily physically accessible to potential attackers. Vulnerable devices can serve as entry points to larger networks and allow bad actors to laterally move within networks to compromise critical system components and infrastructure. Especially when devices are deployed in easily accessible areas or are connected to the internet, we need ways to make those devices trustworthy.

This can be achieved by establishing a so-called Root of Trust (RoT), which is a tamper-proof and inherently trustworthy system component. A RoT can be provided by so-called Trusted Execution Environments (TEE). Those are isolated environments in which trusted applications (TA) can perform security critical operations, like secure storage of

data and cryptographic key material, cryptographic operations, device authentication and attestation and secure over-the-air (OTA) updates.

Using TEEs instead of an operating system (OS) to perform those functions has some advantages. They can provide a reduced set of operations only required to establish trust between communication partners. Therefore, they expose a smaller attack surface than a rich OS. Secondly an OS could be compromised by malware or through a physical attack. Separating the RoT from the OS provides an extra layer of security and allows for independent attestation and verification.

TEEs can be implemented in software, hardware or a combination of both. In the constrained IoT, hardware-based TEEs can help to protect devices while producing less overhead than software-based implementations (e.g. virtual machines, containers). Popular examples for TEE hardware support are the RISC-V physical memory protection (PMP), Arm Memory Protection Unit (MPU) and Arm TrustZone-M. All need specialized firmware to fully support TEEs and manage secure memory access for applications.

To facilitate adoption of those security mechanisms, Arm provides the open source project Trusted Firmware-M, which is a reference implementation for a secure processing environment (SPE). To leverage TEE technology in the IoT OS RIOT, we partly integrated TF-M into the OS as a third party package and evaluated the overhead it introduces to a simple cryptographic application.

This work focuses on the steps we took to run TF-M with RIOT. In Section 2 we will provide more information about TEEs, Arm TrustZone, TF-M, RIOT and related work. In Section 3 we will describe what is needed to integrate TF-M with an operating system and the steps we took to do so. Further we will disuss the limitations of the current integration. We will measure the overhead TF-M introduces to RIOT and show the results in Section 4. In Section 5 I will sum up this work and give an outlook to what the next steps to provide TEEs in RIOT could be.

2 Background and Related Work

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) provides an isolated environment to perform security critical operations and store and access sensitive data [1]. It ensures that data

and code within said environment can not be read or altered by applications located outside, including by an operating system [13]. This is useful for storing cryptographic key material and performing cryptographic operations to ensure confidentiality and integrity and provide authentication and attestation [16]. The IEEE Standard for Secure Computing Based on Trusted Execution Environments [1] describes a layered architecture of a TEE-based secure computing system. Here, the TEE is located within a secure computing node alongside the untrusted Rich Execution Environment (REE) (e.g. an OS). Both environments interact with each other through a secure interface. The standard defines three TEE layers: the *fundamental layer*, the *platform layer* and, optionally, an application layer.

The *fundamental layer* comprises three components. The first part is the Root of Trust (RoT) of the TEE, defined as a combination of "reliable hardware, firmware and software components that perform specific, critical security functions". It is required that the RoT is protected at all times and can not be tampered with. The second part are the hardware and software components that work together to comprise the TEE and its basic functionalities (e.g. environment isolation and measurement, memory encryption and protection, secure key generation). The third component is the interface to the platform layer, which allows for TEE creation and management.

The *platform layer* contains the actual trusted environment in which trusted applications (TA) can be run. Parts of it are a trusted runtime, storage, resource management and system service. It also contains a trusted connection as a communication channel between TAs and applications outside the TEE. It can also be used for remote attestation of TAs and the whole system.

The standard also defines requirements for TEEs. They need to provide isolation of hardware resources, TEE and REE, as well as TAs, and trusted communication between those isolated components. They also need to be interoperable, meaning they should be able to authenticate, verify themselves to others and provide a specified set of attributes. Further it is required that TEEs have low overhead, guarantee availability of correct data and its protection, as well as provide a specified set of cryptographic operations.

