University of Stavanger

HÅKON TØMTE

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Creating a blockchain in TrustZone using Rust

Bachelor's Thesis - Computer Science - December 2022

I, **Håkon Tømte**, declare that this thesis titled, “Creating a blockchain in TrustZone using Rust” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master’s degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Abstract

Many modern processors have security mechanisms that facilitate the creation of a Trusted Execution Environment (TEE), which is isolated from the general-purpose OS. Privacy and integrity challenges arise in IoT scenarios, characterized by untrusting stakeholders and easily accessible and vulnerable devices. Systems using blockchain and TEE technology can solve privacy and integrity challenges by establishing trust between devices, vendors, and stakeholders. ARM processors are prevalent in IoT scenarios, thus, this thesis investigates the feasibility of implementing such systems on ARM's TEE-enabler, TrustZone. To investigate this, I present a prototype of a core system component, the blockchain issuer, in an ARM TrustZone-based TEE and evaluate it. It builds the blockchain inside the TEE, guaranteeing the recordings' confidentiality, integrity, and immutability. It is vital that TEE applications, like the blockchain issuer, are secure and safely implemented. The Rust OP-TEE SDK is an open-source SDK that allows for writing Trusted Applications in the programming language Rust. The main advantage of Rust is that Rust is memory-safe, and thus greatly increases the security of the application. Rust also provides a rich infrastructure of public crates for developers. However, OP-TEE is incompatible with many of these crates. Rust OP-TEE also lacks documentation, specifically regarding the development of new Trusted Applications. Therefore, this thesis provides Rust OP-TEE documentation for developing TAs, and challenges the claim that Rust OP-TEE supports developers to import third-party crates.
The evaluation shows that, with optimizations, a blockchain issuer written in Rust OP-TEE on ARM's TrustZone securely establishes trust from the TEE, with the price of a 5 times performance overhead compared to a normal world implementation.

Acknowledgements

I would like to thank my supervisor, Leander Jehl, for their enthusiasm and help with writing this thesis.
The Rust OP-TEE SDK contributors have been very helpful, and continually elaborated on and answered my questions/issues on Github.
Gernot Heiser's writing guide is a great and comprehensive guide for technical writing and was used extensively.

Contents

Chapter 1

Introduction

1.1 Motivation

Internet-of-Things (IoT) and automation promise increased efficiency and profits. Sensors, sophisticated AI models, and device-to-device integration realize these promises. New business models are necessary to fully utilize the IoT's potential, and these technologies may belong to several parties. We can imagine a scenario in which a farmer's sensor automatically sends pictures of crops to another company's analytical AI model. The model could then responds with "Ready for harvest" or "not ready". Intruders could try to infiltrate these systems to analyze, steal from, or hurt competitors. This creates the challenge of establishing trust in data usage and data integrity. A blockchain can establish trust and prove the integrity of data. However, the majority of consortium blockchains rely on replication and distribution across numerous locations to establish trust, and they need advanced agreement protocols among nodes. This introduces significant hardware costs and energy use, which acts as a barrier to entry for small scenarios and businesses. To solve challenges of privacy and integrity in IoT scenarios, [Bleeke et al., 2022] presented *ChainBox*, a TEE-based smart contract execution framework. ChainBox uses blockchain and TEE technology to establish trust between devices, vendors, and stakeholders. By building the blockchain in a Trusted Execution Environment (TEE), its validity is trusted without the use of replication and distribution and is therefore applicable in small-business IoT scenarios. The TEE guarantees the confidentiality and integrity of the clients' data and the blockchain ensures the immutable recording of transactions. However,

their ChainBox implementation used an Intel SGX-based TEE, and ARM processors are far more prevalent in IoT scenarios, thus, this thesis investigates the feasibility of porting the ChainBox framework to ARM's TrustZone. Such an implementation will allow for trusted, cheap communication between IoT devices, which is crucial for the digitalization of industries and sectors.

1.2 Approach

I implement a prototype of the core ChainBox component, the Ordering Service Enclave (OSE), using the Rust OP-TEE SDK to investigate the feasibility of porting the complete ChainBox framework to ARM's TrustZone and investigate the viability of the Rust OP-TEE SDK. The Rust OP-TEE SDK facilitates the use of Rust in OP-TEE Trusted Applications (TA), which increases application security, compared to the traditionally used *C*. The development process reveals limitations in the Rust OP-TEE SDK and its documentation. The OSE is deployed to a Raspberry Pi for performance benchmarking and to evaluate the overhead introduced by the increased security of the TEE.

To summarize, the main approach to investigate the feasibility of porting ChainBox to TrustZone and the viability of Rust OP-TEE SDK is to:

- Set up the OP-TEE environment to facilitate the usage of TrustZone-based TEE technology
- Implement an Ordering Service Enclave prototype using the Rust OP-TEE SDK.
- Write in the programming language Rust, to ensure the memory safety of the application
- Deploy the Ordering Service Enclave to a Raspberry Pi for performance benchmarking

1.3 Contributions

The successful implementation of an Ordering Service Enclave prototype indicates that the ChainBox framework can be built on ARM TrustZone. Rust's enforced memory-safety features greatly increase the security of the OSE compared

to the memory-unsafe and OP-TEE alternative, the *C* programming language. The development process revealed that Rust OP-TEE lacked proper documentation to guide new developers and revealed limitations when importing third-party Rust crates. Therefore, this thesis provides Rust OP-TEE documentation for developing TAs, and challenges [Wan et al., 2020]'s claim that Rust OP-TEE supports developers to import third-party crates. The documentation's purpose is to expedite the development process in Rust OP-TEE by providing in-depth guidance on how the SDK is used. Lastly, I evaluated the performance benchmarks of the OSE and implemented a throughput-optimized version.

To summarize, the main contributions of this thesis are:

- The implementation of an Ordering Service Enclave prototype in Rust on ARM TrustZone (Source code: [Tømte, 2022])
- Providing Rust OP-TEE documentation for developing Trusted Applications.
- Reveal limitations of the Rust OP-TEE SDK when importing third-party crates.
- Analyzed the performance benchmarks and implemented a throughput-optimized OSE

Chapter 2

Background

2.1 ARM TrustZone

Arm processors control the majority of mobile and embedded markets, in over 60% of all embedded devices and 4.5 billion mobile phones [Pinto and Santos, 2019]. Arm has extended TrustZone-support for the smallest low-end devices, Arm estimates the number of these to reach nearly 1 trillion by 2035. Such a large number of networked devices will produce and share a large volume of data containing privacy- and security-sensitive information, which frequently attracts cybercriminals. To store, manage, and carry out sensitive activities on sensitive data, a secure environment must be created inside an IoT device. This is the first step in achieving security in the IoT.

This section describes ARM TrustZone, and how it creates a secure environment. Hardware and software architecture, TrustZone-enabled technology, and vulnerabilities are covered.

Arm TrustZone is a hardware security extension introduced to Arm processors. TrustZone implements System-on-Chip (SoC) and CPU system-wide security. The technology's basis is the concept of partitioning all of the SoC's hardware and software into a *normal world* and a *secure world*. At any given time, the processor operates exclusively in one of these worlds. The worlds are hardware separated and serve different functions. The normal world is rich in features and flexibility, while the secure world should only perform security-sensitive operations, have far fewer features, and be isolated from the untrusted normal world. By limiting the number of features in the secure world, the susceptible surface

area is reduced. The worlds are completely hardware-isolated and granted uneven privileges. Normal world software cannot directly access secure world resources. However, the secure world can access all resources. The normal world can only *request* a world-switch.

This robust hardware-enforced boundary between worlds creates new possibilities for application and data security. Particularly, by binding the main Operating System (OS) to only operate inside the normal world, crucial applications can reside in the secure world and be isolated from the Rich OS, even if the Rich OS is infiltrated or corrupted.

Arm TrustZone is key-enabler hardware that is leveraged in Trusted Execution Environments (TEE) technology. A TEE is a secure area of the main processor (like the secure world) that a Trusted OS resides in to execute crucial applications and establish cross-world communication through API calls. Examples of Trusted OSes on TrustZone technology are Linaro's OP-TEE and Qualcomm's QSEE. Section 2.2 discusses TEEs further.

2.1.1 Hardware Architecture

This subsection describes key architectural features of Arm Cortex-A processors. Cortex-A is the type of processor found in the Raspberry Pi 3, this type is largely used in mobile devices.

The hardware-based security built into SoCs provides a foundation for improved system security, by creating two protection domains. As mentioned, the processor operates exclusively in one of these worlds at any time. The value of a new 33rd processor bit, called the Non-Secure (NS) bit, determines the current operating world. The Secure Configuration Register (SCR) reads the NS value and passes it through the system [Pinto and Santos, 2019]. Figure 2.1 illustrates the concept of the two worlds. When NS is set (1), the processor is in the normal world and can only access the normal world's resources. After a world switch, the secure world is active, the NS bit is unset (0), and the secure world's resources become available.

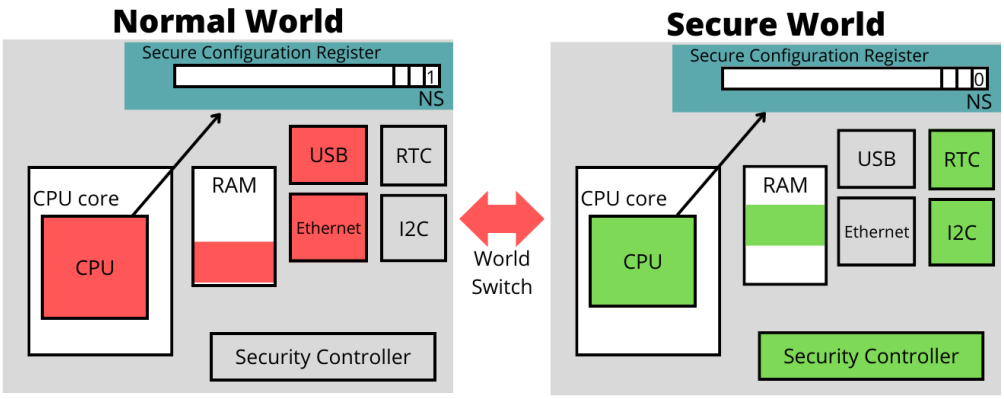

Figure 2.1: The NS bit determines which world the processor executes in. The worlds only have access to specific resources. In this example, the RAM is partially owned by the normal world and the secure world. Remade from [Genode,].

TrustZone has an extra processor mode, *Monitor Mode* that preserves the processor state when world-switching occurs. Monitor Mode saves the state of the current world and restores the state of the world being switched to. The processor enters Monitor Mode when switching worlds. A new privileged instruction, Secure Monitor Call (SMC), allows Monitor Mode to be invoked. This architecture is illustrated in Figure 2.2. The Secure Monitor controls world switching and resides in the secure world.

