Bachelor's Thesis

Trusted Peripherals in TrustZone-M

[redacted]

[redacted]

[redacted]

[redacted]

Abstract

Internet-of-Things (IoT) devices find increasing deployment in untrusted environments while managing and sending a growing amount of data that is captured from the outside world using peripherals. The correctness of that data is often critical to the receiver and the nature of it can be sensitive. However, security concerns in IoT development have historically been an afterthought, as resource constraints did not allow for expensive protection of data and code. ARM's TrustZone technology for Cortex-M processors (TrustZone-M) aims to provide an isolation mechanism that is low in overhead. With this, the prospect of securely — but cheaply — isolating IoT peripherals has surfaced. If successful, assurances to the accuracy and secrecy of peripheral data could be made by the IoT application developer.

In this work, in order to ensure that such peripherals can be trusted, we construct a threat model that clearly defines what security considerations have to be made in the specific case of IoT devices in untrusted environments. Using this model, we give a definition for what we call trusted peripherals. On the basis of this definition, we propose an API that facilitates the secure capturing and transmission of peripheral data under guarantees of integrity, authenticity and optionally confidentiality. This API can accommodate three interaction scenarios that we have found to be most common in practice.

We employed TrustedFirmware-M (TF-M) as as trusted execution environment (TEE) to fully implement this API as a secure service, often denoted as a trusted application (TA). Any real-time operating system (RTOS) with TF-M integration can make use of this framework. Thus, our implementation preserves the separation of concerns between the TA and normal application developer, which we foresee to enable an efficient and secure software development process.

We have evaluated this implementation on a prototype using the NUCLEO-L5552ZE-Q development board and a temperature sensor as an attached peripheral. We have found the performance overhead of using TrustZone-M and TF-M to be minimal and the trade-offs in terms of other resources to be within reason.

We conclude that our findings have the potential to improve a field of IoT development where security guarantees to a device's peripheral data were previously not possible. In this regard, an infrastructure could be established, where independent TA developers, consisting of security experts, could provide secure implementations of our API to application developers, who then leverage the trust in their peripherals to provide new security claims to their customers.

Contents

1 Introduction **1**

2 Background **5**

2.1 Trusted Execution Environment . . . 6

2.1.1 Requirements of a TEE . . . 6

2.1.2 TEE Technologies . . . 8

2.2 TrustZone-M . . . 8

2.2.1 Execution Modes . . . 9

2.2.2 Secure and Non-Secure Worlds . . . 10

2.2.3 Transitions . . . 10

2.2.4 Security Attribution . . . 11

2.2.5 Secure Peripherals & Interrupts . . . 12

2.2.6 TrustZone-M TEEs . . . 13

2.3 Platform Security Architecture . . . 13

2.3.1 Security Goals . . . 13

2.3.2 Threat Models and Security Analyses . . . 14

2.3.3 Certified Levels . . . 14

2.3.4 Firmware Framework . . . 14

2.4 TrustedFirmware-M . . . 16

2.4.1 Root-of-Trust . . . 16

2.4.2 Secure Partitions . . . 17

2.4.3 TEE Terminology . . . 17

2.4.4 Architecture . . . 18

2.4.5 Isolation Level . . . 19

2.4.6 SFN and IPC Mode . . . 19

2.4.7 Integration . . . 20

2.4.8 TF-M as a TEE . . . 20

3 Related Works **21**

4 Design **23**

4.1 Threat Model . . . 23

4.2 Definition: Trusted Peripheral . . . 24

4.3 Trusted Capture . . . 27

4.4 Trusted Delivery . . . 29

4.5 Trusted Transformation . . . 31

4.6 Discussion . . . 34

4.7 Design Goals 36

5 Implementation **37**

5.1 Trusted Peripherals API 37

5.1.1 Trusted Capture API 38

5.1.2 Trusted Delivery API 39

5.1.3 Trusted Transformation API 40

5.1.4 Trusted Transformation API with handles 43

5.2 Hardware and Software 45

5.3 Trusted Peripherals as a Secure Service 45

5.3.1 SFN vs. IPC 47

5.3.2 Secure Peripherals 52

5.3.3 Complications 53

6 Evaluation **57**

6.1 Security Analysis 57

6.2 Performance 60

6.2.1 Performance: Trusted Capture 63

6.2.2 Performance: Trusted Delivery 65

6.2.3 Performance: Trusted Transformation 66

6.2.4 Transmission Rates 69

6.2.5 Footprint 71

6.3 Usability 72

6.3.1 Usability for Trusted Application Developers 72

6.3.2 Usability for Normal Application Developers 73

7 Future Work **75**

8 Conclusion **77**

Bibliography **79**

List of Tables

2.1 Determination Logic of the Security Attribution in TrustZone-M 11
2.2 TEE Terminology Differences 18

6.1 Packet Sizes of all Involved API Calls (in Bytes) 70
6.2 Flash Usage in Trusted and Untrusted Setup (in Bytes) 71
6.3 RAM Usage in Trusted and Untrusted Setup (in Bytes) 72
6.4 Division of Flash Memory for our Board Using Different Layouts (in KB) . 72
6.5 Lines of Code for the API Implementation in Trusted and Untrusted Setup 73

1 Introduction

As Internet of Things (IoT) devices and embedded systems in general become more and more prevalent in consumer markets and industrial infrastructure, the security requirements of these devices become more demanding[32, 82]. A major part of the value proposition for IoT systems comes from the ability to capture data from the outside world, process them internally and consequently transmit or display the data in prepared form. This is made possible by connected peripherals, ranging from sensors to input devices like keyboards to output devices like displays. The collected data is often associated with privacy concerns in the consumer case or otherwise demand secrecy from prying eyes[4]. This problem gets exacerbated with the physical location of these devices, since many use cases envision a scenario in which the product is stationed outside the control and supervision of authorized personnel — such as the manufacturer or owner of the device. This creates a prime environment for any attacker to perform not only software- but also hardware-based attacks. A successful software attack could mean the introduction of malware into the system to send fabricated peripheral data to the original destination servers and sending any genuine data to the attacker's machines, all while remaining undetected. A hardware attack on the other hand could be performed through a class of side-channel attacks, analyzing the device's physical signals to extract the data — breaking all confidentiality and secrecy in the process.

With devices being located in these insecure areas, the question arises, as to how common information security goals of confidentiality, integrity and availability can be assured, especially in regard to any sensitive peripheral data. The typical example that can be given for hardware devices being deployed in untrustworthy environments and mostly successfully shielded from hardware attacks are security cameras. However, these devices are employed in an unusual setting when compared to typical IoT devices. They are normally placed inside tamper-resistant cases and are positioned out-of-reach, with the potential of being recorded serving as a deterrence for any attackers. This scenario is not applicable to the majority of devices and as such, surveillance cameras can not be used as a model against hardware attacks — particularly since the software side of them has several examples of successful remote attacks against devices from various manufacturers[46, 47], rendering any protection of the confidentiality and integrity of video footage through physical means as useless.

An approach to this problem that addresses vulnerabilities in software and — as far as possible — hardware, could lead to a wider adoption of IoT devices in areas where the acceptance and application of them is contingent on the data being reliable and genuine.

ARMs TrustZone technology promises to provide a hardware mechanism for isolation in supported processors[9]. This isolation is a key component of the concept behind the so-called *trusted execution environment* (TEE), which aims to establish an environment for secure execution of code and protected storage of data. This environment can exist outside the normal application and even operating system and assures that attacks, which target the two, can never put the TEE itself into any danger — thus the problem of the before mentioned one vulnerability is restricted to the untrusted side. The trusted environment instead relies on a small Trusted Computing Base (TCB) by design and can accordingly give us more confidence that no such vulnerability exists in its world.

The benefits of this code isolation can further be leveraged and secured by the separation of development into a trusted application (TA) and the normal, non-secure application. The developers of the TA can then consist of a small team of security experts that keep the size of the TCB at a minimum and assure its security. The application developers on the other side can rely on the secure functions that the TA exposes for them and be relieved in a major way of the complexity that is associated with security considerations in software development.

In this thesis, we will explore the possibility of implementing peripherals that can be considered secure and trusted — which we refer to as *trusted peripherals* — by isolating them inside a TrustZone-based TEE. Our goals include the secure capturing, transformation and delivery of peripheral data in an untrusted IoT setting. This solution is primarily required to protect the data and the peripherals in their *integrity* and *authenticity*, while a guarantee in *confidentiality* is also desirable. Additionally, any approach to achieve these security traits should preserve the *separation of concerns* between the trusted application and the normal application. If a design foresees the access of trusted application code by the normal application developers in their day-to-day operations, we consider that solution to have failed in this separation. Our hope is to present the feasibility of our design in practice by weighing the gain in security we have achieved against the costs that we needed to pay in terms of performance and software complexity.

In the following chapters, we introduce and explain the background that is needed to understand our work in §2. We will present a definition of TEEs in §2.1, a hardware technology that can be used to implement them in §2.2, an official specification that can help guide our security goals in §2.3, as well a reference implementation of a TEE that satisfies all our requirements in §2.4. In §3, we present works, which we have discovered to be related on the topics of peripheral isolation, trusted execution environments, TrustZone or trusted APIs. We will discuss which of these works we have chosen to build upon and consequently explain why other papers receive only a mention, because — while related to an overlying subject — they differ enough in their approaches that they turned out to be not applicable for our purposes.

Afterwards, we explain our design in its general form in §4. It is based on security considerations that we gain on a threat model that we will construct in §4.1. To establish a common understanding, we are going to provide a definition of trusted peripheral by means of distinct peripheral interaction scenarios that we envisioned and intend to support in §4.2. We elaborate on the specific applications and how they differ in their security guarantees in §4.3, 4.4 and 4.5. To conclude the design chapter, we discuss possible design alternatives in §4.6 and present how our approach achieves the goals that we put forward here in §4.7.

In our implementation of this design in §5, we will provide a brief summary on the hardware and software that we chose for our implementation and the motivations behind these choices. Afterwards, we provide the proof-of-concept of a trusted peripheral in the form of a temperature & humidity sensor that can securely capture and transmit its data and optionally display the data on an OLED display that is only accessible from the secure world. In practice, a device like this could be located in an agricultural setting to securely capture and transmit temperature data to be able to make assertions on the quality of grown products. A different deployment could be in the field of supply chain management[80]. In order to ensure that the right environmental conditions of perishable goods are met throughout the entire transportation process, a version of our prototype could accompany the products on its journey.

To then assess its usefulness, we compare this prototype with an equivalent setup in untrusted form in §6. This evaluation aims to determine the degree to which we gained security in §6.1, the amount of performance it has cost us in §6.2, and the extent to how realizable this design is in practice §6.3. Finally, we present ways in which this or related topics can be contributed to in §7 and draw our own conclusion in §8.

2 Background

In the year 2022, the Linux kernel has had 476 vulnerabilities discovered inside of its codebase[40]. The situation for embedded systems developers using a real-time operating system (RTOS) to meet their application's requirements is similar; the popular RTOS *FreeRTOS* having 14 discovered vulnerabilities that are rated high to critical as of time of this writing[40, 38]. The developer's reliance on an RTOS in embedded systems to meet their real-time constraints is now commonplace and with the growing IoT market in mind, the prediction of the number of RTOS vulnerabilities increasing is not far-fetched.

Outside of the enormous software stack that forms the operating system, a considerable amount of code in the application itself now originates from third-party sources, such as with included libraries and software development kits. These add more unknowns into the system and can increase the attack surface of any application. With these facts in mind — even after rigorous analysis of their own code — developers have to assume that vulnerabilities exist somewhere in their application, possibly outside of their own control. And the nature of any one vulnerability in a program is that it has the potential to take down the security of the entire rest of the system.

This susceptibility to any one weakness can be traced back to the way software is developed. In the conventional approach to software development in the embedded world and elsewhere, several pieces of code with often wildly differing security demands are directly combined via the compiler and linker into one program, or — in the case of embedded development — one big memory image that is then flashed onto the device. Without any precautions taken, security-critical cryptographic functions used in the application will then effectively have the same protection as any other function; secret keys will be stored in memory regions that require the same access privilege as any other data. Even if the developer strives to establish barriers between these parts, this monolithic model of development hinders the feasibility of any isolation mechanism.

Instead, we need to be able to separate in advance the parts of an application that we know we can — and sometimes are required to — trust from the parts that are potentially insecure. With these measures in place, a vulnerability in a less trusted part of the system can not bring down the security of a more trusted part. The term that encapsulates the idea of this compartmentalization is the *Trusted Execution Environment* (TEE).

We will start this chapter by discussing existing definitions of TEEs in §2.1 and arrive at a list of requirements that a security framework needs to meet in order to be considered a TEE. Apart from TrustZone, we will briefly mention other hardware mechanisms that can help to implement a TEE. Then we explain how the TrustZone technology for Cortex-M devices — also known as TrustZone-M — works in §2.2 and how it achieves isolation between two worlds inside the same microcontroller. Related to this, we will present ARM's *Platform Security Architecture* (PSA) in §2.3, which was created to establish a standard for IoT security. Since many of its objectives overlap with our own, we will summarize some of its various goals, specifications and other documents that can help us in laying a security foundation for our mission. Linked to this, we will give a detailed background of *TrustedFirmware-M* (TF-M) in §2.4, the reference implementation to one of PSA's specifications. We utilize this software in our own implementation, and many of its features will prove critical to our trusted peripherals.

2.1 Trusted Execution Environment

The first attempt to define the term trusted execution environment was carried out by the Open Mobile Terminal Platform (OMTP) in 2009[1]. In it, OMTP defines a TEE as either a set of components that protects against only software attacks — which would make it a *profile 1* TEE according to their terminology — or a *profile 2* TEE that has built-in protections against software and hardware-based attacks. Their standard was not followed up with renewed versions and the effort to standardize the TEE was subsequently picked up by GlobalPlatform in 2010[19]. Their specification of a TEE encompasses several documents with ongoing work put into them — their latest TEE related document being released in July 2023[21].

However, Sabt et al.[60] present inconsistencies between existing TEE definitions — including definitions from OMTP and GlobalPlatform — and instead propose their own. In the following, we will present the requirements to a TEE that we have identified from the literature and the established standards.

2.1.1 Requirements of a TEE

The **Root of Trust** (**RoT**) in a TEE is the trust foundation upon which other trusted structures can be built. As a component's separation from the RoT increases, the degree to which we trust the component usually decreases. As such, the RoT serves as the reference point for the trustworthiness of every other component in the system. A hierarchy of trust grows from this process, in which the only part that is absolutely trusted is the Root of Trust. In practice, the most fundamental RoT can come in the form of a cryptographic key pair, as we will demonstrate in our explanation of the secure boot requirement. If those keys — and by extension the Root of Trust — get compromised, the entire system from a security perspective has to be assumed to be compromised as well.

Secure boot refers to the assurance that the bootloader will only load trusted code. The secure boot is considered to be either part of the root of trust[60] or built directly upon it. This is simply because the trust in any other component can not be given in any way if the bootloader can load code provided by an attacker. A way to achieve secure boot is by signing the firmware image before flashing it onto the device with the private key from the aforementioned key pair that is part of the Root of Trust. The bootloader on the other hand is equipped with the corresponding public key to verify the firmware image by checking its signature. As long as the security of the public key inside the bootloader and the security of the private key belonging to the developer or manufacturer can be assured, this can be considered a minimum viable form of a secure boot. Since the public key needs to be stored on the IoT device, the memory in which it is stored needs to be protected by hardware mechanisms such as locked on-chip flash memory or One Time Programmable (OTP) memory.

The **Secure Processing Environment** (**SPE**) is a prerequisite of any TEE, as it refers to the previously touched upon ability of a processor to isolate parts of its code and data from other parts. This environment needs to preserve the authenticity of code, the integrity of runtime states and the confidentiality of all code, data and runtime states[60]. This environment typically excludes the operating system itself. Its non-secure equivalent is the Non-Secure Processing Environment (NSPE).

The requirement of **Inter-Environment Communication** refers to the ability of a TEE to communicate with untrusted components outside of it. This means that the isolation between the SPE and NSPE cannot be absolute, which creates new targets for attackers[60]. This is seen as a necessary trade-off, since the ability for the untrusted application to call secure functions is a core benefit of the TEE. The specifications of a whole class of interfaces originates from this demand[5], with the most common example being APIs that offer secure cryptographic functions to the untrusted environment.

Another requirement for any TEE is the feature of **Remote and Local Attestation**. Attestation refers to the ability to uniquely identify a device, with the added constraint that an attacker should not be able to change the devices identity. With the help of Local Attestation, the identity can be proven to a local verification entity. Remote Attestation extends this ability to remote identification, which is particularly important in the field of IoT, where the manufacturer's physical access to the device after deployment can not be given. Using this process, secure communication channels can be established between the device and the manufacturer[49], which can be used to securely transmit a new firmware image, for instance.

The **Trusted I/O Path** of a TEE should guarantee that the communication between a TEE and peripherals should not be observable by untrusted components.

The **Secure Storage** system of a TEE needs to provide assurance that data stored inside of it is sealed from the untrusted environment and cannot be changed or accessed by it.

2.1.2 TEE Technologies

To implement an isolation between two execution environments — an secure and non-secure one as mentioned in above criterion —, one could use two separate processors that communicate with each other over a secure channel. This however, proves costly and complicated. To facilitate this isolation on a single chip, hardware manufacturers have introduced various technologies.

Intel's Software Guard Extensions (SGX) was first released in 2015 and is an example of creating so-called secure enclaves from user-space. As their Root of Trust, Intel specifies a Trusted Platform Module (TPM)[49]. Intel SGX is focused on desktop and server-end applications[83, 49]. For RISC-V platforms, MultiZone Security is available, which uses RISC-V's Physical Memory Protection (PMP) interface[39].

TrustZone is ARM's technology that provides a hardware mechanism for a separation of the processor resources into a secure and normal world[9]. As of now, there are two TrustZone technologies that both enable this kind of separation, but achieve it in different ways and are ultimately incompatible with each other. These are the TrustZone technology for the Cortex-A architecture (TrustZone-A)[77] and the TrustZone technology for the Cortex-M architecture (TrustZone-M)[78]. The Cortex-M architecture specifically targets embedded systems using microcontrollers, which fits our use case.

There also exist implementations and approaches to purely software based TEEs[26, 36]. These systems rely on a component to provide virtualization of all required TEE features. As such, the security of the system is dependent on the security of the virtualization component itself[5]. Thus, these software implementations come with a performance overhead and weaker security assurances when compared to hardware-based TEEs. For these reasons, we will focus our attention on hardware TEE technologies — specifically TrustZone.

2.2 TrustZone-M

TrustZone-A was introduced with the Cortex-A architecture in 2004[2]. Cortex-A describes the application profile of Cortex chips and is intended for use in handheld devices — typically running Linux or Android. TrustZone-M describes the TrustZone technologies for the Cortex-M architecture and was first introduced in 2017 with the ARMv8-M architecture of the Cortex-M23[6]. Cortex-M describes the microcontroller profile of the Cortex chips and is intended for use in microcontrollers with real-time constraints.

With the introduction of TrustZone-M, the original technology from 2004 can be called TrustZone-A for clarity purposes. However, most works in the literature and elsewhere do not consider the possibility of confusion and still simply refer to TrustZone-A as TrustZone[71, 61, 31, 27]. This will turn out to be a reoccurring nuisance in later chapters.

In this chapter, we will focus on the operating principles of TrustZone-M. The reasons for this lie in ARM's motivations of introducing a new TrustZone technology in the first place.

A major contributing factor are the architectural differences between Cortex-M and Cortex-A chips. Cortex-M makes no use of a Memory Management Unit (MMU), meaning there is no concept of virtual memory[13]. Instead, everything from flash to ram to peripherals is mapped into the 32 bit address space. To provide memory protection, there is a Memory Protection Unit (MPU) that partitions the memory into secure and non-secure regions. TrustZone-A on the other hand, relies on the MMU for address space isolation and thus memory protection. Further, Cortex-M microcontrollers do not include CPU cache memory in most cases, while Cortex-A chips rely on them for performance gains. Caches introduce complexities in managing security and isolation and have been a previous source of vulnerabilities in TrustZone-A[87, 30]. Another reason were the different constraints between the two architectures. TrustZone-A was designed with handheld devices in mind, while Cortex-M devices use microcontrollers in IoT settings with less computing power and memory. TrustZone-M also needed to consider the requirements of real-time applications, most prominently predictability in execution and low latency. Lastly — and perhaps most importantly — TrustZone-A is a technology from 2004. The field of IT security has changed since then and the prospect of reimplementing TrustZone meant that new insights could be adopted.

For all of these reasons, TrustZone-M is the more applicable technology for our purposes. Hence, for the remainder of this work, we will not explain any more technical specifics of TrustZone-A.

As previously mentioned, TrustZone sets up a division of resources into a secure and non-secure world. This applies to both TrustZone-A and TrustZone-M. In the following sections, however, we will provide a specific explanation as to how this separation is achieved in TrustZone-M. Unless explicitly referenced, source information is derived from ARM's documentation[12].

2.2.1 Execution Modes

Prior to the introduction of TrustZone-M, the ARMv6-M, ARMv7-M architectures provided their own mechanism related to security in the form of two distinct execution modes: thread mode and handler mode. These modes can differ in their respective privilege levels, as illustrated in Figure 2.1. The handler mode always has privileged access level, while the thread mode can be one of the two. Typically, unprivileged thread mode is the execution mode for any application code, while the operating system runs in privileged thread mode. Handler mode on the other hand, is the mode that will be entered by the raising of an exception, e.g. through an interrupt that is triggered by a peripheral[81].

The TrustZone-M enabled ARMv8-M architecture inherited this design, but orthogonally added to it in the form of the secure and non-secure state. This is depicted in Figure 2.1.

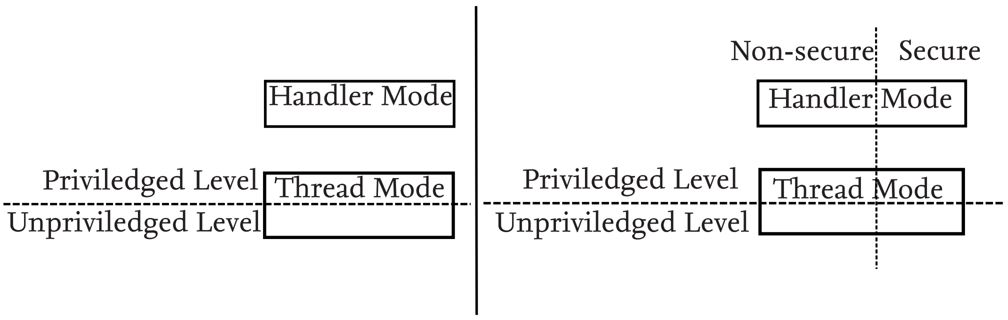


Figure 2.1: Execution modes before ARMv8-M (left) and execution modes in ARMv8-M (right)

2.2.2 Secure and Non-Secure Worlds

As mentioned, all system resources (Flash, RAM, Peripherals etc.) in Cortex-M processors are mapped into the 32-bit memory address space. With the addition of two distinct worlds, these resources can be assigned to one or the other by the developer. This is achieved by marking the memory of a corresponding system resource as secure, meaning it can only be accessed while the processor is in the secure state. Non-secure resources on the other hand, can be accessed by the non-secure and secure world alike. The processor receives its security state from the currently executing instruction. Put most simply, an instruction that resides in a secure region puts the processor into the secure state. In turn, a non-secure instruction causes a non-secure state. Since a processor in the non-secure state cannot access secure resources, what would follow from this, is that code from the secure world could never again be executed. To address this issue, a third security attribute is added to mark instructions. These attributes now being *non-secure* (NS), *secure* (S) and the newly introduced *non-secure callable* (NSC). The NSC security attribute can mark memory regions that contain code that serve as entry points for the non-secure world. Thus, the NSC security attribute is required for the transition mechanism from the non-secure to the secure world.

