DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Running a language interpreter inside the ARM TrustZone: An exploration of dynamic code execution in trusted execution environments

Adrian Steffan

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Running a language interpreter inside the ARM TrustZone: An exploration of dynamic code execution in trusted execution environments

Betreiben eines Interpreters in der ARM TrustZone: Eine Untersuchung dynamischer Programmausführung in vertrauenswürdigen Laufzeitumgebungen

Author:	Adrian Steffan
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	M.Sc. Teemu Kärkkäinen
Submission Date:	19.10.2020

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 19.10.2020 Adrian Steffan

Abstract

Dynamic execution environments allow clients to remotely execute arbitrary functions and, as a result, have special needs in terms of security. Servers can offer security-critical functionality by utilizing trusted execution environments (TEE) to run certain parts of a program in isolation. However, using TEEs for dynamic code execution presents challenges, as the programs targetting these secure environments are often not portable and unfit for dynamic loading. Conversely, scripts written in interpreted languages are easily transmittable and can run cross-platform.

In this work, we propose a system that combines the flexibility of using interpreted languages with the isolation provided by TEEs. Our system enables the execution of scripts inside a TEE. Additionally, it allows for persistently storing scripts in the TEE for repeated invocation. We further present a prototype implementation using the Lua language, the source code of which was made publicly available as an open-source project. Our evaluation shows that our system introduces a performance overhead when running Lua scripts over native applications in the TEE. This slowdown falls in the same order of magnitude as the overhead normally found when comparing Lua and C. We conclude that our system can offer developers a tradeoff between flexibility and performance when developing for TEEs.

Contents

1 Introduction

In environments where computing power is limited, it is common to delegate computationally heavy tasks to a remote server. In traditional setups, these servers offer a predefined set of functions or application programmable interfaces (APIs) that expect data input from the client. Typical examples of such a setup are mobile voice assistants: Here, the client sends voice data to the vendor's server, which then runs the data against a pre-deployed voice analysis algorithm.

In contexts where more dynamic task offloading is required, servers in this static model might not offer all of the functionality a client desires. One approach that solves this is to allow clients to send the code required to run their desired functions to the server (alongside the input data). The server then acts as a general computing platform that provides a runtime environment (RE) for the transmitted code. Similar approaches have been used for popular services like Amazons AWS Lambda [1].

As the offloaded functions grow in complexity, they might have special requirements in terms of software security. These requirements could include: Protecting sensitive data, isolating the executed functions from other programs running on the system, and ensuring the calculations' integrity with cryptographic algorithms.

One way remote and potentially untrusted machines can offer such security-critical functionality is the employment of trusted execution environments (TEE). These environments use specialized hardware that allows for the execution of certain program parts in isolation and guarantees the integrity of calculations performed in them. For example, systems like VoiceGuard [2] and Occlumency [3] utilized TEEs to protect user data in static task offloading scenarios.

However, using TEEs for dynamic code execution presents challenges, as the programs utilizing these secure environments are often not portable. Conversely, scripting languages offer a portable format for their programs, but interpreters for these do not run natively inside of trusted environments.
The goal of this thesis is to bridge this gap by embedding a language interpreter inside a TEE. More specifically, our approach targets the Arm TrustZone platform, as similar work has been conducted for Intel SGX [1] [4].

[1] Intel SGX is an implementation of TEEs available on Intel processors.

Our main contributions are as follows:

- We propose a system that enables the execution of scripts inside a TEE using language interpreters. We extend this system to support the persistent deployment of scripts to the TEE and make them callable by interpreters running outside of the secure environment.
- We present a prototype implementation of our system using the OPTEE platform and the Lua scripting language.
- We demonstrate the practicability of our system and examine general performance properties by evaluating our prototype implementation.

This thesis will first explain the concepts behind the used technologies and discuss related research. We then detail our conceptual system design and highlight relevant design decisions. Afterward, we describe our reference implementation and focus on the Lua interpreter's modifications required to run in the TEE. Following this is a discussion of the experiments that we ran on our implementation and the notable results. Lastly, we summarize our findings and provide a list of topics for further research.

2 Background

This chapter will discuss the concepts that build the technological basis for the system that we created. Furthermore, it will examine related research that deals with similar topics to those presented in this work.

2.1 Dynamic Runtime Environments

The primary motivation for this work is to provide an extension for dynamic runtime environments. We define these environments similar to Guerin, Kärkkäinen, and Ott [5]: They offer flexible computing platforms that dynamically receive and execute code from clients. This concept allows clients to dictate how they use the resources available on the server that performs the computation. In the use case presented by [5], the code is run on microcontrollers to capture sensor data and control actuators in their surroundings. They also orchestrate multiple microcontrollers with their loaded code, which allows them to compose significantly more complex services than static functions would be able to provide. To make the code snippets portable across different machines [5] uses Lua, an interpreted language. We adopt this approach in our system design.

2.2 Interpreted Languages

Traditionally, compiled languages like C use a compiler to translate the high-level source code written by the developer to low-level machine code understandable by machines. This machine code makes up the executable binary and is tailored to the architecture of a specific platform. The executable artifact is, therefore, distinct from the source code that the developer wrote.

Conversely, programs written in interpreted languages are translated concurrently to the program's execution on an instruction-by-instruction basis. The so-called interpreter handles this translation. The developer's code and the executable artifact are the same. In the remainder of this thesis, we will refer to such a piece of code as "a script".

As the translation happens repeatedly with every execution of a program as opposed to only once during compile time, interpreted languages can introduce a performance overhead compared to compiled languages. Even though compiled languages can be more performant, interpreted languages bring other advantages in simplicity and portability. The syntax focuses on developer productivity, and the same script can be reused on multiple different

machines. The only part that needs to be adjusted to the target platform is the interpreter itself, whereafter all scripts written in the given language can be executed. This property makes scripting languages attractive for dynamic runtime environments: Clients looking to run their code only need to know whether a given interpreter is available on the target machine. Therefore, servers do not have to expose information about their architecture, and scripts stay compatible with platforms introduced in the future.

Hybrid languages combine properties from both compiled and interpreted languages. They translate a program's source code into bytecode, which is then run on virtual machines that fill a role similar to interpreters. This partial translation reduces the performance overhead for each run while still keeping the code's representation portable between virtual machines running on different platforms. Java is a prevalent example of such a hybrid language, popularizing its Java Virtual Machine with the slogan "write once run everywhere" [6].

2.2.1 Lua

One example of an interpreted scripting language is Lua. It was designed as an extension language that dynamically extends the functionality of other programs [7]. Even though Lua is referred to as an interpreted language by its authors, it is translated to bytecode and runs on a virtual machine for improved performance, sharing properties with hybrid languages [7]. For this thesis's reference implementation, we chose Lua as the scripting language to run inside the TEE. Lua is open source, has an interpreter with a small footprint, and offers an easy to use API to communicate with C programs [8]. These properties greatly facilitated the development of our prototype. Furthermore, as Lua was developed as an extension language usable by non-programmers, the syntax and structure of Lua scripts are simpler than those of other programming languages. This property makes it a prime choice to showcase our system, as the language itself does not add unnecessary complexity.