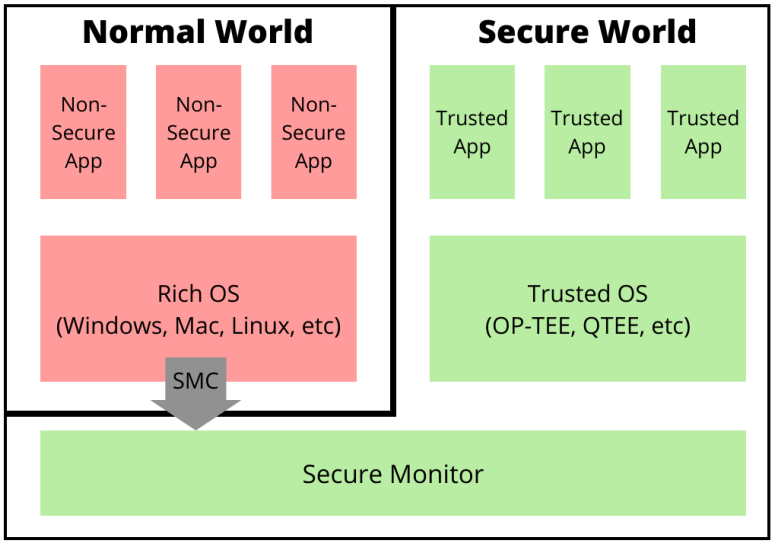

Figure 2.2: The TrustZone architecture for Cortex-A processors. The Secure Monitor can be used to switch between the worlds. Remade from [Pinto and Santos, 2019]

ARM has implemented this split-environment processor with some system IP additions. These components enforce security restrictions and maintain the advantages of ARM's designs, such as low power consumption.

The TrustZone Address Space Controller (TZASC) allows the dynamic classification of memory as secure or non-secure. Controlled by the secure world, the TZASC enables the partitioning of a single memory unit, instead of requiring separate secure and non-secure units. The TZASC allows the partitioning of the RAM shown in Figure 2.1, in which half of the RAM belongs to the secure world and the other half to the normal world.

The TrustZone Memory Adapter (TZAM) allows a similar partition of memory, but for on-chip static memory (SRAM) [Ngabonziza et al., 2016].

The hardware architecture allows the separation of two protection domains by designating physical memory and resources. The processor operates in only one domain at any time and can enter Monitor Mode to switch domains. Both worlds can request to enter Monitor Mode using an SMC instruction.

2.1.2 Software Architecture

The secure world can be implemented as a fully-fledged OS or as a software library. The considerations when implementing a secure and a non-secure operating system on TrustZone hardware will be the focus of this subsection. Generally, a standard operating system, such as Linux or Windows, runs in the normal world, and a Trusted OS, like OP-TEE, runs in the secure world. The functions of TrustZone for the device are: first, to securely boot both OSes; second, to provide a secure and reliable means of communication between the two, while protecting the secure world from the normal world [Ngabonziza et al., 2016].

Secure boot

Attackers may attempt to break the software while the device is powered off, by for example replacing the OS image in flash with a malicious version. Without verification of both OS images at boot time, the device may boot a malicious version of an OS and expose the system to attacks. ARM has therefore designed TrustZone-enabled systems to use a Secure Boot Sequence. The sequence establishes a Chain of Trust (CoT). The Root of Trust (RoT) is the foundation of the chain and must be implicitly trusted, then all other components must be authenticated before being executed. Authentication is done using cryptographic signatures, also called digital signatures. One creates a digital signature with an asymmetric key pair (e.g RSA encryption algorithm), by signing some value using the private key and verifying it later using the public key. Verification proves that the private key signed the signature. This verification is done at each step of the booting sequence to ensure that the CoT has not been broken.
This Secure Boot Sequence will prevent the booting of external software, like a malicious OS image, because that external software does not have the correct signature. As long as the RoT is secure, the device will securely boot the OSes.

Secure cross-world communication

Normal world to secure world switching is only possible through these exceptions: an interrupt, an external abort, or an explicit call with the SMC instruction. The hardware interrupts and aborts only support full world switching, but SMC also supports passing messages without a complete switch. Monitor mode and

the Secure Monitor establish the means of cross-world communication.

The software architecture allows for secure booting using a Chain of Trust (Cot), provided that the Root of Trust (RoT) is secure. Normal world to secure world switching is strictly controlled, to protect the secure world.

2.1.3 TrustZone as a key-enabler of TEEs

A TEE is an isolated environment that can execute Trusted Applications (TA) without the interference of the local and function-rich OS. Most importantly, the TEE must be isolated and insusceptible from the local OS, so the TEE can guarantee the confidentiality and integrity of its operations. TrustZone is a natural enabler for building TEE support systems. Essentially, the secure world provides an isolated execution environment where the TEE can reside. Monitor Mode and the SMC instruction facilitate world-switching and message passing.

2.1.4 Vulnerabilities

This subsection describes the key vulnerabilities of TrustZone and TrustZone-based TEEs. In practice, operations performed in the TEE are implicitly trusted and the TEE can be treated as a Root of Trust (RoT) in systems. Therefore, it is crucial to be aware of the underlying vulnerabilities of TrustZone and know that the TEE is not perfectly secure.

Establishing a Root of Trust in TrustZone

A Chain of Trust (CoT) is never more secure than its Root of Trust (RoT). The secrecy of the RoT's private key ensures the trust of the entire computational system [Ngabonziza et al., 2016]. AMD and Intel use a Trusted Platform Module (TPM) as their RoT. Thus, the device's physical features guarantee the secrecy of that key. However, TrustZone does not explicitly specify what the RoT of the system should be or provide secure key storage. TrustZone provides system-wide isolation of two execution environments, but an external physical device is required to establish a secure RoT. Alternatively, [Zhao et al., 2014] have prototyped an alternative RoT based on physical manufacturing variations and a software-based

TPM.
TrustZone-based TEEs are in practice implicitly trusted, but TrustZone does not provide a designated RoT. A secure and preferably physical RoT should be established for the TEE to be trusted.

TEE Vulnerabilities

[Pinto and Santos, 2019] found more than 130 vulnerabilities regarding TrustZone and TrustZone-based TEEs. The majority of these flaws come from bugs that exist in some providers' implementations of the TEEs. These vulnerabilities have for example been exploited to execute arbitrary code in the TEE or in some cases defeat secure boot. [Rosenberg, 2014] describes a vulnerability in Qualcomm's QSEE, which caused any applications using TrustZone for security assurance to be completely compromised. [Suciu et al., 2020], a more recent example, describes vulnerabilities that allowed for Horizontal Privilege Escalation through applications running in the TEE leaking stored data.

Essentially, even though the security design of TrustZone might be correct, that is, secure architecture and true isolation, the code running inside the TEE may contain vulnerabilities. The vulnerability can be in the Trusted OS (OP-TEE, QSEE, etc.) or in the Trusted Applications that run in the TEE. Attackers may take advantage of these flaws to corrupt the TEE and jeopardize the system's trust state.

Remote Attestation

A key function of a TEE is to provide remote attestation, to prove to third parties that the application is correctly running inside of the TEE. While Intel SGX provides remote attestation, TrustZone lacks built-in mechanisms for attestation. Nevertheless, researchers have proposed variants of remote attestation protocols for Arm TrustZone, thus extending the built-in capabilities for attestation [Mé nétrey et al., 2022]. These propositions require a root of trust in the secure world; a secure source of randomness for cryptographic operations; and a secure boot mechanism.
So although TrustZone lacks built-in support for remote attestation, a Trusted OS or a Trusted Application can utilize a secure RoT, and cryptographic operations to issue evidence (digital signatures). Therefore, each Trusted OS (like OP-TEE)

should provide its own remote attestation process. As a result, remote parties or suppliers may still confirm the validity of the attesters and the issued evidence in a TrustZone-based TEE. [Li et al., 2015] and [Ménétrey et al., 2022] are examples of remote attestation features extended to ARM TrustZone.

ARM TrustZone enables the separation of two worlds, one functionally rich and one secure world. ARM TrustZone is a hardware security extension, that software relies on for implementing a secure and a non-secure OS in their respective execution environments. The widespread use of ARM processors in IoT scenarios establishes ARM and TrustZone as the natural foundation on which IoT security is built.

2.2 GlobalPlatform TEE

A GlobalPlatform-certified TEE is a logical and effective approach for utilizing TrustZone hardware to split and control system resources in a secure environment [Globalplatform, 2022f].

An Execution Environment is a set of hardware and software components that provide the infrastructure required to allow the execution of programs. Devices, from smartphones to servers, provide a Rich Execution Environment (REE), a tremendously extendable and flexible operating environment. Common REEs are Windows, Android, and iOS. This gives the device flexibility and capability, but it also makes it susceptible to a variety of security risks. The Trusted Execution Environment (TEE) exists alongside the REE, to provide an area of higher security for the device to protect assets and execute trusted code. The REE is abundant with features, while the TEE is reserved for vital functions, often security-related. Applications executed by the TEE are Trusted Applications (TA).

GlobalPlatform is a technical standards organization that specializes in digital security. This section outlines a TEE's hardware and software architectures by the GlobalPlatform TEE Standard.

GPD TEE

A TEE compliant with the GlobalPlatform TEE Standard is a GPD TEE.
It must meet the following criteria:

- All code executing inside the TEE has been authenticated.
- The TEE resists known remote and software attacks, and a set of external hardware attacks.
- The TEE must provide separation from other environments in the device.
- The TEE must be compliant with the TEE Client API and the Internal Core API specifications[Globalplatform, 2022d][Globalplatform, 2022a].

Trusted Application

Trusted Applications are applications with distinct privileges that perform security-related functions. This does not mean that the ”Trusted” Application is necessarily safe, or bug-free. However, it is vital that they are implemented as such. TrustZone’s function is to separate and shield the security-related functions from the functionality-rich normal world. Attackers can use a vulnerable TA in the TEE to attack other TAs and the trusted OS itself, defeating the purpose of the TEE. Therefore, TAs should be as simple as possible and only perform security-sensitive functions. A TEE should attempt to limit the number of TAs, to reduce vulnerable surface area.

TEE Security Functionality

The TEE’s function is to securely host and run TAs, enforce their isolation from another and the REE, and protect the TEE’s assets [Globalplatform, 2022b]. The core TEE functionalities for the end user are:

- Isolation of the TEE services, the TEE resources, and the Trusted Applications from the REE.
- Protected communication interface between CAs in the REE and TAs in the TEE, including communication endpoints. This allows the REE to request usage of the features of the TEE, without sacrificing the TEE’s integrity.

- Trusted storage of TA and TEE data.
- Random Number Generation (RNG)
- Cryptographic API for end-user, the functionality may include: Generation and derivation of keys and key pairs, hashing, symmetric and asymmetric operations
- Instantiating the TAs and ensuring consistency of TA code.
- Monotonic TA instance time.

RNG, the cryptographic API, and monotonic TA instance time are functionalities called *System Services*. They are the Trusted OS's system calls and provide functionality for the TAs. Subsection 4.2.3 shows a consequence of TEEs having their own system calls.

The TEE is isolated from the untrusted REE, it securely hosts and executes TAs and provides necessary services and functionality for the TAs. GPD TEEs are TEEs that comply with GlobalPlatform security standards, including utilizing GPD-standardized TEE APIs for cross-world communication. Subsection Libteec and libutee describes an implementation of the GPD TEE APIs.