2.2.3 Transitions

The corresponding processor instruction for this mechanism is the new secure gateway instruction `SG`. This instruction needs to be marked as NSC and is located before the jump to secure code. Executing `SG` sets the processors security state to secure, which enables the processor to execute the secure code that it jumps to. To transition back into the non-secure world, the Branch and Exchange NS instruction `BXNS` is used with the register `Rm` as its argument. `Rm` contains the address that the processor jumps back to after it has entered the non-secure state. TrustZone-M also includes an instruction for the calling of non-secure code while in a secure state. This is done by the branch with link and exchange to non-secure state instruction `BLXNS` with the register `Rm` from before as its argument.

Between these transitions from one security state to another, some registers are banked, meaning they are saved for a later transition back into the former security state. One of the banked registers is the stack pointer, to enable separation between the secure and non-secure stacks[12].

2.2.4 Security Attribution

To define partitions of memory as non-secure, non-secure callable and secure, so-called attribution units are used, of which there are two different kinds. The Security Attribution Unit (SAU) is the default attribution unit that is present on all TrustZone-M enabled devices. The Implementation Defined Attribution Unit (IDAU) is an optional unit outside of the processor and device-specific. These attribution units can disagree in their partitioning, in which case the combination of the two gives the effective security attribution. The logic by which we determine this final attribution is shown in Table 2.1.

Table 2.1: Determination Logic of the Security Attribution in TrustZone-M

IDAU	SAU	Result
NS	S	S
NS	NSC	NSC
NS	NS	NS
NSC	S	S
NSC	NSC	NSC
NSC	**NS**	**NSC**
S	S	S
S	**NS**	**S**
S	**NSC**	**S**

Surprisingly, the security attribution is biased towards giving the more secure attribution in cases of disagreement. Those cases are highlighted in bold in Table 2.1. The reason for the logic being defined this way is the fact that the IDAU is mostly used by vendors to provide reasonable default secure memory partitions. With the help of the SAU a developer can then override these default partitions if needed. Figure 2.2 illustrates how this process looks in practice with memory regions that are marked differently by the IDAU and SAU.

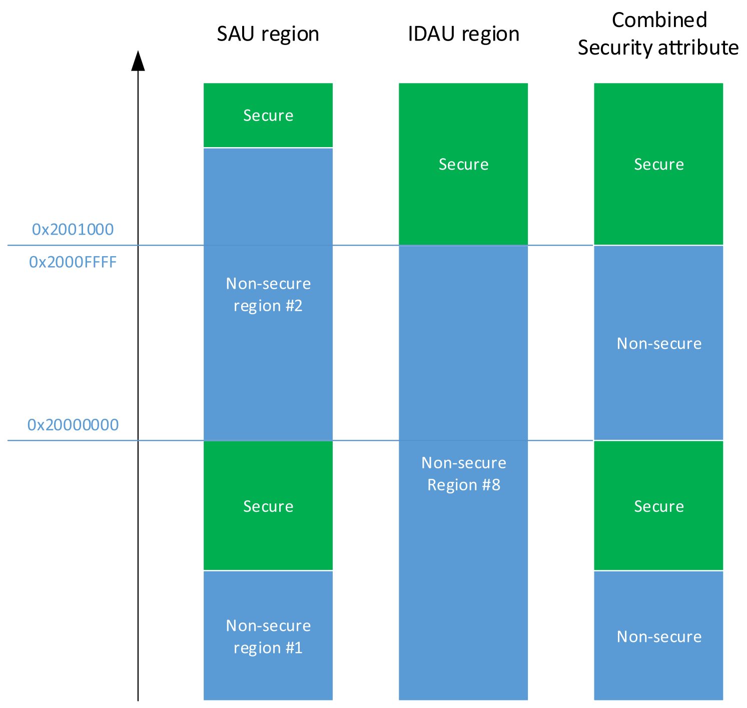


Figure 2.2: Security attribution from IDAU and SAU regions, adapted from [81]

2.2.5 Secure Peripherals & Interrupts

On a given device, there can be more more than just the processor that can issue transactions and interact with peripherals. An example can be the Direct Memory Access (DMA) controller, which might not understand the security policies set by the SAU and IDAU. Hence, transactions issued by other master components to slave components can not be secured without an additional mechanism.

To specifically create secure peripherals, TrustZone expects at least one of two scenarios: The peripheral itself being TrustZone-aware (i.e. understanding the SAU/IDAU configuration) or the transaction can be blocked by an additional security gate beforehand. In the latter case, these security gates have to be configured by the Global Trust Zone Controller (GTZC), which is another component on any TrustZone-M device that is responsible for the security configuration outside the processor.

As such, if a TrustZone-unaware peripheral needs to be secured, additional setup code on the device has to run that tells the GTZC to set the corresponding secure flags for the peripheral. Any non-secure transactions then result in an exception, triggered by the GTZC.

Next to the existence of secure peripherals, TrustZone-M also has facilities to enable the existence of secure and non-secure interrupts. To achieve this, the Vector Table Offset Register (VTOR) is banked the same way as the stack pointer described before. This allows for the vector tables for the secure and non-secure world to be separated. Similar to the assignment of secure and non-secure memory, interrupt sources can be configured as secure and non-secure. Interestingly, a non-secure interrupt can occur during execution of secure code and will be handled. This is an intentional design decision based on the real-time constraints of microcontrollers[81].

2.2.6 TrustZone-M TEEs

The technologies of TrustZone-M form a foundation to build a Trusted Execution Environment. While TrustZone cannot be considered a TEE in itself, it gives us the ability to implement the Secure Processing Environment that is vital to the operation of any TEE. We will conclude this chapter by taking a look at existing TrustZone-M based TEEs. While there are various well-known examples of TEEs using TrustZone-A[5, 60], the cases of TEEs relying on TrustZone-M as their isolation mechanism are sparse as of the time of this writing. Examples include ProvenRun's ProvenCore-M[51] and Trustonic's Kinibi-M[29]. Outside of these, another group of TrustZone-M based TEE's does exist, but this group is characterized by their reliance on the PSA compliant TF-M. We will give an overview on what PSA and TF-M are in the following chapters.

2.3 Platform Security Architecture

To establish a standard in IoT security, the Platform Security Architecture (PSA) was created by an industry consortium led by ARM[52]. PSA provides guidance in the security design process through a four step model that includes analysis through threat modeling and security analysis (TMSA), planning security architecture with the help of PSA specifications, implementation of the firmware source code on the basis of standardized APIs, and lastly certification through the PSA Certified certification scheme[65].

2.3.1 Security Goals

At the base of PSA's efforts lies the Platform Security Model document, on top of which all other specifications are build upon[50]. In it, PSA defines 10 security goals that it sees as essential to establish trust. These goals consists of the ability for unique identification, the support of a security lifecycle, device attestation, secure boot, secure updates and anti-rollback mechanism, isolation between trusted and untrusted parts, the possibility of interaction between those isolated parts, secure storage and the existence of a set of trusted services that are required to contain cryptographic functions[53].

While the document makes no mention of a trusted execution environment, we can compare the above goals with our criteria for a TEE and see the extent to which they overlap. The objectives of a secure boot and secure storage exist verbatim in both. The responsibilities of the bootloader in PSA's case have been extended to feature the ability to upgrade to newer images while preventing downgrades. Our requirement of inter-environment communication has been worded to "Interaction across isolated boundaries", which carries the same meaning. The combination of the goals of unique identification and attestation lay the foundation to enable remote attestation in the TEE case. The security goal of isolation indicates the need for a secure processing environment.

As we can see, both our TEE definition and the list of PSA's security goals aim for a similar type of security architecture. The stated goals encompass core attributes of a TEE to the point where an implementation that meets all of its demands aligns closely with the characteristics of a TEE — as we will discuss in the next chapter.

2.3.2 Threat Models and Security Analyses

To aid in the aforementioned analysis step of its framework, PSA provides threat model and security analysis documents that cover specific examples of IoT devices. As of time of this writing, the collection of documents covers five use cases: an asset tracker[14], a smart water meter[68], a network camera[41], a smart speaker[67] and a smart camera[66]. In §6.1, we will decide on one of these examples to guide our own security analysis.

2.3.3 Certified Levels

With the end of PSA's certification process, vendors receive a certificate in the form of a PSA Certified Level. These levels range from one to three and make different claims of security assurances. The first level demonstrates that best security practices in accordance to the 10 security goals were followed[54]. PSA Certified Level 2 builds on the first level and includes an evaluation in a laboratory to showcase that the product can protect against remote software attacks[55]. Only the third level includes tests to prove "substantial protection" against hardware attacks[56].

2.3.4 Firmware Framework

The primary document that standardizes PSA's isolation goal is the Firmware Framework specification[11]. It specifies how the Secure Processing Environment should be implemented, with the Root of Trust sitting at the foundation of everything. In Figure 2.3, we can see how this specification builds its components on top of its Root of Trust — similar to the way we have mentioned in the chapter on TEEs.

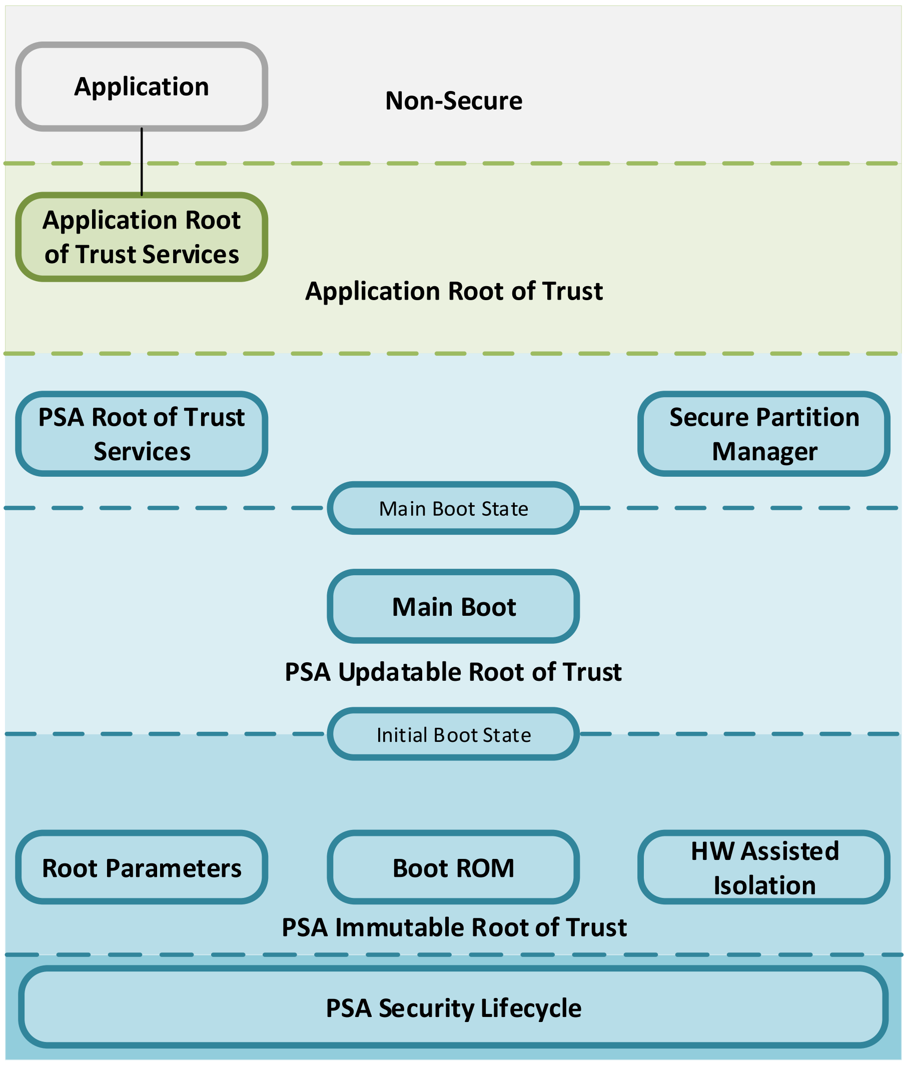


Figure 2.3: PSA Firmware Framework System Architecture, from [11]

In addition to the documents, the PSA also provides reference implementations for some of their specifications. The reference implementation for the Firmware Framework and thus the Secure Processing Environment is called TrustedFirmware-M (TF-M)[74].

2.4 TrustedFirmware-M

TrustedFirmware-M (TF-M) is the PSA certified reference implementation of the Secure Processing Environment targeting M-profile Cortex processors, starting with ARMv8-M[74]. It is an open source project that was released in version 1.0 in 2020 and is designed and maintained by IoT security professionals and claims to implement best practices in cryptography, device attestation, secure storage and secure boot. While the current implementation relies on TrustZone-M for hardware isolation, it is TEE agnostic in theory and could be implemented with any isolation mechanism[75]. Its goals include the minimization of production efforts, streamlining of the integration process into projects and attaining the benefits of security by scale[22] — meaning as TF-M becomes more common, vulnerabilities will be discovered and addressed. These are on the whole the same goals of the previous project *TrustedFirmware-A* (TF-A), which is ongoing that precedes TF-M by several years[23] and implements an SPM on the basis of TrustZone-A[73].

2.4.1 Root-of-Trust

At the core of TF-M's security lies its Root of Trust architecture, which is based on the PSA design as seen in Figure 2.3. The component that is absolutely trusted is referred to as the PSA Immutable Root of Trust, which contains root keys, the secure boot and the reliance on hardware-based isolation mechanisms. On top of it, the PSA Updatable Root of Trust is built. As the name infers, this part contains elements of the secure world that can be changed once the IoT device is deployed, e.g. over a firmware update mechanism using TCP/IP. Continuing the Root of Trust's hierarchy, we see the Application Root of Trust; a less trusted domain that is nonetheless part of the secure world. The hierarchy ends with it and transitions into the non-secure world. Here all components are by definition untrusted and it is where the real-time operating system (RTOS) would exist in most cases.

In the PSA way of understanding, the trust level of all these domains depends on the device's current phase of its security life cycle — which is why it is positioned under all other elements. For instance, parts of the PSA Immutable Root of Trust can change when the device is still in its development phase, since the bootloader will typically be flashed onto it during this process. This, however, changes when the device enters its deployment phase, where the bootloader should never be able to change.

In TF-M's implementation, the bootloader used to satisfy the secure boot requirement is *MCUboot*[37]. It verifies the secure and non-secure images before loading them using the process that we outlined in §2.1. In addition to this, it can be enabled to provide roll-back protection and a firmware update mechanism.

2.4.2 Secure Partitions

Inside Figure 2.3, we also encounter so-called Root of Trust services. We briefly mentioned these services in §2.3 within the context of PSA's 10 security goals and in §2.1 regarding the TEE requirement of inter-environment communication. In essence, the Root of Trust services are interfaces that facilitate the interaction between the non-secure and secure world. They come in the form of PSA Root of Trust (PRoT) services and Application Root of Trust (ARoT) services, which are less trusted than their PRoT counterparts. These secure services in turn live inside of secure partitions, which determine their isolation and access to outside resources[44].

The currently provided PRoT services in TF-M are Audit Logging (logging of critical system events), Firmware Update, Initial Attestation (proving of the device's identity), Platform (interaction with platform-specific components), Internal Trusted Storage (storing of secrets such as cryptographic keys) and Crypto (providing secure cryptographic services). The only provided ARoT service is Protected Storage, which allows the secure storage of larger data than its Internal Trusted Storage equivalent. However, custom ARoT services can be created by the developer using TF-M in a project, which will prove useful for our purposes later on.

To coordinate the communication between these services, as well as between the NSPE & SPE in general, the Secure Partition Manager (SPM) is introduced as a fundamental component[11] — the responsibilities of which we will discuss later in this chapter.

2.4.3 TEE Terminology

Before continuing with the architecture of TF-M, in order to clear any confusion, Table 2.2 shows differences in TEE terminology that we have identified between the GlobalPlatform standard, the literature and TF-M. Some of the examples do not translate one-to-one, but carry the same meaning in the TEE context, while others can be considered fully synonymous. It should be of course noted that the language around TEEs in the literature is not uniform; most works that we have found tend to comply with GlobalPlatform's naming scheme for the various concepts. The table captures some terms where academic works deviate — which primarily stem from the work of Sabt et al.[60], on which we based many of our TEE requirements. Especially noteworthy is the difference between the term "Trusted Execution Environment" itself. In the GlobalPlatform's way of understanding, the existence of a trusted execution environment implies the existence of all other components outside of it, while other sources see a secure environment, in which trusted execution can take place as only a part of the TEE and instead refer to the entire system as a trusted execution environment.

GlobalPlatform[20]	**Literature**[60]	**TrustedFirmware-M**[74]
Trusted Application	Trusted Service	Secure Service
Trusted Kernel	Separation Kernel	Secure Partition Manager (SPM)
Trusted Storage	Trusted Storage	Secure Storage
Trusted UI	Trusted UI	*no equivalent*
Execution Environment (EE)	Execution Environment (EE)	Processing Environment (PE)
Trusted EE (TEE)	Secure EE (SEE)	Secure PE (SPE)
Regular EE (REE)	Non-Secure EE (NSEE)	Non-Secure PE (NSPE)
TEE Client API	Inter-Environment Communication	Interaction across Isolated Boundaries

Table 2.2: TEE Terminology Differences

2.4.4 Architecture

With all the elements introduced and clarifications out of the way, we can now look at the PSA Firmware Framework architecture — and thus TF-M's architecture that complies with this design — in its entirety in Figure 2.4.

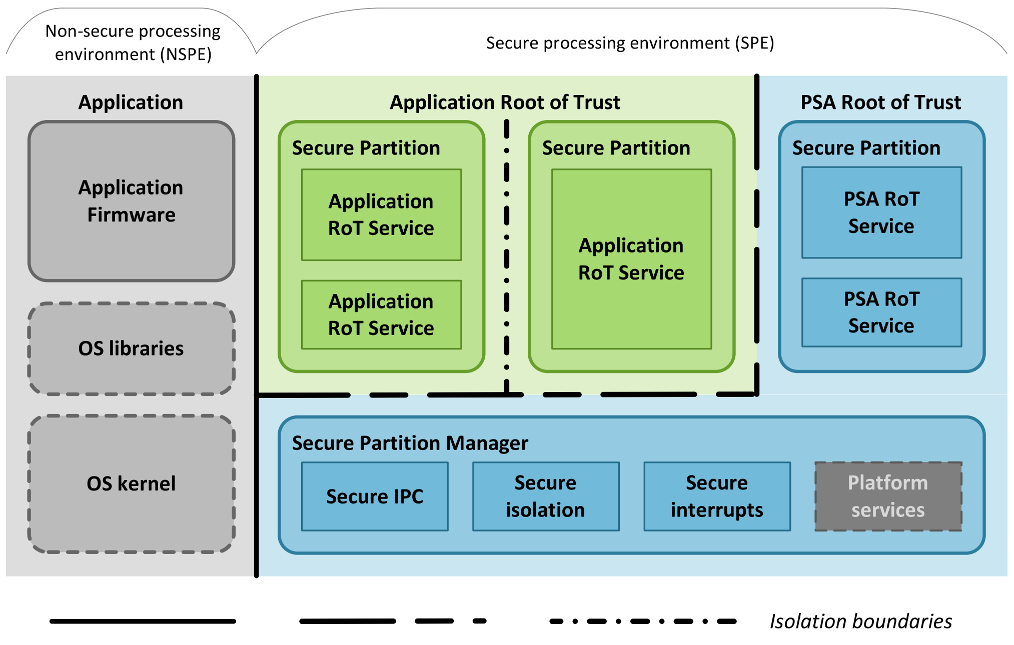


Figure 2.4: Architecture of the PSA Firmware Framework and TF-M, adapted from [11]

We rely on the PSA Immutable Root of Trust to set up this architecture at booting time, after which its responsibilities are mostly carried out. Thus, for the sake of simplicity, it is left aside in the figure. Other than that, we encounter the mentioned components and now see what the Secure Processing Environment is made up out of. To go into more detail, we will explain the depicted isolation boundaries and the SPM's secure IPC mechanism.

2.4.5 Isolation Level

While TrustZone-M only provides hardware support for the separation into two worlds (the secure and normal world), the PSA standards include designs for additional separation inside the secure world[11]. These separations are achieved in software by the Secure Partition Manager seen in 2.4 and constitute a form of runtime isolation. The degree to which they isolate secure components from each other can be configured by the developer. These options are called isolation level and range from one to three. In Figure 2.5, we see the different levels and their impact on the secure world illustrated.

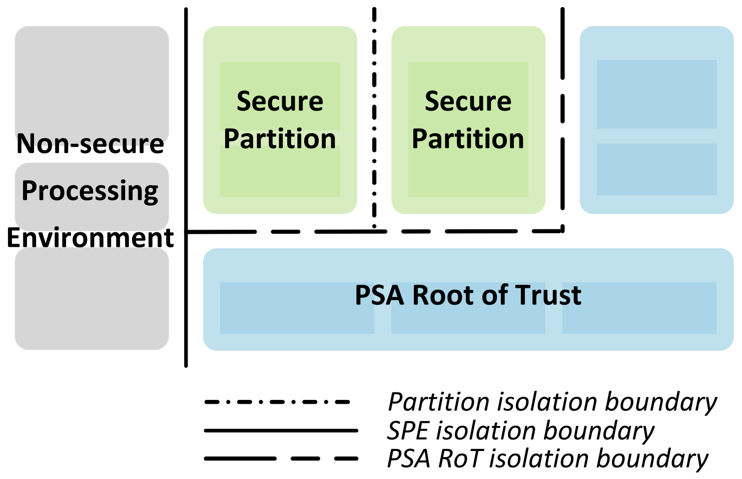

Non-secure Processing Environment
Secure Partition
Secure Partition
PSA Root of Trust
Partition isolation boundary
SPE isolation boundary
PSA RoT isolation boundary

Figure 2.5: PSA Isolation security levels, adapted from [11]

The first isolation level describes the SPE isolation boundary. This is the major separation between the SPE and NSPE, which in the current case of TF-M is implemented using hardware that supports TrustZone-M. In this configuration, there is no further separation inside the secure world, meaning there is no distinction to be made between the PRoT and the ARoT. This distinction instead is introduced in Isolation Level 2 with the PSA RoT isolation boundary. With this level active, an ARoT secure service has its access to PRoT services restricted to public APIs. The final and highest isolation level extends this measure by creating a boundary between one ARoT service and another. While it provides the strongest separation, it comes at the cost of overhead and possible code duplication[85].

2.4.6 SFN and IPC Mode

To allow code in secure partitions to communicate with each other and to enable the non-secure world to access the secure services provided by these partitions, PSA requires the SPM to implement a secure inter-process communication (IPC) mechanism[11]. With this mechanism, the inter-environment communication is not architecture dependent — meaning TrustZone-M, physical separation of processors or any other isolation mechanism could be used in conjunction with IPC[45]. Its downsides however, include an increase in complexity and more overhead in performance and memory requirements[45].

TF-M's design introduced an optional second approach to this problem: the library model or secure function (SFN) mode. Here, secure services are implemented as functions, which creates less overhead but couples the software to the ARMv8-M architecture, i.e. TrustZone-M. Compared to IPC mode, it also has no support for isolation levels higher than the first[44]. This mode proved beneficial, but was not part of the PSA Firmware Framework specification, which prompted — among other things — an extension to the specification[10].