2.1.1 TEE Implementation Approaches

There are different approaches to implement TEEs. On the one hand there are hardware-based solutions, which utilize hardware mechanisms like memory protection and inter-

rupts. Software-based solutions rely on virtualization and containers to isolate processing environments [17, 18]. Software TEEs can be a good option in specific use cases and on devices that don't support memory separation in hardware, though they introduce large memory and runtime overhead compared to hardware-based solutions.

Some newer IoT devices implement hardware components that can be used to support TEEs. For RISC-V architectures there are multiple available options: Keystone [10] provides an open-source framework for building customized TEE. MultiZone [8] separates the system into multiple equally secure zones. Both utilize the RISC-V Physical Memory Protection (PMP) to make sure that certain memory areas can only be accessed with corresponding privilege levels [11].

Many Arm Cortex-M/R architectures support a similar mechanism called Memory Protection Unit (MPU), which also protects predefined memory areas from unprivileged read, write and execute operations. Arm MPU is not very widely supported by operating systems for various reasons, including high performance overhead and financial cost [19].

As an improvement, Arm has designed an alternative approach: TrustZone [15] operates system-wide, not only protecting specific memory areas, but separating the whole system into a secure (S) and a non-secure (NS) world. *TrustZone-A* for Cortex-A devices has been around for some time now and has recently been adapted to the smaller, more constrained Cortex-M architecture (supported by the newer Armv8-M and Armv8.1-M devices). *TrustZone-M* provides a memory-map based division between S and NS memory regions, peripherals and some specific registers, as shown in Figure 2. The processor can operate in two states, also secure and non-secure. Software executed in NS state is blocked from accessing resources marked as secure, protecting them from unauthorized access. Some memory areas, like secure and non-secure SRAM are completely separated.

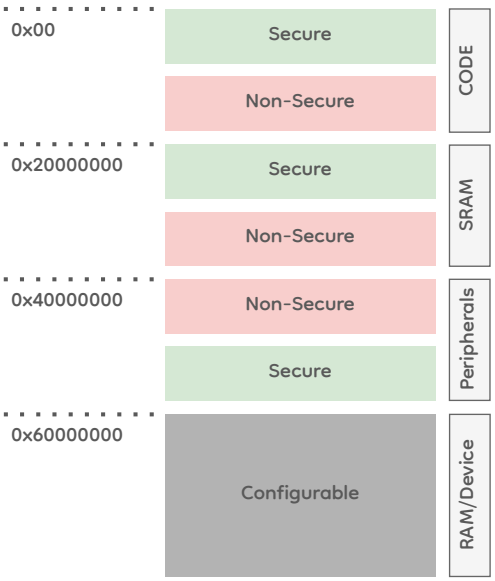

Figure 1: Example TrustZone-M memory map (based on Nordic nRF9160 and TF-M region defines).

In other cases, like some GPIOs, one component can be accessed through two different addresses (e.g. a UART has a secure and a non-secure address and can thus be used in both states). If one component has two addresses, it can depend on the state, whether it is writable or read-only, and what type of data and functionality can be accessed.

While *TrustZone-A* needs a software component called secure monitor to handle the transitions between states, in *TrustZone-M* the transition is handled by the hardware and can be triggered by secure function entry points in so-called non-secure callable memory regions, as well as interrupts. This reduces the overhead and makes it more suitable for constrained, low-power devices. *TrustZone-M* optionally supports MPU usage to further divide between privileged and unprivileged access within the S and NS states.