2.2.1 Hardware Architecture

The REE and the TEE utilize many resources such as processing core(s), ROM, RAM, cryptographic accelerators, etc [Globalplatform, 2022c]. Figure 2.3 shows a simplified example of a device's resources, and highlights the components that can host a TEE. This subsection describes how TEEs separate the system's hardware resources into two Execution Environments.

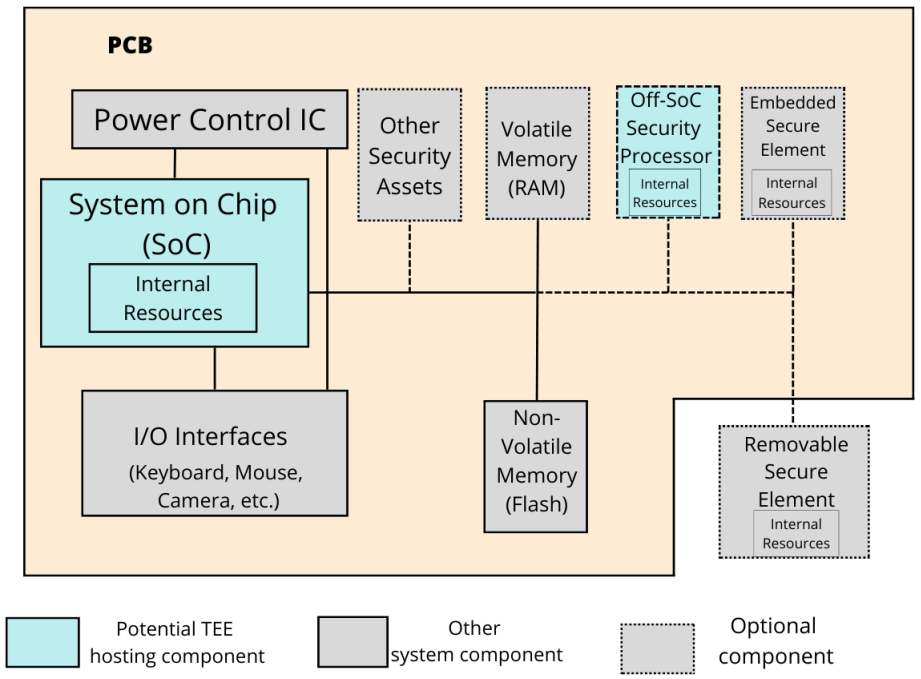

Figure 2.3: Typical chipset architecture. The figure shows an example of resources that can exist for a device. A Printed Circuit Board (PCB) that connects components such as SoC processing units, RAM, etc. Remade from [Globalplatform, 2022c]

Resources are either controlled by the REE or TEE, however, they are transferable. Control over parts of or a whole resource can shift between the environments. If the TEE controls a resource then that resource is isolated from all other execution environments, unless explicitly authorized by the TEE. The controlling TEE regards all of its resources, that it has not shared, as trusted resources. These trusted resources are isolated from all non-trusted resources and become a closed system, protected from all other execution environments. Other execution environments, like the REE, can allow its resources to be accessible by the TEE without any specific permissions. However, a TEE may **not** allow the opposite. Other execution environments can only access a TEE's resources with specific permissions.

REE and TEE Resource Separation

Figure 2.4. illustrates the separation of REE and TEE resources. The REE has access to untrusted resources, but none of the trusted resources. The access control can be implemented either through physical-, hardware- or cryptographic isolation. The REE can only access trusted resources through API entry points, like the TEE Client API. The separation does **not** prevent the REE from passing buffers to the TEE, and vice versa, in a secure manner[Globalplatform, 2022c].

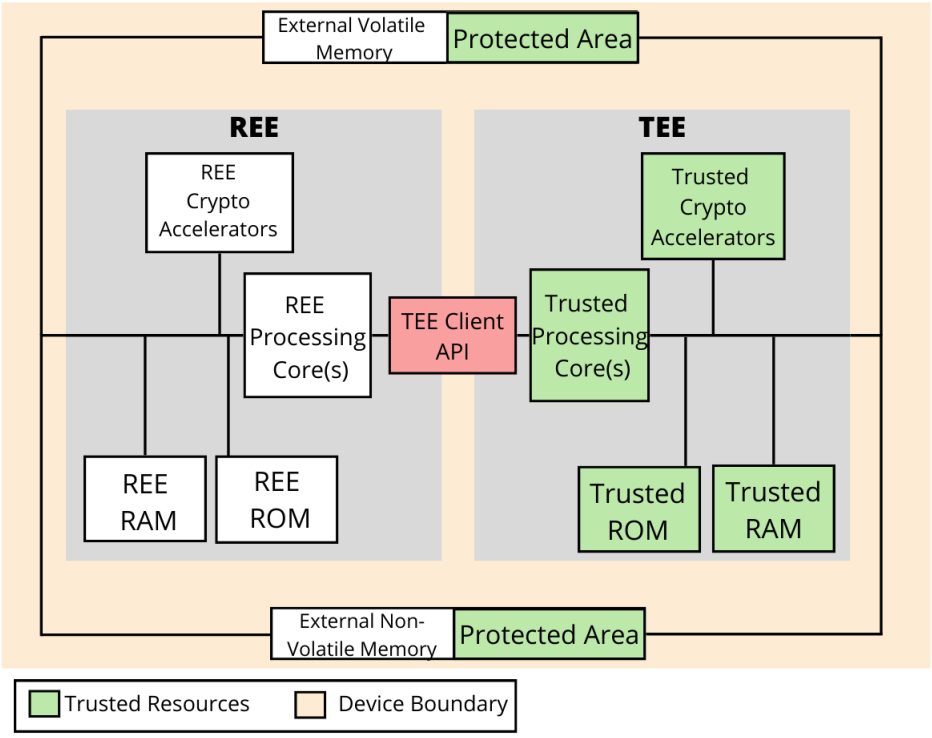

Figure 2.4: The REE and TEE resources are separated, but the TEE Client API facilitates communication. Buffers can be passed between REE and TEE. Remade from [Globalplatform, 2022c]

2.2.2 Software Architecture

As mentioned, the TEE provides APIs to enable communication from the REE to facilitate the usage of TA functionality. This subsection outlines the software architecture of the TEE and the interfaces defined by GlobalPlatform for cross-

environment communication.
The TEE is, as mentioned, reserved mainly for security-sensitive applications. Such applications usually consist of two separate applications, a *Client Application* (CA) and a TA. This application pair is represented by ”CA/TA”. The CA resides in the REE and performs non-sensitive functions, such as user interaction. The CA requests TA functionality with the TEE Client API. The TA receives any needed function parameters from the CA and performs the sensitive operations. However, the TA and TEE never trust the CA, because it resides in the less secure REE. The TA runs on a Trusted OS inside the TEE, which validates the parameters supplied by the CA. TrustZone's hardware-assisted isolation protects the integrity and secrets of TAs from the untrusted REE. Figure 2.5 shows a simplified version of the GlobalPlatform Software Architecture.

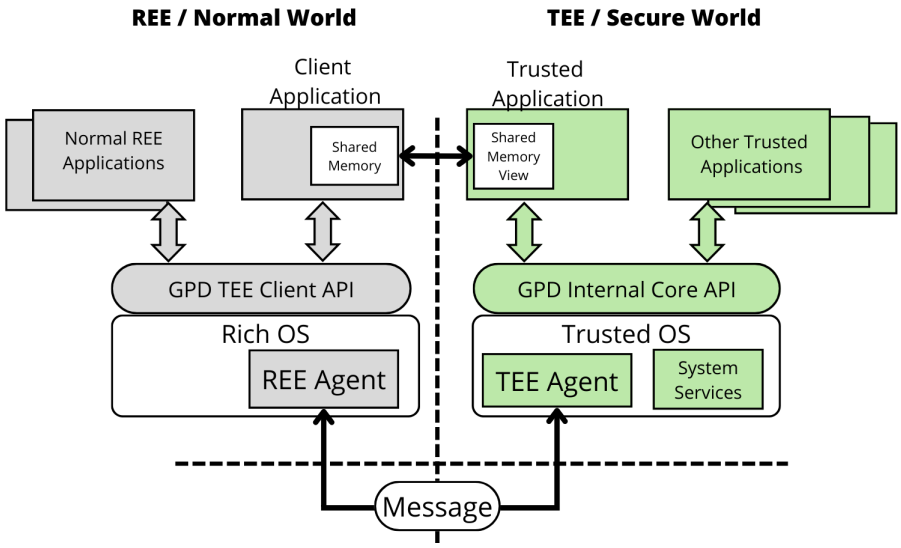

Figure 2.5: GlobalPlatform TEE Architecture. Communication is done between the agents. Memory is shared between the CA and TA, and the APIs facilitate communication for the applications. The Trusted OS includes System services, the TA accesses these through the Internal Core API. Remade from [Globalplatform, 2022c]

As the CA requests the trusted execution of a particular TA, it relies on the *REE Agent* to deliver the *Message* to the TEE. Initially, the CA calls the *GPD TEE Client API* to ask the REE Agent to pass the Message to the TEE. When the

TEE Agent receives the Message, the Trusted OS and the *GPD Internal Core Api* connect the agent with the correct TA and provide the message. The CA exchanges data by sharing memory with the TA, the TA can then access the shared memory and write to it.

The architecture allows the CA/TA to perform security-related functions in the TEE, with normal world inputs, and return the result to the CA.

2.2.3 OP-TEE

Open Portable TEE (OP-TEE) is a Trusted OS designed as a companion to a non-secure Linux kernel running on Arm's TrustZone and complies with the GlobalPlatform standard. The OP-TEE OS provides communication between a Linux-based REE and the TrustZone TEE based on the ARM Secure Monitor Call (SMC) instruction. OP-TEE supports the GPD TEE Client API and GPD TEE Internal Core API. The main goals of OP-TEE are to:

- Isolate the TEE from the non-secure OS.
- Remain small enough to reside on the on-chip memory of ARM-based systems, like IoT devices.
- Be easily portable to different architectures.

Most importantly, OP-TEE is a publicly available Trusted OS developed for TrustZone. OP-TEE is open-source and developed by Linaro, now owned by TrustedFirmware.org [Doc, 2022]. Other Trusted OSes for TrustZone, like Qualcomm's QSEE, are proprietary, so access is restricted from the public. To use OP-TEE, their GitHub repository must be installed and built. Subsection 4.1.1 explains the build process.

This subsection describes the architecture of Trusted Applications running on the OP-TEE OS, referred to as just a *TA* for the remainder of this section.

Libteec and Libutee

OP-TEE has developed two libraries that implement the GPD TEE Client and GPD TEE Internal APIs: *Libteec* to be used by Client Applications (CA) in the REE, and *Libutee* to be used by TAs in the TEE.

Libteec - TEE Client API (Used by CA)

OP-TEE provides the library Libteec, to establish communication between the CA and the Trusted OS, in accordance with the TEE Client API. TEE Client API abstractions establish communication [Globalplatform, 2022e], these abstractions are:

- A TEE Context, which is the connection between a CA and the TEE
- A Session, which is the connection between a CA and a specific TA
- A Command, which is the unit of communication between a CA and a TA within a Session. Commands are sent with a command id, representing a specific function to invoke.
- The Operation Parameters, which are function parameters that are sent with a command. They can only be integers or memory references

The Libteec library supplies the CA with these abstractions, opens a session with a TA, establishes shared memory, and invokes commands.