2.4.7 Integration

As mentioned in the beginning of this chapter, one of TF-M's goals was the adoption of it through widespread integration into existing real-time operating systems. Consulting the list of official PSA certified software, we can see popular operating systems such as Mbed[8], FreeRTOS[18] and Zephyr[84] relying on TrustedFirmware-M to achieve PSA Certified level 1[57]. In the case of ARM's Mbed, there is little documentation on its TF-M integration to be found other than the claim that "the PSA support is based on a specialized TrustedFirmware-M implementation"[8]. FreeRTOS offers some explanations of its TF-M usage in the form of blog posts[63, 62]. Out of these three, Zephyr offers the most extensive documentation along with sample projects that test various of TF-M's features. Thus, we have chosen to use TF-M in conjunction with Zephyr for our purposes.

2.4.8 TF-M as a TEE

It should be noted, that there are conflicting information from the official sources on whether TF-M provides a TEE by itself[7], or if it is only the foundation on which developers can build a trusted execution environment upon[7, 22].

This confusion probably stems from two sources: On one hand, the TEE secure boot requirement is met by MCUBoot, which is integrated into the TF-M project, but is nonetheless considered a separate project under the Trusted Firmware name. Further, TF-M is designed to be integrated into the workflow of an existing project — namely an existing RTOS. On the other hand, the term "Trusted Execution Environment" has also found use referring to just the Secure Processing Environment itself — as we have previously touched upon in regards to existing TEE terminologies.

Nevertheless, we can state with confidence that a project using TF-M to establish an NSPM that contains the RTOS, which is isolated from the SPM that is guaranteed to be given first execution from a secure bootloader such as MCUboot can be considered a Trusted Execution Environment.

3 Related Works

As we have mentioned, TrustZone remained the label for the TrustZone-A technology on the whole. This complicates the research process, since even recent works that seemingly talk about TrustZone in general actually are only applicable to TrustZone-A devices and use cases. Nevertheless, in this chapter we will look at previous works that have tried to isolate peripherals inside a TEE to achieve various security goals.

Lentz et al.[31] propose SeCloak, a system specialized to provide on-off control over a smartphone's peripherals, such as the camera or its microphone. Their implementation is based on a fork of OP-TEE[43], a popular TrustZone-A based TEE, and their use case is primarily concerned with preserving the privacy of users by giving them confidence that a turned off peripheral device is truly off. Hence, their work is focused on providing a secure mechanism to manage the availability of peripheral data, while we endeavor to make claims about maintaining data integrity. For their future work, they mention the prospect of extending the system to allow for more than just binary (on or off) peripheral control.

Related to this, Brasser et al.[17] implement a form of reliable remote on-off control of peripherals for use in restricted spaces. They mainly envision a scenario where personnel is granted access to premises containing confidential information, e.g. a government facility, while at the same time being unauthorized to take video footage or make audio recordings using their personal computing devices. Their system can give assurance that these guests cannot access their device's peripherals by securely turning them off after entering a specified location. Their proposal is based on a TrustZone-A device that communicates with a policy server that enforces the on-off rules of peripherals by having direct remote access to the device's memory. Once again, the work revolves around control of the availability of peripheral data — this time extended to remote control.

Yuhala [83] aims to address privacy issues in IoT smart home systems by isolating specific peripherals and their drivers into a TEE and requiring any captured data — such as video by a camera — to be filtered of any sensitive data using machine learning techniques. However, their proof-of-concept implementation goes into no specifics as to how they achieve peripheral isolation.

Salman and Du[61] provide a proof-of-concept to achieve data integrity of peripheral data using OP-TEE. They focus their attention on the use case of Quick Response (QR) payment systems, where the integrity of the merchant's QR code is vital to assure that the payment goes where it is intended to. Linked to this, they present a way to preserve the integrity of location data that can be used in payment verification. The integrity is maintained from the moment of data capture to the delivery to a destination server. This work exemplifies that integrity of peripheral data using a TEE is achievable.

Liu et al.[33] present two software abstraction for Trusted Sensors, which they define as sensors that can make guarantees of their data integrity. Their first abstraction is called sensor attestation and involves the signing of sensor data using a key that is part of the TCB. Since they use the same key for Remote Attestation, they also achieve authenticity of the sensor data. The second abstraction requires the encryption of the sensor data in a process they call "sensor seal". In addition to the sensor data, a policy is contained in the ciphertext that enables a mechanism to decrypt the data and hand it over to the non-secure world, provided that the requirements set by the policy have been met. The authors also consider the need to process some peripheral readings locally. However, they do not provide an approach to achieve sensor data transformation outside the TEE — meaning the processing code needs to live in the secure world and thus inflates the TCB. They have applied their framework on both x86 and TrustZone-A platforms and demonstrate minimal overhead in performance. However, they exclude physical access of an attacker in their threat model. Their implementation also aspires to achieve privacy through plausible deniability by leveraging differential privacy — an approach that involves the deliberate injection of uncertainty into a larger data set that can later be compensated for mathematically.

Building on the previous work, Janjua et al.[27] provide their own framework for trusted sensor data. The aforementioned calls to sensor attestation and sensor seal are combined into one step in their solution. Contrary to the previous work, the researchers here also claim to support arbitrary operations on sensor data from the secure world while maintaining the chain-of-trust, which would be a desirable property for such an API. However, they are unclear about the precise implementation of this mechanism and works that reference the paper are unconvinced of this claim[61]. Despite this, we will try to build on their design in §4.5, extending it to optionally provide the guarantee of confidentiality at the cost of flexibility.

Although their implementation is based on a Raspberry Pi 3, which cannot be considered representative for an actual deployment scenario by their own admission, they anticipate the usage of their framework in the context of participatory sensing — a crowdsourcing approach where sensor data is collected by the devices of various people moving around in an environment. In this envisioned setup, a smartphone containing a TrustZone-A based TEE would securely receive data from a sensor node. These sensor nodes are in fact IoT devices in themselves, which are able to hold keys and encrypt their own sensor data before sending them to the smartphone. In the work, these devices have no trust considerations, since the authors exclude the possibility of the sensor node itself being compromised in their threat model. Instead, they are assumed to have their own security mechanism to protect their data and cryptographic keys. This raises the question: what if the sensor nodes cannot be trusted in the first place? To a certain extent, our work can be understood to close this gap.

4 Design

In this chapter, we present our design to achieve trusted peripherals inside a TEE. To begin, we first construct our general threat model in §4.1. We then provide a definition of trusted peripherals in §4.2, on the basis of the three most common use cases that we have found. We examine these three scenarios under the constraints that we have put forward in our threat model, resulting in designs in §4.3, 4.4 and 4.5 that adhere to our security goals. Lastly, in §4.6 we are going to discuss alternative designs that we encountered and explain why we ultimately opted not to make use of them, before we present how our final design decisions have achieved the goals that we put forward in our introduction in §4.7.

4.1 Threat Model

Our general use case is the secure capturing of peripheral data in untrusted environments, followed by the secure delivery of said data to to an external entity in the form of a trusted remote machine. Here, we identify our peripheral data as the primary asset to protect. Following from this, all code that interacts with the peripheral device during the data capturing process, as well as the communication channel that carries the data to its destination, require guarantees of protection.

The main component of our system that we absolutely trust to be secure is the device's trusted execution environment. The most security relevant parts contained within it for us are its secure boot process, secure storage facilities and the Secure Processing Environment. Related to this, we trust the hardware mechanisms that are involved in implementing any of these elements to be secure, as well. Specifically, these entail immutable memory for secure booting, attacker-inaccessible, tamper-resistant flash and RAM for secure storage, and a secure operation of the isolation mechanism enabling the Secure Processing Environment — in our case, TrustZone-M. Any cryptographic keys that are stored on the device are assumed to be safe by making use of the TEE's services, or by relying on device-specific features, such as One-Time-Programmable (OTP) memory containing, e.g., public keys. Additionally, keys that are held by the remote server are trusted to secure. This set of hardware and software is considered to be our Trusted Computing Base (TCB) and any attacks targeting these security assets are thus excluded from the threat model.

Potential adversaries in our scenario come in two forms: Individuals in the environment with physical access to the device and remote attackers. We expect both groups to aim to access or manipulate the data — their motivations to do so being of no interest.

The main attack surface for them consists of all code and data inside the untrusted world of the device and all communication that takes place between it and its remote target. This means that we consider the non-secure application to be compromised, as well as the communication channel to be under surveillance. From these follow the concrete threats of data interception and data tampering.

Our main security objective that guides our design considerations is the maintaining of peripheral data integrity. Connected to this, we mandate the authenticity of the data to be upheld, meaning any fabricated data by a non-authorized entity will be exposed as such. Further, we aim to provide confidentiality of the data, if possible. However, we consider the availability of our peripheral data to not be a hard requirement in any case. The reasoning stems primarily from an attacker's physical access to the device, as well as the assumption of a compromised OS in the untrusted world. In §4.6, we consider designs that appear to promise guaranteed availability, but, for the purposes of this threat model, the continuous availability of peripheral data is never given.

To achieve the security goals that we have narrowed down on, we will consider three situations with differing requirements in §4.2. We look at the concrete processes surrounding them in §4.3, 4.4 and 4.5 and elaborate on how our design mitigates the threats that we have stated here.

4.2 Definition: Trusted Peripheral

To arrive at a definition for trusted peripherals, we can think of following three scenarios that encapsulate the different life cycles peripheral data can have:

1. The data needs to be captured and delivered as securely as possible.
2. The data needs to be captured, seen by the non-secure world, and delivered as securely as possible.
3. The data needs to be captured, processed in some way, and delivered as securely as possible.

The term "as securely as possible" is still vague, so we need to ask ourselves more precisely: What guarantees to security can we make to each scenario in relation to common security goals regarding our introduced threat model?

As we have explained in §4.1, the availability of peripheral data in any case can not be given. Hence, the achievement of this goal is preferable, but not an obligation.

The security traits that we instead focus on are the confidentiality, integrity and authenticity of peripheral data. However, our definition will not specify any mechanism by which these traits are achieved. Instead, we will go into specifics for these in §4.3, 4.4 and 4.5.

With these disclaimers out of the way, we can look at the first scenario from above and see what security traits we can maintain in the chain-of-trust. We will call this scenario

Trusted Delivery, and it composes of the following steps:

D1. Sensor data is securely captured.

D2. Sensor data is given to the untrusted world with guarantees to its confidentiality, integrity and authenticity.

D3. Sensor data is delivered to a destination, maintaining confidentiality, integrity and authenticity.

Since in the case of trusted delivery, the untrusted world does not need to see or process the data in any way, we can make guarantees to its confidentiality (e.g. through encryption of the data). Guarantees to the data's integrity and authenticity can, e.g., be made by hashing the data and signing the hash with a private key that we know belongs to the trusted world in step D1, before passing this data packet to the non-secure side in step D2, which issues the transmission of encrypted sensor data along with verification data in step D3.

An example where this scenario applies, would be any situation where peripheral data needs to be simply forwarded to a server that verifies its authenticity and integrity and also is able to decrypt the data to, e.g., store it in a secure database.

The next scenario that we will look at is the second scenario from above. We will call this scenario **Trusted Capture** and it consists of the following steps:

C1. Sensor data is securely captured.

C2. Sensor data is given in plain text to the untrusted world, but with guarantees to its integrity and authenticity.

C3. Sensor data is optionally inspected by the untrusted world and the result of the inspection can determine if the next step happens or not.

C4. Sensor data is delivered to a destination, maintaining integrity and authenticity.

In this case, we can no longer make any claims about the confidentiality of the data, since the untrusted environment has had access and the chain-of-trust has been broken in this regard in step C2. However, since the untrusted environment does not require to make changes to the data, we can maintain the integrity and authenticity all throughout. Also, the untrusted world can decide not to perform step C4, depending on a predicate that it has.

For instance, this could be the case when sensor data should only be transmitted when, e.g., a certain value threshold is exceeded — the rationality possibly being the preservation of power on the IoT device or the desire to not overload the target server. Performing this check inside the trusted environment is also conceivable, but comes with drawbacks that we elaborate on in §4.6.

The last scenario that we cover is possibly the most difficult to implement and the one most open to variation. We call this scenario **Trusted Transformation** and its steps are as follows:

T1. Sensor data is securely captured.

T2. Sensor data is given to the untrusted world with guarantees to its integrity, authenticity and /optionally/ confidentiality

T3. Sensor data is processed or transformed in a way that maintains integrity and authenticity and optionally confidentiality

T4. Sensor data is delivered to a destination, maintaining integrity and authenticity and optionally confidentiality

This means that when the trusted transformation scenario is determined to be needed for a trusted peripheral to serve its purpose properly, an implementation needs to offer a way for transformations of the data to take place in step T3 without compromising on the integrity and authenticity.

Optionally, the confidentiality can be maintained throughout the transformation process and beyond. This circumstance could occur when the untrusted world knows that a transformation of the data needs to happen, but still does not need to inspect the data. For example, one could imagine a scenario where the untrusted world receives photos from a trusted camera peripheral in step T2 — either in encrypted form or as a handle. The untrusted world can then decide that any faces in the picture need to be censored *without knowing whether there are any faces in the photo at all.* The untrusted world then informs the TEE, which will perform the face-censoring transformation while maintaining the integrity, before it encrypts the data in step T3 and hands it back to the untrusted side to perform step T4.

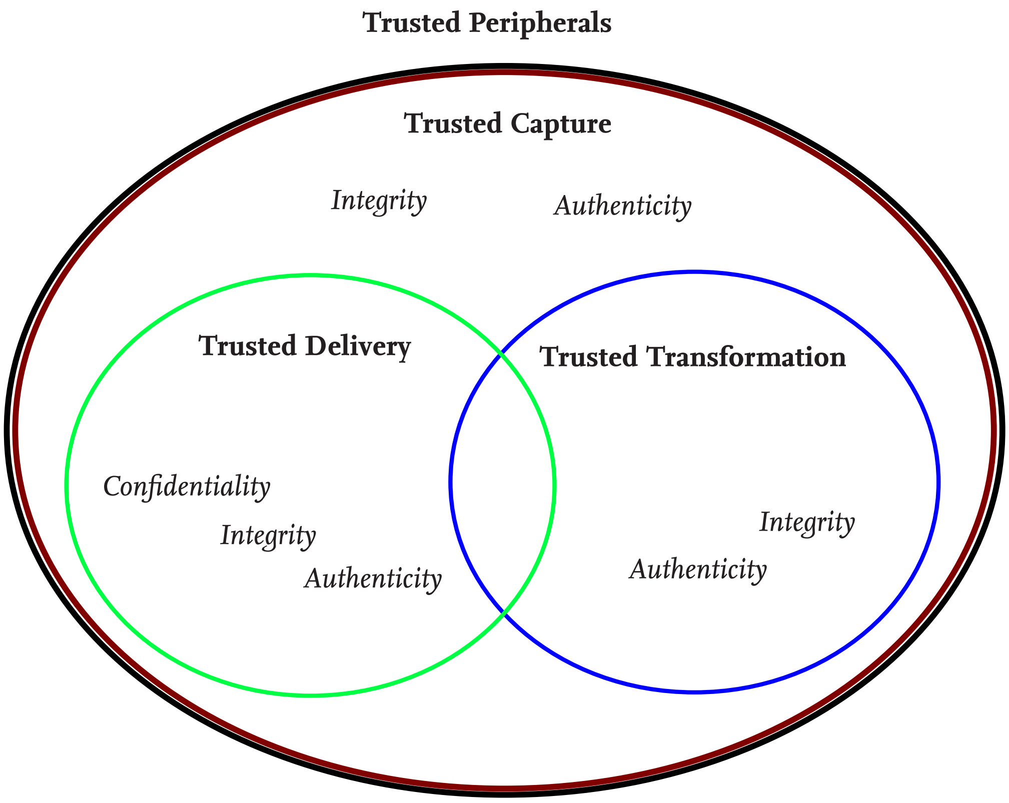


Figure 4.1: Possible Scenarios a Trusted Peripheral can support

In Figure 4.1, we can see the three scenarios for a trusted peripheral illustrated. It shows what security traits can be achieved for a trusted peripheral in every scenario. The most restrictive scenario can gain the most traits: The trusted delivery. The overlap between it and the trusted transformation represents the optional mode of operation that we have hinted at. In our definition, a peripheral needs to at least support the trusted capturing of data to be considered a trusted peripheral.

In the following chapters, we will explain each scenario in more detail and provide concrete mechanisms by which we achieve the possible security goals for the entire life cycle of the peripheral data.

4.3 Trusted Capture

The simplest scenario, where we consider a peripheral to be trusted, is when it supports the trusted capturing of sensor data. The diagram in 4.2 illustrates this process. To initiate trusted capture, the non-secure world in the Non-Secure Processing Environment (NSP) calls a function that is offered by the trusted world in the form of a secure service. This function passes execution to the Secure Processing Environment (SPE), where access to

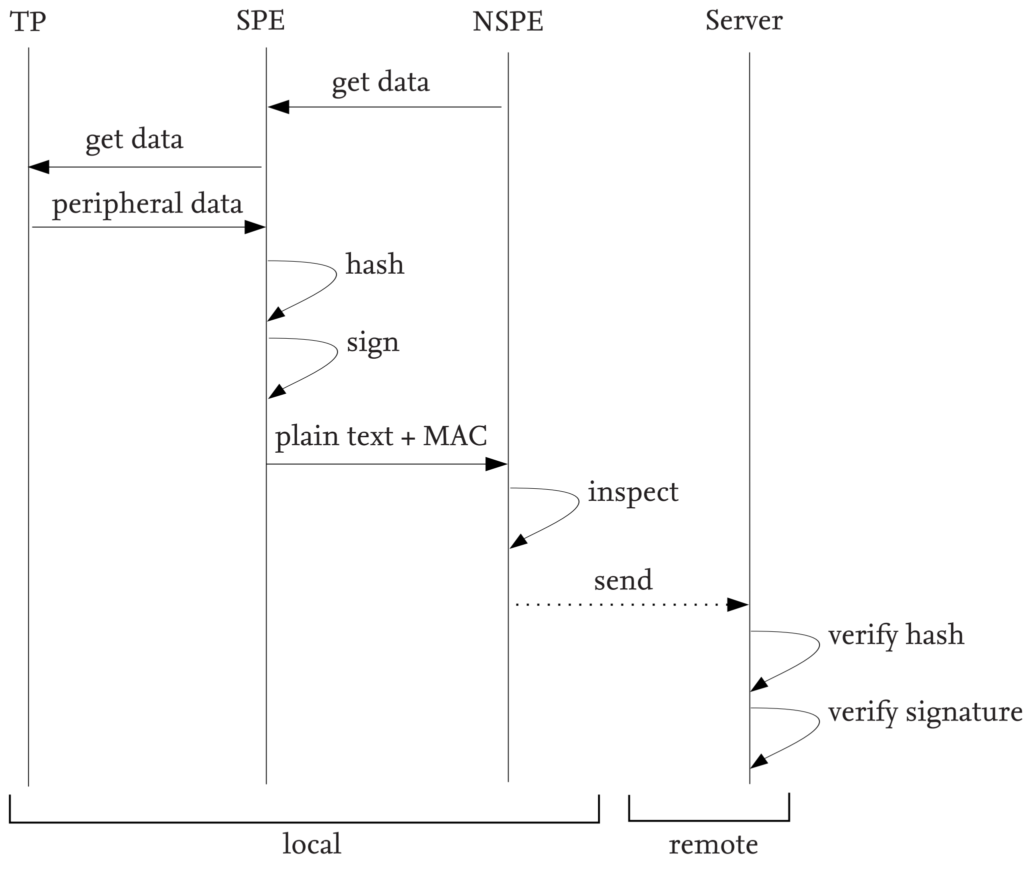


Figure 4.2: Diagram of the trusted capturing process

the trusted peripheral (TP) is granted. From here, peripheral driver code can be used to communicate with e.g. a sensor to capture a reading.

The trusted world then aims to generate a message authentication code (MAC) from the sensor data by hashing it with a cryptographically secure hashing algorithm and then signing the output digest with a private key that is stored securely on the device and can only be accessed by the trusted world.

This MAC is returned to the untrusted world along with the data in plain text — breaking any form of secrecy in the process. The NSPE can then inspect the data and decide, based on the application developer's discretion, whether this specific data packet should be sent to a target computer or not. The optionality of this step is illustrated in 4.2 with a dashed line. The destination could be any machine — local or remote —, including another IoT device. In our explanations, however, we will narrow it down to a remote server. If the application decides to send the data, it will be received by the server, which then verifies its integrity and authenticity by computing the digest, decoding the signature and checking the two for equality. The public key to perform this verification is assumed to be owned by the destination. We will not go into details about this, but a way to achieve this in practice could be through a corresponding key distribution during the provisioning process of the device at the manufacturer or with a key exchange protocol.

It should be mentioned that the above decision of sending or dropping the data is unique to the trusted capturing process. Although we make no guarantees to the availability of peripheral data, the trusted capture scenario is specifically allowed to never send the data.

As we have stated earlier, this is a design choice that is intended primarily for IoT purposes in order to save on power in the form of fewer transmissions. One could also imagine an application where a remote server sees itself overloaded with too many inconsequential sensor readings from various IoT devices in deployment. To counter this, it could reconfigure the application remotely to only send data that meets a certain requirement. This reconfiguration can happen without making changes to the trusted part of the firmware image — a property that would be unique to trusted capturing.

The security traits that are thus achieved by this process are integrity and authenticity of the data.

4.4 Trusted Delivery

The trusted delivery process that is depicted in Figure 4.3 is mostly identical to the previous trusted capturing process in Figure 4.2. The only exception is the inclusion of an encryption step inside the trusted world, which in turn makes the inspection step inside the SPE impossible. This encryption is mandatory and results in the untrusted world only receiving the peripheral data as ciphertext. The encryption is done with the help of a public key that is stored securely on the device using the TEE's secure storage mechanism.

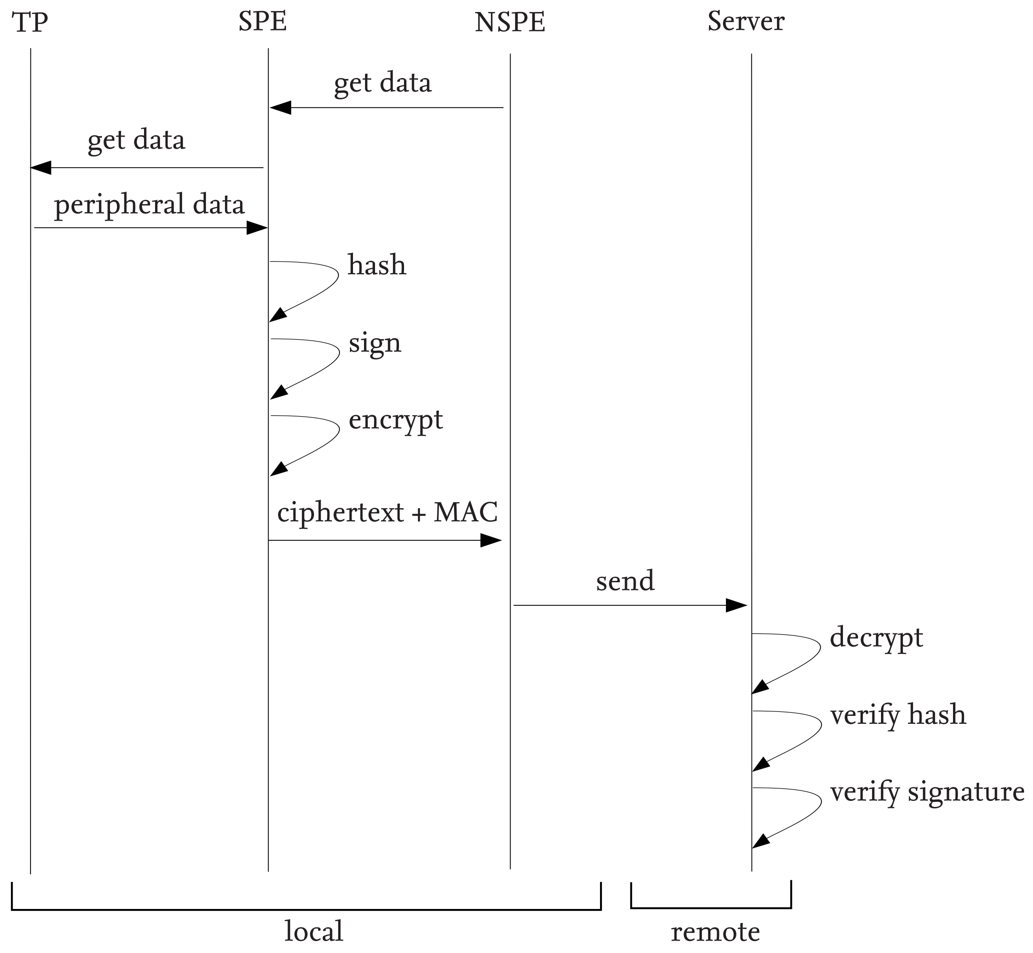


Figure 4.3: Diagram of the trusted delivery process

The private key counterpart is stored on the remote server, making the decryption of the data impossible by anyone except the target machine. A successful delivery thus gives the assurance of confidentiality, coupled with guarantees of integrity and authenticity from the previously discussed mechanisms.