2.3 Trusted Execution Environments

[5] states that allowing for the execution of code from external sources can be problematic for a system's security. Furthermore, isolating scripts provided by clients could also offer protection to the client, who might want to hide computed data from other programs running in the dynamic runtime environment. One way these REs could offer isolation and related security features to clients is by employing trusted execution environments.

A trusted execution environment is a processing environment that runs on a kernel that is separated from the normal operating system (OS) [9]. It guarantees security-critical aspects like the authenticity of the code that runs in it and the integrity of runtime states like CPU registers and memory [9]. If it offers persistent data storage, the confidentiality of the stored data is also guaranteed [9]. These features provide an elevated level of security compared to execution on the normal OS kernel. Therefore, it makes sense to delegate the execution of security-critical code (e.g., verification of cryptographic hashes, key derivation) to this

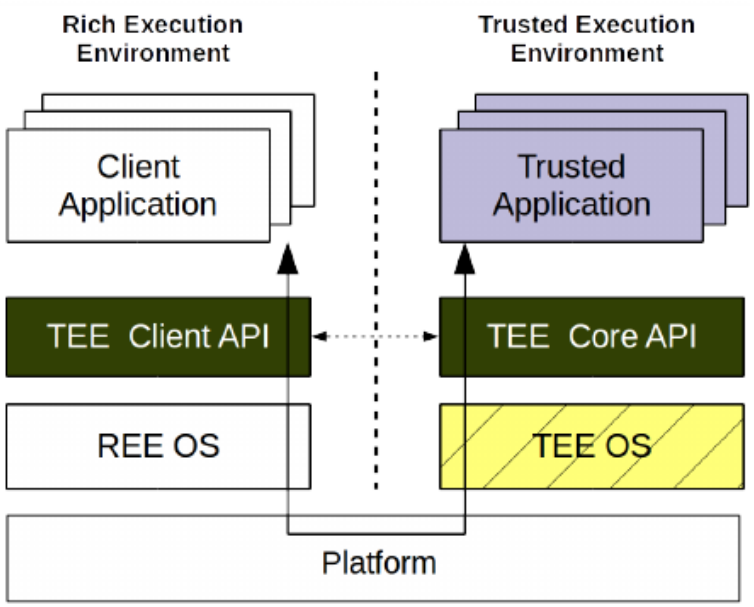

Figure 2.1: Structure of a TEE architecture. Source: [13]

secure environment. Due to the code running on a separate kernel, the execution is isolated from other processes running on the same machine. Furthermore, as only a small portion of programs are executed on the trusted side, the overall attack surface is minimized.

GlobalPlatform API. For the rest of this thesis, we will use the terminology and structure from the TEE specification proposed by GlobalPlatform [10]. Both our design and implementation section are based on the GlobalPlatform TEE Internal Core API [11] and the TEE Client API Specification [12].
Figure 2.1 shows the overall structure of a system that incorporates a TEE. The non-trusted side (also referred to as "rich world" or "rich OS") is separated from the trusted side (also referred to as "TEE", "secure world", or "secure side") on a hardware level. The programs running on both sides can only communicate by accessing the APIs defined by GlobalPlatform.

Programs running on the trusted side are called trusted applications (TA). These are not executed as standalone programs, but they act as APIs that provide callable functions for the non-trusted side. Once a program running on the rich OS calls a trusted function, the underlying OS handles the context switch to the trusted environment and prompts the TA's execution.

A more detailed view of this process can be seen in Figure 2.2. A rich world program that accesses a trusted function is called a client. Clients start their interaction with a TA by opening a session, which triggers the creation of a TA instance with its own physical memory space. The session represents the connection between the client and the TA instance and logically groups the commands issued by a single client. The client can now repeatedly invoke

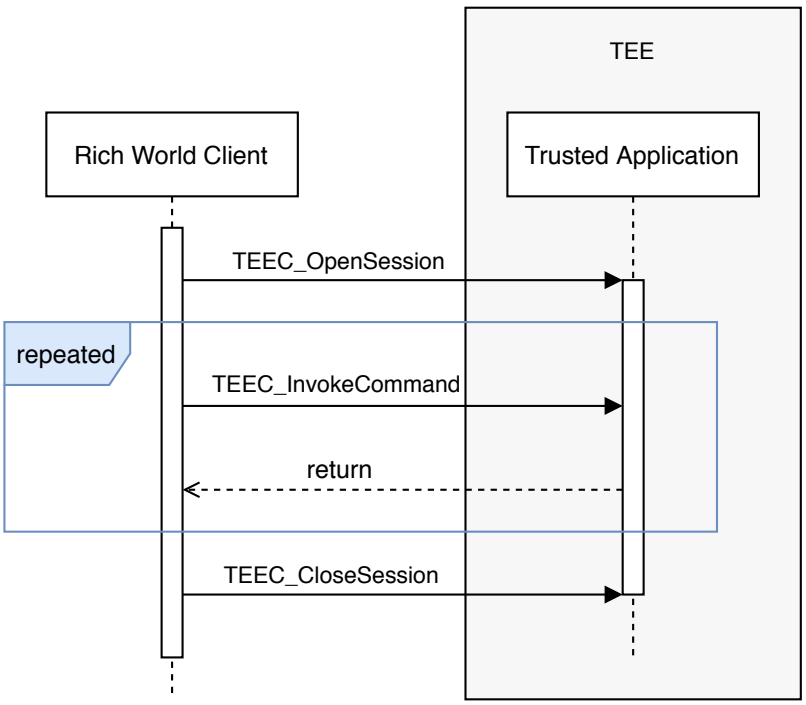

Figure 2.2: Basic lifecycle of a TA instance.

commands to call functions of the TA and pass argument data. When the client finishes its communication with the TA, it closes down the session, whereafter, the resources used by the TA instance are freed.

Arm TrustZone. In order to isolate resources for the execution in the different worlds, the underlying hardware needs to support this kind of separation. For this thesis, we looked towards Arm TrustZone. TrustZone is a collection of hardware extensions built into both Arm application processors (Cortex-A) and Arm microcontrollers (Cortex-M) [14]. It supports the partitioning of systems into two worlds with a special hardware mode. A physical processor core is virtualized as two virtual cores that represent the two distinct execution environments [9]. This virtualization ensures that the core operates in exclusively one of the worlds at any given time [14]. TrustZone uses a 33rd processor bit, the so-called Non-Secure bit, to signal in which world the current instruction is being executed [14]. Security-critical peripherals can read this bit via a bus to determine the current processor state [14]. A privileged instruction, the "Secure Monitor Call", was introduced to set this bit and switch between the secure and non-secure world [14]. The separation of physical memory for the two worlds is enabled by the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) [15]. Certain parts of the memory can be restricted so that they are only accessible by the secure world. This mechanic enables TAs to have memory separated from rich world programs and still get input from clients via a shared memory space. [14], [9], and [15] provide further, more detailed surveys of the TrustZone architecture.

TrustZone does not pose any restrictions on the TA developer, as TEE implementations based on GlobalPlatform abstract away the details of the underlying hardware system. Therefore, the code written for our TA is not dependant on the hardware platform that enables the isolation.