Libutee - TEE Internal Core API (Used by TA)

OP-TEE provides the static library Libutee, to provide services to the TAs, in accordance with the TEE Internal Core API. Libutee calls services implemented in OP-TEE Core (privileged level of the TEE) through system calls. Other services are statically implemented in the library. The main services provided are:

- Communication means with Client Applications (CAs) running in the REE (TA entry points and command parameters)
- Cryptographic API
- Time API
- Arithmetical API
- Debugging features (printing to Secure World terminal)

The Libutee library supplies the TA with these functionalities and is necessary when opening a session with the CA and receiving commands.

These libraries are both written in the programming language C. C is the common denominator for almost all application frameworks and operating systems that host CAs and TAs [Globalplatform, 2022d].

Trusted Applications in OP-TEE

Trusted Applications in OP-TEE are Executable and Linkable Format (ELF) files, their filename is the TAs Universally Unique Identifier (UUID). A UUID is, as the name suggests, an identifier that should be unique. Every TA requires a UUID, so it can be specifically requested by the TEE Client API. TAs have the suffix ".ta", and are signed with a digital signature and optionally encrypted; therefore, they can reside in the untrusted REE filesystem. OP-TEE Core validates and loads them when executed [Doc, 2022].

OP-TEE TAs must have five mandatory entry points. Entry points are functions in the TA, that are called by the OP-TEE OS during CA/TA communication.

In chronological order for a typical CA/TA:

- Ta_CreateEntryPoint(), is called when the TA loads, returns success
- Ta_OpenSessionEntryPoint(), is called when a session is established, returns success
- Ta_InvokeCommandEntryPoint(), this is where TA logic is done, called whenever the CA invokes a command. The entry point can contain many different commands,
- Ta_closeSessionEntryPoint(), is called when the session is closed
- Ta_DestroyEntryPoint(), is called when the TA is finished

Development with cross-world communication is explained thoroughly in Subsection 4.2.2.

The Trusted Execution Environments provide an area of higher security for the device to protect assets and execute trusted code. Hardware isolation like ARM TrustZone enables TEE systems. The TEE's main functions are to securely host and run TAs, enforce their mutual isolation and isolation from the REE, protect the TEE's assets and provide system services.

2.3 Rust

Rust is a programming language designed for developing reliable and efficient systems. Rust's static type system is safe and expressive and provides strong guarantees about isolation, concurrency, and memory safety. The language's type system and runtime guarantee the absence of data races, buffer overflows, stack overflows, and accesses to uninitialized or deallocated memory[Matsakis and Klock, 2014]. Thus, the Rust language and compiler help developers create safer applications.

Rust offers the following mechanisms to achieve reliability in memory safety and thread safety:

- Claiming the ownership of each data object
- Automatically checking the read/write permissions (mutability) of each object
- Disabling dangerous raw pointer operations (such as dereferencing)

During compilation, if the code violates any of Rust's security criteria, the compiler will raise an error and provide helpful error messages.

In addition to code security, Rust provides an infrastructure of thousands of public *crates* (similar to libraries).

Ownership

One of the main concepts of Rust is that values are generally not copied, but *moved*. When a variable gets passed between functions or threads, the associated ownership is transferred or moved with it.
If *Function A* passes a variable to *Function B*, then Function A no longer has access to the variable, because ownership of the variable has moved to Function B.

Rust-unsafe

Rust's design achieves and follows strict security criteria by default. However, to guarantee that any program can be written in Rust, it also provides the keyword

unsafe for developers to inject memory-unsafe code segments. Rust provides this unsafe option for two primary reasons: to allow developers to create some special functions that cannot pass the compiler's default inspection and to allow the code to directly interact with system/hardware components. By wrapping code inside the *unsafe* block, it allows the code to bypass the built-in compiler checks and conduct vulnerable operations, such as writing on an immutable reference [Wan et al., 2020].
Rust-unsafe allows the writing of memory-vulnerable code in Rust. The unsafe blocks also serve to mark unsafe areas of code, which makes the code more easily peer-reviewed for security.

2.3.1 Rust OP-TEE SDK

OP-TEE expects TAs to be written in C. C is not a memory-safe language; thus there have been reported numerous memory corruptions on such TAs [lagini-maineb, 2016]. Researchers have found that Rust, a memory-safe language, can be used instead of C.
Their research suggests that writing an OP-TEE TA in Rust greatly increases the application's memory safety while introducing only minor performance overhead [Wan et al., 2020]. They have published their Rust OP-TEE SDK (Rust OP-TEE) as an open-source development toolkit [ApacheTeacleave, 2020]. Rust OP-TEE does the heavy lifting of making Rust work in the OP-TEE environment, it also provides example TAs to help the development process. Section 4.2 covers development using Rust OP-TEE.

Rust OP-TEE crates

The Rust OP-TEE SDK, to port Rust into OP-TEE, implements Rust versions of OP-TEE's API-implementing libraries: Libteec and Libutee are replaced by the crates *optee_teec* and *optee_utee*. They serve the same functions as their C counterparts. Subsection 4.2.2 covers how these crates are used in development.

Chapter 3

Related Work

This chapter describes the ChainBox framework implemented by [Bleeke et al., 2022] on Intel SGX. This thesis investigates porting ChainBox to ARM TrustZone; therefore, this section presents the architecture of ChainBox and what a complete implementation requires.

3.1 ChainBox

Smart contracts are transaction protocols that execute the contractual terms of an agreement. Contractual clauses in smart contracts are enforced automatically when a certain condition is satisfied [Zheng et al., 2020].

ChainBox is [Bleeke et al., 2022]'s proposed solution for establishing trust between IoT devices. Its basis is a Trusted Execution Environment (TEE), which generates a blockchain on a single-edge device and enables low-cost on-site deployment. Instead of replication and distribution, ChainBox relies on TEEs to establish trust. Some form of attestation allows participating parties to trust that the code executes correctly inside a TEE. That is, the blockchain is created as an append-only log, and transactions submitted to the blockchain are properly authenticated. Attestation, like Intel SGX's remote attestation feature, should verify three things: the application's identity, that it has not been tampered with, and that it is running securely within the Intel SGX Enclave (or Trusted Execution Environment) [Lab, 2019].

The design includes a smart contract runtime secured by a dedicated enclave. It allows for smart contracts to execute inside the TEE, this execution can be at-

tested. Therefore, tenants can know that the smart contract is correctly executed. The design ensures that tenants can be held accountable for the data shared with others, and the blockchain ensures that shared data is consistent for all participants.

ChainBox creates an immutable append-only logbook, and the deployment and trusted execution of smart contracts while enabling data exchange through recorded blockchain transactions. It does this without relying on replication and distribution for establishing trust, thus, reducing costs.

3.1.1 System Architecture

The ChainBox consists of three key components as shown in Figure 3.1. The green components operate within the TEE and are trusted, while the gray ones are untrusted. This subsection presents the ChainBox design and provides context to the Ordering Service Enclave (OSE).

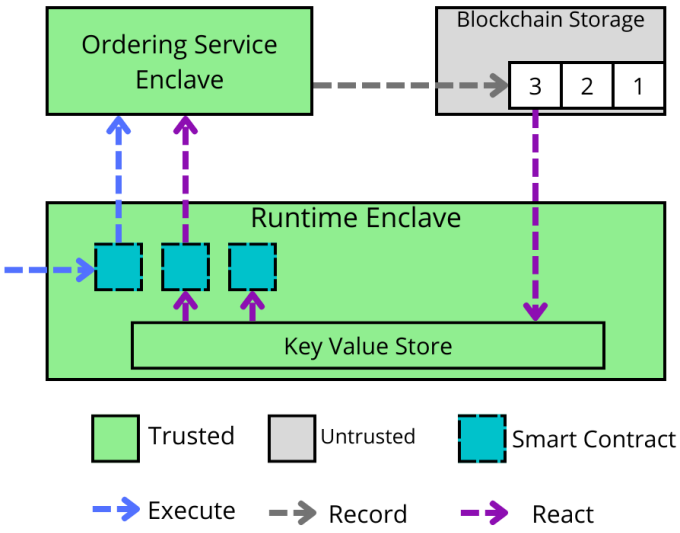

Figure 3.1: The Runtime Enclave supplies the OSE with smart contract transactions to create the block to be stored in the blockchain.

The Ordering Service Enclave is the core component. It deterministically handles smart contract transactions and creates the blocks that are stored in the blockchain. Since the ordering service is running inside a TEE and its code is publicly known, vendors can trust that it runs correctly.

The Blockchain Storage appends the newly created blocks and writes them back into the storage. Each block contains a signature from the ordering service for verification and ensures that it's only created once. The blocks are distributed to all vendors and are regarded as public information. The ordering service's signature and the hash chain protect the blockchain's integrity.
The Runtime Enclave is responsible for the loading, instantiating, and execution of smart contracts. It supplies the Ordering Service Enclave with smart contracts to be logged in the blockchain.
The design allows for attestation at three places: vendors can attest that the Runtime Enclave executes the contracts correctly; the Runtime Enclave can attest that the Ordering Service Enclave runs correctly; the Ordering Service Enclave attests the integrity of the smart contracts before accepting their transactions.

ChainBox implemented their Ordering Service Enclave in Intel SGX. In [Bleeke et al., 2022]'s Future Work they wished to investigate porting the system to ARM TrustZone.

The Blockchain

A blockchain is an append-only data structure, it's a distributed ledger used to record transactions from multiple vendors. These vendors do not trust each other, but blockchain can solve the problem of establishing trust in distributed systems. More specifically, the problem of creating a distributed storage of timestamped logs where no party can tamper with the content of the data without detection. Inside a blockchain, blocks are connected by including a cryptographic hash of the previous block, thus allowing modifications of blocks to be detected, as modifying one block changes all later blocks.
Figure 3.2 illustrates the structure of the blocks that the Ordering Service Enclave (OSE) issues, and how they together make up a blockchain. The hash of each block is cryptographically made from the previous block's hash, the time of creation, and the transaction data to store.

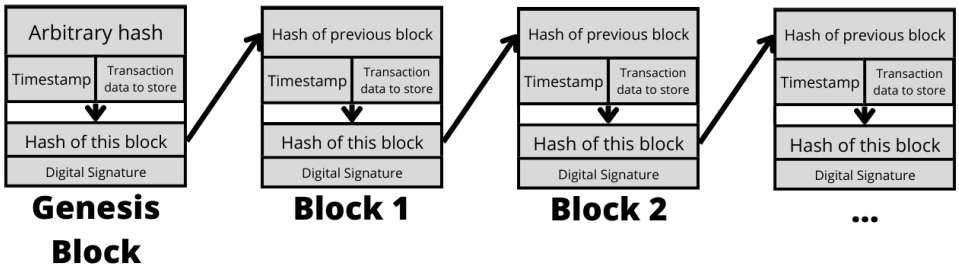

Figure 3.2: A Blockchain consists of blocks connected through the previous block's hash. The block's hash is signed with a digital signature. The signature proves that the Ordering Service Enclave built the block. The genesis block is the first block in a blockchain and is arbitrarily created.