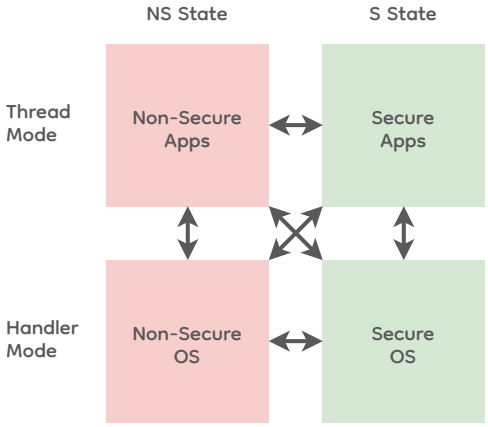

Figure 2: Transitions between S and NS states as well as thread and handler mode in TrustZone-M.

The MultiZone approach has also been implemented for Cortex-M devices with a MPU [14]. Similarly, Oliveira, Gomes and Pinto [12] published uTango, an open-source TEE for IoT devices based on Arm TrustZone-M. Like the MultiZone developers they criticize existing TrustZone-based TEEs for separating the system into only two worlds, implementing too many services in the secure worlds and thus increasing the attack surface. They claim that by creating multiple equally secure execution environments, their own approach will provide better security. They measure the performance overhead of their TEE and perform a security analysis.

2.2 Trusted Firmware M

In order to introduce new standards to IoT security and to streamline the design process of secure IoT systems, Arm has developed a Platform Security Architecture (PSA) framework. PSA provides standardized resources to help build the system in all areas, including hardware, firmware and software design, security analysis and assessment. Products that fulfill all PSA specifications can be certified by the *PSA Certified* program.

Part of the firmware and software specification are the PSA Firmware Framework and PSA Secure Services (cryptography, secure storage, attestation and secure updates). To make it easier for product developers to fulfill those specifications, Trusted Firmware M (TF-M) provides a reference implementation of a Secure Processing Environment (SPE) for Armv8-M and Armv8.1-M architectures as well as dual-core platforms. It integrates with TrustZone-M, and complies with *PSA Certified* guidelines. Using TF-M allows vendors to easily get their products certified without needing to implement the standards themselves.

TF-M is open source and freely available. It comes with its own secure bootloader to authenticate and update secure and non-secure firmware images separately. The TF-M core controls the isolation and communication between the images. Isolation can be applied on multiple levels, depending on a plaform's capabilities and needs. The PSA Secure Services can optionally be added to the build, including their own memory partitions to operate in. This allows for different configurations and binary sizes, to suit different types of IoT devices and their constraints. Additionally, TF-M comes with tests and additional tools and extras (e.g. usage examples and third-party modules).

2.3 The RIOT Operating System

RIOT OS [5] is an open source operating system for IoT devices. It aims to have a small memory footprint, support a wide range of architectures and devices, and be user-friendly and easily accessible. Recently support for secure elements and crypto hardware accelerators have been added and evaluated [9]. In other previous work, we have integrated and evaluated the Arm PSA Crypto API [7]. Recently Blischke [6] has used and evaluated the RISC-V PMP in RIOT.

3 TF-M Integration

3.1 Basic Build and Functionality

TF-M provides an SPE, which acts as an intermediary between a *non-secure processing environment (NSPE)* and the secure hardware. It provides secure services, which are necessary to verify system integrity and increase security. Those services include secure updates, cryptography, secure storage (e.g. for keys and certificates) and attestation.