2.3.1 Related Work: Using TEEs in remote execution

Multiple research projects have been dedicated to using TEEs in order to protect clients' data in remote execution scenarios. For example, Voiceguard [2] uses Intel SGX to protect user voice data in remote voice analysis tasks. In their architecture, the client can share a symmetrical encryption key with the secure environment by leveraging Intel's remote attestation service. The voice data is then sent to the server encrypted and is only deciphered inside the TEE for analysis. This setup ensures that neither the machine owner nor other programs running on the server can access the unencrypted voice data.

While voice processing has been shown to have acceptable performance in TEEs, Intel SGX has memory limitations that introduce a significant performance overhead for deep learning tasks involving convolution [3]. Occlumency [3] proposes a new way of performing convolution to accelerate deep learning inference, speeding up the computation by a factor of 3.6 compared to non-modified convolution in Intel SGX. Research in this area can make TEEs viable for running more intensive inference tasks on images or videos.

Similar works show the potential of building applications around security features of TEEs that protect user data in static remote execution environments. Our work extends this idea and aims to make the TEE's security features themselves available to users deploying scripts to a server.

2.3.2 Related Work: Running language interpreters inside of TEEs

This thesis is not the first work that runs language interpreters inside a TEE. Previous research by Wang et al. proposed ScriptShield [4], a framework that enables the execution of unmodified interpreters inside Intel SGX TEEs. The main challenge of porting programs to these secure environments is that certain code (like system calls) is incompatible with TEEs [4]. ScriptShield solves this issue by wrapping the language interpreter and redirecting the execution of system calls back to the normal world. Using this method, they manage to run interpreted scripts inside Intel SGX TEEs with a minor performance overhead.

Our work differs from theirs by explicitly targeting Arm TrustZone. As Arm TrustZone offers no feature for redirecting system calls back to the normal world, the approach presented by [4] is not applicable. To run under TrustZone implementations, language interpreters would need to be modified. Even though this increases development overhead compared to ScriptShield, porting these interpreters enables machines with Arm architectures to act as dynamic execution environments.

Feifan Chen and Mehrotra [16] successfully ported a Lua interpreter to OPTEE, an open-source implementation of TrustZone TEEs. They used Lua scripts to define access policies for data, which were to be evaluated inside a trusted environment. As they did not need all functionality of the Lua interpreter for their purposes, their port resulted in a reduced feature set (further elaborated in Section 4.3). Their implementation of the interpreter targeted a specific use case. This thesis builds upon that implementation to generalize its applicability: We design a system that wraps the interpreter to provide a dynamic execution environment similar to the one shown in [4].

3 System Design

This chapter will look at the general system design of an architecture that allows dynamic code execution on platforms with trusted execution environments. We will first briefly overview the entire system with its core functionality and then discuss the critical design considerations taken while examining the system's parts in more detail.

3.1 High-Level Overview

Platforms that support TEEs allow for the partitioning of programs, running certain parts isolated in the trusted environment. Our overarching goal was to make the loading and execution of new programs on these platforms as frictionless as possible, while still retaining the ability to use the TEE for critical security operations. For this purpose, we propose a system that utilizes language interpreters so that programs written in scripting languages can dynamically be loaded to extend a system's capabilities. The overall structure of the system can be seen in Figure 3.1.

The following is an overview of the main set of features that our system provides at a high level:

- Run language interpreters both inside the TEE and on the rich OS side
- Load scripts into both sides and execute them using the language interpreter
- Enable the communication between scripts on both sides using the underlying interfaces provided by the TEE[1]
- Give the TEE persistent access to scripts deployed to it so that they can be used to build up callable trusted APIs
- Enable scripts in the TEE to call other scripts living there in order to allow the composition of functionality
- Provide a mechanism to control what scripts are allowed to run in the TEE

[1]This communication only flows in one direction. Rich world scripts invoke trusted scripts and provide arguments, while trusted scripts pass return values back to the rich side.

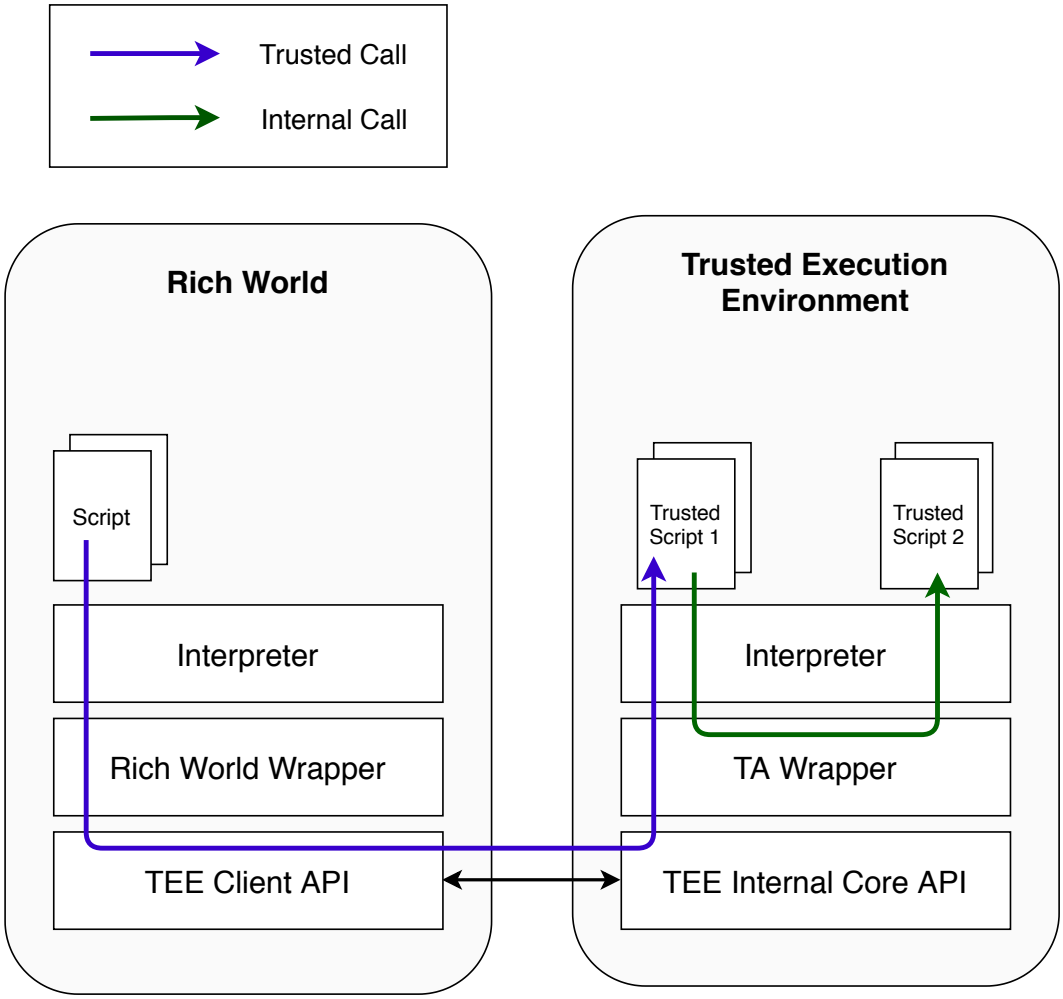

Figure 3.1: High-Level System Overview

These features allow us to compose applications that utilize both the trusted and untrusted side of a given trusted platform. Because these applications consist of interpreted language snippets that need no compilation, they can run on any platform that supports our system without modification. The interfaces offered between scripts inside the TEE and to scripts outside the TEE allow building the application's secure parts in a modular fashion, reusing previously deployed functionality.