In this design, the NSPE is expected to send the encrypted data to the target at some point. If the application fails to transmit, it indicates the loss of the data's availability to the server — either by an attacker or otherwise.

Hence, trusted delivery is not equipped to facilitate the selective dropping of data packets, motivated by the goal of power savings or other reasons. In §4.6, we present a design that could enable this mechanism and elaborate on why we consider it to be impractical. In any case, we consider the ability to decide to selectively ignore sensor readings to be exclusive to the trusted capturing process.

In conclusion, the security traits that are achieved by the trusted delivery design are confidentiality, integrity and authenticity throughout the entire chain.

4.5 Trusted Transformation

Our last trusted peripheral scenario is shown in Figure 4.4 and is the most involved out of the three, but shares most of its design with the other two. This design shares resemblance with a previous proposal for trusted operations in the literature[27], but includes the option of preserving confidentiality of the data while maintaining the option for data transformations. This option comes in the form of returning only a data handle to the untrusted world, instead of the actual data. Any operations in the figure that are associated with this mode are depicted with dashed lines. First, we will consider the version that passes plain data to achieve trusted transformations. These transformations are the first novelty in the diagram compared to the previous processes. Equipped with the data and MAC, the application can call the SPE again and request it to perform a transformation on the data from a list of predefined operations. Doing so, the trusted world will take the data and verify its integrity and authenticity with the MAC that was passed along with it. With the data at hand, the TEE now executes the desired operation on it. To keep track of all changes to the data, any modification causes the appending of a transformation identifier to a list inside the TEE. This list will be attached to the peripheral data. Any modified data goes through the same hashing & signing process as after the initial capturing. The new data is then returned to the NSPE, which can choose to issue more transformations. The eventual delivery of the data to the target is once again expected to occur.

When operating in handle mode, steps are added in between and only a handle to the data is returned to the NSPE. The additional steps mainly come from the assigning and resolving of the handle by the secure world. Here however, before the untrusted client can send the data to the remote target, it needs to invoke a special transformation that

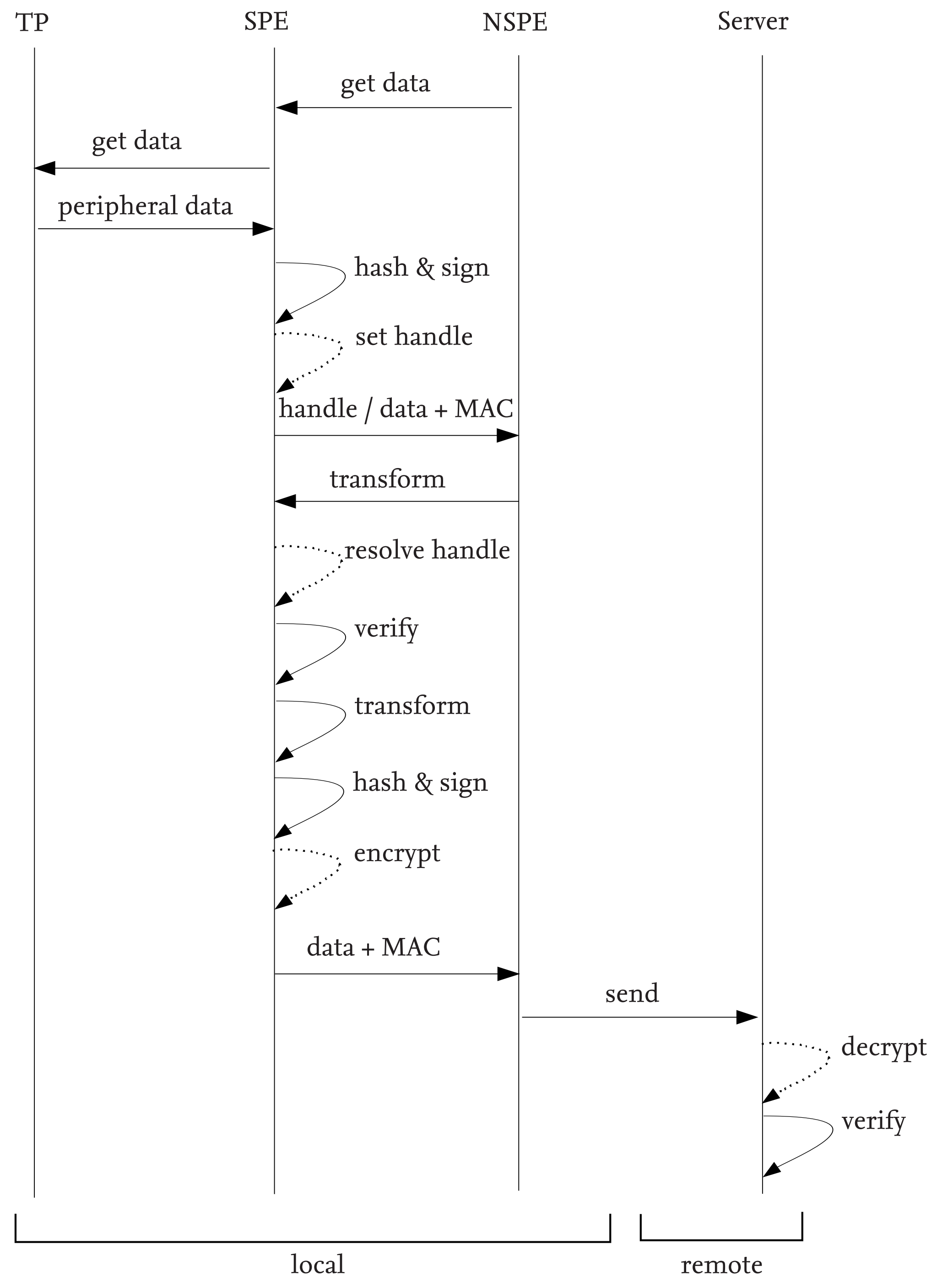


Figure 4.4: Diagram of the trusted transformation process

results in the return of the data in encrypted form. In essence, the NSPE exchanges its handle for a ciphertext and MAC, which it then can transmit to the target. The intended sequence of interactions for the non-secure side thus are to receive the handle at first, invoke transformations on it that change the data inside the secure storage of the trusted world, and then swap the handle for the encrypted data. When the data is then delivered to the remote machine, it can know with certainty that confidentiality has been preserved, since the untrusted world has never seen the contents of the data.

Whether the transformations can be reversed at the destination after delivery is dependent on the requirements of the developer. If the complete reconstruction of the data to its state right after capturing should be possible, any possible parameters of the specific transformation need to be appended along with the identifier. Though this should come with considerations for the security, since these arguments have been passed into the TEE by the non-secure world and could have been tampered with by an attacker. In any case, the target server can still reconstruct the original data after applying the inverse transformations. In essence, a minimal form of version control for the peripheral data needs to be implemented to support this scenario.

An example where this would come in use, could be a temperature sensor that delivers temperature measurements in degrees Celcius, while the target machine is hard coded to require values in Fahrenheit. The untrusted application could issue a conversion before sending the data. On the other hand, if the target machine then changes the expected unit to degrees Celcius, it can use the list of transformations to understand why incoming values are off.

If no full reconstruction is needed, only the IDs of the transformations are included in order. In §4.2, we have mentioned the example of censoring faces inside a photograph without the non-secure world ever being able to see those faces to demonstrate the use of handles in trusted transformation. It also would be an example where the reversing of the censoring step should not be possible, but the destination machine might still want to have a record confirming that the transformation took place.

Concerning the optional confidentiality through the use of handles: Another possibility to achieve this goal is through the use of local encryption and decryption. In §4.6, we explain how this would work in practice and why we instead opted for the solution using handles.

In §4.6, we elaborate on the downsides of a design alternative where the untrusted world is excluded from any decision surrounding the data transformations. The security goals that can be achieved by our design, however, are integrity, authenticity and optionally confidentiality of the data — at the loss of inspection capabilities by the non-secure world.

4.6 Discussion

Throughout the design phase, we have encountered alternative approaches that could achieve our goals in different ways or appear to offer even greater benefits beyond our initial aspirations. However, we determined them to be ultimately inferior to our design after careful consideration. Here we list these alternatives and the reasoning on why we left them out of our main design.

In §4.1, we excluded the availability of data from our security objectives. The reasoning stems primarily from the physical access property of our use case, which includes attacks such as the disconnecting of cables, encasing of the system into a Faraday cage to stop wireless communication or just simply the destruction of the entire device — just to name a few. These attacks could be partly mitigated by, e.g., a tamper-proof encasing of the device combined with mechanisms that can detect physical intrusion. For the purposes of a secure software design however, these methods are considered to be out of scope.

Apart from the physical access on the pure software side of things, the assumption of a compromised operating system running in the untrusted world also implies an attacker's ability to stall any execution, including the delivery of sensor data to a target — possibly without the attacker ever getting near the device. To address this issue, one could think of moving the entire transmission process into the secure world and take up the task of peripheral data delivery that was previously given to the untrusted world. This appears to give a larger degree of availability in the case of remote attacks.

However, since the addition of, e.g., an entire TCP/IP stack along with ethernet drivers into the Trusted Computing Base would to a large extent defeat the purpose of having a TEE, this approach would result in costs that do not outweigh the benefits in our view. Additionally, the data transmission could still be targeted by malicious scheduling from the compromised operating system running in the untrusted world — effectively starving the trusted environment of execution. Hence, we see the exclusion of this security goal from the threat model as inevitable.

In §4.3, we presented a design that allows for the possibility of checking the contents of peripheral data for the purpose of optionally forgoing the transmission step.

On the other hand, we see the design in §4.4 as obligated to attempt the delivery eventually, on the basis of the untrusted world not being able to know what the data is.

To enable this mechanism, one could imagine a different design: The conditional check that drops the data is instead performed inside the trusted world, with a failed check resulting in the return of no sensor data to the NSPE. There are two downsides to this approach.

Firstly, inspecting the data to control its delivery can be considered code that is specific to the application, thus breaking our set goal of the separation of concerns between developers of trusted code and developers on the untrusted side. Secondly, trying to update the conditional checks remotely would involve relying on an expensive firmware update mechanism. If these updates are frequent, it defeats the purpose of any power savings.

For these reasons, we consider the ability to decide to selectively ignore sensor readings to be exclusive to the trusted capturing process.

In §4.5, we foresee transformations to take place in the trusted world that can be issued by the non-secure world. An alternative could implement these transformation to take place entirely inside the trusted world, excluding the untrusted world from having any say in them. This approach would lead to more code inside the TEE, increasing the size of the TCB. The more aggravating problem however, is the removal of any separation of concerns between the trusted application and the normal world. TA developers in this scenario would not only have to implement the transformations, but also decide when, when not and in what order to perform them, depending on outside conditions. For this reason, we have opted to rely on a mechanism where the non-secure world can specify its own order of transformations.

Another design alternative that we have mentioned in §4.5 was the use of encryption and local decryption instead of handles to achieve confidentiality within trusted transformations. In this solution, the TEE would only return encrypted data to the untrusted world. It can then pass the ciphertext back with a specified transformation, upon which the SPE will decrypt the data and perform the operation. Afterwards it would create the MAC and re-encrypt the data to return it to the NSPE.

Regarding this approach, it should first be noted how the requirements to the encryption have changed from the trusted delivery case. Previously, the local decryption of sensor data was not needed, meaning an asymmetric encryption scheme can be used, where the device is unable to recreate the plain text data out of any encrypted data packets.

This means that in a "Store Now, Decrypt Later" (SNDL)[72] attack, even after the device and all its keys are fully compromised, an attacker cannot decrypt the data packets that he might have acquired in a man-in-the-middle (MIDM) setup. The private key that would be necessary to perform this decryption is instead held by the server, which is assumed to be stationed in a secure environment.

Contrary to this, a device that relied on trusted transformation for the data's confidentiality is now vulnerable to an SNDL attack. This is because for the local decryption to be possible, the device must have stored either the private key of its asymmetric encryption, or relied on symmetric encryption using only one key. For this reason, our solution using data handles can give us more confidence in confidentiality. Concerning device resources, handles come at the cost of higher storage requirements in the trusted world, while encryption and decryption comes at the cost of performance.

4.7 Design Goals

Our design achieves the goals that we have mentioned in our introduction and outlined throughout this chapter. The stated objectives were the protection of peripheral data integrity and authenticity. Optionally, we desired the confidentiality of data to be given, if possible. Any solution needed to preserve the separation of concerns between the trusted application (TA) development and the normal application developers.

We have considered three main scenarios for trusted peripherals and categorized our definition accordingly. Each scenario can make guarantees to a different set of security requirements. However, all three of them maintain the integrity of peripheral data in all cases. This is achieved by the usage of a message authentication code containing a signed digest. Any tampering of the data — encrypted or not — will be detected, as the only entity that can create the signature is the TEE in the device itself. An authorized server is equipped with the corresponding public key. By the same mechanism the authenticity of the data in all cases is maintained.

In the case of trusted transformation, operations on the peripheral data without compromising its integrity are supported. This is done by appending all performed transformations to the data set, updating the MAC securely with every change of the data.

In the case of trusted delivery, the confidentiality of the data is maintained through asymmetric encryption. The device is storing the public key for this inside of the TEE's secure storage facilities, while the corresponding private key for decryption is stored on a destination machine chosen by the developer.

Optionally, trusted transformation can also support a scenario in which the confidentiality is maintained. This is achieved through the use of data handles, the contents of which are stored securely inside the TEE.

None of our designs presuppose the intimate cooperation of TA and normal application developers. Instead, every process can be implemented as a predefined API that the trusted world programmers implement and the developers of the untrusted side can interact with. Thus, the separation of concerns is preserved.

5 Implementation

Based on our design, we present an API in §5.1 that covers all three usage scenarios. We go into implementation specifics for every function of this API in §5.1.1, 5.1.2, 5.1.3 and §5.1.4. Finally, we elaborate on our concrete implementation surrounding employed hardware and software in §5.2 and the considerations around including our API as a secure service in §5.3.

5.1 Trusted Peripherals API

With all our definitions surrounding trusted peripherals at hand, we can propose a simple API that is shown in Listing 1.

```
int tp_trusted_capture(periph_data_t* data_out, mac_t* mac_out);

int tp_trusted_delivery(void* data_out, mac_t* mac_out);

int tp_trusted_transform(periph_data_t* data_io, transform_list_t* transforms_io,
                         mac_t* mac_io, transform_t transform);

/* handle version */
int tp_trusted_transform(tp_handle_t handle_io, mac_t* mac_io,
                         uint8_t* ciphertext_out, transform_t transform);
```

Listing 1: Trusted Peripherals API

The API is implemented by the trusted world and exposes its functionality to the untrusted side. A non-zero return from any function indicates an error code.

In the cases of trusted capture and delivery, pointers to empty buffers for the peripheral data are passed and filled out by the trusted world. The size of these buffers depends on the specific peripheral and are known to both the trusted and untrusted side.

The specific memory layout of the peripheral data however, is known only for the scenario involving trusted capture — as indicated by the `periph_data_t` pointer. After returning from these functions, the NSPE will have the data in plaintext with a MAC (in the trusted capturing case) or the encrypted data with a MAC (in the trusted delivery case).

In the case of trusted transformation, only ever the handle version or the plain text version should be implemented for one trusted peripheral. This is why the two versions here share the same function name with different arguments. In the plain text version, all parameters are input/output parameters, with the exception of the specified transformation. In the handle version, the passed in handle gets initialized at first call, while subsequent calls expect a valid handle. The last call will fill out the ciphertext buffer, hence it is considered an output parameter. Since the untrusted world does not have access to the data, the MAC can be an input/output or just output parameter — in our case we decided on the former as an additional safety guard and sanity check for the application developer. This way, the MAC can be used to verify that the data is in the state that the developer expects.

We now go into further detail regarding what the usage code of every API call looks like and their implementation.

5.1.1 Trusted Capture API

In Listing 2, we see how the API is used inside the untrusted world when trusted capturing is needed. The buffers are allocated by the application developer and filled after return from the secure function. Afterwards, the non-secure side has the option to inspect the data and decide to send or drop the reading.

```
periph_data_t data = {0};
mac_t         mac  = {0};

int ret = tp_trusted_capture(&data, &mac);

if (ret != 0) { /* error handling */ }

/* inspection step */
if (data.value > THRESHOLD_TO_SEND)
{
    /* send data & mac to target */
}
else
{
    /* drop reading */
}
```

Listing 2: Usage code of Trusted Capture API

In Listing 3, we show a possible implementation in the trusted execution environment. Here, the communication with the peripheral takes place and the MAC for verification purposes is generated by hashing and signing the data.

```c
int tp_trusted_capture(periph_data_t* data_out, mac_t* mac_out)
{
    int status;

    /* securely capture peripheral data */
    status = peripheral_capture(data_out);
    if (status != 0) { return status; }

    /* hash and sign the data */
    status = compute_mac(mac_out, data_out);
    if (status != 0) { return status; }

    return status;
}
```

Listing 3: Implementation of Trusted Capture

5.1.2 Trusted Delivery API

In Listing 4, we show example usage code for the trusted delivery API call. The untrusted world once again supplies any buffers to be filled. This time, a ciphertext buffer that can fit the encrypted data is required. In our implementation, we set a compile-time size limit that fits our encrypted sensor readings precisely. This simplifies the implementation and usage code of the API. If ciphertexts with varying sizes need to be supported, the length of it needs to be returned to the untrusted caller and the API may require to be broken up into several calls.

```c
uint8_t ciphertext[MAX_TRUSTED_DELIVERY_CIPHERTEXT_SIZE] = {0};
mac_t   mac  = {0};

int ret = tp_trusted_delivery(ciphertext, &mac);

if (ret != 0) { /* error handling */ }

/* send ciphertext & mac to target */
```

Listing 4: Usage code of Trusted Delivery API

In Listing 5, a possible implementation is presented. Since all steps up to the encryption are identical to the trusted capturing process, its function can be used to simply the procedure.

```
int tp_trusted_delivery(uint8_t* ciphertext, mac_t* mac_out)
{
    int status;
    periph_data_t data;

    status = tp_trusted_capture(&data, mac_out);
    if (status != 0) { return status; }

    status = encrypt_data(&data, ciphertext);
    if (status != 0) { return status; }

    return status;
}
```

Listing 5: Implementation of Trusted Delivery

5.1.3 Trusted Transformation API

For the transformations used in the trusted transformation case, a `transform_t` struct that acts as a simple discriminated or tagged union can be used. The value of a `transform_id` enum inside the struct then determines its memory layout. The first entry of this enum with the value zero can then signal the initial data capture. In Listing 6, we show what these data structures could look like, along with some example transformations. The untrusted world has access to all of these. The list keeping track of all performed transformations can be of any form, but should preferably be contiguous in memory. For our implementation, we used a fixed-length array and set a compile-time limit of the allowed number of transformations.

```
typedef enum
{
    TRANSFORM_ID_INITIAL,                  /* signals initial data capture */
    TRANSFORM_ID_SUBMIT_HANDLE_AND_ENCRYPT, /* for handle version           */

    /* transformations */
    TRANSFORM_ID_CENSOR_FACES,
    TRANSFORM_ID_CROP_PHOTO,
    TRANSFORM_ID_CONVERT_C_TO_F,
    /* ... */

    TRANSFORM_ID_COUNT,
} transform_id;

typedef struct
{
    transform_id type;
    union
    {
        censor_mode censor_param;
        crop_t      crop_params;
        float       convert_param;
    };
} transform_t;
```

Listing 6: Transformation Data Types for Trusted Transformation

The first version of this API is the plain text trusted transformation, where the untrusted world has access to all peripheral data and the list of transformations. However, any modification to either one causes the verification step using the MAC to fail.