The block's hash is cryptographically signed with the OSE's private key, which creates a digital signature. Vendors can verify the digital signature by using the OSE's private key, verifying that the block was issued by the OSE.

3.2 Other Related Work

Currently, there are other works on confidential smart contract execution within a TEE. Almost all of these works, such as ChainBox, use Intel SGX as the underlying TEE technology, instead of TrustZone. Notably, [Müller et al., 2020] is an exception, they have because of ARM's prevalence in IoT scenarios implemented an extension of Hyperledger Fabric to leverage ARM TrustZone for smart contract execution. Their work indicates that the Runtime Enclave component of ChainBox, which handles smart contract execution, is portable to ARM TrustZone.

While there are fewer works using TrustZone than Intel SGx, there is almost no work or cases of implementing Trusted Applications in Rust using Rust OP-TEE. To the best of my knowledge, the only work of such kind is [Jung et al., 2022]'s Trusted Monitor, which partially utilizes Rust OP-TEE in an intrusion detection system that runs inside the ARM TrustZone TEE.

Chapter 4

Implementation

4.1 Introduction

I have implemented the Ordering Service Enclave prototype in OP-TEE because it utilizes TrustZone hardware, which is abundant in IoT devices, and is open-source. To increase the security of the TA, I have written it in the language Rust and used the Rust OP-TEE SDK to port Rust into the C-based OP-TEE. ARM TrustZone is a security feature implemented in hardware. TrustZone splits physical computer resources between a normal world (REE) and a secure world (TEE). ARM processors are necessary to utilize TrustZone. ARM CPUs are ubiquitous in IoT scenarios, but not in personal computers. QEMU, an open-source emulator, can virtualize a machine that supports TrustZone. I used QEMU as a virtual machine (VM) during development but deployed the OSE to a Raspberry Pi 3 Model B for performance benchmarking.

The following chapter presents the implementation of the Ordering Service Enclave prototype. The first challenge of implementation is building the OP-TEE environment to facilitate the use of TEE and TrustZone technology. The second challenge is to understand how to develop CA/TA pairs and perform cross-world communication using Rust OP-TEE. The last challenge is developing the OSE to create a blockchain in TrustZone.

4.1.1 Setting up Rust OP-TEE building environment with QEMU

The first step in building OP-TEE is to make the switch to Linux. OP-TEE communicates with a Linux REE and embracing Linux also makes the building process

easier. The OP-TEE maintainers primarily use Ubuntu-based distributions, but other Linux distributions should work.

The Rust OP-TEE SDK has a setup guide [ApacheTeacleave, 2020]. Before following that documentation, I recommend installing the prerequisites [Documentation, 2022a].
To test the CA/TAs, establish a shared folder between the host (the computer) and the guest (QEMU). This shared folder should contain the executable of the Client Application and its corresponding TA file (named after its UUID and suffixed with ".ta"). Sharing the folder allows us to quickly deploy the code to QEMU, without having to build QEMU over again, which takes considerable time. After sharing the folder with QEMU, copy the TAs into "/lib/optee_armtz" (the standard location where the OP-TEE OS searches for Trusted Applications), to help the Trusted OS find the TAs.
This should allow for modifying Rust OP-TEE's provided examples, compiling the changes, and testing the applications on QEMU.

4.1.2 Deploying on a Raspberry Pi 3 Model B

To deploy on a Raspberry Pi, the Pi's SD card must be flashed, and the OP-TEE image is built on it. OP-TEE build instructions for Raspberry Pi 3: [OPTEE, 2022]. When the OP-TEE image has been built on the SD card, I copy the CA executable and the TA onto the Pi's SD card. I then worked directly on the Pi using HDMI, thus, I did not need UART cables to connect to the Pi. After booting, I copied the TA files to the Pi's "/lib/optee_armtz". The Raspberry Pi can then execute the CAs and request the trusted execution of TAs.

4.2 Development in the Rust OP-TEE SDK

The SDK provides two resources for guiding development: Development documentation and examples. The SDK's development documentation is not comprehensive enough[Teacleave, 2022c]. As a result, the examples provided serve as the primary source of guidance. The examples do little to explain the logic of communicating between the worlds. They neither explain the necessary steps to create your client and trusted application pair (CA/TA), nor the limitations third-party crates have in the TA. Therefore, I make an effort to clarify these topics in

this section. Hopefully, this can provide insight and expedite the development process for other RUST OP-TEE developers.

4.2.1 Building a new CA/TA

This subsection describes how to use the SDK to build a CA/TA pair in Rust OP-TEE. As previously mentioned, Trusted Applications running in the OP-TEE OS follow a certain structure and require mandatory files, headers, TA entry points, etc. [Documentation, 2022b]. This can seem daunting, but luckily Rust OP-TEE provides examples and *Makefiles* that make the building process rather simple. Makefiles are computer files that perform a set of operations, in Rust OP-TEE they are provided frequently and support the developer, e.g., with compiling the CA/TA. An SDK CA/TA has three main folders:

- Host, which is the Client Application.
- The TA folder, which is the Trusted Application.
- Proto

Proto among other things, defines the commands that CA can give TA. These commands serve as the communication means between the worlds.

Copy an example

The easiest way to create a new CA/TA is to copy one of the examples given by the SDK and modify it. To create a new CA/TA:

- Duplicate an example
- Decide commands the CA sends to TA and modify proto/src/lib.rs as such
- Modify the folder name in host/Makefile and host/Cargo.toml
- Implement CA logic in host/src/main.rs
- Generate a **new** random Universal Unique Identifier (UUID) and edit uuid.txt
- Implement TA logic in ta/src/main.rs
- Compile the CA/TA (use the SDK Makefiles, see: [ApacheTeacleave, 2020])

Every TA requires a UUID, so the Trusted OS can uniquely identify it. Therefore, the example UUID **cannot** be reused. To quickly generate a random UUID one can use one of many online UUID generators or Linux's *uuidgen* command. The UUID must be 36 characters(32 hex digits, and 4 "-" symbols).
This concludes building a new CA/TA pair in Rust OP-TEE.

4.2.2 Working between the worlds

Opening a session

Before invoking a command, a session between the CA and TA must be opened. The TA resides inside the TEE, which is separated from the REE where our CA is. The SDK provides two necessary crates for establishing cross-world communication, optee_teec and optee_utee [Apache, 2022a]. Optee_teec implements the TEE Client API in the REE, while optee_utee implements the TEE Internal Core API in the TEE. Optee_teec creates an abstraction of the logical connection to the TEE, called Context. The Context opens a session with the specific TA using the desired TA's UUID. Listing 4.1 shows how the CA opens a session with the TA, which allows the CA to invoke commands.

```
let mut ctx = Context::new()?;
let uuid = Uuid::parse_str(UUID).unwrap();
let mut session = ctx.open_session(uuid)?;
```

Listing 4.1: The CA initializes the TEE Context and supplies it with the TAs UUID to open a session

Listing 4.2 shows the open_session entry point in the TA, which is called when the session has been successfully opened. This entry point can initialize a Rust structure, which lasts for the entire session, it maintains the state of the application between commands. The optional parameter *_sess_cxt* (session context) is passed in Listing 4.2, with the type *State*. This State structure must implement Rust's default trait, which sets the initialization values for that structure.

```
#[ta_open_session]
fn open_session(_params:&mut Parameters,_sess_ctx:&mut State){
    trace_println!("[+] TA open session");
    Ok(())
```

```
5 }
```

Listing 4.2: The open session entry point in the TA. Uses the optional parameter session context to initialize a State structure

Optee_utee establishes the entry points for the TA using Rust attributes, which are recognized by "#[]" above the function. In Listing 4.2, line 1, the #[ta_open_session] attribute allocates memory on the heap, with the session context referencing the State structure[Teacleave, 2022b]. The session context is also supplied as a parameter to the invoke command entry point, which allows the State to be updated and used for each command in the session. The memory allocated for the session context is secure world memory, thus making it inaccessible to the normal world. Maintaining a state is crucial for the implementation of the Ordering Service Enclave, as the previous block is used in the creation of the next block.

Invoking commands

A session facilitates the invocation of many commands, a command is essentially a function call. The session's invoke_command function takes two arguments, the command id (defined in Proto) and an Operation structure. The command id represents which command to invoke. Operation contains four TA parameters, these are arguments sent to the TA, and they can only be in the form of a ParamTmpRef or a ParamValue. ParamTmpRef is a TA parameter that holds a temporary memory reference. This memory must be in the form of a byte array and is shared memory between the CA and TA. Therefore, the TA can write to this shared memory and CA will be able to read the changes. ParamTmpValue is similar, but instead of holding a temporary memory reference, it carries small raw data in the form of integers. ParamTmpValue can hold two integers: a and b, the TA can change these and the CA can read the changes from the Operation structure.

This means that the data sent between the worlds can *only* be in the form of byte arrays or integers.
Listing 4.3 shows an example of a CA invoking a command.

```
1 let mut buffer = &mut [0u8; 256];
2 let p0 = ParamTmpRef::new_output(buffer);
3 let p1 = ParamValue::new(32, 0, ParamType::ValueInout);
4 let mut operation = Operation::new(0, p0 , p1, ParamNone, ParamNone);
```

```
session.invoke_command(Command::Increment as u32, &mut operation)?;
let new_integer = operation.parameters().1.a();
println!("new integer: {}, new_integer)"; // prints 33
println!("updated buffer: {:?}, buffer)"; // prints updated buffer
```

Listing 4.3: CA invokes a command by supplying a memory buffer and a ParamValue containing the integer 32. It then reads

In Listing 4.3, ParamTmpRef creates a memory reference, which the TA can write to (output), there is also the option to create an input-only memory reference, which can not be written to. Likewise, Paramvalue carries the integer 32, and the ParamType::ValueInout tags the parameter as both an input and output. The Operation always requires 4 TA parameters, so in Listing 4.3 the Operation includes two empty ParamNones. When the command has been completed, the variable new_integer is set to the updated *a* of ParamValue, which in this case is 33. The CA prints the buffer after the command, therefore, anything written to this memory in the TA is displayed.

Commands inside the TA

The TA's invoke_command function is the entry point for all commands, and the command id identifies the specific action to perform. Listing 4.4 is an example invoke_command function, it takes in the command id, the TA Parameters. Optionally, the session context can be requested by including it as a function parameter. The function receives the session context from its optee_utee-defined attribute. If the TA did not initialize the session context when opening the session but is still requested here, the function will return a security error.

```
#[ta_invoke_command]
fn invoke_command(sess_ctx:&mut State,cmd_id:u32,params:&mut Parameters)->Result<()>{
    match Command::from(cmd_id) {
        Command::Increment => {
            return increment(sess_ctx, params);
        }
        Command::Decrement => {
            return decrement(ses_ctx, params);
        }
        _ => {
            return Err(Error::new(ErrorKind::BadParameters));
        }
    }
```

```
}
```

Listing 4.4: Invoke command (TA side)

The invoke_command entry point matches the command id supplied from CA with its corresponding function. Inside each function the TA must unpack the parameters, to access the shared memory and values.

```
fn increment(state: &mut State, params: &mut Parameters) -> Result<()> {
    let mut p0 = unsafe { params.0.as_memref().unwrap() }; // ParamTmpRef
    let mut p1 = unsafe { params.1.as_value().unwrap() }; // ParamValue
    let mut buffer = p0.buffer(); // byte array
    let integer_value = p1.a(); // 32
    p1.set_a(integer_value + 1); // sets the parameter to 33
    buffer.write(&[1u8;256]); // writes an array of 1s to the shared memory.
    Ok(())
}
```

Listing 4.5: This arbitrary Increment command first unpacks the parameters. Then accesses and writes changes to the buffer and the integer

Listing 4.5 shows how a command unpacks the Parameters and accesses the data. The arbitrary Increment command increments the integer and updates that Parameter's *a* value. It also writes to the shared memory buffer.