While we can use the system to develop both the trusted and untrusted parts of the application in a scripting language, we do not necessarily have to use the same scripting language on both sides. If the implementation handles the serialization of the data that is passed over the underlying TEE interface accordingly, two different interpreted languages can be employed on both sides. The TEE scripts could even be invoked by other rich world applications that are not part of our system. This flexibility allows for integrating our system and the scripts with existing applications that rely on TAs. Therefore, developers can implement specific parts of an application in lower-level languages if they better fit their needs while still accessing the APIs provided the deployed scripts.

In the following section, we will take a closer look at how we designed the various parts of our system to include the features listed above.

3.2 Detailed Design

While the general concepts of this architecture can be applied to any TEE, we will focus on TEEs that follow the GlobalPlatform TEE System Architecture for the rest of the design section. Additionally, we devised the system design to target ARM TrustZone [17]. While the choice of the hardware platform had no significant impact on the overall design, we want to highlight that similar approaches have already been found for other architectures, like the Intel SGX [4].

At its core, our goal is to add functionality to an operating system in the form of new trusted applications. These should then later be consumable by programs in the rich OS. Some implementations of the GlobalPlatform API, like OP-TEE [18], already offer the ability to add pre-compiled TAs while the system is running. However, we argue that this method of adding TAs has downsides that make it unfit for various situations.

Due to differences in the platforms that implement the GlobalPlatform API, a TA is compiled for a specific target platform [19]. This fact might cause problems if we want to use the same application on different machines or deploy a TA over a network where information about the target platform is not exposed. Even if this information is available, adding new functionality to a system would be restricted to sources that either a) carry a set of pre-compiled binaries or b) have the toolchains necessary to build one. Scenarios in which clients with restricted operating systems (e.g., mobile devices) act as sources would be problematic. Likewise,

situations, where we add machines with different architectures over time, would require updates for all sources.

Furthermore, by adding new TAs as pre-compiled binaries, we rely on the access control mechanisms set by the TEE implementation. TEEs need to ensure that they only execute TAs from trusted sources. For example, OP-TEE achieves this by signing TAs with a single key pair [20]. We may want to differentiate between sources and restrict access to certain trusted environment features (e.g., access to storage) for different TAs. When relying on pre-compiled binaries, this might not be achievable without modifying the specific TEE implementation itself.

Our proposed solution to these problems is the use of interpreted languages to build TAs. Like we have discussed in Section 2.2, scripts for these can run on any platform that offers an interpreter for the given language. This property makes the TAs portable, not only for different TrustZone platforms but also for different TEEs and architectures. On top of that, we can define more fine-grained mechanics for controlling the deployment of new TAs by extending the interpreter. For example, we could define a set of trustworthy sources and identify these by verifying digital signatures when loading in new scripts. The main drawback of using scripting languages is the decreased performance compared to lower level implementations. This work will address these drawbacks by performing performance measurements in Section 5.3 .

The remainder of the design section will focus on the following two parts: We will first discuss how we can utilize language interpreters so that scripts can act as trusted applications. Once that is established, we will examine how we can enable interpreters on the rich OS side to consume these scripts.

3.2.1 The Language Interpreter

Theoretically, one could use any interpreted language with our design. However, the language interpreter running inside the trusted environment needs to have two properties, which are important to consider when deciding what language to use in an implementation.

For one, the version of the interpreter used needs to be able to run inside a TEE. Because TEEs try to minimize their attack surface, they often do not offer the same system calls or standard C library functions as the rich OS. For example, OP-TEE does not offer any Linux system calls, and their libc is not complete [21]. Accordingly, if the language interpreter depends on syscalls or standard library functions, the interpreter needs to be modified in order to operate in the TEE. Modifications involve removing functionality that is unnecessary for our system and reimplementing the library functions needed for operation. An example of this can be found in Section 4.3, where we look at a modified Lua interpreter that runs on OPTEE-OS.

Secondly, the interpreter needs to offer an interface that the TA can call in order to execute scripts. More specifically, the TA needs to be able to set up an interpreter environment, load

in the script and arguments, execute the script, and get return values from the interpreter. Popular interpreted languages like Python and Lua offer APIs that accomplish this [22] [8].

3.2.2 Running Scripts

The basis of our system is made up of a traditional TA that wraps the modified language interpreter. For applications to be able to use the interpreter running on the secure side, our TA needs to offer the following interface to the rich world:

```
run_script(script, args)
```

The first parameter refers to the script we want to run inside the trusted application. It represents a single function with no external dependencies that takes an input and provides a return value. It can either be represented by a string containing the script text or bytes containing the intermediate bytecode if the chosen interpreter supports that.

The second parameter can be any data we wish to pass to the script as input. As we are working with TAs like they are defined in the GlobalPlatform TEE Internal Core API [11], our interface targets the C language. It is unlikely that every datatype of the chosen interpreted language has a direct equivalent in C. This is why the consumer of the TA's API needs to serialize the script's input arguments into an intermediate form that can be passed to our interface. Once our wrapper TA receives the data, it will then deserialize it into a format that the interpreter running the script can work with. Consequently, the script's return value also needs to be serialized before passing the data back to the rich world.

Figure 3.2 depicts the sequence of events that happen when the `run_script` interface is called. Note how the TA acts as a mediator between the interpreter and the rich word caller by setting up the interpreter environment, loading in the script, and serializing/deserializing the script's input and output values.

Since every script gets a new execution context in the interpreter, the scripts being run in the TEE act like stateless functions. Variables, functions, and loaded modules are not shared between script calls. As this work mainly focuses on the interfaces needed to make such a system operate, we found this to be sufficient. Additionally, separating state prevents accidental variable shadowing between scripts. However, if one wanted data to persist between script calls, this could be achieved by keeping the current interpreter state in memory and reusing it with each subsequent call to the TA.

Similarly, every instance of the TA gets its own memory space. If multiple clients in the real world were to invoke trusted scripts, they would each get served by different trusted interpreter instances with separate memory spaces. We found this separation to be practical, as memory overflows in one instance would not affect the other instances of the interpreter. GlobalPlatform does, however, allow for a single TA instance to serve all incoming rich world calls, so the system could be changed to operate on shared memory space [23].