A usage example is depicted in in Listing 7. In this case, the passed pointers are only output parameters that contain empty buffers that will be filled by the trusted world. All subsequent calls to `tp_trusted_transform` must then include a filled out `transform_t` struct, containing the operation to be performed along with its parameters. In this example, the untrusted world issues the cropping of a photo. If the developers intend this transformation to be reversible at the target, the structure from 6 could contain a buffer of the cropped out part. A different implementation could use callbacks instead[27].

```
periph_data_t      data      = {0};
transforms_list_t  list      = {0};
tp_mac_t           mac       = {0};
transform_t        transform = {0}; // sets type to TRANSFORM_ID_INITIAL_CAPTURE

/* initial capture */
int ret = tp_trusted_transform(&data, &list, &mac, transform);

if (ret != 0) { /* error handling */ }

/* issue transformation */
transform.type        = TRANSFORM_ID_CROP_PHOTO;
transform.crop_params = { /* ... */ }; /* include buffer if reversible */
ret = tp_trusted_transform(&data, &list, &mac, transform);

if (ret != 0) { /* error handling */ }

/* send data & list & mac to target */
```

Listing 7: Usage code of Trusted Transformation API

The implementation of this scenario is the most complicated out of the three. Our approach is depicted in Listing 8. On first call, a special initial transformation type is expected, triggering the basic sequence of events that would take place for trusted capturing. Here, however, the MAC needs to be computed from the data and the list of transformations. Afterwards, the caller is equipped with data that can be transformed via the predefined operations seen in Listing 6. If such a transformation has been passed, the trusted world first verifies the MAC. Afterwards, a switch-case is used to dispatch between the different transformations. Finally, the transformation is appended to the list and the MAC recomputed.

```c
int tp_trusted_transform(periph_data_t* data_io, transform_list_t* transforms_io,
                         mac_t* mac_io, transform_t transform)
{
    int status = 0;

    if (transform.type != TRANSFORM_ID_INITIAL)
    {
        if (list_is_full(transforms_io)) { return -1; }
    }
    else /* initial capture */
    {
        /* empty the list */
        list_empty_out(transforms_io);

        /* securely capture peripheral data */
        status = peripheral_capture(data_io);
        if (status != 0) { return status; }

        /* hash and sign the data and the transforms list */
        status = compute_mac_tt(mac_io, data_io, transforms_io);
        if (status != 0) { return status; }

        return status;
    }

    /* verify mac regarding data & list integrity */
    status = verify_mac_tt(mac_io, data_io, transforms_io);
    if (status != 0) { return status; }

    switch(transform.type)
    {
        case TRANSFORM_ID_CROP_PHOTO:
        {
            /* perform transformation */
            status = crop_photo(data_io, transform.crop_params);
            if (status != 0) { return status; }

        } break;

        default: { /* unreachable */ };
    }

    /* append to list */
    list_append(transforms_io, transform);

    /* recompute mac */
    status = compute_mac_tt(mac_io, data_io, transforms_io);
    if (status != 0) { return status; }
}
```

Listing 8: Implementation of Trusted Transformation

5.1.4 Trusted Transformation API with handles

In Listing 9, we have written out the use case of censoring faces from a photo that we have described in §4.2 as code. The usage of the handle version of trusted transformation is mostly identical to its plain text equivalent. As an important difference, the calling untrusted side needs to always end its interaction with the API by submitting its handle and receiving encrypted data before it can send anything. This signal to the API is sent with the special transformation ID `TRANSFORM_ID_SUBMIT_HANDLE_AND_ENCRYPT`. Similar to the trusted delivery case, a buffer containing the ciphertext needs to be supplied by the non-secure environment. As long as normal transformations take place, the buffer can be passed as `NULL`.

```c
tp_handle_t handle;
mac_t       mac       = {0};
transform_t transform = {0}; // sets type to TRANSFORM_ID_INITIAL_CAPTURE

int ret = tp_trusted_transform(handle, &mac, NULL, transform); /* sets handle */

if (ret != 0) { /* error handling */ }

/* issue transformation */
transform.type        = TRANSFORM_ID_CENSOR_FACES_FROM_PHOTO;
transform.censor_mode = CENSOR_MODE_PIXELATE; /* irreversible */
ret = tp_trusted_transform(handle, &mac, NULL, transform);

if (ret != 0) { /* error handling */ }

/* exchange handle for ciphertext */
uint8_t ciphertext[MAX_TRUSTED_DELIVERY_CIPHERTEXT_SIZE] = {0};
transform.type = TRANSFORM_ID_SUBMIT_HANDLE_AND_ENCRYPT;
ret = tp_trusted_transform(handle, &mac, ciphertext, transform);

/* send ciphertext (containing data & list) and mac to target */
```

Listing 9: Usage code of Trusted Transformation API with handles

The implementation adds a few steps to assign and resolve the passed handle compared to its plain text version. In Listing 10, we show the additional lines of code while commenting out steps that stayed the same. The handle mechanism can be implemented with a hash table or the handle can simply be the index into an array located in secure storage. If the caller specifies the encryption transformation, its handle gets taken away and the ciphertext buffer gets filled with the encryption of peripheral data and the list of transformations.

```c
int tp_trusted_transform(tp_handle_t handle_io, mac_t* mac_io,
                         uint8_t* ciphertext_out, transform_t transform)
{
    int status = 0;

    /* points to secure storage */
    periph_data_t*    data;
    transform_list_t* transforms;

    if (transform.type != TRANSFORM_ID_INITIAL)
    {
        /* resolve handle */
        handle_get_buffers(handle_io, data, transforms);

        /* as before: check if list is full */
    }
    else
    {
        /* assign handle */
        status = handle_assign_free_buffers(handle_io, data, transforms);
        if (status != 0) { return status; }

        /* as before: empty the list */
        /* as before: securely capture peripheral data */
        /* as before: hash and sign the data and the transforms list */

        return status;
    }

    /* as before: verify mac regarding data & list integrity */

    switch(transform.type)
    {
        case TRANSFORM_ID_CENSOR_FACES_FROM_PHOTO:
        {
            /* as before: perform transformation */
        } break;

        case TRANSFORM_ID_SUBMIT_HANDLE_AND_ENCRYPT:
        {
            /* invalid buffer */
            if (!ciphertext_out) { return -1; }

            status = encrypt_data_tt(data, transforms, ciphertext_out);
            if (status != 0) { return status; }

            /* take away handle */
            status = handle_free(handle_io);
            return status;
        } break;

        default: { /* unreachable */ };
    }

    /* as before: append to list */
    /* as before: recompute mac */
}
```

Listing 10: Implementation of Trusted Transformation with handles

5.2 Hardware and Software

For our prototype, we have used the NUCLEO-L552ZE-Q development board from STMicroelectronics[69]. It features a STM32L552ZE microcontroller unit (MCU) that is based on a Cortex-M33 architecture — thus supporting the TrustZone-M security extensions. It is equipped with 512 KB of flash memory and 256 KB of SRAM. The board is PSA Certified level 1[70], meaning — according to PSA — best security practices were applied in its development and design.

As our demonstration peripheral, we have decided upon the temperature and humidity sensor HYT 221 from Innovative Sensor Technology[24]. It sends environmental data over the Inter-Integrated Circuit (I^2C) communication protocol and works in polling mode.

Additionally, we have integrated a 0.96inch OLED display from Waveshare[25] into our setup. When turned on, the trusted environment can communicate with this screen to display the temperature and humidity data on it using the Serial Peripheral Interface (SPI). This is intended to showcase that our approach is agnostic to the used communication interface. If confidentiality is needed, the display can be disabled. It also serves an an example of an output peripheral that is entirely contained within the SPE. We have not covered this use case in our design, since it puts no special requirements on the API, but it could open possibilities to a wider range of applications.

We have opted for Zephyr as our real-time operating system (RTOS). As previously mentioned, the main motivation behind this decision was its extensive documentation regarding its TF-M integration[86]. Additionally, it provides helpful support in tools that automate the process of compiling itself, TF-M and its bootloader MCUboot and the subsequent merging of these three images into one.

The selection to rely on TrustedFirmware-M (TF-M) for our TEE on the other hand, has been mainly driven by the lack of TrustZone-M-based alternatives. Related to this, we can briefly summarize why we still opted to use TF-M as our TEE instead of using TrustZone-M in a more bare-metal fashion to satisfy all our specific requirements ourselves. As we made clear before, TrustZone-M serves mainly as the hardware mechanism to implement a Secure Processing Environment (SPE). Although the code and memory isolation that we gain from this environment are a key prerequisite for peripheral to be considered trusted, the requirements that we see as essential go beyond this foundation. In essence, our design relies on so many parts of a TEE that, in our view, a developer trying to implement trusted peripherals will in effect have to implement an entire TEE.

5.3 Trusted Peripherals as a Secure Service

To relocate any parts relating to the memory or code of peripherals into the trusted world, we have opted to implement our trusted peripheral API as a custom Application Root-of-

Trust (ARoT) service inside TF-M. Unless explicitly referenced, all information is derived from TF-M's documentation[12].

The process by which one adds their own services in TF-M involves the configuration of YAML files and changes of TF-M's CMake build system scripts. Listing 11 shows the definition of a bare-bones trusted peripheral secure partition, which can contain one or more secure services.

```
{
  "psa_framework_version": 1.1,
  "name": "TFM_SP_TP",
  "type": "APPLICATION-ROT",
  "priority": "NORMAL",
  "model": "SFN", # or "IPC"
  "stack_size": "0x800",

  "services": [
    {
      "name": "TFM_TRUSTED_PERIPHERAL",
      "sid": "0xFFFFF001",
      "non_secure_clients": true,
      "connection_based": false,
      "version": 1,
      "version_policy": "STRICT",
    }
  ],
  "dependencies": [
    "TFM_CRYPTO"
  ]
}
```

Listing 11: Definition of a TF-M Custom Secure Partition

To let TF-M know of this new service, it has to be included in a manifest list in another YAML file. Listing 12 shows the addition of our defined partition into this list.

```
{
  "description": "TF-M secure partition manifests",
  "type": "manifest_list",
  # ...
  "manifest_list": [
    {
      "description": "Trusted Peripheral Partition",
      "short_name": "TFM_TP",
      "manifest": "${APPLICATION_SOURCE_DIR}/trusted_peripheral/tfm_trusted_peripheral.yaml",
      "output_path": "${TFM_BINARY_DIR}/trusted_peripheral",
      # ...
      "linker_pattern": {
        "library_list": [
           "*tfm_*partition_tp.*"
         ],
      }
    },
  ]
}
```

Listing 12: Addition of a Custom Secure Partition into TF-M

The giving of names in these files is not arbitrary, as TF-M's build system will use the provided identifiers for the purposes of code generation.

As seen in Listing 11, we have specified the PSA Root-of-Trust (PRoT) Crypto service as a dependency to our Application Root-of-Trust (ARoT) service. This provided service uses a reference implementation of PSA Crypto[58] to securely perform cryptographic functions. It supports the hashing, signing and encrypting of data using various cryptographic keys and algorithms. For our implementation, we rely on two 1024-bit RSA key pairs using the PKCS 1 v1.5 standard[28]. The private key of the first pair is stored on the device and is used to sign SHA256 hashes. The public key of the other pair is used for encrypting peripheral data. The counterpart to both pairs is stored on the remote server. In a deployment scenario, the device keys could be stored immutably in memory, e.g. inside our MCU's One-Time-Programmable (OTP) memory during the provisioning process.

In any case, after completing the inclusion of our own secure service, we have the possibility of writing code that gets compiled into the secure TF-M image, rather than the non-secure image of the RTOS. TF-M's build system and tools take care to mark our service code and the memory used in it as secure, using the general process we have described in §2.2.

5.3.1 SFN vs. IPC

As explained in §2.4, we can specify the degree to which a custom service is isolated from other secure components. The requirement to communicate with TF-M's Crypto PRoT service in effect makes this decision for us. Only ARoT services with an isolation level of 1 are able to communicate with PRoT services. This also means that we can make use of the SFN model, which only works in this isolation level. For comparison purposes, we implemented both modes of operation. To showcase the differences between the two, we will look at the API of the trusted capturing scenario. The code in §5.1.1 accurately describes a possible implementation in an untrusted setup. As a secure service using SFN however, the function implementation of the API call in the non-secure world can be seen in Listing 13.

```
psa_status_t tp_trusted_capture(sensor_data_t* data_out, tp_mac_t* mac_out)
{
    psa_status_t status;

    psa_outvec out_vec[] = {
        { .base = data_out,     .len = sizeof(sensor_data_t)     },
        { .base = mac_out,      .len = sizeof(tp_mac_t)          },
    };

    psa_invec  in_vec[]  = { { .base = NULL, .len = 0 } }; /* no input parameter*/

    status = psa_call(TFM_TRUSTED_PERIPHERAL_HANDLE, TP_TRUSTED_CAPTURE,
                      in_vec, IOVEC_LEN(in_vec),
                      out_vec, IOVEC_LEN(out_vec));
}
```

Listing 13: Trusted Capture Implementation in the Non-Secure World (SFN)

As we can see, the actual implementation of trusted capture is not directly included, as we have not yet transitioned into the secure world. To perform this transition, the PSA specification requires the use of input/output vectors (IOVECs). In these data structures, we define the address and size of our input and output parameters, respectively. As trusted capture requires no input parameters, the corresponding data structure is left empty. With these preparations done, the call to a PSA C interface is performed. This function is used to call Root-of-Trust services and represents the actual secure function call — meaning the memory of this code will be marked non-secure-callable and be prefixed by the secure gateway instruction for us. To transfer execution to the correct service and function, we pass a service handle — which has been generated by TF-M, based on our chosen service name — and an API call identifier, which has been defined by us and will be used in the next step.

Listing 14 shows where execution is continued after a successful `psa_call`. Once again, the name of this function is determined by our service definition and chosen model. This function lives in the secure world (i.e. the TF-M image) and dispatches execution to our secure API implementations. Here, the rationale for using output parameters exclusively for our API also becomes apparent, as the return type for all service functions is required to be a PSA status code.

```
psa_status_t tfm_trusted_peripheral_sfn(const psa_msg_t *msg)
{
    switch (msg->type) {

    case TP_TRUSTED_CAPTURE:   { return tfm_tp_trusted_capture(msg->handle);   }
    case TP_TRUSTED_DELIVERY:  { return tfm_tp_trusted_delivery(msg->handle);  }
    case TP_TRUSTED_TRANSFORM: { return tfm_tp_trusted_transform(msg->handle); }
    case TP_TRUSTED_HANDLE:    { return tfm_tp_trusted_handle(msg->handle);    }

    default:                   { return PSA_ERROR_PROGRAMMER_ERROR; }
    }

    return PSA_ERROR_GENERIC_ERROR;
}
```

Listing 14: Dispatching API Calls in the Secure World (SFN)

The secure world implementation for trusted capturing is shown in Listing 15. Here, the actual code that performs the necessary work is involved as before. However, the argument that is passed here is only a handle. To return the results of this function to the non-secure caller, another PSA function has to be invoked that writes the results of the computation into the output parameters at positions 0 and 1. Additionally, we can identify overhead in the form of allocated stack memory for the sensor data and MAC. These were not needed in the untrusted implementation, as the passed buffers could be filled directly.

```
static psa_status_t tfm_tp_trusted_capture(psa_handle_t handle)
{
    psa_status_t status;
    sensor_data_t sensor_data;
    tp_mac_t mac;

    /* as before: securely capture peripheral data */
    psa_status_t status = internal_capture(&sensor_data);
    if (status != PSA_SUCCESS) { /* error handling */ }

    /* as before: hash and sign the data */
    status = compute_mac(&mac, &sensor_data);
    if (status != PSA_SUCCESS) { /* error handling */ }

    /* write to output parameters */
    psa_write((psa_handle_t)handle, 0, &sensor_data, sizeof(sensor_data));
    psa_write((psa_handle_t)handle, 1, &mac,         sizeof(mac));

    return status;
}
```

Listing 15: Trusted Capture Implementation in the Secure World

The implementation for IPC mode is similar, but contains additional steps. In that mode, the service is required to be connection-based, which changes our implementation to what can be seen in Listing 16.

```c
psa_status_t tp_trusted_capture(sensor_data_t* data_out, tp_mac_t* mac_out)
{
    psa_status_t status;

    psa_outvec out_vec[] = {
        { .base = data_out,     .len = sizeof(sensor_data_t)     },
        { .base = mac_out,      .len = sizeof(tp_mac_t) },
    };

    /* api call encoded into the in_vec to be able to dispatch in IPC mode */
    uint32_t api_call = TP_TRUSTED_CAPTURE;
    psa_invec  in_vec[]  = { { .base = &api_call, .len = sizeof(uint32_t) } };

    psa_handle_t handle;
    handle = psa_connect(TFM_TRUSTED_PERIPHERAL_SID, TFM_TRUSTED_PERIPHERAL_VERSION);
    if (!PSA_HANDLE_IS_VALID(handle)) { return PSA_ERROR_GENERIC_ERROR; }

    status = psa_call(handle, PSA_IPC_CALL,
                      in_vec, IOVEC_LEN(in_vec),
                      out_vec, IOVEC_LEN(out_vec));

    psa_close(handle);
}
```

Listing 16: Trusted Capture Implementation in the Non-Secure World (IPC)

The handle for our service must now be acquired by establishing a connection. As an additional restriction, we lose the ability to pass our API call identifier directly. Instead, we encode it in our input IOVEC. As mentioned in §2.4, this mode is also intended to support a fully separated architecture, e.g., two separate processors that are in execution and communicate with each other using signals. To accommodate this, IPC mode requires the specification of an entry point, which can be seen in Listing 17.

```c
psa_status_t tfm_tp_req_mngr_init(void)
{
    psa_signal_t signals = 0;

    while (1) {
        signals = psa_wait(PSA_WAIT_ANY, PSA_BLOCK);

        if (signals & TFM_TRUSTED_PERIPHERAL_SIGNAL) {
            tp_signal_handle(TFM_TRUSTED_PERIPHERAL_SIGNAL, tfm_trusted_peripheral_ipc);
        }

        else {
            psa_panic();
        }
    }

    return PSA_ERROR_SERVICE_FAILURE;
}
#endif
```

Listing 17: Entry point for IPC mode in the secure world

The SPM acts as the scheduler that passes execution back to this entry point for an incoming signal. If that signal is relevant to the secure partition, a signal handler is invoked, which can be seen in Listing 18.

```
typedef psa_status_t (*tp_func_t)(psa_msg_t *);
static void tp_signal_handle(psa_signal_t signal, tp_func_t pfn)
{
    psa_status_t status;
    psa_msg_t msg;

    status = psa_get(signal, &msg);
    switch (msg.type) {
    case PSA_IPC_CONNECT:
        psa_reply(msg.handle, PSA_SUCCESS);
        break;
    case PSA_IPC_CALL:
        status = pfn(&msg);
        psa_reply(msg.handle, status);
        break;
    case PSA_IPC_DISCONNECT:
        psa_reply(msg.handle, PSA_SUCCESS);
        break;
    default:
        psa_panic();
    }
}
```

Listing 18: Signal Handler in IPC Mode

This signal handler passes execution to a dispatcher, which can be seen Listing 19 and is mostly identical to the SFN dispatcher, except the API call is extracted out of the IOVEC. After that, the secure world implementation of Trusted Capture from Listing 15 is called.

```
psa_status_t tfm_trusted_peripheral_ipc(psa_msg_t *msg)
{
    /* extract api call to dispatch */
    uint32_t api_call = 0;
    size_t ret = psa_read(msg->handle, 0, &api_call, msg->in_size[0]);
    if (ret != msg->in_size[0]) /* psa_read should return number of bytes copied */
    {
        return PSA_ERROR_PROGRAMMER_ERROR;
    }

    switch (api_call) {
        /* dispatch api call as in the sfn case */
    }

    return PSA_ERROR_GENERIC_ERROR;
}
```

Listing 19: Dispatching API Calls in the Secure World (IPC)

As we have seen, the IPC mode comes with the added baggage of more code complexity and overhead in the form of more function indirections. A restriction that we previously left unmentioned, which applies to both modes, is the set limit of IOVECs. PSA foresees a maximum number of input/output parameters and in TF-M's case, this number is four. This means that only a combination of four input or output parameters can be passed to a secure function. We will elaborate on what this means for the usability for developers in §6.3.

5.3.2 Secure Peripherals

With the previous chapters, we have established a way to develop code and allocate memory that is located in the secure world. This includes peripheral code and the data that we receive from them. However, we still lack a mechanism to isolate the peripherals themselves. As we have shown, our secure service is called by the non-secure world — meaning any isolation that we perform in those secure functions is not guaranteed to be run. Instead, any peripheral devices need to be locked up before execution is passed to the non-secure world by the Secure Partition Manager (SPM).

To facilitate this use case, TF-M services include the possibility of defining memory-mapped IO (MMIO) regions in the service definition. This way, memory-mapped peripherals can be specified to be isolated. Listing 20 shows an example definition of an I^{2}C peripheral with a corresponding permission setting.

```
{
  "name": "TFM_SP_TP"
  # ...
  "mmio_regions": [
    {
      "name":       "TFM_PERIPHERAL_I2C",
      "permission": "READ-WRITE",
    },
  ],
  # ...
}
```

Listing 20: Definition of an MMIO Region in a TF-M Custom Service

The name of this peripheral is designed to be defined per platform in TF-M. Listing 21 shows how and where this identifier is defined and how it directly maps to the address of a platform specific struct.

```
/* in tfm_peripherals_def.h */
extern struct platform_data_t tfm_peripheral_i2c;
#define TFM_PERIPHERAL_I2C   (&tfm_peripheral_i2c)

/* in mmio_defs.h    */
const uintptr_t partition_named_mmio_list[] = {
     (uintptr_t) TFM_PERIPHERAL_I2C,
     /* ... */
 };
```

Listing 21: Definition of a platform-specific Peripheral Identifier

By including this named peripheral in our service definition, the code to isolate the memory regions of the peripheral will be executed by the Secure Partition Manager (SPM) on start-up — before any untrusted code can run.

5.3.3 Complications

While the before mentioned mechanism is intended to facilitate peripheral isolation, we have found the support for our board in this regard to be non-existent, as none of its peripherals have been included in the MMIO definitions. Regarding this, the definition of the platform-specific struct for our device inside TF-M's source code can be seen in Listing 22.

```
/* Holds the data necessary to do isolation for a specific peripheral. */
struct platform_data_t
{
    uint32_t periph_start;
    uint32_t periph_limit;
    int16_t periph_ppc_bank;
    int16_t periph_ppc_loc;
};
```

Listing 22: Peripheral Isolation Structure for STM32L5 Boards

In this data type, the start and end address of the memory-mapped peripheral is specified. Additionally, information can be provided for the Peripheral Protection Controller (PPC) that is used for peripherals that are not TrustZone-aware as described in §2.2. This is relevant for us, as all I^2C and SPI peripherals in the case of our board are of this kind.

While we could add the definitions of our peripherals that require isolation ourselves, we also have found the functionality that iterates through this array and performs the isolation using the data to be not implemented for our device. Instead, we have opted to perform this isolation ourselves, by including the necessary calls to the STM32 HAL inside an initialization function for our board, as depicted in Listing 23.

```
void gtzc_init_cfg(void)
{
    /* ...       */
    /* TF-M code */

    /* our secure peripherals */
    HAL_GTZC_TZSC_ConfigPeriphAttributes(GTZC_PERIPH_I2C2, GTZC_TZSC_PERIPH_SEC);
    HAL_GTZC_TZSC_ConfigPeriphAttributes(GTZC_PERIPH_SPI1, GTZC_TZSC_PERIPH_SEC);

    /* TF-M code */
    /* ...       */
}
```

Listing 23: Isolation of TrustZone-unaware Peripherals

This code configures the security gates in front of the peripherals to disallow non-secure interactions with them. If still attempted, it will trigger an exception by the Global TrustZone Controller (GTZC).

To communicate with our board and its peripherals, we make use of its Hardware Abstraction Layer (HAL). TF-M already includes the STM32 HAL, but is missing files relating to the I^{2}C and SPI interfaces. This is further indication that the inclusion of peripherals inside its TEE has not yet matured. At any rate, we have added the necessary files and modified TF-M's build scripts to include them.

As a further complication with this setup, the MCUboot bootloader implements its firmware update mechanism for this board in the form duplicated image slots. The main image to boot from resides in the primary slot, while the secondary slot is used to store the old image. In the case of a firmware update, a new image is written to the secondary slot for verification, before MCUboot swaps the contents of the primary and secondary slot — thus completing the update. Thus, the flash needs enough memory to be able to store both images.In the case of Zephyr & TF-M however, we now have two images from the start: The secure TF-M image and the non-secure Zephyr image. To enable this firmware update mechanism, these both need to be duplicated in memory with a secondary slot. The standard flash layout that TF-M uses for our board is laid out in Listing 24. The effective space that is left for the Zephyr image is only 36 KB.

```
/*
 * 0x0000_0000 BL2 - MCUBoot                 ( 38 KB)
 * 0x0000_e000 OTP / NV counters area        (  8 KB)
 * 0x0001_0000 Secure Storage Area           (  8 KB)
 * 0x0001_2000 Internal Trusted Storage Area (  8 KB)
 * 0x0001_4000 Secure image     primary slot (180 KB)
 * 0x0004_1000 Non-secure image primary slot ( 36 KB)
 * 0x0004_a000 Secure image     secondary slot (180 KB)
 * 0x0007_7000 Non-secure image secondary slot ( 36 KB)
 */
```

Listing 24: Standard flash layout of the STM32L552ZE-Q in TF-M

We have found this amount to be far too insufficient for the untrusted application and have thus forced a flash layout that is intended for testing inside TF-M. Listing 25 depicts this layout, which comes at the drawback of disabling the firmware update mechanism of our implementation, as it removes the secondary slot for both images.

```
/*
 * 0x0000_0000 BL2 - MCUBoot                ( 38 KB)
 * 0x0000_e000 OTP / NV counters area  area  (  8 KB)
 * 0x0001_0000 Secure Storage Area           (  8 KB)
 * 0x0001_2000 Internal Trusted Storage Area (  8 KB)
 * 0x0001_4000 Secure image     primary slot (224 KB)
 * 0x0004_c000 Non-secure image primary slot (172 KB)
 * 0x0007_7000 Unused                        ( 40 KB)
 */
```

Listing 25: Our flash layout of the STM32L552ZE-Q in TF-M

With these issues out of the way, we implemented our trusted peripheral API inside TF-M as a custom service. An application can choose to enable this service and securely interact with it.