The TA must unpack the parameters inside a rust unsafe block because the Parameter functions themselves contain unsafe blocks [Apache, 2022b]. The Rust compiler can not know what is being sent between the two programs, only where to access it. Therefore, the optee utee crate creates a raw pointer to that memory. The TA must access the data contained in the Parameter, for reading, writing, and accessing functions. To perform these operations, the TA must dereference the raw pointer. A raw pointer is not guaranteed to point to valid memory and is not even guaranteed to be non-NULL [edu, 2022]. Among other things, this makes dereferencing a raw pointer a memory unsafe action. Therefore, the TA must use an unsafe block, making it the developer's responsibility to ensure that it's not pointing to somewhere that would be incorrect.

Development in the TA is then equivalent to the normal world, except for system calls.

4.2.3 Third-party crates

Third-party crates are Rust crates that have no immediate connection to the TrustZone SDK. The SDK publication states that trusted third-party crates can be imported into the TA development [Wan et al., 2020]. Although this is true, using third-party crates in our TA can be troublesome.

The issue comes from the fact that the TEE operates on the OP-TEE OS, and has different system calls than for example Linux. System calls are requests to the operating system's kernel. If an included crate uses an incompatible system call then the TA will not compile. If it has none, it can be imported like you would in a normal Rust application. The crate *Serde* is an example of such an "out of the box" compatible crate, which I used in the development of the OSE. However, it is quite likely that development includes coming across incompatible crates. Luckily, some crates are portable to OP-TEE. By replacing the incompatible system calls with alternatives provided in the SDK the crate becomes compatible. These alternatives are found in the optee_utee crate and are the system services or TEE functionalities of OP-TEE. Creating a random number or retrieving the system time are examples of common system calls [man, 2022]. If there is a crate that, for example, uses Linux's *getrandom* system call, it's OP-TEE incompatible. However, by replacing getrandom with optee_utee::Random, one ports the crate to OP-TEE. If a crate uses a system call without a Rust OP-TEE equivalent, the system call is irreplaceable and, thus, the crate **cannot** be ported.

Because the SDK is open source, others have ported some originally incompatible crates to OP-TEE. In these cases, an SDK example showcasing the ported crate will be added [Teacleave and HakonToemte, 2022]. Although as of writing, there are only two such examples, for the crate Ring and Rustls.
To conclude, Rust OP-TEE TAs does as claimed support third-party crates. However, development may include porting these crates to OP-TEE, and some crates are unsupported(i.e. not portable).

I have in this chapter attempted to demystify the Rust OP-TEE SDK. In particular, building your own CA/TA and how to perform inter-world communication.

4.3 Implementing the Ordering Service Enclave

In this section, I will present a prototype of ChainBox's Ordering Service Enclave (OSE). The prototype shows that a TrustZone-based TEE can create a blockchain, and issue it to the normal world. This section will be mostly illustrations as I have already described most of the SDK-related code in Subsection 4.2. The implementation of hashing, digital signatures, and block creation in code is not the focus of this thesis.

4.3.1 Design

My implementation of the Ordering Service Enclave is not within the ChainBox system (described in Section 3.1). Therefore, I have made some simplifications: I have not implemented any Blockchain storage, the data to log is a string instead of a smart contract, and the orderer exists in the untrusted world; therefore, the orderer is not attested. I justify these simplifications with the fact that the main purpose of the OSE is to create and issue signed blocks from the TEE, and this thesis investigates the feasibility of doing this in OP-TEE. Therefore, the storage of the blocks, the data content of the blocks, or the source of the data is not of importance in this thesis. The OSE does *not* execute the smart contracts, only log the transactions as blocks, therefore a string serves essentially the same purpose.

The purpose of the Ordering Service Enclave is to be a *block issuer*, as shown in Figure 4.1. The CA serves as the *block orderer* and sends data to the issuer for logging, the TA returns the data contained in a block. Subsection 3.1.1 describes the blockchain created by the OSE.

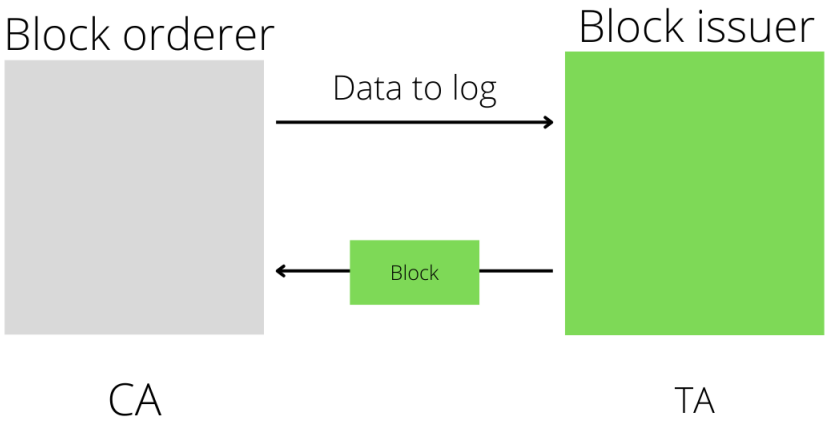

Figure 4.1: The Ordering Service Enclave prototype in its simplest form. The OSE receives data and returns that data contained in a block to the orderer.

The Ordering Service Enclave's (OSE) functions are to build and issue blocks from the TEE. The blocks work to log data and because they come from the TEE, they are implicitly trusted. Later, any vendors can ensure that the OSE created the block because it is signed with a digital signature. The defining trait of a blockchain is that every block depends on all previous blocks. This means that one would only need to verify the latest block's signature to know that all blocks in the blockchain come from the OSE.

4.3.2 Implementation

The OSE is a TA with 209 lines of code and provides two commands. New_block and get_public_key. New_block is the heart of the application, and creates a block and writes it to the shared memory provided by the normal world. The REE can then access and store the block by reading that memory. The block can then be used and stored in the normal world by reading that memory. Get_public_key is a command that gets the OSE's public key. The OSE uses an ED25519 key pair. The private key signs the digital signature and the public key verifies it.
The implementation requires maintaining the state between blocks, therefore, a state structure is stored for each session. The state consists of two objects.

- The previous block
- An ED25519 key pair

These are stored in the state because they need to linger from one creation to the next.

New block

The new block command takes three parameters from the normal world.

- Memory buffer to write the Block to
- Data (string) to log
- An output Paramvalue that represents the array length of the serialized block. This makes deserializing easier for the CA.

Figure 4.2 shows the OSE's block generation process. The data is hashed with the previous block's hash and a timestamp, using the SHA256 algorithm, then an ED25519 digital signature is signed on this hash. The REE's system time is used as a timestamp, using optee_utee::Time in the TA. When the block has been issued, the state's previous block is updated.

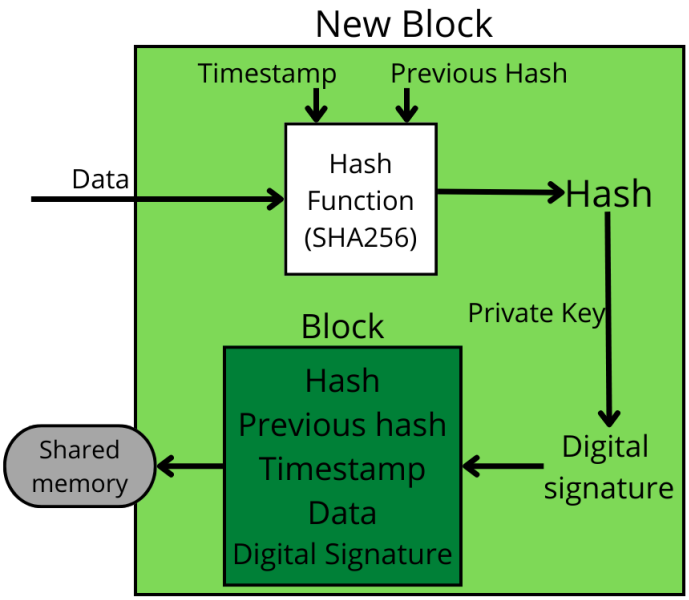

Figure 4.2: The OSE generates the new block's hash from the data, the previous hash, and the timestamp, then it creates the block and writes it to the shared memory.

```
pub struct Block {
    pub id: u32,
    pub hash: String,
    pub previous_hash: String,
    pub timestamp: String,
    pub data: String,
    pub signature: String,
}
```

Listing 4.6: Block structure

Listing 4.6 shows the Block structure in the code. Because communication between the worlds is done through the Parameters, which are only in the form of integers or byte arrays, the Block cannot be returned as a Block. To write the Block structure to the buffer, the Block structure must be serialized, and to be used in the normal world it must be deserialized. I use the third-party crate Serde for serializing. Serde serializes the block to a JSON string, this JSON string is trans-

formed into bytes and written to the buffer.

There are two third-party crates used in the OSE. Serde for serializing, and the crate Ring for ED25519 key generation and signing. I have not implemented any digital signature verification in the OSE, because in practice this will be done in the normal world by participating parties.

Get public key

The get public key command takes a memory buffer and writes the OSE's public key to that shared memory. The public key is necessary for verification purposes.

Verification

Anyone can verify that the block has been made by the OSE by verifying it with its public key. Verification is done by hashing the block's data, previous hash, and timestamp using the same algorithm as the OSE (SHA256), this must return a hash equal to the hash of the block. Then verify the hash and signature using the public key. If this verification is a success, then all blocks before this block in the chain are also verified.

4.3.3 Analysis

This subsection discusses the implementation of the OSE, not in respect of performance, but in respect of potential errors, security and future work to be done.

Errors

If the Block created by the OSE is larger than the shared memory buffer the application will return an Error::Errorkind::ShortBuffer. The TA also has a RAM size reserved for the TA heap; it's called TA_DATA_SIZE and is defined in the TA configurations at the bottom of each TA. If TA_DATA_SIZE is too small the application will freeze, without giving any error. This happens when available heap memory is not enough, e.g, the session context (state structure) grows too big or the Operation Parameter buffers are too big. TA_DATA_SIZE can be configured by each TA but is limited by several factors, such as the total memory available for the TAs [Teacleave, 2022a].

Security

The Ordering Service Enclave TA includes four unsafe blocks. All of them are from unpacking the parameters, and although unsafe, they are necessary to access the shared memory and values. There can also be unsafe code in the crates that are used.

Future Work

In terms of security, future work should be done to evaluate the security of the unpacking of Operation Parameters.