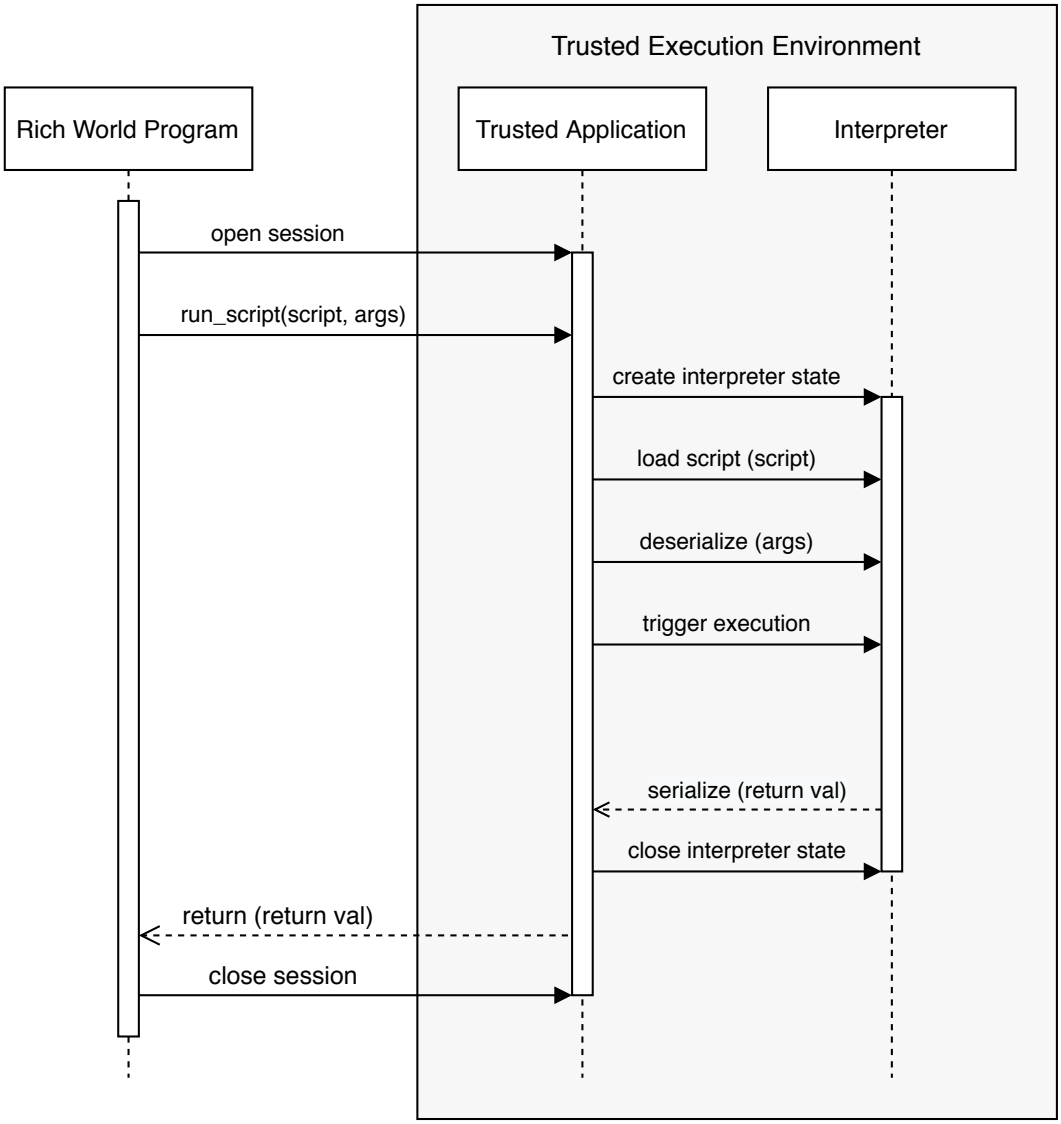

Figure 3.2: The sequence diagram for calls to run_script

Combining these two properties allows for three possible setups:

- Every script gets its own interpreter state
- Interpreter state is shared between scripts of the same rich world client
- All scripts share the same interpreter state (if multiple rich world apps want to run scripts concurrently, access control mechanisms would need to be used to keep a consistent interpreter state)

We designed the system with the first option in mind; however, many of the presented concepts also apply to the other setups.

3.2.3 Persistent Scripts

The above interface allows us to input scripts into our TA and execute them with the modified interpreter. However, this requires us to pass scripts into the TEE every time we want to run them. It might be desirable to deploy scripts to the TEE in a persistent way so that we can invoke them later without repeatedly providing a particular script as input.

There are several advantages to this approach: For one, it enables us to preload scripts in order to optimize performance. In situations where getting the script into the TA is time-consuming (e.g., a large-sized script being provided via poor network conditions), it can be cheaper to store it and perform a lookup when invoked. Storing scripts also allows other rich world applications to use the interface that our TA provides. Suppose there is functionality that is commonly used in the TEE (e.g., checking signatures, performing symmetrical encryption). In that case, we can deploy scripts for these tasks so that other rich world applications can use that functionality without creating TAs of their own. Lastly, scripts running in the TA can be enabled to directly call other scripts that were previously deployed without leaving the TEE. This mechanic allows for the composition of multiple TAs in order to create more complex programs. It also makes it possible for TAs to reuse standard functionality already present in the TEE, making the development of new scripts more efficient.

To store and access scripts on the trusted side, we use the permanent storage offered by the GlobalPlatform API. All instances of a particular TA share access to the same part of the secure storage [24]. That means that if one instance of the wrapper TA saves a script, a different instance can later access the script form the storage. We consider writing to the storage to be necessary to make the scripts reusable between runs of the TA. However, not every call that invokes a saved script also needs to access the permanent storage. Keeping scripts in the memory of a TA instance can reduce the total computation time, as memory access is generally faster than reading from permanent storage. We will explicitly measure and further discuss this difference in Section 5.2.3.

To enable the saving of scripts, our trusted application offers the following interface to the rich world:

```
save_script(script, script_id)
```

The `script` argument can either be a string containing the script text or intermediate bytecode. The format of the script is the same for the `run_script` interface.
The `script_id` is an identifier that will later be used to access the saved script. For small scale setups, we propose using a string representation of the function name to make calling the scripts from the rich world intuitive. In larger systems with multiple independent sources, the usage of Universally Unique Identifiers (UUIDs) [25] could be considered, as our design does not include an explicit way of handling identifier collisions. For comparison, the TEE Internal Core API Specification also employs UUIDs to reference the binary TAs [26].

We do not specify what return value this function passes back to the real world, as we think this aspect should be adjusted to the specific implementation. An example would be the transmission of error codes to signal possible failures during the saving process (insufficient memory or storage space, script_id collisions, invalid script formats).

To later call the saved scripts, we define the following interface:

```
run_saved_script(script_id, args)
```

The `script_id` specifies which of the deployed scripts is to be invoked. It has the same form as in `save_script`. The second parameter is the intermediate representation of the data that should be passed to the script as input. It has the same form as in `run_script`. The function returns the serialized value that the interpreter returns after running the specified script.

During a run of the function, the TA loads the referenced script from either the storage or memory. It then proceeds to deserialize the data and call the interpreter in the same way as in `run_script`. In Figure 3.3 , the sequences of events for both `save_script` and `run_saved_script` are shown.

In setups where the interpreter state is to be shared between script calls, the additional serialization and deserialization are not needed. We can use the interpreter specific mechanics of passing open arguments between script functions in a single state (e.g., passing arguments to a function via the call stack).

The three interfaces `run_script`, `save_script`, and `run_saved_script` describe the core functionality of the trusted interpreter. Even though additional interfaces like `delete_script` make sense for the system's practical implementation, we deemed the three core interfaces sufficient to explore the ideas behind this system.