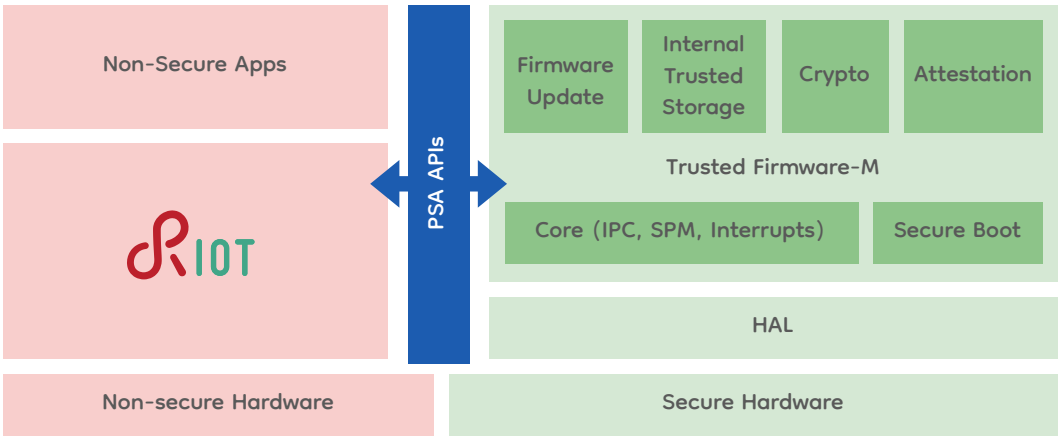

Figure 3: TF-M in combination with a non-secure operating system like RIOT.

The NSPE can be either a bare-metal application or an operating system, and runs in non-secure memory areas. Communication between SPE and NSPE happens through several service APIs, which are provided by the SPE.

The TF-M build system generates three binaries: a bootloader image, a secure firmware image and a non-secure firmware image. All images are signed with a key to be verified by the bootloader and concatenated to a single binary. When flashing, at first the bootloader is written at memory address 0, followed by the merged binary.

When booting the system, the bootloader first verifies the image signatures. If they are valid, it boots the SPE, which sets up the platform. The setup includes the configuration of memory regions and the creation of partitions for the secure services. All of this is done in a secure CPU state. The SPE then loads and boots the NSPE and switches the CPU to non-secure mode. The NSPE can now run application code in non-secure memory regions. If it, for example, uses PSA secure services, it can call the service APIs, which then trigger the transition back to the secure state to perform the requested operations. Afterwards the system switches back to non-secure mode and returns the results of the operation to the NSPE.

The goal of this work is to replace the TF-M NSPE with RIOT and reduce the operations systems access to non-secure area. Regarding the issue of integrating new operating systems with TF-M, the documentation is quite thin. The requirements it states are:

- The OS must be able to run in non-secure mode

- The OS must initialize PSPLIM register and handle it during thread context switch operations
- The OS needs to ensure that link register value can differentiate between S and NS builds

This is a very generic description and required us to find out what exactly this means for RIOT. The necessary steps are described in Section 3.2.

3.2 Integration Steps

Preparing RIOT RIOT currently supports two platforms with TrustZone-M technology, the Nordic nRF5340 and nRF9160. For this work we focused on the latter. When RIOT runs on the nRF9160, it runs in secure mode. This means per default it has access to the whole system and uses secure RAM, peripherals and registers. The first step was to find all the instances where secure addresses are accessed and change them to non-secure addresses. To make this optional, we added the modification as a compile-time option, as shown in Listing 1.

Listing 1: Example of optional access to secure and non-secure LED ports in RIOT

```
#if IS_ACTIVE(MODULE_TRUSTED_FIRMWARE_M)
    #define LED_PORT            (NRF_P0_NS) /**< Default LED PORT */
#else
    #define LED_PORT            (NRF_P0_S) /**< Default LED PORT */
#endif
```

To switch between secure and non-secure images, supervisor calls (SVC) are needed. Per default, RIOT does not use SVC, so they have to be enabled explicitly for TF-M.

After this I had to modify RIOT RAM length and start address. TF-M requires a secure RAM range of 0x16000 bytes, which means that RIOT RAM can only start at address 0x20016000. Usually RIOT RAM starts at 0x20000000, so we needed to modify RIOT RAM length and start address. This can be configured individually for RIOT platforms.

Since TF-M uses MCUboot we could use RIOTs existing partial support for MCUboot to facilitate the integration. MCUboot usage requires the definition of the image header size and the new start address of the binary. Like the modified RAM start address, this can be configured in the CPU specific makefiles in RIOT.