Future work is to place the OSE within the ChainBox system and expand the prototype into a complete and production-ready OSE. This includes implementing Runtime Enclave and Blockchain Storage compatibility, a constant key pair, and doing attestation and logging of the Runtime Enclave's smart contracts. As mentioned, TrustZone does not have built-in support for remote attestation, and neither has OP-TEE implemented remote attestation (though they have tried) [Teacleave and elias vd, 2022]. Therefore, a remote attestation scheme must be implemented through a TA, luckily, there is literature on such schemes for TrustZone.
As a component of their upcoming Armv9 CPU design, Arm unveiled Confidential Compute Architecture (CCA), an architecture that will offer remote attestation procedures [Mé nétrey et al., 2022]. So TrustZone built-in support for attestation is coming.

My implementation is a prototype of an Ordering Service Enclave to be used in the Chainbox framework. The prototype establishes that a more complete implementation can be made in Rust OP-TEE, which supports ARM TrustZone.

Chapter 5

Experimental Evaluation

This chapter presents the performance evaluation of the Ordering Service Enclave on a Raspberry Pi 3 Model B. The OSE is measured by the Client Application. The OSE implementation is expected to introduce significant overhead compared to a strictly normal world implementation. Therefore, these are compared, such that the TEE's sluggishness and the cost of increased security can be evaluated. Finally, optimizations are hypothesized and implemented.

5.1 Experimental Setup

A Raspberry Pi 3 Model B version 1.2 has 1GB RAM and a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU. I am doing the evaluations on the Raspberry Pi, as opposed to QEMU, because the Raspberry Pi is more interesting in the context of real implementations and IoT scenarios. It is also easier to analyze the data, as there are fewer variables with the Pi than on a VM with among other things, turbo-boosting CPUs. The function time measurements were performed in the CA using the rust std::time module. The TA performed measurements using the optee_utee::Time module.

5.2 Experimental Results

First, I evaluate the execution time of the two available commands and an empty command, on the Raspberry Pi. The empty command has no logic and returns immediately. It serves as a reference for how much of the execution time is context-

switching (world-switching) and how much is logic inside the TA. These commands are invoked after the TA is loaded and a session is opened, thus, these operations are not part of the latency.

To test the variance in latency, I measured the latency for each command and then used the Rust module statrs::Statistics to calculate the average-, minimum-, and maximum latency, and the standard deviation (σ). 1000 iterations were measured. Table 5.1 shows the result.

Command	average (ms)	min (ms)	max (ms)	σ	throughput
Empty command	2.38	2.35	2.52	0.02	426/s
GetPublicKey	2.38	2.35	3.718	0.02	426/s
NewBlock	4.20	4.11	4.952	0.05	238/s

Table 5.1: Performance of the different commands.

Expectedly, Table 5.1 shows that the standard deviation increases with the complexity of the command. Still, the standard deviation for the New Block command is very low at 0.05. The maximum time for the New Block command is 15 standard deviations away from the mean. Such a result must be an outlier, implying that the vast majority are much closer to the average of 4.20.

I have calculated the throughput in Table 5.1 from the average latency. It shows that even with no logic in the TA, the throughput is only 426 commands per second. When including the logic of creating blocks, the throughput decreases to 238 blocks issued per second.

The results show that even an empty command takes about two and a half milliseconds. This time is time used for context-switching and I will call this for *invoke time*. [Amacher and Schiavoni, 2019] found their invoke time to be 1.31 milliseconds for their OP-TEE TA, developed in C (”..time spent to execute an empty function inside the TA once it is loaded (1.31 ms), to give a baseline of comparison”). They deployed on an equivalent Raspberry Pi, thus, the cause of the discrepancy between their and my invoke time is uncertain. Further research should establish if Rust OP-TEE is the cause of this increased invoke time.
The latency of a simple command like GetPublicKey is almost entirely decided by invoke time. NewBlock has far more logic and thus, a considerable TA logic latency.

I assume that the latency is affected by the invoke time and the time it takes

to perform TA logic. Therefore, latency is given by Equation 5.1.

$$Latency = InvokeTime + TALatency \tag{5.1}$$

Invoke time was found to be about 2.38; therefore, while optimizations on the New Block logic could reduce latency, it will still be *at least* 2.38 ms. Invoke time currently stands for more than half of the latency, thus, optimization efforts should be to lower the total invoke time, by invoking the command fewer times.

I have also evaluated the latency of the NewBlock command with varying block sizes, Table 5.2 shows the result.

Block size (bytes)	latency (ms)
512	4.20
8192	4.38
131072	5.44
262144	7.92

Table 5.2: Generating big blocks takes more time.

The result shows that generating big blocks takes more time. Although the size of the hash is constant, hash computation takes longer for larger data inputs. Only the data to log increases the block's size, and that relationship is one-to-one. The buffer size (the size of the shared memory) ha s to be big enough to contain the block, thus, it too increases when logging big data entries. This buffer size affects the invoke time, as shown in Table 5.3.

Buffer size (bytes)	Invoke time (ms)
512	2.38
8192	2.41
131072	3.18
262144	3.90

Table 5.3: This table shows that invoke time increases with the size of the shared memory

Strictly Normal World Implementation

I implemented a strictly normal world Ordering Service Enclave; it builds an equivalent block in a single untrusted application. The throughputs of the trusted and untrusted applications are compared in Table 5.3. It shows that the untrusted application is almost 14 times faster. A cost of trusted execution is performance.

Ordering Service Enclave	block throughput
Trusted	238/s
Untrusted	3220/s

Table 5.4: A strictly normal world OSE is significantly faster than the secure world implementation

5.2.1 Optimizations

The optimization that follows aims to increase the number of blocks the OSE can issue per second: the throughput. To reduce total invoke time, the CA can pass multiple data entries to the OSE in the same invoke command. In a real implementation, this could be done by invoking when a fixed-sized batch of entries is ready, or after exceeding a timer. The OSE would then have to issue a list of blocks back to the CA, instead of just one. The method of reducing context switches to increase throughput is known, particularly in Intel SGX. [Tian et al., 2018] found that their switchless calls could reduce latency significantly (5 to 8 X speedup). Unfortunately, I have not found similar evaluations in ARM TrustZone, but the idea is the same. By reducing the number of context switches, there is less total invoke time and thus a lower latency.

Optimization Hypothesis

This subsection proposes a hypothesis of the effect batching will have, to estimate the effectiveness of batching, explain why it works, and help find the natural next step in optimizations. With batching, and expecting a full batch, our latency per block issued would be given by Equation 5.2.

$$LatencyPerBlock = \frac{InvokeTime + BatchSize * TALatency}{BatchSize} \tag{5.2}$$

By only invoking the command once per batch, the invoke time has a smaller total effect. Equation 5.3 shows the expected latency per block when issuing 10 blocks with an original buffer size of 8192 and a batch size of 10. With batching, the buffer size needs to be at least $8192 * 10 = 81920$. Therefore, the invoke time per command call is slightly increased, but the batching reduces the number of command calls, which overshadows this effect. For the case of this calculation, I use Table 5.3 to approximate that a buffer size of 81920 has an invoke time of 2.80 ms.

$$LatencyPerBlock = \frac{2.80ms + 10 * (4.38ms - 2.38ms)}{10} = 2.28ms \tag{5.3}$$

My hypothesis estimates that it would take 22.8 ms to issue 10 blocks with batching, compared to 43.8 ms without batching.

Optimization result

I implemented an alternative to the OSE that supports batching, to test my hypothesis and hopefully improve the OSE's throughput.

Batching	latency (ms)	latency per block (ms)	throughput (per second)
10	21.4	2.14	467
100	173	1,73	578

Table 5.5: Batching improves performance

Table 5.4 shows that batching increases throughput as expected, and it does so to a greater degree than my hypothesis suggested. This discrepancy between my hypothesis and the measured effect of batching can be due to my hypothesis not accounting for time saved from serializing and unpacking parameters once per batch, rather than once per block.
I measured how many blocks the OSE issued per second, for ten seconds. The test was performed with varying batch sizes, to compare performance, the result is shown in Figure 5.1.

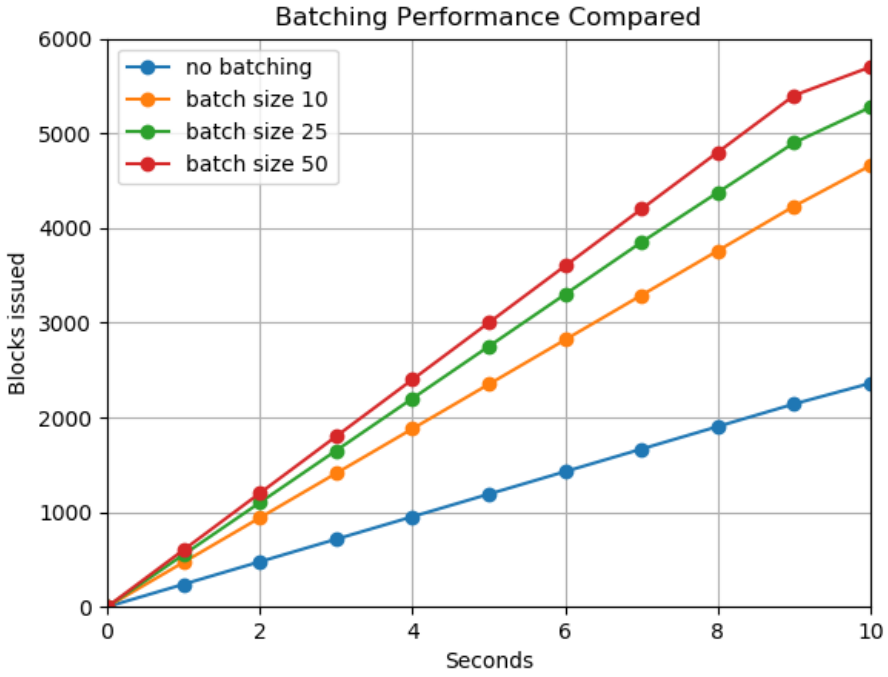

Figure 5.1: Big batches improve performance

Figure 5.1 shows that batching increases throughput. The difference between no batching and a batch size of ten is great, but further increasing the batch size has a reduced effect. The invoke time per block decreases from 2.38 ms to about 0.24 ms with just a batch size of ten; therefore, there is a diminished return in time saved by increasing the batch further.

There is also the option to combine data entries, such that one block contains multiple data entries. This would allow the logging of more data entries per second. Combining data entries for each block would also increase the buffer size needed.

The evaluations show that the Ordering Service Enclave has a latency of 4.2 ms per block issued. Which is about 14 times slower than the equivalent normal world implementation. [Göttel et al., 2019] found that TEE throughput yields a 12 to 17 times performance overhead on the Raspberry Pi. Batching the commands can reduce performance overhead to about 5 times a normal world implementation. The implementations are compared in Figure 5.2.