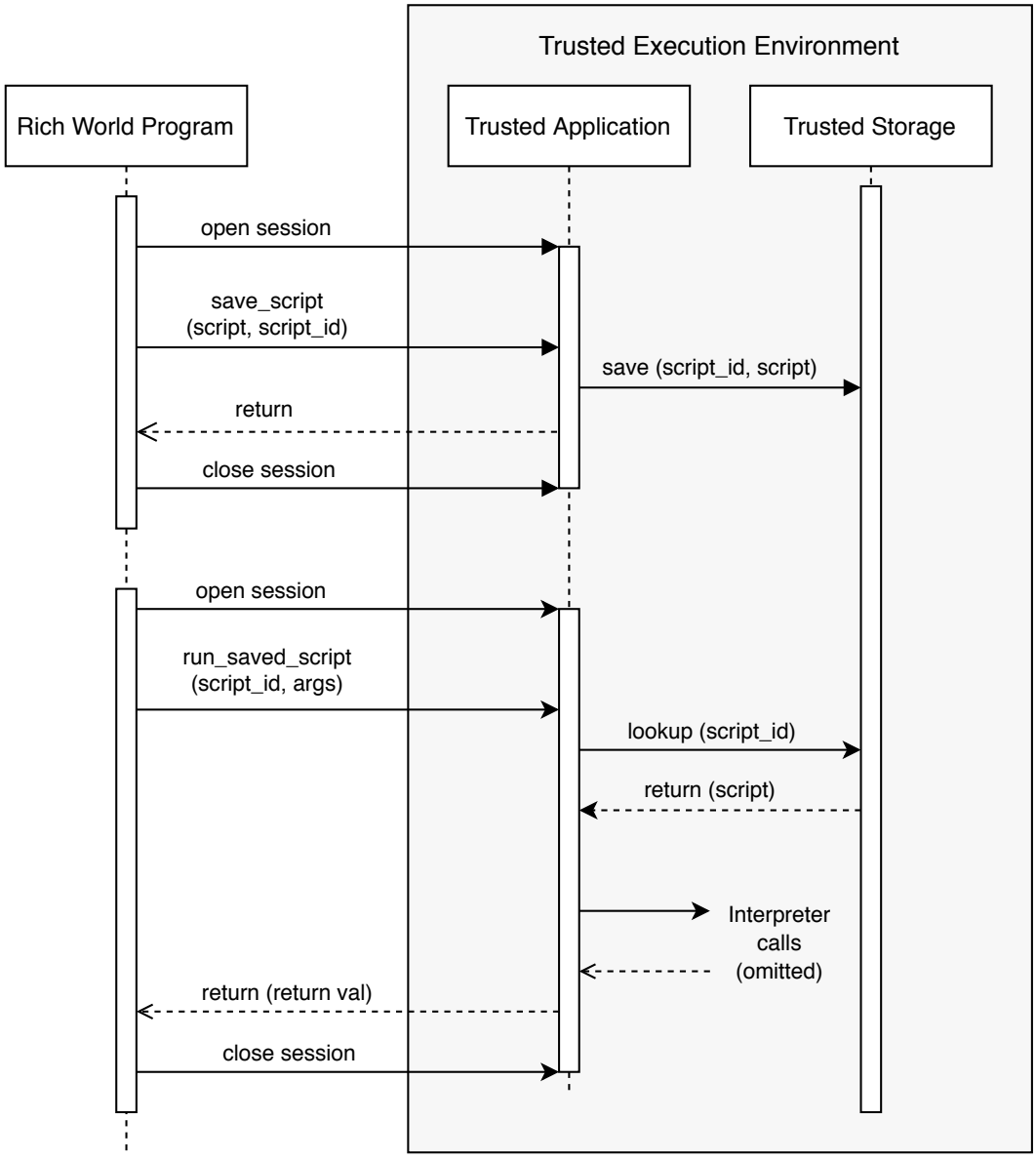

Figure 3.3: The sequence diagram for calls to `save_script` and `run_saved_script`

3.2.4 Security Considerations

Until now, our interfaces permit any rich world application to deploy and execute arbitrary scripts in the TEE. To prevent untrusted sources from doing so, we mimicked the principles employed by OP-TEE for regular TAs. TAs are cryptographically signed during the build process in order to make sure only scripts coming from trusted sources are executed [20]. Optionally, they are also encrypted to keep their contents secret while stored in the untrusted storage [20].

To replicate this behavior for our interpreter, we employ both encryption and signing in an Encrypt-then-MAC scheme [27]. We first need a shared secret that we communicate with our interpreter out of channel. As Encrypt-then-MAC requires two keys, we employ a key derivation function (KDF) to generate them from our shared secret. Scripts that should run inside the TEE are now first encrypted symmetrically using the first generated key. Afterward, we append the initialization vector (IV) used by the encryption algorithm to the ciphertext. We then use the second key to generate a message authentication code (MAC) over the ciphertext and IV. Our new script representation is the concatenation of the MAC, ciphertext, and IV.

To integrate this into the system described above, we can redefine our interfaces `run_script` and `save_script`. Instead of taking in a string or bytecode representation of the script, they now take in the composite of MAC, ciphertext, and IV. Before passing the script to the interpreter, the system uses the shared secret to derive the necessary keys, check the MAC, and decrypt the ciphertext to restore the script. This sequence allows us to reject any input that either comes from non-trusted sources or was tampered with. It also protects the contents of the script while being stored outside the TEE. If the TEE implementation is following the GlobalPlatform standard, functions to perform all of the actions specified above are available as APIs from inside the trusted application [11].

Naturally, this is only one possible way of increasing the security of such a system, and different use cases might require more sophisticated approaches. A comprehensive security analysis of the system falls out of the scope for this work.

3.2.5 Untrusted Side

With the system described in the last section, we can use scripting languages to implement TAs. However, programs on the rich world can only call these TAs via the C interface specified in the GlobalPlatform API. We also want to enable scripting languages in the rich world to consume the APIs offered by the scripts running in the TEE. To achieve this, we need to define interfaces that connect the calls made by the rich world scripts to the C interfaces defined in the last section.

We propose a modification to a rich world interpreter that would enable this. The interpreter's input consists of both the scripts running on the non-secure side and those that will act as TAs.

An additional function, `TA_call(script_id, arguments)`, is offered by the interpreter. It takes in both an identifier of the script and arbitrary arguments. When called, the interpreter fulfills three tasks: First, it serializes the arguments from the language-specific data types into a general representation that can be passed to the TAs API. Next, it calls the TA script specified by `script_id` by passing the needed script alongside the arguments to the `run_script` interface. Optionally, the interpreter can call `save_script` on all the TA scripts it received as input on startup, in which case `TA_call` only needs to pass the scripts identifier and the arguments to the `run_saved_script` interface. (We will discuss the performance difference of these approaches in Section 5.2.3.) After the TA returns, the interpreter needs to deserialize the return values and pass them back to the rich world script.

Using the described modified interpreter in the rich world combined with the one integrated with the TA allows us to write applications that have communicating parts in both the rich world and TEE. Finally, it is important to state that the rich world interpreter's integrity is not guaranteed because it runs in the non-secure part of the os. For important return values coming from the TEE, it would therefore make sense to employ a signing process using the TEE's cryptographic functions. When looking at the results later, we can verify that these values were computed in the TEE and that they were not tampered with by a compromised interpreter. In conclusion, the rich world interpreter provides a convenient way of accessing the secure APIs but does not add an additional security layer.