Per default RIOT can only flash one binary. We added a TF-M specific makefile to RIOT, that contains a new flash target with support for multiple binaries. This makefile is executed after building the secure and non-secure images. It links both images, signs them separately with an RSA key and merges them into one binary. It then flashes the bootloader binary at address 0x00 and the merged binary with the required offset.

3.3 Adding TF-M

Trusted Firmware-M has been added to RIOT as a third-party package. We implemented an interface, through which non-secure calls from the non-secure to the secure side can be executed. A makefile downloads the source code and builds TF-M in two steps. First, the secure image is configured and built. This produces the bootloader binary and secure binary, as well as a folder called *api_ns*. This contains code and configurations that are needed for communication between the secure image and the non-secure image. In the next step, we compile this *api_ns* and create a library that shall be linked with the RIOT binary.

3.4 Limitations

Usually RIOT runs applications in a main thread, which is created during kernel initialization and has its own stack with the stack size configured at runtime. After thread creation RIOT initiates a context switch to execute the program until it is done. The current integration with TF-M does not permit RIOT to create its own threads for applications. This means, core threading needs to be disabled and OS and applications are run in the same thread. Since the required stack size for an application is defined when creating the main thread, there is currently no way to dynamically increase the stack size without threading. As a workaround, the stack size is hard-coded in the linker file for the nRF9160.

4 Evaluation

To evaluate the impact of TF-M on RIOT OS, we compare memory consumption and execution time of cryptographic operations with and without a secure processing environment. TF-M uses the PSA Crypto reference implementation from the MbedTLS

library [3]. For comparison, we include MbedTLS as a third-party package in RIOT. We use the same version TF-M uses and build it with the same configuration for the library and the PSA Crypto module. The only difference is that TF-M builds MbedTLS with the SPM (secure partition manager) option, to separate code into secure and non-secure parts. Since RIOT does not support this split we can't use this option in our case.

Compiler optimizations RIOT optimizes compilation for size rather than speed. TF-M builds MbedTLS with level 2 optimizations, which result in a bigger binary size, but faster execution times. For these measurements we also build RIOT with level 2 optimization.

Memory allocation Some MbedTLS operations, like ECC operations, use dynamic memory allocation at runtime. RIOT usually avoids dynamic allocation completely and allocates all sizes statically at compile time. Since RIOT does not provide a good `malloc` implementation, we link MbedTLS with the standard libc implementation. TF-M provides an alternative `malloc` implementation, which allocates memory with a static buffer, which is faster than the libc version.

4.1 Memory Overhead

For measuring memory overhead we build two versions of an application performing an ECDSA operation. One only runs on RIOT with the MbedTLS package, the other runs on RIOT with TF-M and MbedTLS. TF-M can be built with different size profiles, to adapt the firmware to different device constraints. For this evaluation we build the medium profile, since it is the smallest version that also provides asymmetric crypto operations. We then compare the amount of memory used by the text, data and bss sections of the code. As shown in Figure 4, the binary containing only the application with RIOT and MbedTLS uses $\approx 41\ KB$ of ROM and $\approx 7\ KB$ of RAM. The TF-M build consists of three different binaries: the bootloader, the secure image and the non-secure image. They add up to over 200 KB in ROM and $\approx 70\ KB$ in RAM. It is noticable that the RIOT image shrinks by almost 35 KB when we run it as the NSPE. This can be attributed to MbedTLS now being built as part of the secure image instead of the RIOT image.