6 Evaluation

In this chapter, we will perform an evaluation of our design and implementation under the aspects of security in §6.1, the aspect of performance in §6.2 and the aspect of usability for developers in §6.3.

To guide all comparisons, we have implemented an untrusted version of the prototype that we introduced in §5.2 and 5.3. This untrusted setup uses only a Zephyr image with no TrustZone-M capabilities. We remove any unneeded variability between the two by, firstly, not making use of the simpler Zephyr API to interact with the peripheral in our implementation. Instead, we directly call functions from the STM32 HAL — the same exact ones used by the trusted version. Secondly, the untrusted version also uses PSA Crypto for any of its cryptographic functions. The same keys and algorithms are shared between both setups. The difference here is that PSA Crypto is not part of a secure service and, instead, merely another library. As such, the comparison should be on equal grounds and any differences should be caused by TrustZone-M or the TEE. As a complication however, we have found out that there are implementation differences between the secure service version of PSA Crypto and its library equivalent. In our performance chapter, we will go into specifics of this disparity and perform first measurements that hopefully isolate the relevant differences.

As further evidence to confirm our findings, we have taken steps to support the execution of our application on the board ARM MPS2+ AN521[3], which is supported as an emulation target in Zephyr that can be run on the Quick Emulator (QEMU) virtualization software[59]. It supports both trusted and untrusted execution. Since we cannot attach our peripherals to an emulated board, any calls to our API will generate mock sensor values for this board in our test.

6.1 Security Analysis

The deployment scenario that we envision for our prototype is a setting in which temperature and humidity data needs to be securely transmitted. An example could be in the agricultural industry, where the growing of products under favorable environmental conditions should be guaranteed. As such, our prototype makes use of trusted capture, or — if confidentiality is needed — trusted delivery. As a proof-of-concept for trusted transformation, the untrusted caller can issue a conversion from degrees Celcius to Fahrenheit, while maintaining the integrity of sensor data. For the purpose of analyzing the security of this deployment, we can refer to our general threat model from §4.1 and further narrow it down to our specific use case.

To help with this process, we look at the threat modeling and security analysis (TMSA) Smart Meter example that PSA provides[68], which we have determined to be the closest application scenario to our prototype. In their example, they describe a situation where an IoT water meter is installed in a home or office. The water distributor expects accurate and genuine peripheral data in the form of measured water flow to calculate a bill that the client has to pay. Hence, the device is located in an untrusted environment where individuals could stand to benefit from the tampering of the captured data and the data's integrity needs to be preserved. They exclude the possibility of over-the-air firmware updates, due to the device's limited network capabilities and battery power. As we can see, all of these higher level requirements and constraints also apply to our scenario.

Further comparisons can be drawn in the asset identification. The document identifies the firmware and the certificate that is used to authenticate it as primary assets to protect. Within our context, these are our application and the public key that is used by the bootloader to verify the application image. The water flow measurements (for us temperature & humidity data) are also considered to be important assets — same as per our threat model. Lastly, the credentials to authenticate and protect communication are identified, which in our case, are the employed key pairs.

Building on these insights, the TMSA analyzes possible threats. By applying these to our scenario, we can evaluate how our security measures mitigate them.

As a first possibility, an attacker could perform a Man-In-The-Middle (MITM) attack or go as far as to impersonate the remote server. If this attack is successful, exchanged messages could be modified, which includes the tampering of our sensor readings. Our use of a MAC can protect the data integrity in this case. However, if impersonation should still be rendered impossible, the TMSA insists that only connections from configured servers should be accepted. For our purposes, the trusted world could communicate securely with only the intended server using encryption. In this case, the non-secure world acts as an intermediary between the two. If the untrusted side tries to pass messages from an unknown source, this source has no ability to authenticate itself to the trusted world. Nonetheless, secure direct communication between a server and only the untrusted world can not be given. Our design does not rely upon this type of communication.

A different attack that is specified in the PSA document is the abuse of firmware. In this situation, an attacker is able to install an authorized firmware and thus directly threatens all assets. This can, e.g., be accomplished if the bootloader's public key used for verification can be changed by the attacker. As we have touched upon in our threat model, we consider an attacker to be unable to do this, due to the use of hardware mechanisms, such as OTP memory to store the public key. However, the attacker may also perform this attack with the aid of debug features that were left active in deployment. The board used for our prototype provides a joint test action group (JTAG) and serial wire debug (SWD) port for debugging purposes[69]. Likewise, a serial port is used to update software locally in IoT devices [35]. Hence, to successfully prevent this threat, we require that these ports are closed before deployment.

The last threat is the most difficult to protect against. Due to the physical access, attackers may try to perform all classes of hardware attacks on the device. Although we now exclude the possibility of an attack using the debug facilities of the device, the field of hardware attacks in this context still includes physical attacks that tamper with the device, side-channel attacks that analyze physical signals (e.g. through a power analysis) to extract unintended information that is carried within[16], and fault injection attacks, where an attackers provokes the malfunctioning of the system through, e.g., voltage glitching[64]. As a mitigation, the PSA TMSA example only vaguely recommends to protect and react against physical tampering attempts. Although there are Cortex-M devices that are PSA Certified level 3 and thus make claims about protection from hardware attacks[56], our board is not one of them. Devices using TrustZone-enabled Cortex-M processors previous to the Cortex-M33 have been successfully attacked using fault-injection before[79], making it possible to skip over instructions, including, e.g., instructions that mark memory regions as secure or ones that check for the validity of the firmware. Regarding side-channel attacks, we have found instances of the not TrustZone-enabled Cortex-M4 and Cortex-M7 being vulnerable[16][15]. The vulnerability of our specific prototype to these attacks however, is uncertain. Among the software countermeasures against them are the inclusion of random delay to complicate timing-based attacks, duplicated execution of security critical instruction to avoid skipping over them, and the monitoring of control flow[48]. These however, only decrease the probability of a successful physical attack. Thus — although we see it to be out of scope for this work — the use of a tamper-proof container that can detect intrusion attempts is still appropriate in practice.

As TF-M makes up the majority of our Trusted Computing Base (TCB), it is important that we can assess its security metrics. The TF-M image in our implementation has a binary size of 132 KB. In terms of code complexity, the implementation for the NUCLEO-L552ZE-Q spans approximately 33.000 source lines of code (SLOC)[42]. A form of attack that has been unmentioned thus far are code reuse attacks. Here, the attacker can use Return Oriented Programming (ROP) to take advantage of existing code fragments in the form of so-called gadgets to chain together a malicious execution without any code injection[35]. The implementation here has been found to contain around 10.000 of such ROP gadgets[42]. There has been a proposal of TrustZone-M based Address Space Layout Randomization (ASLR) that could possibly mitigate the threats that result from this[34]. Concerning known vulnerabilities, TF-M has so far published six security advisories that address the Common Vulnerabilities and Exposures (CVEs) that it discovered[76]. Out of these six CVEs, four have been rated as having high severity by the National Vulnerability Database (NVD)[40].

In conclusion, our security analysis has identified areas in which full assurance of protection cannot be given. These stem primarily from hardware attacks, which usually target the specific type of device at hand. For our board, we can make no definitive judgment in its security against this type of attack, since we lack concrete evidence that points to a negative or positive answer. The reasonable approach in this case is to assume vulnerability by default. For our specific use case however, we also recognize that such an attack is quite involved and requires the necessary equipment and expertise. The probability of a person in the physical environment of an agricultural setting being qualified to carry out such an attack is not impossible, but significantly lower than the probability of such an individual existing remotely. Here, we can give confidence in the protection from remote aggressors, in which case our solution can be considered up to the task.

6.2 Performance

In this chapter, we will benchmark the performance of both of our setups regarding every API call. The ambition was to demonstrate the overhead caused by the implementation of trusted peripherals as a secure service. However, we soon found the trusted setup to be faster in all cases — against all expectations. We determined this difference to be caused by calls to the PSA Crypto API that we make strong use of. The exact culprit is the inclusion of a specialized crypto hardware accelerator in the case of TF-M's Crypto service[44]. Although our board features no dedicated accelerator for cryptographic operations, the implementation in TF-M's case makes particular use of the hardware abstraction layer in the case of STM32 devices. The implementation for the library in the untrusted case is instead generic and not hardware specific. We have not discovered an equally efficient cryptographic library that can be used in place of the untrusted one, nor have we found the intentional exclusion of this accelerator to be possible, as the Crypto secure service relies on it.

To still perform an evaluation that can make claims about the overhead caused by the TrustZone-M context switch between worlds and TF-M's secure service, we have implemented a secure function that is called by the non-secure world and performs a (mildly) computationally intensive task to simulate workload. The result of this computation is returned to the non-secure world. In the untrusted setup, it is a simple function call. This test setup is outlined in Listing 26 and mimics a typical call of our API by using both an output and input parameter. This setup should give us a starting point to reason about the combined performance cost of entering the secure world and returning to the non-secure world.

```c
/* in non-secure world */
for (int i = 0; i < 10000; i++)
{
    timing_t begin = timing_counter_get();

    uint64_t result = 0;
    measure_context_switch(&result, i);

    timing_t end = timing_counter_get();
    log_value(end - begin);
}

/* in secure world (trusted) or in non-secure world (untrusted) */
psa_status_t measure_context_switch(uint64_t* res_out, uint32_t number)
{
    #ifdef TRUSTED_SETUP
        uint32_t number;
        size_t ret = psa_read((psa_handle_t) handle, 1, &number, sizeof(uint32_t));
        if (ret != sizeof(number)) { return PSA_ERROR_PROGRAMMER_ERROR; }

        uint64_t res = compute_large_value(number);

        psa_write((psa_handle_t)handle, 0, &res, sizeof(uint32_t));
    #else
        *res_out = compute_large_value(number);
    #endif

    return 0;
}

/* test function to simulate workload */
static uint64_t compute_large_value(int value)
{
    uint64_t result = 0;
    for (int i = 0; i < 100; i++) {
        result = result + (i * value) / 2;
    }
    return result;
}
```

Listing 26: Test Setup to Measure Overhead of Context Switching

All measurements are performed with compiler optimizations turned on in both cases. We made use of Zephyr's timing functions, which gives use the execution time between a start and end point in nanoseconds [86]. Figure 6.1 shows the measured overhead of world switching (in IPC or SFN trusted setups) compared to a simple function call (in the untrusted setup) in microseconds. Figure 6.2 presents the same information for our emulation target in QEMU.

Figure 6.1: Overhead of World Switching in Trusted Setup

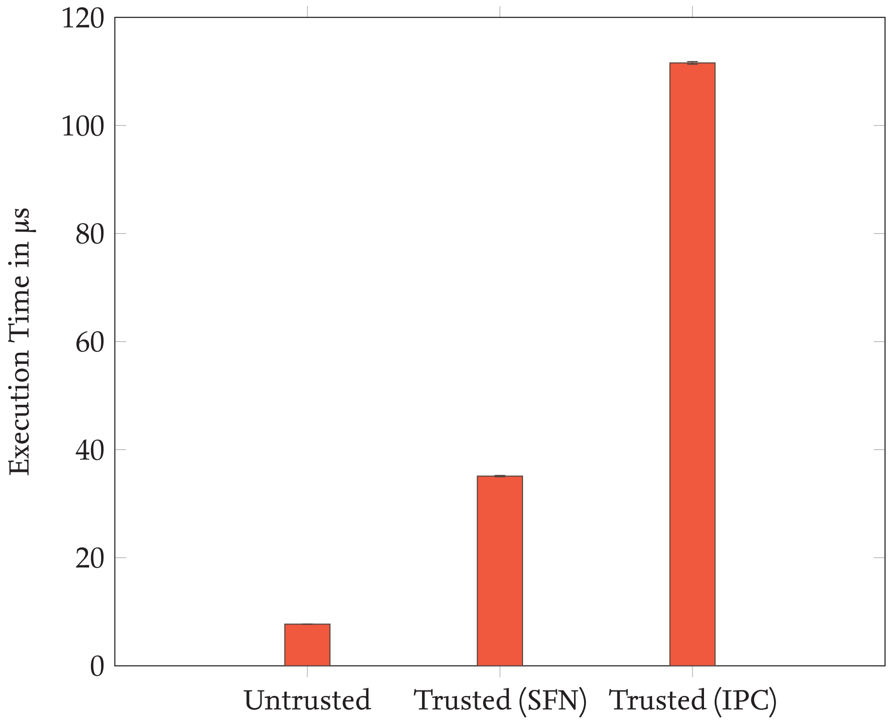
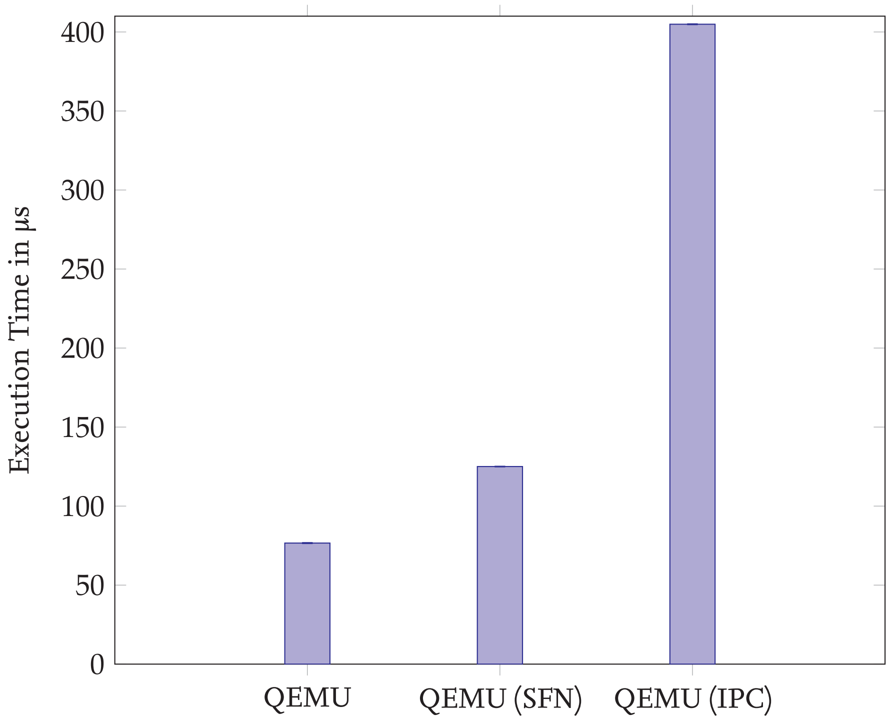


Figure 6.2: Overhead of World Switching in QEMU



As we can see, there are orders of magnitudes in difference between the test cases. The normal application on our board can perform the computation in just under 8 µs, while the SFN version takes 35 µs and IPC mode requires just under 112 µs. While the specific values for the emulated target depend on the accuracy of the software implementation of the board and the specific machine that ran the simulation, the findings here can serve as a confirmation of these ratios. The standard deviation is pictured as well, but is so negligibly small that it is mostly not visible. This is in line with TrustZone-M's set goal of satisfying real-time constraints while gaining security. The deviation is, however, lowest for the untrusted setup — 0.004 µs, while the trusted versions exhibit standard deviations of 0.12 µs (SFN) and 0.24 µs (IPC). While this is once again a difference in orders of magnitudes, we will see that these fluctuations are completely offset by the deviations that are caused by more realistic and involved workloads.

As we have explained in §2.2, a call from the non-secure to secure world requires only a single instruction at the entry point of the function, which cannot solely explain these performance differences. However, when considering the implementation specific to TF-M secure services that we have shown in §5.3.1, these disparities make more sense. Our trusted setups have more function indirections, require additional memory allocations, as well as additional function calls to perform the copying of arguments and writing to parameters. These additional steps are most frequent in the IPC case. One can thus assume that the overhead of a simple secure service API call will be in the realm of about 100 µs. For our purposes, we will be overly cautious and assume a worst case of 1000 µs or 1 ms in both IPC or SFN mode to serve as a buffer for any uncertainties.

With these findings in mind, we can perform the evaluation of all our API calls in practice. It should be noted that the next measurements will be in milliseconds, while the cost of context switching has remained in the range of microseconds. For the remainder of the performance evaluation, we will look at our board communicating only with the temperature and humidity sensor, meaning the SPI display will be disconnected. The peripheral is running in a polling mode and requires 100 ms between readings to compute an updated reading. When adding the code to issue and fetch the measurements from the sensor without any security mechanisms in place, the execution time comes out to about 102 ms in all setups.

6.2.1 Performance: Trusted Capture

In this chapter, we will benchmark the trusted capture API in isolation. The timer is started before the API call and ends with the receiving of sensor data and the MAC in the non-secure world. In Figure 6.3, we see the average execution time of our trusted and untrusted setup after capturing 1000 sensor readings. Figure 6.4 presents the same information for the emulation target. Additionally, we included an estimate for our board that represents the trusted version (IPC or SFN) without a cryptographic accelerator.

Figure 6.3: Execution Times in Trusted Capture

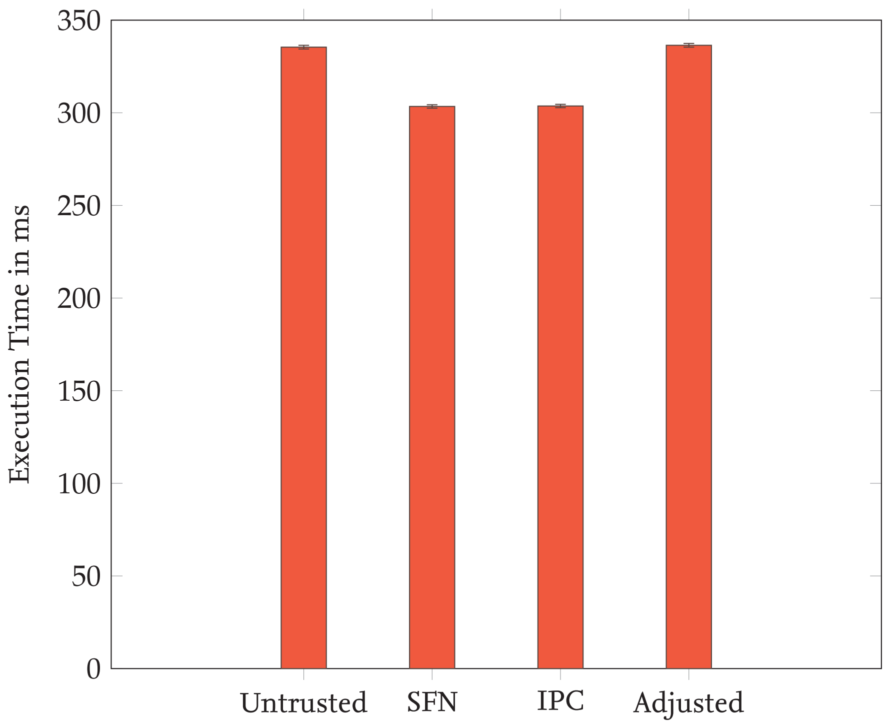
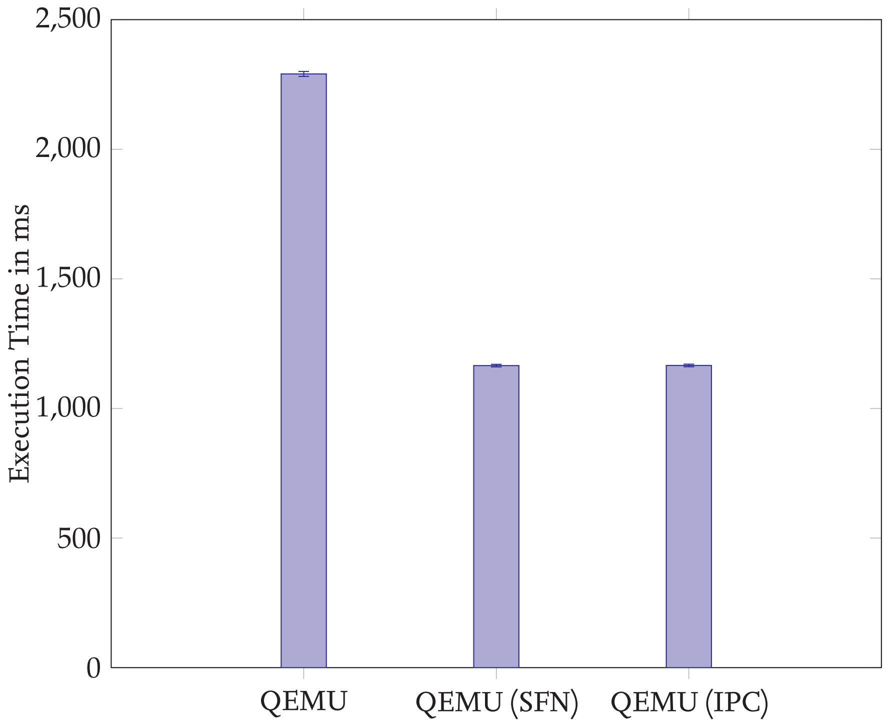


Figure 6.4: Execution Times for Trusted Capture (Emulated)



As we can see, our trusted prototype outperforms the reference setup by about 30ms — due to the cryptographic acceleration factor that we have mentioned earlier. The performance difference between IPC and SFN has become insignificant. The untrusted version needs about 335 ms to perform the task and the trusted capturing scenario relies on only one API call. Thus, we can add the conservative estimate of 1 ms overhead from context switching to arrive at a execution time of 336 ms for the trusted version in the worst case when adjusting for the cryptographic acceleration. The standard deviation is just under 1 ms for our board in all cases, meaning the nanosecond differences of context switching from before are no longer relevant. The emulated target confirms these tendencies.

When accounting for the 102 ms that are caused by the communication with the peripheral, we can determine that trusted capturing in the untrusted setup (henceforth referred to as untrusted capturing for brevity) spends about 230 ms to compute the MAC, while trusted capturing does the same in just under 200 ms.

6.2.2 Performance: Trusted Delivery

To benchmark the trusted delivery API in isolation, we repeated the same steps as before. In Figure 6.5 and 6.6, we see the average execution time of all setups after capturing a sensor reading and returning the corresponding ciphertext and MAC to the non-secure world 1000 times in a row.