3.3 Summary

We have established that portability and the ability to run cross-platform make scripting languages a primary choice for writing dynamically loadable TAs. These properties lead us to design a system around the idea of executing scripts on an interpreter that runs in the TEE. For this system, we specified the required steps to load the scripts and accompanying arguments into the interpreter's environment. Our system treats scripts as stateless functions; however, minor modifications could enable us to keep some state in between script calls. Both storage and memory were deemed viable options to store scripts in the TEE across multiple invocations. Furthermore, persisting scripts in such a way can enable scripts in the TEE to call each other, allowing for the orchestration of more complex trusted programs.

Even though our system does not dictate any security measures to control the source of scripts, a combination of encryption and signing could be used to emulate the security provided for regular TAs.

We defined the three interfaces, `run_script`, `save_script`, and `run_saved_script`, for our system to enable rich world applications to deploy and run scripts on our TA. One possible consumer of these interfaces could be a modified language interpreter on the rich side, which would allow users to develop entire applications utilizing the TEE in scripting languages.

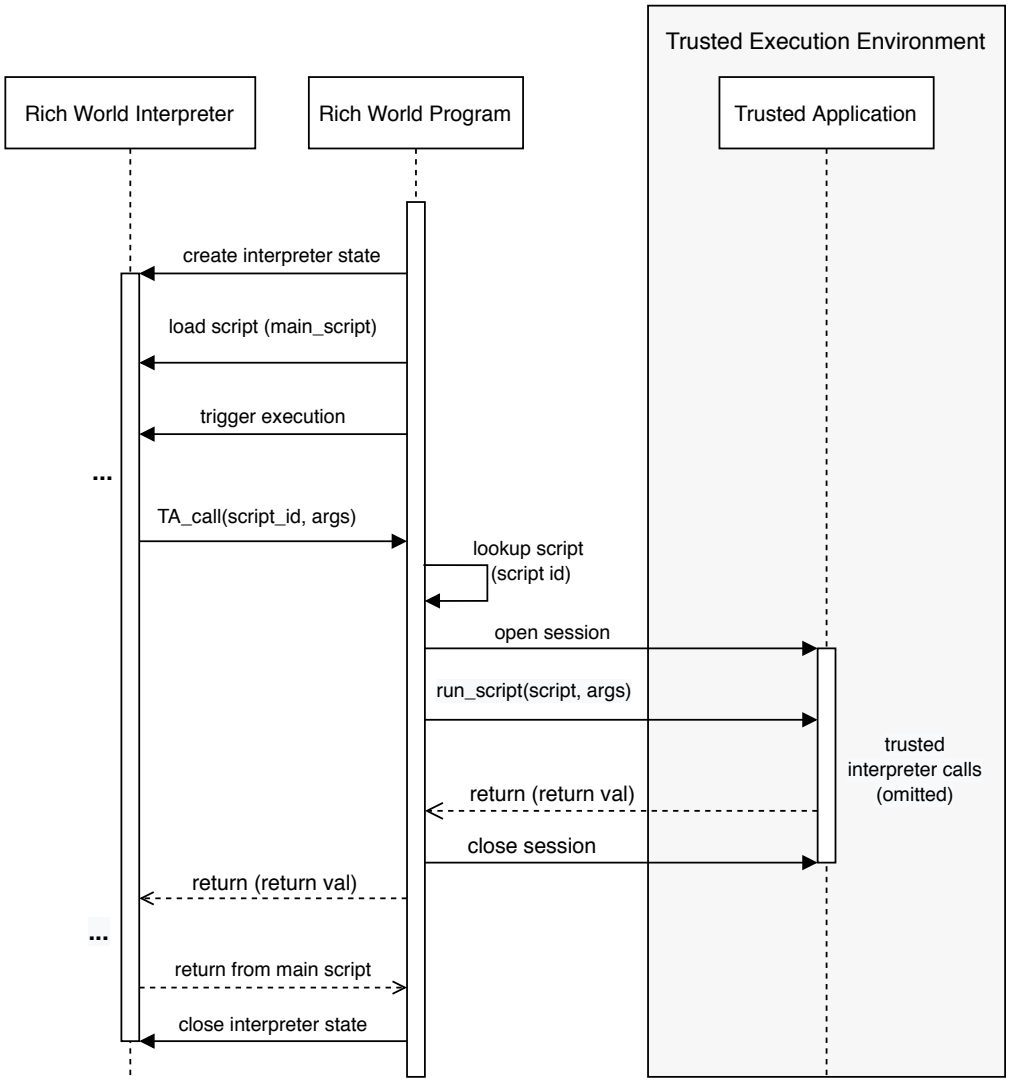

Figure 3.4: The sequence diagram showing the wrapper communicating between the TA and the rich world interpreter

4 Implementation

This section will detail a reference implementation[1] of the described system used to assess the overall feasibility and measure different performance aspects. We will briefly overview the technologies used and go over the concrete implementation details of the most important parts.

4.1 OPTEE

The Open Portable Trusted Execution Environment (OPTEE) [18] [13] is a TEE implementation based on Arm TrustZone developed by Linaro[1]. OPTEE is open-source and has extensive documentation, which is why we chose it as the platform for our reference implementation. The builds of OPTEE act as operating systems on target devices and provide both a rich world and a trusted environment. While OPTEE supports a variety of devices [19], we opted to use a Raspberry Pi 3 for our implementation and experiments. While this made for a convenient prototyping platform, we do not recommend this combination for production environments: At the time of writing, the Raspberry Pi family does not offer the full feature set of TrustZone, so running OPTEE on it will not result in a secure setup [28]. In addition to being based on TrustZone, OPTEE implements its TEEs according to the GlobalPlatform Specification. Our implementation can therefore be built for any platform that implements this specification.

4.2 Lua

We chose Lua as the interpreted language for our scripts. The main advantage for our purposes was the fact that Lua can be seamlessly embedded in C [8]. All of the functionality we need from an interpreter was exposed as a C API that our TA could use. To access the API, we linked the source of Lua to our TA during the build. Lua is also one of the faster scripting languages [29], which meant that we expected the performance tradeoffs compared to TAs written in C to be less significant. Most importantly, the interpreter is both open source and relatively small, so porting it to the TEE was feasible. To exchange data with C, Lua stores values on a virtual stack that can be manipulated using API calls [8]. The stack is part of a Lua "state" that needs to be passed to every API function. For example, a function call that

[1] https://gitlab.lrz.de/adriansteffan/optee_lua_runtime
[1] https://www.linaro.org/

we made use of would be `const char lua_pushstring (lua_State *L, const char *s)`, which pushes the C string s onto the Lua stack.

Another convenient feature of Lua is the fact that C functions can be setup up to be directly callable by the scripts running on the interpreter [8]. This interface enabled us to implement the internal calling between trusted scripts without further modifying the interpreter.

4.3 Modifications made to the Lua Interpreter

OP-TEE does not offer Linux system calls and comes with a reduced libc [21]. Running an unmodified Lua interpreter in the TEE is therefore not possible. Previous work by Feifan Chen and Mehrotra [16] produced a modified Lua interpreter with reduced functionality that is capable of running as part of an OPTEE TA. We integrated the interpreter into our TA and compiled a list of notable changes compared to an unmodified Lua 5.3.3. This list does not claim to be complete but instead aims to give an example of the modifications necessary to port an interpreter to a TEE platform.