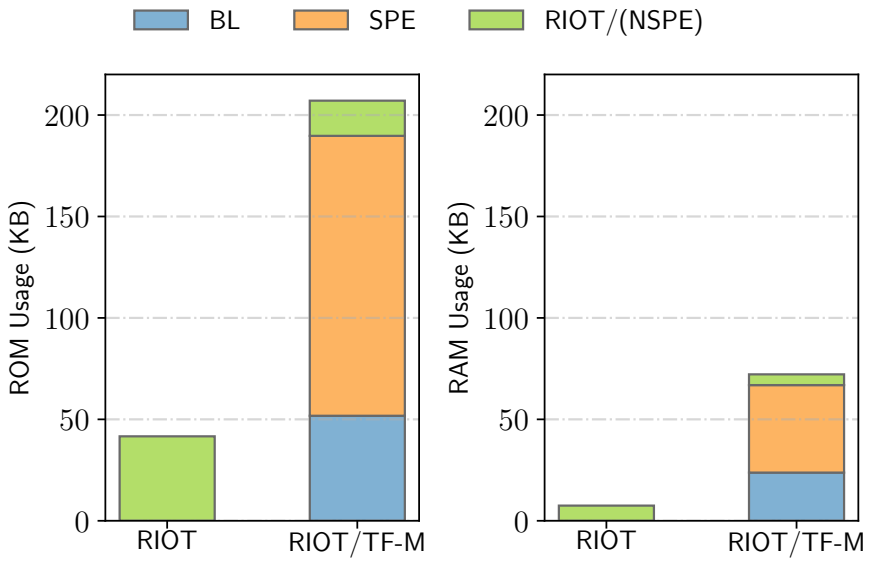

Figure 4: RAM and ROM usage of RIOT and RIOT/TF-M builds.

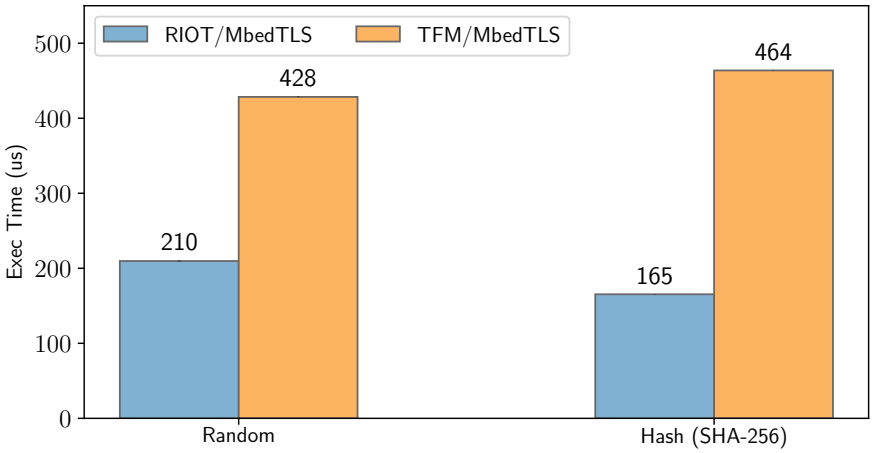

Figure 5: Random and hash execution times by operation.

4.2 Execution Time

To evaluate execution time overhead, we measure the duration of a random number generation (RNG) and a hash calculation. To compare asymmetric operations we measure an ECC key pair generation, hash signature generation, hash verification and a Diffie-Hellman key agreement, all with a NIST-P256 elliptic curve. We perform 1000 iterations of each operation and toggle a GPIO output pin before and after each execution. We sample the data with a logic analyzer at a rate of 6 MS/s.

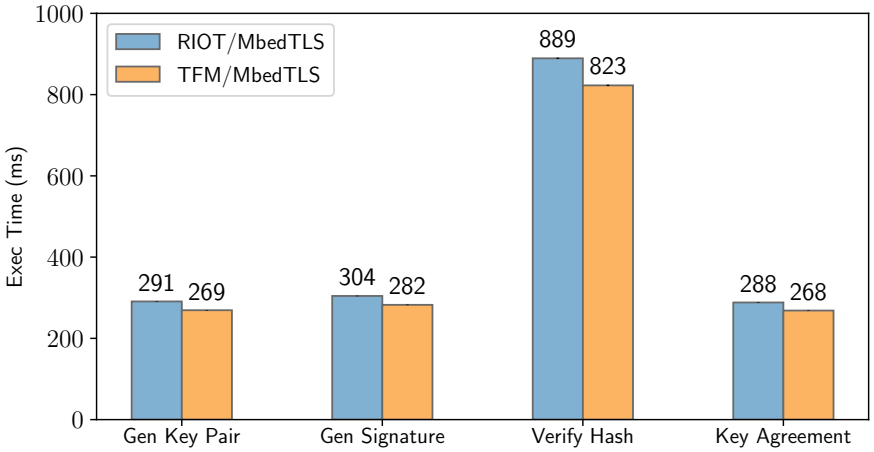

Figure 6: ECC execution times by operation.