Figure 6.5: Execution Times for Trusted Delivery

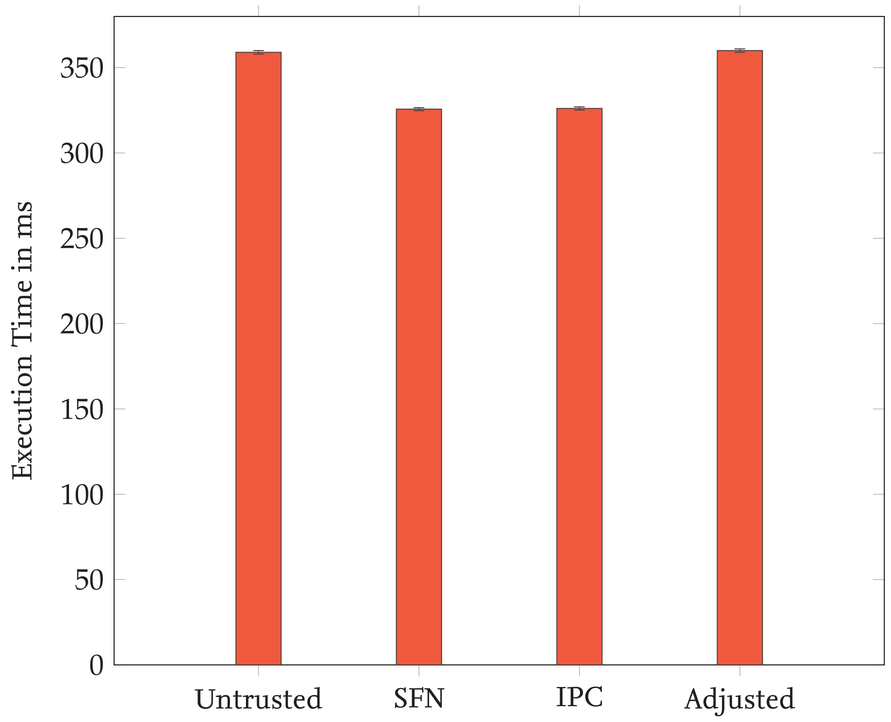
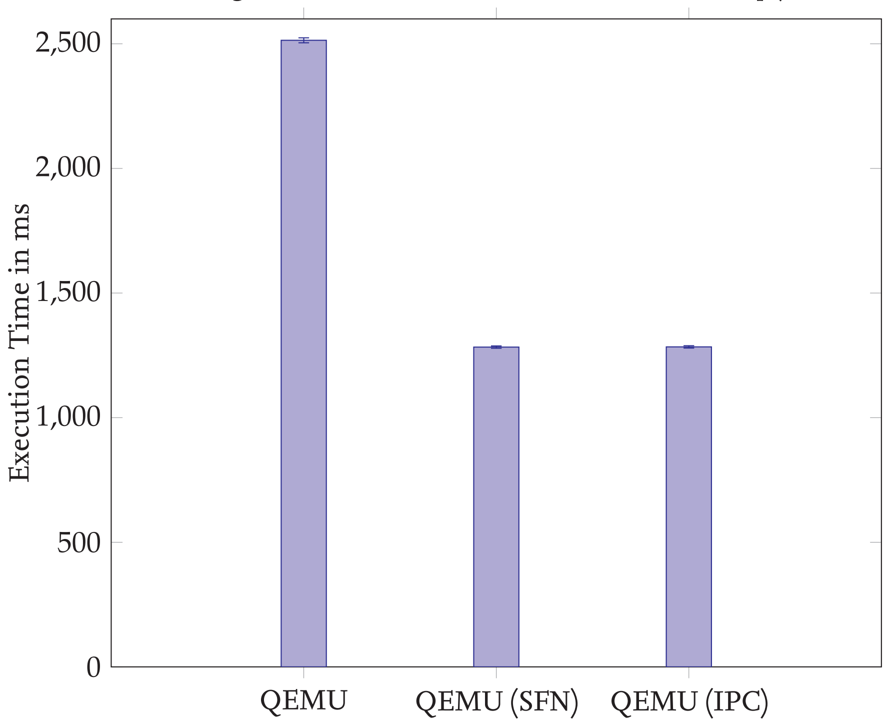


Figure 6.6: Execution Times for Trusted Delivery (Emulated)



The trusted setups now outperform their untrusted equivalent by 33 ms. After accounting for the peripheral code and MAC computation, the additional step of encryption in this scenario costs us about 26 ms in the untrusted case and 23 ms in the trusted one. Hence the gap from before is widened by 3 ms. The measurements of the emulated target confirm these ratios. The standard deviation is again around 1 ms for our board.

As the trusted delivery scenario relies on one API call, we can estimate an execution time of 360 ms for the trusted version in the worst case when adjusting for acceleration compared to the 359 ms for the untrusted setup.

6.2.3 Performance: Trusted Transformation

For the purposes of benchmarking the trusted transformation API in both variants, we included five example transformations in our test setup. These transformations perform the simple task of converting the sensor data back and forth from degree Celcius to Fahrenheit. A measurement was taken when all transformations were completed. In Figure 6.7 and 6.8, we see the average execution time of the normal trusted transformation case after repeating this 100 times in a row.

Figure 6.7: Execution Times for Trusted Transformation

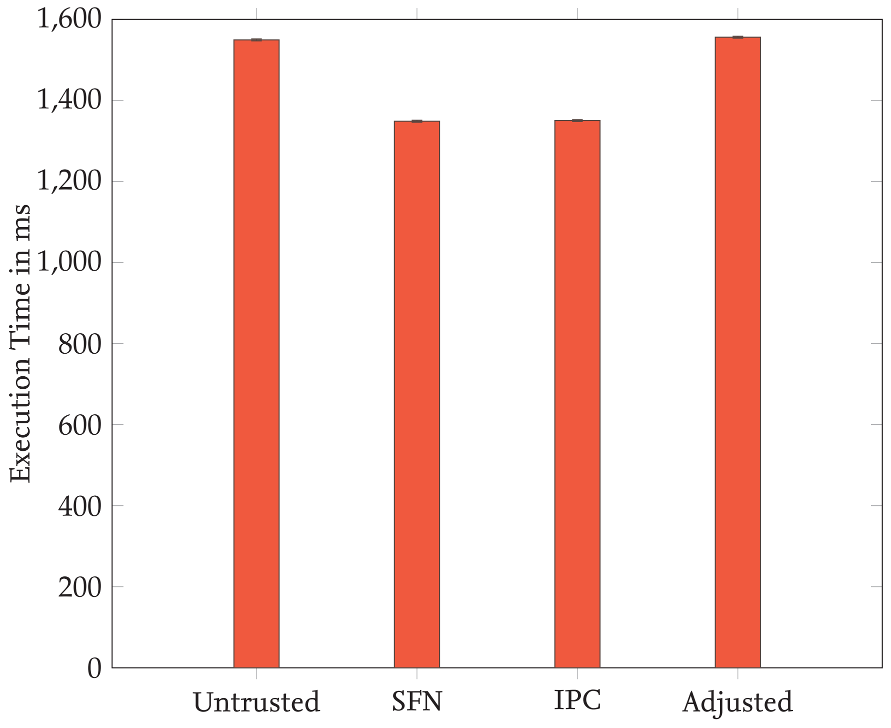
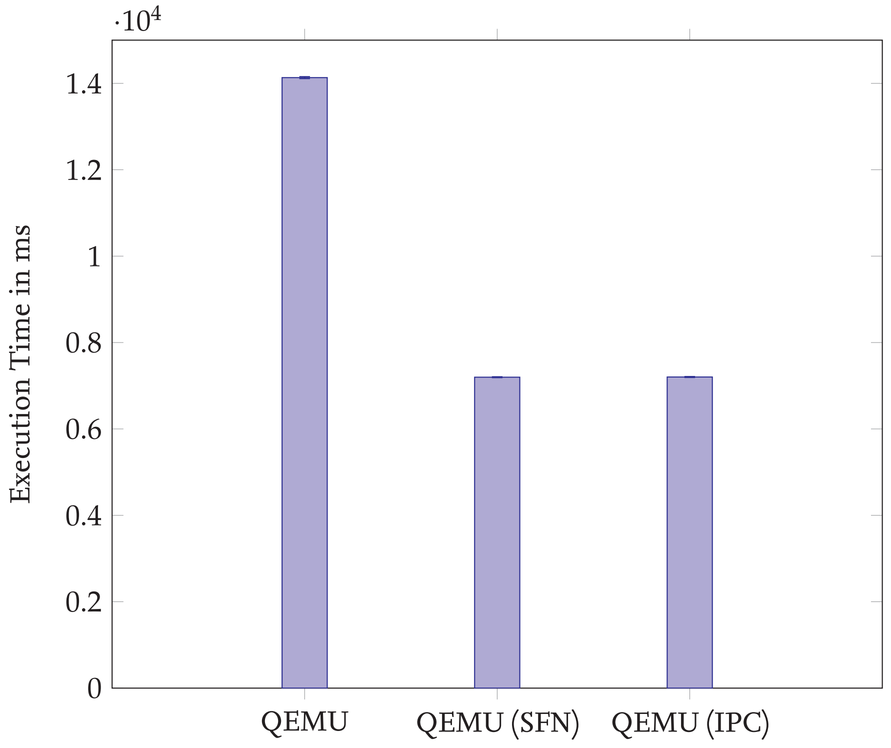


Figure 6.8: Execution Times for Trusted Transformation (Emulated)



The trusted cases here outperform the non-secure version by about 200 ms. Due to the fact that we perform five transformations after the initial call, we arrive at a total of six API calls, resulting in a worst-case overhead of 6 ms. Thus, the adjusted trusted case is estimated to be at around 1556 ms compared to the 1550 ms of the untrusted case.

In Figure 6.9 and 6.10, we show the average execution time for the trusted transformation using handles in the same test setup. As this requires an additional API call to finalize the interaction (by submitting the handle and encrypting the data), the overhead in the adjusted worst case is now 7 ms. As we can see, the differences between the two versions are marginal. Thus, when the normal application doesn't have to inspect the peripheral data in any way, confidentiality can be achieved without a big impact on performance.

Figure 6.9: Execution Times for Trusted Transformation in Handle Mode

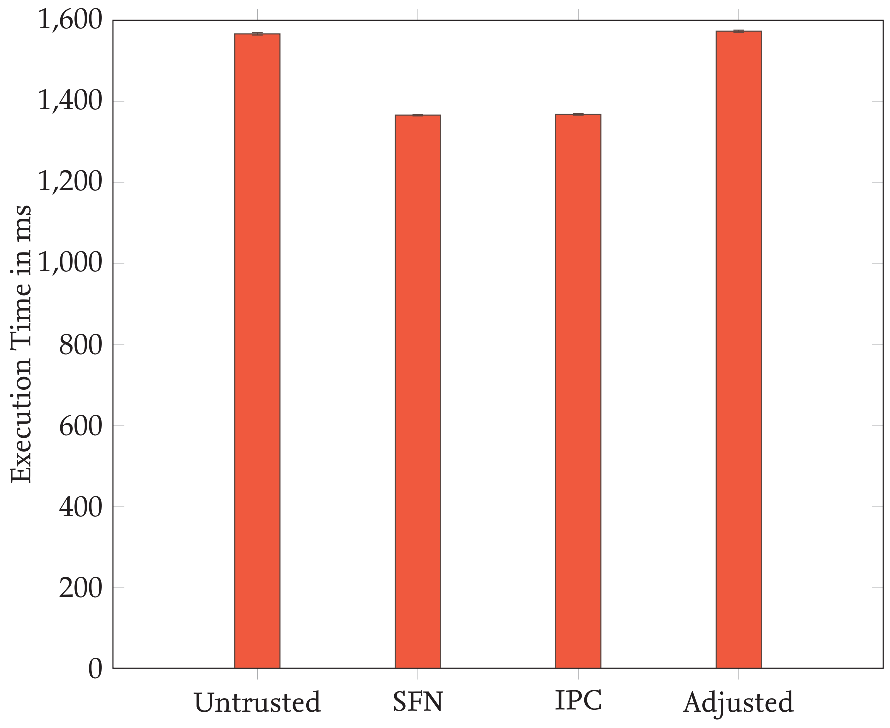
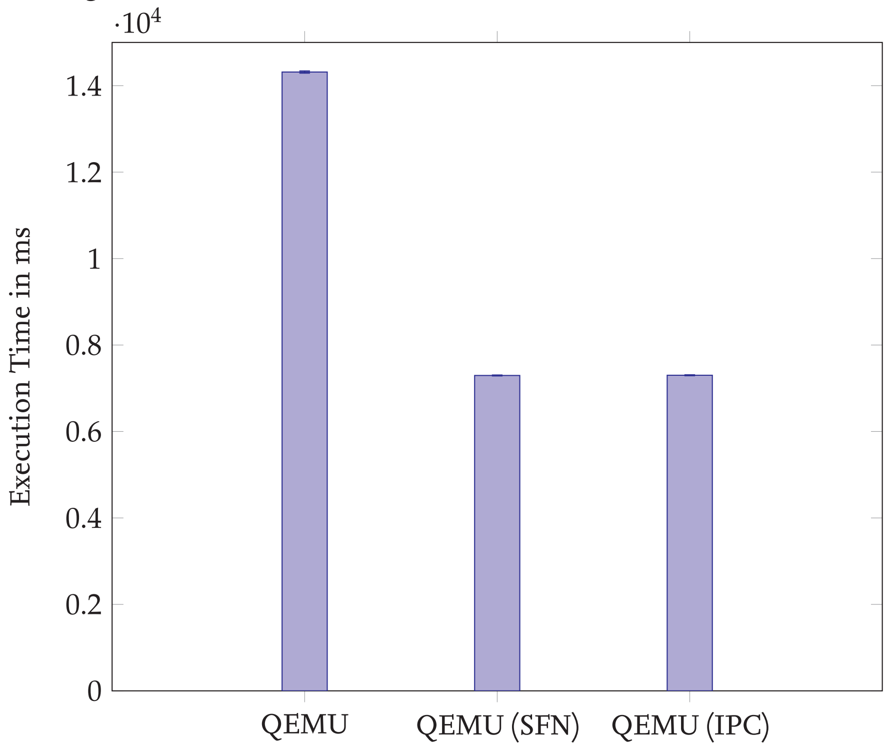


Figure 6.10: Execution Times for Trusted Transformation in Handle Mode (Emulated)



6.2.4 Transmission Rates

For the typical IoT use case, the collected peripheral data packets will have to be transmitted in some form. Although in practice, these transmissions will often be kept to a minimum for power savings, we here will look at constantly outgoing transmissions of our different scenarios in the trusted and untrusted setups for the purposes of measuring their effective transmission rates.

For this evaluation, the board was connected with our development machine over USB. The communication protocol to transfer data was the Universal Asynchronous Receiver Transmitter (UART) with a configuration of 115200 bps and 8-N-1 (8 data bits, no parity, 1 stop bit). This allows for a maximum transmission rate of 11.52 KB/sec, which we were able to confirm in the optimal case (i.e. no data processing between transmissions). In our test setup, the application was continuously capturing sensor readings and immediately transmitting them. The measurement started with the first received byte and ended with the last. Table 6.1 lists the total size of every sent packet in all scenarios and what they are composed of. The indicator *crypt* declares that the corresponding data is contained in the ciphertext. In our case, encrypted data was always able to fit into 128 Bytes of ciphertext.

In Figure 6.11, we show the transmission time for the first to the 100th transmitted sensor readings. As our connection was reliable, there were no dropped packets or significant

Table 6.1: Packet Sizes of all Involved API Calls (in Bytes)

	TC	TD	TT	TH
Sensor Data	8	*crypt*	8	*crypt*
Ciphertext	-	128	-	128
Hash (MAC)	32	32	32	32
Signature (MAC)	128	128	128	128
Transformations	-	-	72	*crypt*
Total	168	288	240	288

deviations in the data rate. Figure 6.12 shows this data rate for every scenario in Bytes per second. Although untrusted delivery (UD) and trusted delivery (TD) are the fastest in this regard, Figure 6.11 shows that the transmission rate in terms of sensor readings per seconds is slower than trusted capture (TC) and untrusted capture (UC). This is because the data packet is larger due to the encryption, so more time is proportionally spent with the transmission than in the other cases. All cases however, are still far below the maximum possible transfer rate.

Figure 6.11: Transmission Times for all Scenarios in Trusted and Untrusted Setup

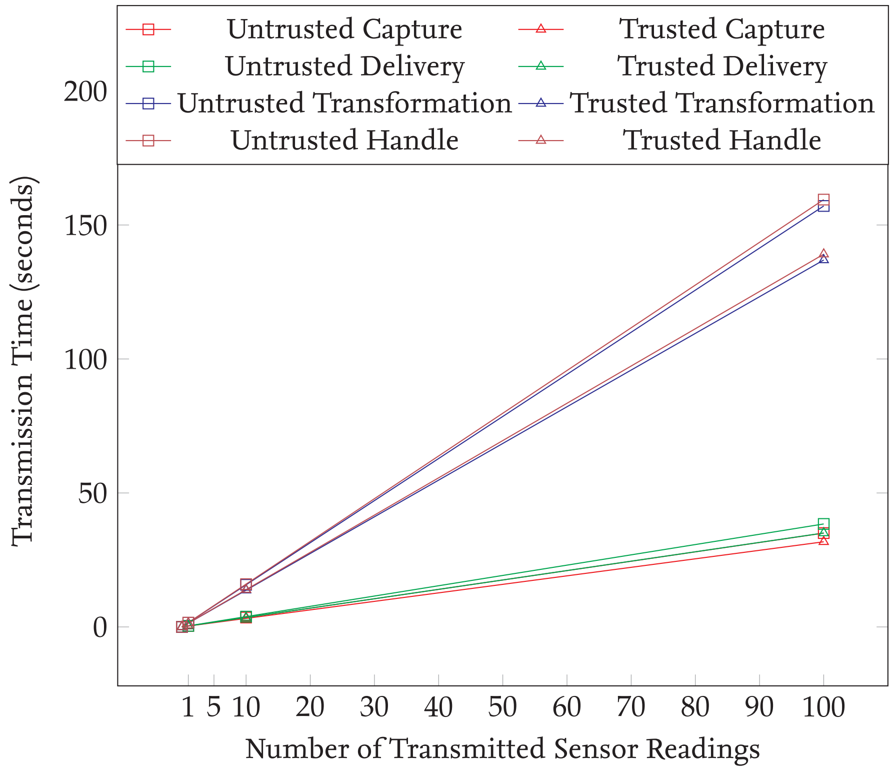
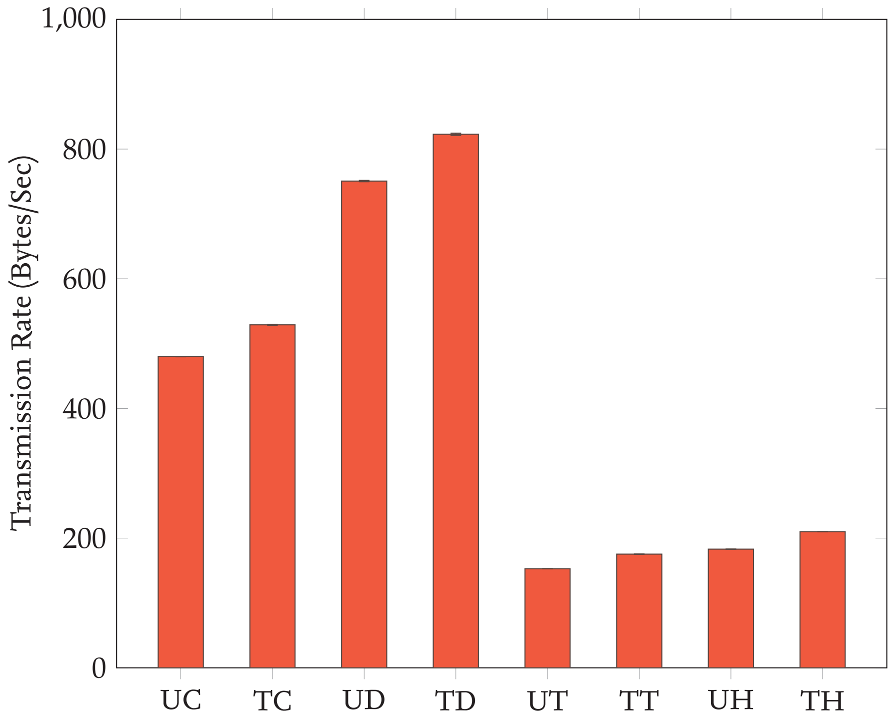


Figure 6.12: Transmission Times for all Scenarios in Trusted and Untrusted Setup



6.2.5 Footprint

In terms of footprint, our untrusted version requires more flash (as seen in Table 6.2) and RAM (as seen in Table 6.3). Other works in the field confirm this order of magnitude[42].

Using a Zephyr tool that can break down the contributions to the overall binary size by individual source files and folders, we have found that our code — including peripheral driver code and added STM32 HAL files — makes up only about 5 % of the binary. An amount of over 60 % is made up out of the cryptographic module. It includes components that were completely unused by us (e.g. AES, TLS, ECDSA), meaning there seems to be no mechanism to selectively exclude or include parts of this code. For the smaller, untrusted image the Crypto module only takes up 40 %.

Table 6.2: Flash Usage in Trusted and Untrusted Setup (in Bytes)

	Trusted	Untrusted
Flash (BL2)	32588	–
Flash (TF-M)	132088	–
Flash (Zephyr)	69530	131629
Flash Total	234206	131629

Table 6.3: RAM Usage in Trusted and Untrusted Setup (in Bytes)

	Trusted	Untrusted
RAM (BL2)	32588	–
RAM (TF-M)	132088	–
RAM (Zephyr)	69530	131629
RAM Total	234206	131629

As we have previously mentioned, TF-M and its bootloader also require a set flash memory layout, which divides the flash into different sections at build time. In Table 6.4, we depicted this division for our trusted setup and the TF-M default layout. An untrusted setup suffers no such division.

Table 6.4: Division of Flash Memory for our Board Using Different Layouts (in KB)

	TF-M Default	Trusted	Untrusted
Flash (BL2)	38	38	–
Flash (TF-M)	180	224	–
Flash (Zephyr)	36	172	512

6.3 Usability

In this chapter, we will assess the usability of our framework for developers in practice. The term usability in this context encompasses the ease of use, the addition or subtraction of code complexity, to what degree it is seamless to integrate and the amount of changes it requires to one's workflow. Based on our goal of the separation of concerns between trusted application (secure world) and normal application (non-secure world) developers, we split this evaluation into two parts to cover both cases.

6.3.1 Usability for Trusted Application Developers

As we have shown in §5.3.1, the TF-M implementation of a simple API such as ours can require invasive changes in the code. The restrictions that are put on the trusted application developer by TF-M are to the point, where the design of the API needs to be changed to meet these requirements. This can be witnessed most obviously in the fixed return type and in the limit of input/output parameters.

The code complexity and overhead is reduced in SFN mode, when compared to the IPC mode. The use of IPC mode can enable a higher isolation level and the possibility of accommodating an architecture using separated processors. As these benefits usually are not needed, We predict that the requirements of most use cases will be met by the SFN mode. As such, further implementation of IPC mode could be left out in order to lessen the workload on the trusted application developer.

In Table 6.5, we list the source lines of code (LoC) for the implementation of our API in the case of our trusted setup supporting both IPC and SFN and in the case of implementing it as a normal C program (untrusted setup). A difference of about 300 LoC can be seen, a large part of which can be considered boilerplate code to setup the IPC or SFN mechanisms. While changes and additions to the API cause ripple effects in the form of code changes throughout the code base, it is reasonable to assume that the TA developer is familiar with the nature of TF-M secure services and can handle any complications.

Table 6.5: Lines of Code for the API Implementation in Trusted and Untrusted Setup

	Trusted	Untrusted
Lines of Code	1273	968

Our evaluation here is based to a large part on the assumption of a preserved separation of concerns between both developers. However, as the trusted application developer would be the entity that is faced with the task of implementing this API for a wide array of different peripherals, we need to investigate whether this separation is respected in all scenarios of our API.

For the trusted capturing and trusted delivery cases, we can confidently state that the separation can be upheld, as the API is simple to implement and clear in what is returned from it. The most common implementation for these scenarios will come in the form of uncomplicated sensors. An API for these devices can be can be realized without much or any communication with the normal application developer.

In the case of trusted transformation, concessions must be made regarding the separation of concerns. Since the TA developers are required to implement all transformations, they either need to accurately predict all transformations that the application developer will need, or — the more likely case — a closer communication with the application developers is needed, in order to accommodate their use cases. Additionally, the implementation of many transformations would cause an increase in code size, which stands at odds with the TA developer's goal of a small TCB. Hence, supporting the trusted transformation scenario would cause the most complications.

6.3.2 Usability for Normal Application Developers

We see the usability for the normal application developer in terms of programming ease as satisfactory. We find the API straightforward to use and a peripheral that was previously untrusted could be exchanged easily for a trusted peripheral provided by the TA developer.

However, measurements that relate to the iteration times for the developer have worsened. On our development machine, the from-scratch build times and flash times (i.e. the time it takes to flash the new firmware onto the board) have doubled. While these are values that are heavily dependent on the specific machine, the amount to which they differed for us show a clear trend. Related to this; the booting times of the firmware went from instantaneous after flash (<10ms) to a noticeable delay that can be seconds long.