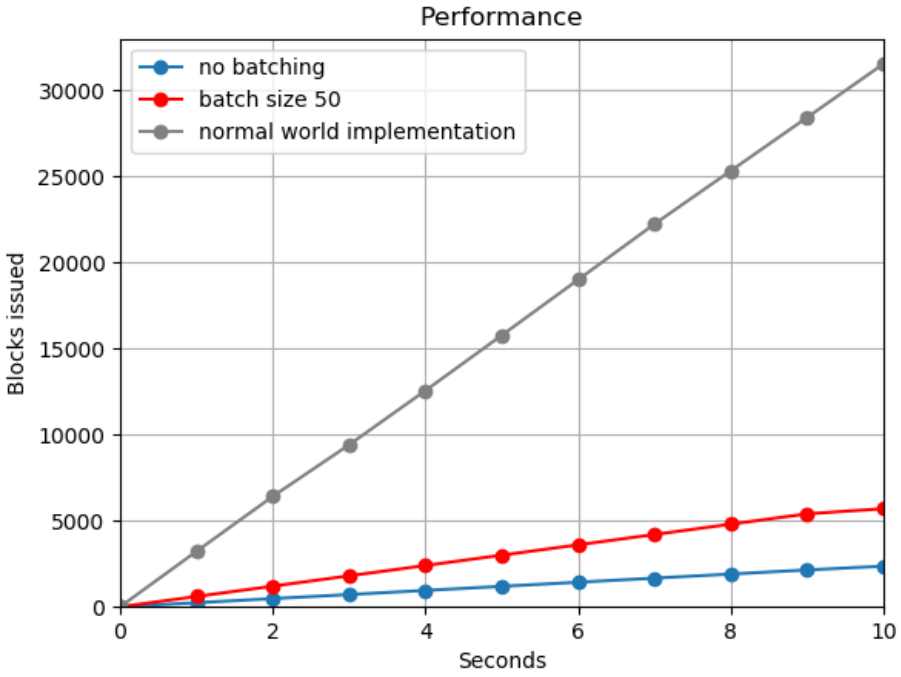

Figure 5.2: The Ordering Service Enclave is slow compared to an equivalent normal world version, even with batching

5.2.2 Further work

This section details what should be investigated further to optimize the throughput of the OSE or other RUST TAs.

Further work should be done to find out if there's a way to reduce the latency of the invoke command, which we have named invoke time. Batching is not always a viable solution, and an invoke time of two milliseconds can greatly reduce the throughput of Trusted Applications. Further optimization will be to reduce the latency inside the TA, by having faster hashing, signing, and serializing. As previously mentioned, verifying a block in the blockchain validates all previous blocks, this allows us to create fewer signatures. This can be utilized by for example only signing the last block of a batch.

To compare the performance of the various TEEs, implementations of the OSE for other TEEs like Intel SGX or AMD SME should be examined.

Chapter 6

Conclusions

Establishing a trusted and distributed ledger in systems of non-trusting parties is traditionally done through resource-heavy replication and distribution. By creating the blockchain in a Trusted Execution Environment (TEE), and performing verification, it can be trusted with lower development costs.

In this thesis, I presented a Trusted Application (TA) that executes the trusted creation and issuance of blocks. The TA is a prototype of [Bleeke et al., 2022]'s Ordering Service Enclave (OSE), but on ARM TrustZone technology and written in Rust. The OSE receives data to be logged from the REE and returns a block, as a part of a blockchain, with a digital signature proving that the block was created by the OSE. In this thesis, I have also demystified the development process using the Rust OP-TEE SDK, which showed that OP-TEE TAs can be effectively implemented in Rust. Rust makes it easier to write safe code and gives access to a rich infrastructure of public crates; however, some of these crates are not compatible and must be ported to OP-TEE for TA development. Rust's enforced memory-safety features increase the security of the TA; however, my implementation introduced four unsafe Rust blocks, which were necessary for passing parameters. These unsafe blocks bypass Rust's security features, thus, future work should be done to evaluate their and included crates' security. All other code is certain to be memory safe, as they pass Rust's built-in compiler checks. The Ordering Service Enclave is considerably slower than a normal world implementation, as is expected when using TEE technology. The OSE can be optimized to reduce the sluggishness of the TEE by sending batches of commands, thus, reducing the number of world switches. With optimizations, the Ordering Service

Enclave establishes trust from the TEE, with the consequence of a 5 times performance overhead compared to a normal world implementation. This overhead is an acceptable cost for the security and trust of the TEE. This thesis establishes that the OSE can be ported to the IoT prevalent TrustZone technology, and for security reasons, should be written in Rust rather than C.

Bibliography

[Amacher and Schiavoni, 2019] Amacher, J. and Schiavoni, V. (2019). On the performance of arm trustzone. In Pereira, J. and Ricci, L., editors, *Distributed Applications and Interoperable Systems*, pages 133–151, Cham. Springer International Publishing.

[Apache, 2022a] Apache, T. (2022a). Optee_teec documentation. https://teaclave.apache.org/api-docs/trustzone-sdk/optee-teec/optee_teec/index.html.

[Apache, 2022b] Apache, T. (2022b). Optee_utee documentation. https://github.com/apache/incubator-teaclave-trustzone-sdk/blob/master/optee-utee/src/parameter.rs.

[ApacheTeacleave, 2020] ApacheTeacleave (2020). Teaclave trustzone sdk. https://github.com/apache/incubator-teaclave-trustzone-sdk.

[Bleeke et al., 2022] Bleeke, K., Mahhouk, M., Almstedt, L., Jehl, L., and Kapitza, R. (2022). Chainbox: Using tees and webassembly to run smart contracts on the edge.

[Doc, 2022] Doc, O.-T. (2022). Op-tee documentation. https://readthedocs.org/projects/optee/downloads/pdf/latest/.

[Documentation, 2022a] Documentation, O.-T. (2022a). Prerequisites for op-tee. https://optee.readthedocs.io/en/latest/building/prerequisites.html.

[Documentation, 2022b] Documentation, O.-T. (2022b). Trusted applications in op-tee. https://optee.readthedocs.io/en/latest/building/trusted_applications.html.

[edu, 2022] edu, M. (2022). Mit rust the programming language. https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/raw-pointers.html.

[Genode,] Genode. An exploration of trustzone technology. https://genode.org/documentation/articles/trustzone.

[Globalplatform, 2022a] Globalplatform (2022a). Internal core api. https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[Globalplatform, 2022b] Globalplatform (2022b). Protection profile. https://globalplatform.org/specs-library/tee-protection-profile-v1-3/.

[Globalplatform, 2022c] Globalplatform (2022c). System architecture. https://globalplatform.org/specs-library/tee-system-architecture/.

[Globalplatform, 2022d] Globalplatform (2022d). Tee client api. https://globalplatform.org/specs-library/tee-client-api-specification/.

[Globalplatform, 2022e] Globalplatform (2022e). Tee client api. https://higherlogicdownload.s3.amazonaws.com/GLOBALPLATFORM/transferred-from-WS5/TEE_Client_API_Specification-V1.0_c.pdf.

[Globalplatform, 2022f] Globalplatform (2022f). Tee white paper. https://globalplatform.wpengine.com/wp-content/uploads/2018/04/GlobalPlatform_TEE_Whitepaper_2015.pdf.

[Göttel et al., 2019] Göttel, C., Felber, P., and Schiavoni, V. (2019). Developing secure services for iot with op-tee: A first look at performance and usability. In Pereira, J. and Ricci, L., editors, *Distributed Applications and Interoperable Systems*, pages 170–178, Cham. Springer International Publishing.

[Jung et al., 2022] Jung, B., Eichler, C., Röckl, J., Schlenk, R., Hönig, T., and Müller, T. (2022). Trusted monitor: Tee-based system monitoring. In *2022 XII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8.

[Lab, 2019] Lab, S. S. . S. (2019). Sgx attestation. https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation.

[laginimaineb, 2016] laginimaineb (2016). Qsee privilege escalation vulnerability and exploit (cve-2015-6639).

[Li et al., 2015] Li, W., Li, H., Chen, H., and Xia, Y. (2015). Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 75–88, New York, NY, USA. Association for Computing Machinery.

[man, 2022] man, L. (2022). Getrandom manual. `https://man7.org/linux/man-pages/man2/getrandom.2.html`.

[Matsakis and Klock, 2014] Matsakis, N. D. and Klock, F. S. (2014). The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA. Association for Computing Machinery.

[Mé nétrey et al., 2022] Mé nétrey, J., Göttel, C., Khurshid, A., Pasin, M., Felber, P., Schiavoni, V., and Raza, S. (2022). Attestation mechanisms for trusted execution environments demystified. In *Distributed Applications and Interoperable Systems*, pages 95–113. Springer International Publishing.

[Ménétrey et al., 2022] Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). Watz: A trusted webassembly runtime environment with remote attestation for trustzone. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1177–1189.

[Müller et al., 2020] Müller, C., Brandenburger, M., Cachin, C., Felber, P., Göttel, C., and Schiavoni, V. (2020). Tz4fabric: Executing smart contracts with arm trustzone : (practical experience report). In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 31–40.

[Ngabonziza et al., 2016] Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. (2016). Armexplained.

[OPTEE, 2022] OPTEE (2022). Optee build. `https://optee.readthedocs.io/en/latest/building/devices/rpi3.html#build-instructions`.

[Pinto and Santos, 2019] Pinto, S. and Santos, N. (2019). Demystifying arm trustzone: A comprehensive survey.

[Rosenberg, 2014] Rosenberg, D. (2014). Qsee trustzone kernel integer overflow vulnerability. https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf.

[Suciu et al., 2020] Suciu, D., McLaughlin, S., Simon, L., and Sion, R. (2020). Horizontal privilege escalation in trusted applications. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[Teacleave, 2022a] Teacleave (2022a). Ta data size limit. https://github.com/OP-TEE/optee_os/issues/1429.

[Teacleave, 2022b] Teacleave (2022b). Ta open session attribute. https://github.com/apache/incubator-teaclave-trustzone-sdk/blob/master/optee-utee/macros/src/lib.rs.

[Teacleave, 2022c] Teacleave (2022c). Teacleave development. https://teaclave.apache.org/trustzone-sdk-docs/overview-of-optee-rust-examples/.

[Teacleave and elias vd, 2022] Teacleave and elias vd (2022). Poc attestation pull request. https://github.com/OP-TEE/optee_os/pull/4011.

[Teacleave and HakonToemte, 2022] Teacleave and HakonToemte (2022). Third party crates. https://github.com/apache/incubator-teaclave-trustzone-sdk/issues/99.

[Tian et al., 2018] Tian, H., Zhang, Q., Yan, S., Rudnitsky, A., Shacham, L., Yariv, R., and Milshten, N. (2018). Switchless calls made practical in intel sgx. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, page 22–27, New York, NY, USA. Association for Computing Machinery.

[Tømte, 2022] Tømte, H. (2022). Blockchain github repo including optimized version. https://github.com/HakonToemte/BlockChain.

[Wan et al., 2020] Wan, S., Sun, M., Sun, K., Zhang, N., and He, X. (2020). RusTEE: Developing Memory-Safe ARM TrustZone Applications. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20.

[Zhao et al., 2014] Zhao, S., Zhang, Q., Hu, G., Qin, Y., and Feng, D. (2014). Providing root of trust for arm trustzone using on-chip sram. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, TrustED '14, page 25–36, New York, NY, USA. Association for Computing Machinery.

[Zheng et al., 2020] Zheng, Z., Xie, S., Dai, H.-N., Chen, W., Chen, X., Weng, J., and Imran, M. (2020). An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491.

University of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Hein Meling