Figure 5 shows that RNG takes $\approx 428\ \mu s$ when using TF-M, which is twice as long as the application version using only RIOT. Hash computation time almost triples, from 165 μs up to $\approx 464\ \mu s$. This much overhead is expected.

Figure 6 shows that asymmetric operations look a little different. Here the operations executed by only RIOT and MbedTLS take longer than the ones executed by TF-M and MbedTLS. A likely cause are the different `malloc` implementations mentioned in paragraph 4. It is also possible that TF-M does some other optimizations. Also, asymmtric operations take a lot longer than simple operations like hash computation and RNG. While the overhead for switching between secure and non-secure state can take several hundred μs (as shown in Figure 5), asymmetric operations have execution times of several hundred ms. This shows that when perfoming longer and more complex operations, the relative overhead added by TF-M is still quite small.

4.3 Limitations

This evaluation only measures a small subset of TF-M operations and configurations. For example, TF-M provides different size profiles with different capabilities, different levels of memory isolation and internal secure storage for persistent key storage. It also offers two types of communication between secure and non-secure worlds, depending on the level of isolation. All of those could possibly introduce more or less overhead and produce quite different results. Also, we only measured software implementations for the crypto operations. TF-M supports hardware acceleration for crypto operations on

some platforms. Since RIOT does not offer the same capabilities, yet, those could not be compared.

For an extensive analysis, more measurements of different configurations are needed.

5 Conclusion and Outlook

In this work we describe a way to integrate the open source project Trusted Firmware-M with the IoT operating system RIOT and measured the overhead it introduces to memory usage and execution times of cryptographic operations. Our measurements of execution times show that the impact of TF-M is high when performing simple executions (like RNG and hash computation) and negligible when performing complex operations such as asymmetric cryptography. There is no definite conclusion whether the use of TF-M pays off comparing only the execution times.

Regarding memory usage, we can see that TF-M adds significant overhead compared to a RIOT-only binary. While it is possible to build smaller configurations of TF-M, this comes with a trade-off in functionality and security, and even small TF-M builds will still add significant overhead on the ROM and RAM side. This makes it only feasible for devices that have a large amount of flash and RAM available.

TF-M is integrated with the operating system Zephyr and, for Nordic platforms, with the Nordic Software Development Kit (SDK). Both are well documented and TF-M developers as well as the Nordic developer support strongly encourage people to work with those integrations.

On the other hand, it is difficult to get support when developing individual solutions and integrating TF-M with other operating systems. The documentation is quite generic and parts of it were incomplete or outdated by the time we tried to work with them (some parts have been updated recently). During our work with TF-M, changes to the code and build process were introduced, sometimes breaking the code for our platform. Additionally, even their minimal bare-metal example didn't run out of the box on our board and required modification and debugging. This leads to the assumption that not all code changes in the main repository are regularly tested on all platforms. This makes it harder to support and maintain a TF-M integration in RIOT.

For RIOT it is also an issue that TF-M only supports a subset of Arm Cortex-M devices. To use TF-M, a device needs to support TrustZone-M or a Cortex-M dual-core architecture. RIOT, on the other hand, aims to support many different platforms and architectures. As mentioned before, another relevant platform with a hardware protection mechanism is RISC-V, which would require a separate solution.

This leads to the conclusion that for RIOT it might be more feasible to provide its own implementation of a TEE, that is more reduced in size and can support other platforms. Therefore, in future work, we will explore alternative approaches. This work will be used as a benchmark, to compare our implementation to a vendor solution and improve efficiency as much as possible.