Additionally, a constraint is put on the developer in the form of flash memory division. As we have presented in §5.3.3, we ourselves were faced with the issue of insufficient flash when using TF-M. This is a problem that the TA developer ultimately cannot solve and instead the responsibility would need to lie with the TF-M developers. In our view, further efforts should be made to decrease TF-M's binary sizes, which in turn could reduce the size of the TCB and result in better security assurances.

In terms of specific API calls, both trusted capturing and trusted delivery are the least likely to cause problems. The worst scenario that we foresee in these calls would be an erroneous implementation by the TA developer. In that case, the normal application developer would have to communicate the bug to the TA developer, which could be a time consuming affair. Problems that result from this would be aggravated in case of devices being already in deployment. For trusted transformation, a wrong implementation would have the same problem. But the usability here would also be hindered when the normal application developer is in dire need of a specific transformation that is missing from the secure service. Once again, the two developers would have to begin time-consuming communications, which could result in unmet deadlines.

7 Future Work

In this chapter, we outline potential approaches to extend our trusted peripheral API in order to make it more useful in practice.

A worthwhile extension to our API could be the support of arbitrary trusted transformations. Although there are works that claim to support this use case[27], in our view, such a design can only be truly useful if it enables a way for arbitrary trusted transformations to take place without involving the trusted application developer. In this concept, the normal world could issue transformations as they wish without disturbing the data integrity. We foresee that any such implementation would need to keep a history of the performed operations in some form. A naive implementation could involve the inclusion of an interpreter inside the TEE. The normal world would pass the source code of a transformation program to the trusted side, upon which the code is executed and the transformation performed. To keep a history, the source code could be attach to the data. However, this obviously would lead to serious security concerns — outweighing any gains in the process. Work towards this could be done by identifying the most common, granular operations to perform on peripheral data.

For the purposes of deciding on the feasibility of our API in this work, we have implemented a prototype that is only interested in the data of one peripheral at a time. Though, when the normal application is faced with several peripherals that all individually capture data, it is conceivable that the application developer would like to aggregate the collected the data into one small-sized data packet. With our current API however, the MAC would be duplicated for every reading of every peripheral. Thus, the API could be extended to be able to package together several readings with just one MAC. A simple implementation could be just another API call that passes two sensor readings with two MACs and returns one packet with one MAC. A more interesting solution, however, could go a step further and implement a mechanism by which only one MAC is computed for the entire package in the first place. Perhaps the application can specify a list of peripherals for which it is interested in the data. The trusted application acknowledges this list and begins the capturing process for all of them. Upon full completion, a MAC is computed for the entire package, which are then both returned to the caller. This can get more complicated, as the different peripherals might vary wildly in their times to capture the data, so a performant implementation might have to return execution to the untrusted side.

8 Conclusion

To conclude this thesis, we summarize the steps we took and the findings we acquired.

We started out with a common scenario in mind; IoT devices in untrusted environments which capture peripheral data that is either sensitive, mission-critical or both. The complications to security that can arise from this situation motivated our ambitions to isolate the peripheral from any non-secure elements.

We first identified the use of a trusted execution environment to be appropriate for this task. However, we soon encountered inconsistencies around the various definitions of a TEE, which could be ascribed to its previous history of being associated with primarily smartphone devices. To establish a clear foundation, we put forward a list of requirements that we determined to be key characteristics of a TEE. To achieve specifically the TEE isolation requirement, we determined that the TrustZone technology for Cortex-M devices would be the best candidate, as it builds on the insights of the widely used TrustZone-A without compromising on an IoT device's ability to satisfy real-time constraints.

While searching for available TrustZone-M based TEEs, we soon discovered TF-M to be the only realistic option. As the reference implementation of the Platform Security Architecture Firmware Framework, we classified it as a fully qualified TEE when used in conjunction with a secure bootloader and RTOS. In the process of researching TF-M, we learned about the possibility of implementing custom secure services inside it. This later would become our avenue to implement a trusted peripheral API.

When looking for related works in the literature, we found TrustZone-M specific works that focus on peripheral isolation to be sparse. Instead, we acquired our most valuable insights from works that predated TrustZone-M and were concerned with the design of a secure peripheral API. We proceeded to build upon these works with a focus on our IoT scenario.

To formalize the security aspects of our scenario, we constructed a threat model. We identified all security goals that are important to us and at the same time, explicitly excluded the security aspect of availability, which we consider to be infeasible to achieve.

With the specific threats and goals in mind, we provided a definition of trusted peripherals that centered around three interaction scenarios. We predict these scenarios to be the most common in deployment and as such, categorized our definition accordingly. We then determined what security traits can be preserved for every scenario and at what cost.

With these findings at hand, we were able to propose an API that satisfies the needs of all three use cases with a straightforward implementation for each one. To separate the implementation into the secure world, we performed the necessary steps to convert this code to a TF-M secure service. We identified possible sources of friction and overhead during this process.

Nevertheless, we moved to demonstrate the feasibility of our trusted peripherals in practice using a simple prototype: A board that captures temperature and humidity data from a connect sensor. While straightforward, this setup served as an important proof-of-concept and could still be imagined to have real world applications. In a security analysis of this prototype, we have found the benefits to security to be substantial.

In the evaluation of the implementation's performance, we encountered a problem: Due to circumstances that we found to be out of our control, our secure setup outperformed an ordinary implementation. To compensate for this uncertainty, we devised a way to include an adjusted measurements by adding a worst-case overhead to our findings. With this data, we were able to conclude that the performance overhead from TrustZone-M and TF-M were acceptable even when being pessimistic.

However, when assessing the usability for the developers, we had to make concessions — many of which can be ascribed to the choice of TF-M as a TEE. These included bloated binary sizes, a large TCB and overly severe constraints on the flash memory layout. For our API specifically, we determined that the usability for both developers is satisfactory and preserves a separation of concerns in most cases.

In conclusion, we believe that our work presents a contribution towards more secure devices in the field of IoT. We anticipate that interest in this topic will pick up with time, which could present an opportunity to gain more widespread adoption of best practices in IoT security — the use of trusted peripherals being one part of it.

Although a solution to the overall problem seems to be merely beneficial for now, we predict that — as IoT devices gradually permeate every aspect of our lives — a reasonable strategy to address this issue is going to become increasingly necessary.

Bibliography

[1] *Advanced Trusted Environment: OMTP TR1*. Tech. rep. (accessed on September 25, 2023). May 2009. URL: https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtpadvancedtrustedenvironmentomtptr1v11.pdf.

[2] Thaynara Alves and D. Felton. "Trustzone: Integrated Hardware and Software Security". In: (Jan. 2004).

[3] *AN521 - Example SSE-200 Subsystem for MPS2+*. (accessed on September 25, 2023). ARM Limited. URL: https://developer.arm.com/documentation/dai0521/latest/.

[4] Noah Apthorpe, Dillon Reisman, and Nick Feamster. *A Smart Home is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic*. 2017. arXiv: 1705.06805 [cs.CR].

[5] Ghada Arfaoui, Saïd Gharout, and Jacques Traoré. "Trusted Execution Environments: A Look under the Hood". In: *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. Apr. 2014, pp. 259–266. DOI: 10.1109/MobileCloud.2014.47.

[6] *ARM Cortex-M23*. (accessed on September 25, 2023). ARM Limited. URL: https://developer.arm.com/Processors/Cortex-M23.

[7] *Arm Developer: Trusted Firmware-M*. (accessed on September 25, 2023). ARM Limited. URL: https://developer.arm.com/Tools%20and%20Software/Trusted%20Firmware-M.

[8] *ARM Mbed*. (accessed on September 25, 2023). ARM Limited. URL: https://os.mbed.com.

[9] *ARM Security Technology: Building a Secure System using TrustZone® Technology*. Tech. rep. (accessed on September 25, 2023). 2005. URL: https://documentation-service.arm.com/static/5f212796500e883ab8e74531.

[10] *Arm® Firmware Framework for M 1.1 Extension*. Tech. rep. (accessed on September 25, 2023). Jan. 2023. URL: https://documentation-service.arm.com/static/63d10cf5eae484354a6c9d56.

[11] *Arm® Platform Security Architecture Firmware Framework 1.0*. (accessed on September 25, 2023). ARM Limited. URL: https://documentation-service.arm.com/static/64a2ed35df6cd61d528c4132.

[12] *Arm® TrustZone Technology for the Armv8-M Architecture*. (accessed on September 25, 2023). ARM Limited. URL: https://documentation-service.arm.com/static/5f873034f86e16515cdb6d3e.

[13] *Armv8-M Architecture Reference Manual.* Tech. rep. (accessed on September 25, 2023). 2015. URL: `https://documentation-service.arm.com/static/64ac1658df6cd61d528c5cd8`.

[14] *Asset tracker TMSA example.* (accessed on September 25, 2023). ARM Limited. URL: `https://developer.arm.com/-/media/Files/pdf/PlatformSecurityArchitecture/Analyze/TMSA_Asset_Tracker_Bundle_BET01.zip?revision=a55da6e1-b318-4b57-a4b1-af9856aefd6d`.

[15] Alessandro Barenghi et al. "Exploring Cortex-M Microarchitectural Side Channel Information Leakage". In: *IEEE Access* 9 (2021), pp. 156507–156527. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2021.3124761`.

[16] Johannes Bauer and Felix Freiling. "Towards Cycle-Accurate Emulation of Cortex-M Code to Detect Timing Side Channels". In: *2016 11th International Conference on Availability, Reliability and Security (ARES).* Aug. 2016, pp. 49–58. DOI: `10.1109/ARES.2016.94`.

[17] Ferdinand Brasser et al. "Regulating ARM TrustZone Devices in Restricted Spaces". In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services.* MobiSys '16. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 413–425. ISBN: 9781450342698. DOI: `10.1145/2906388.2906390`. URL: `https://doi.org/10.1145/2906388.2906390`.

[18] *FreeRTOS.* (accessed on September 25, 2023). Amazon Web Services, Inc. URL: `https://www.freertos.org`.

[19] *GlobalPlatform Device Technology: TEE Client API Specification.* (accessed on September 25, 2023). GlobalPlatform. July 2010. URL: `https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf`.

[20] *GlobalPlatform Technology: TEE System Architecture v1.3.* Tech. rep. (accessed on September 25, 2023). May 2022. URL: `https://globalplatform.org/specs-library/tee-system-architecture/`.

[21] *GlobalPlatform.org.* (accessed on September 25, 2023). GlobalPlatform. URL: `https://globalplatform.org/`.

[22] *HKG18-212 - Trusted Firmware M: Introduction.* (accessed on September 25, 2023). Linaro. Mar. 2018. URL: `https://static.linaro.org/connect/hkg18/presentations/hkg18-212.pdf`.

[23] *HKG18-212 - Trusted Firmware M: Introduction.* (accessed on September 25, 2023). Linaro. Mar. 2018. URL: `https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/refs/tags`.

[24] *Innovative Sensor Technology.* (accessed on September 25, 2023). Innovative Sensor Technology IST AG. URL: `https://www.ist-ag.com`.

[25] *Innovative Sensor Technology.* (accessed on September 25, 2023). Waveshare Electronics. URL: `https://www.waveshare.com/0.96inch-oled-b.htm`.

[26] Hassaan Janjua et al. “Towards a Standards-Compliant Pure-Software Trusted Execution Environment for Resource-Constrained Embedded Devices”. In: *Proceedings of the 4th Workshop on System Software for Trusted Execution*. SysTEX ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019. isbn: 9781450368889. doi: 10.1145/3342559.3365338. url: https://doi.org/10.1145/3342559.3365338.

[27] Hassaan Janjua et al. “Trusted Operations on Sensor Data †”. In: *Sensors* 18 (Apr. 2018), p. 1364. doi: 10.3390/s18051364.

[28] Burt Kaliski. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313. Mar. 1998. doi: 10.17487/RFC2313. url: https://www.rfc-editor.org/info/rfc2313.

[29] *Kinibi-M*. (accessed on September 25, 2023). Trustonic. url: https://www.trustonic.com/technical-articles/kinibi-m/.

[30] Zili KOU et al. “Cache Side-channel Attacks and Defenses of the Sliding Window Algorithm in TEEs”. In: *Design, Automation and Test in Europe Conference and Exhibition*. Apr. 2023, pp. 1–6. doi: 10.23919/DATE56975.2023.10137116.

[31] Matthew Lentz et al. “SeCloak: ARM Trustzone-Based Mobile Peripheral Control”. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’18. Munich, Germany: Association for Computing Machinery, 2018, pp. 1–13. isbn: 9781450357203. doi: 10.1145/3210240.3210334. url: https://doi.org/10.1145/3210240.3210334.

[32] Zhen Ling et al. “Security Vulnerabilities of Internet of Things: A Case Study of the Smart Plug System”. In: *IEEE Internet of Things Journal* 4.6 (2017), pp. 1899–1909. doi: 10.1109/JIOT.2017.2707465.

[33] He Liu et al. “Software Abstractions for Trusted Sensors”. In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’12. Low Wood Bay, Lake District, UK: Association for Computing Machinery, 2012, pp. 365–378. isbn: 9781450313018. doi: 10.1145/2307636.2307670. url: https://doi.org/10.1145/2307636.2307670.

[34] Lan Luo et al. “fASLR: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems”. In: *IEEE Internet of Things Journal* 9.18 (Sept. 2022), pp. 17120–17135. issn: 2327-4662. doi: 10.1109/JIOT.2022.3190374.

[35] Lan Luo et al. “On Security of TrustZone-M-Based IoT Systems”. In: *IEEE Internet of Things Journal* 9.12 (June 2022), pp. 9683–9699. issn: 2327-4662. doi: 10.1109/JIOT.2022.3144405.

[36] Brian McGillion et al. “Open-TEE – An Open Virtual Trusted Execution Environment”. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, Aug. 2015. doi: 10.1109/trustcom.2015.400. url: https://doi.org/10.1109%2Ftrustcom.2015.400.

[37] *MCUboot*. (accessed on September 25, 2023). Linaro. url: https://www.trustedfirmware.org/projects/mcuboot/.

[38] Gary Mullen and Liam Meany. “Assessment of Buffer Overflow Based Attacks On an IoT Operating System”. In: *2019 Global IoT Summit (GIoTS)*. June 2019, pp. 1–6. DOI: 10.1109/GIOTS.2019.8766434.

[39] *MultiZone Security for RISC-V*. (accessed on September 25, 2023). HEX-Five Security. URL: https://hex-five.com/multizone-security-tee-riscv/.

[40] *National Vulnerability Database*. (accessed on September 25, 2023). National Institute of Standards and Technology. URL: https://nvd.nist.gov.

[41] *Network camera threat model and security analysis example*. (accessed on September 25, 2023). ARM Limited. URL: https://developer.arm.com/-/media/Files/pdf/PlatformSecurityArchitecture/Analyze/TMSA_Network_Camera_Bundle_BET00.zip?revision=e9afcb64-f3a6-47ed-a39f-f22d9f85ee98.

[42] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. “uTango: An Open-Source TEE for IoT Devices”. In: *IEEE Access* 10 (2022), pp. 23913–23930. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3152781.

[43] *OP-TEE*. (accessed on September 25, 2023). Linaro. URL: https://www.trustedfirmware.org/projects/op-tee/.

[44] *OpenIoT Summit Europe 2018: Compartmentalization in IoT - Trusted Firmware M Secure Partitioning*. (accessed on September 25, 2023). ARM Limited. Oct. 2018. URL: https://tf-m-user-guide.trustedfirmware.org/.

[45] *OpenIoT Summit Europe 2018: Compartmentalization in IoT - Trusted Firmware M Secure Partitioning*. (accessed on September 25, 2023). ARM Limited. Oct. 2018. URL: https://elinux.org/images/4/4b/Trusted-Firmware-M-Secure-Partitioning-%E2%80%93-Compartmentalization-in-IoT-Miklos-Balint-Ken-Liu-Arm.pdf.

[46] Danny Palmer. “Critical IoT security camera vulnerability allows attackers to remotely watch live video - and gain access to networks”. In: *ZDNET* (Aug. 17, 2021). (accessed on September 25, 2023). URL: https://www.zdnet.com/article/critical-iot-security-camera-vulnerability-allows-attackers-to-remotely-watch-live-video-and-gain-access-to-networks/.

[47] BBC Panorama. “The tech flaw that lets hackers control surveillance cameras”. In: *BBC* (June 26, 2023). (accessed on September 25, 2023). URL: https://www.bbc.com/news/technology-65975446.

[48] *Physical attack mitigation in Trusted Firmware-M*. (accessed on September 25, 2023). Linaro. URL: https://tf-m-user-guide.trustedfirmware.org/design_docs/tfm_physical_attack_mitigation.html.

[49] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Comput. Surv.* 51.6 (Jan. 2019). ISSN: 0360-0300. DOI: 10.1145/3291047. URL: https://doi.org/10.1145/3291047.

[50] *Platform Security Model 1.1.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/app/uploads/2021/12/JSADEN014_PSA_Certified_SM_V1.1_BET0.pdf.

[51] *ProvenCore-M.* (accessed on September 25, 2023). ProvenRun. URL: https://provenrun.com/products/provencore-m/.

[52] *PSA Certified.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/.

[53] *PSA Certified 10 Security Goals.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/app/uploads/2020/11/PSA_Certified_10_security_goals.pdf.

[54] *PSA Certified Level 1.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/getting-certified/silicon-vendor/overview/level-1/.

[55] *PSA Certified Level 2.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/getting-certified/silicon-vendor/overview/level-2/.

[56] *PSA Certified Level 2.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/getting-certified/silicon-vendor/overview/level-3/.

[57] *PSA Certified Products.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/certified-products/.

[58] *PSA Cryptography API.* (accessed on September 25, 2023). Platform Security Architecture. URL: https://armmbed.github.io/mbed-crypto/html/.

[59] *QEMU.* (accessed on September 25, 2023). QEMU Team. URL: https://www.qemu.org/.

[60] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications.* Helsinki, Finland, Aug. 2015. DOI: 10.1109/Trustcom.2015.357. URL: https://hal.science/hal-01246364.

[61] Ammar Salman and Wenliang Du. "Securing Mobile Systems GPS and Camera Functions Using TrustZone Framework". In: July 2021, pp. 868–884. ISBN: 978-3-030-80128-1. DOI: 10.1007/978-3-030-80129-8_58.

[62] *Secure OTA Updates for Cortex-M Devices with FreeRTOS.* (accessed on September 25, 2023). Amazon Web Services, Inc. URL: https://www.freertos.org/2021/07/secure-ota-updates-for-cortex-m-devices-with-freertos.html.

[63] *Security for Arm Cortex-M devices with FreeRTOS.* (accessed on September 25, 2023). Amazon Web Services, Inc. URL: https://www.freertos.org/2020/07/security-for-arm-cortex-m-devices-with-freertos.html.

[64] Bodo Selmke et al. "Locked out by Latch-up? An Empirical Study on Laser Fault Injection into Arm Cortex-M Processors". In: *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC).* Sept. 2018, pp. 7–14. DOI: 10.1109/FDTC.2018.00010.

[65] *Simplifying Security for OEMs: A Four Step Framework.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/app/uploads/2020/02/Four_Steps_to_Device_Security_PSA_Certified.pdf.

[66] *Smart Camera SESIP Profile (threat model) example.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/app/uploads/2022/08/JSADEN016-Smart_Camera_SESIP_Profile-v0.4_37.pdf.

[67] *Smart Speaker SESIP Profile (threat model) example.* (accessed on September 25, 2023). ARM Limited. URL: https://www.psacertified.org/app/uploads/2022/06/JSADEN015-Smart_Speaker_Voice_Assistant_SESIP_Profile-Release.pdf.

[68] *Smart water meter threat model and security analysis example.* (accessed on September 25, 2023). ARM Limited. URL: https://developer.arm.com/-/media/Files/pdf/PlatformSecurityArchitecture/Analyze/TMSA_Water_Meter_Bundle_BET00.zip?revision=b3a22663-55dd-4d37-8d33-f2c01cf5c170.

[69] *STM32 Nucleo-144 development board with STM32L552ZE MCU.* (accessed on September 25, 2023). STMicroelectronics. URL: https://www.st.com/en/evaluation-tools/nucleo-l552ze-q.html.

[70] *STM32L5 — Certificates.* (accessed on September 25, 2023). Platform Security Architecture. URL: https://www.psacertified.org/products/stm32l5/certificates/#security-level-1.

[71] Lizhi Sun et al. "LEAP: TrustZone Based Developer-Friendly TEE for Intelligent Mobile Apps". In: *IEEE Transactions on Mobile Computing* (2022), pp. 1–18. ISSN: 1558-0660. DOI: 10.1109/TMC.2022.3207745.

[72] Kevin Townsend. "Solving the Quantum Decryption 'Harvest Now, Decrypt Later' Problem". In: *SecurityWeek* (Feb. 16, 2022). (accessed on September 25, 2023). URL: https://www.securityweek.com/solving-quantum-decryption-harvest-now-decrypt-later-problem/.

[73] *Trusted Firmware-A.* (accessed on September 25, 2023). Linaro. URL: https://www.trustedfirmware.org/projects/tf-a/.

[74] *Trusted Firmware-M.* (accessed on September 25, 2023). Linaro. URL: https://www.trustedfirmware.org/projects/tf-m/.

[75] *Trusted Firmware-M Technical Overview*. Tech. rep. (accessed on September 25, 2023). Feb. 2023. URL: https://www.trustedfirmware.org/docs/TrustedFirmware-MTechnicalOverviewQ1-2023.pdf.

[76] *TrustedFirmware-M: Security Advisories*. (accessed on September 25, 2023). Linaro. URL: https://tf-m-user-guide.trustedfirmware.org/security/security_advisories/index.html.

[77] *TrustZone for Cortex-A*. (accessed on September 25, 2023). ARM Limited. URL: https://www.arm.com/technologies/trustzone-for-cortex-a.

[78] *TrustZone for Cortex-M*. (accessed on September 25, 2023). ARM Limited. URL: https://www.arm.com/technologies/trustzone-for-cortex-m.

[79] *TrustZone-M(eh): Breaking ARMv8-M's security*. (accessed on September 25, 2023). Chaos Computer Club e. V. Dec. 2019. URL: https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security.

[80] Karl Wüst and Arthur Gervais. "Do you Need a Blockchain?" In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. June 2018, pp. 45–54. DOI: 10.1109/CVCBT.2018.00011.

[81] Joseph Yiu. *Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors -*. London: Newnes, 2020. ISBN: 978-0-128-20736-9.

[82] Rui Yu, Xiaohua Zhang, and Minyuan Zhang. "Smart Home Security Analysis System Based on The Internet of Things". In: *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*. Mar. 2021, pp. 596–599. DOI: 10.1109/ICBAIE52039.2021.9389849.

[83] Peterson Yuhala. "Enhancing IoT Security and Privacy with Trusted Execution Environments and Machine Learning". In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*. 2023, pp. 176–178. DOI: 10.1109/DSN-S58398.2023.00047.

[84] *Zephyr*. (accessed on September 25, 2023). Zephyr Project. URL: https://www.zephyrproject.org/.

[85] *Zephyr Developer Summit: Trusted Firmware M in Zephyr*. (accessed on September 25, 2023). Linaro. June 2021. URL: https://drive.google.com/file/d/18aque_mH_tifmfiZBKrjbq4KkDMWMtgW/view.

[86] *Zephyr Documentation*. (accessed on September 25, 2023). Zephyr Project. URL: https://docs.zephyrproject.org/latest/.

[87] Ning Zhang et al. "TruSense: Information Leakage from TrustZone". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. Apr. 2018, pp. 1097–1105. DOI: 10.1109/INFOCOM.2018.8486293.