References

[1] IEEE Standard for Secure Computing Based on Trusted Execution Environment. In: *IEEE Std 2952-2023* (2023), S. 1–29

[2] ARM Ltd.: *Arm PSA Firmware Framework 1.0*. https://developer.arm.com/documentation/den0063/a/?lang=en, last accessed 04-10-2024. 2019

[3] ARM Ltd.: *Mbed TLS*. https://tls.mbed.org, last accessed 07-17-2020. 2020

[4] ARM Ltd.: *Arm Firmware Framework for M 1.1 Extension*. https://developer.arm.com/documentation/aes0039/latest/, last accessed 04-10-2024. 2023

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

[6] Blischke, Bennet: *Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices*. 2023. – URL http://inet.haw-hamburg.de/thesis/completed/ba_bennet_blischke.pdf

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

[8] Garlati, Cesare ; Pinto, Sandro: Secure IoT Firmware For RISC-V Processors, 03 2021

[9] Kietzmann, Peter ; Boeckmann, Lena ; Lanzieri, Leandro ; Schmidt, Thomas C. ; Wählisch, Matthias: A Performance Study of Crypto-Hardware in the Low-end IoT. In: *Proc. of Embedded Wireless Systems and Networks (EWSN'21)*. New York, USA : ACM, February 2021. – URL https://dl.acm.org/doi/10.5555/3451271.3451279

[10] Lee, Dayeol ; Kohlbrenner, David ; Shinde, Shweta ; Asanović, Krste ; Song, Dawn: Keystone: An Open Framework for Architecting Trusted Execution Environments. In: *15th European Conference on Computer Systems*. New York, NY, USA : Association for Computing Machinery, 2020 (EuroSys '20). – URL https://doi.org/10.1145/3342195.3387532

[11] Lu, Tao: A Survey on RISC-V Security: Hardware and Architecture. In: *CoRR* abs/2107.04175 (2021). – URL https://arxiv.org/abs/2107.04175

[12] Oliveira, Daniel de ; Gomes, Tiago ; Pinto, Sandro: uTango: an open-source TEE for the Internet of Things. In: *ArXiv* abs/2102.03625 (2021). – URL https://api.semanticscholar.org/CorpusID:231846582

[13] Pei, M. ; Tschofenig, H. ; Thaler, D. ; Wheeler, D.: Trusted Execution Environment Provisioning (TEEP) Architecture / IETF. URL https://doi.org/10.17487/RFC9397, July 2023 (9397). – RFC

[14] Pinto, Sandro ; Garlati, Cesare: Multi Zone Security for Arm Cortex-M Devices, 02 2020

[15] Pinto, Sandro ; Santos, Nuno: Demystifying Arm TrustZone: A Comprehensive Survey. In: *ACM Comput. Surv.* 51 (2019), jan, Nr. 6. – URL https://doi.org/10.1145/3291047

[16] Usman, Ahmad B. ; Cole, Nigel ; Asplund, Mikael ; Boeira, Felipe ; Vestlund, Christian: Remote Attestation Assurance Arguments for Trusted Execution Environments. In: *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems*. New York, NY, USA : Association for Computing Machinery, 2023 (SaT-CPS '23), S. 33—42. – URL https://doi.org/10.1145/3579988.3585056

[17] Zandberg, Koen ; Baccelli, Emmanuel: Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF. In: *CoRR* abs/2011.12047 (2020). – URL https://arxiv.org/abs/2011.12047

[18] Zandberg, Koen ; Baccelli, Emmanuel: Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules. In: *CoRR* abs/2106.12553 (2021). – URL https://arxiv.org/abs/2106.12553

[19] Zhou, Wei ; Guan, Le ; Liu, Peng ; Zhang, Yuqing: *Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems*. 2019

Erklärung zur selbstständigen 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