- The functionality to read in .lua files and the standalone interpreter (that would execute .lua files by calling the C API) were removed, as only the internal C API was needed for operation in the TEE.
- The implementations for the following standard Lua packages were removed: package, io, os, debug, coroutine.
- The libraries string.h, ctype.h, and math.h were not available in full, so functions provided by those were partially reimplemented (e.g., isalpha, toupper). This change required the most effort during development.
- Whenever the Lua interpreter would resolve a mathematical operator, it would now either call one of the reimplemented math methods or, if that specific function were missing, would throw a custom error when parsing the script.
- On occasions where values from the standard libraries locale.h and time.h were used, dummy values were employed, as OPTEE does not provide a full version of those libraries at the time of writing.
- Error printing was rewritten not to invoke Lua specific functions, but rely on MSG and printf

The omission of the Lua coroutine is a notable change, as OPTEE does not offer multithreading for regular TAs [30]. Restoring the functionality would allow building TAs with multithreaded logic.

Certain mathematical functions and Lua standard libraries were still missing due to the limited use case of the project that provided us with the interpreter. We managed to restore

LUA_TYPE_NUMBER	An integer value
LUA_TYPE_STRING	A string value
LUA_TYPE_CODE	Any other arbitrary Lua element, represented as Lua code that returns that element

Table 4.1: The types of arguments accepted by run_script and run_saved_script

mathematical operators like fmod for testing purposes, but a full reimplementation of Lua's math library was out of scope for this work. Along with this thesis, we include a git repository[1] where the changes made to the Lua interpreter can be further explored by inspecting differences in the commit history.

4.4 Interacting with the Lua Interpreter

In the following paragraphs, we will discuss the implementation of the core function `call_lua` in the C language and the internal representation of both the script and the input arguments. We decided to use strings containing the source code as the representation for the scripts. This format helped with error printing and debugging, and we could directly make use of Lua's *load_buffer* interface, which receives the script text as a byte array. For Lua, in particular, bytecode could be a viable alternative [31].

The internal, serialized representation of the arguments consists of two parts: A void pointer `input` that would either refer to an integer or a byte array and a flag `input_type` that denotes how to interpret the data. The three possible types of arguments are listed in Table 4.1. In order to pass the data to the Lua script, we defined the two helper functions `stack_from_args` and `args_from_stack`. The deserializer (`stack_from_args`) manipulates the Lua stack so that a Lua representation of the input data lays on top of the stack for the script to use. The serializer (`args_from_stack`) pops the uppermost value from the Lua stack, determines its type, and creates a representation (output pointer, output_type integer) that can be passed to other C functions.

Both integers and strings have direct representations in C, meaning the interpreter can directly use the data. The serializer only has to know in what format to push the data to the Lua stack. The deserializer can use the C types `int` and `char*` to represent the return values in C. As there are no such equivalents for other Lua data types, we used a special string representation for the remaining types: These values are encoded by strings containing automatically generated Lua scripts that consist of a single return statement, which directly returns the value. To deserialize the input, we let the interpreter execute the script, which automatically leaves the desired value at the top of the stack. For serialization, we use the

[1] https://github.com/adriansteffan/optee-lua-diff

open-source Lua script DataDumper [32], which turns any Lua value into a script containing the described return statement. To run DataDumper, we execute it on the Lua interpreter with the stack we want to serialize. This usage highlights an interesting property of our system: As we have access to the full Lua API inside the TEE, we can not only run input scripts coming from the rich world, but we can also extend the functionality of our underlying TA using Lua, as we did with DataDumper.

Using code to transmit arguments results in a human-readable representation and enables us to transmit arbitrary Lua data types (excluding user data and C functions [32]). However, as the loading of arguments includes the execution of potentially unsigned code, this could lead to security issues. We made use of this solution to show that the passing of arbitrary data is achievable; real-world deployments will need to use serialization that better fits their security needs.

The C function that runs Lua scripts on our interpreter, `call_lua`, can be seen in Figure 4.1. It runs whenever we invoke a Lua script through `run_script`, `run_saved`, or when performing an internal call between scripts. It first creates a new Lua state and loads the available Lua standard libraries. Next, the C function that enables the calling of other trusted scripts is loaded into the state and exposed to scripts. The TA then loads the script into the buffer with `luaL_loadbuffer` and proceeds to deserialize the input arguments. Afterward, the loaded script is executed by calling `pcall()`, the return value is taken from the stack, and `output` and `output_type` are set accordingly. Finally, the Lua state is destroyed with `lua_close()`, as we do not keep state between script runs.

Having established how we represent the data when interacting with the interpreter, we will now look at how it is passed through the system's trusted side.

4.5 The GlobalPlatform Client API

To invoke TA functions from a C program running in the rich world, the TEE Client API defines the TEEC_InvokeCommand interface. This function expects three notable arguments: A reference to the current session, an integer specifying the TA function we want to call, and a TEEC_Operation struct. This struct includes a TEEC_Parameter array of length four, which acts as the primary way of passing arguments to the TA.

Each of the four TEEC_Parameter elements can either represent a "value" which is a struct containing two integers (referred to as a and b), or a pointer to a client owned, shared memory buffer. These TEEC_Parameter will also be manipulated by the TA to contain return values after the function concludes. The tables 4.2, 4.3, and 4.4 give an overview of how we utilized the TEEC_Parameter to pass data to and from the TA for the three interfaces described in the design chapter.

```
    void call_lua(char* script, size_t script_len, void* input, int input_type,
        void** output, int* output_type){

         /* create Lua state */
        lua_State *L = luaL_newstate();

        luaL_openlibs(L);

        /* Register the function for calling interal Lua scripts with the state */
        lua_pushcfunction(L, internal_TA_call);
        lua_setglobal(L, "internal_TA_call");


        /* Load the lua script from the buffer */
        luaL_loadbuffer(L, script, script_len, "lua_script");


        /* Push argument on the stack */
        stack_from_args(L, input, input_type);

        lua_pcall(L, 1, 1, 0);

        /* Return value of operation */
        args_from_stack(L, -1 ,output, output_type);

    lua_close(L);
}
```

Figure 4.1: The code of the TAs core function, `call_lua`

params[0]	(memref) input buffer containing the encrypted (or plaintext) lua script
params[1]	(value) a: Input parameter type specifying what datatype is contained in the params structure b: (Optional) numerical input argument, gets replaced by numerical return value
params[2]	(value) a: A flag to indicate if the input data is plaintext or encrypted+signed b: unused
params[3]	(memref): (Optional) A memory buffer for transmission of strings and arbitrary arguments as Lua code

Table 4.2: Parameter array for `run_script`

params[0]	(memref) input buffer containing the name of the Lua script
params[1]	(memref) input buffer containing the encrypted (or plaintext) Lua script
params[2]	(value) a: A flag to indicate if the input data is plaintext or encrypted+signed b: unused
params[3]	unused

Table 4.3: Parameter array for `save_script`

params[0]	(memref) input buffer containing the name of the Lua script
params[1]	a: Input parameter type specifying what datatype is contained in the params structure b: (Optional) numerical input argument, gets replaced by numerical return value
params[2]	unused
params[3]	(memref): (Optional) A memory buffer for transmission of strings and arbitrary arguments as Lua code

Table 4.4: Parameter array for `run_saved_script`

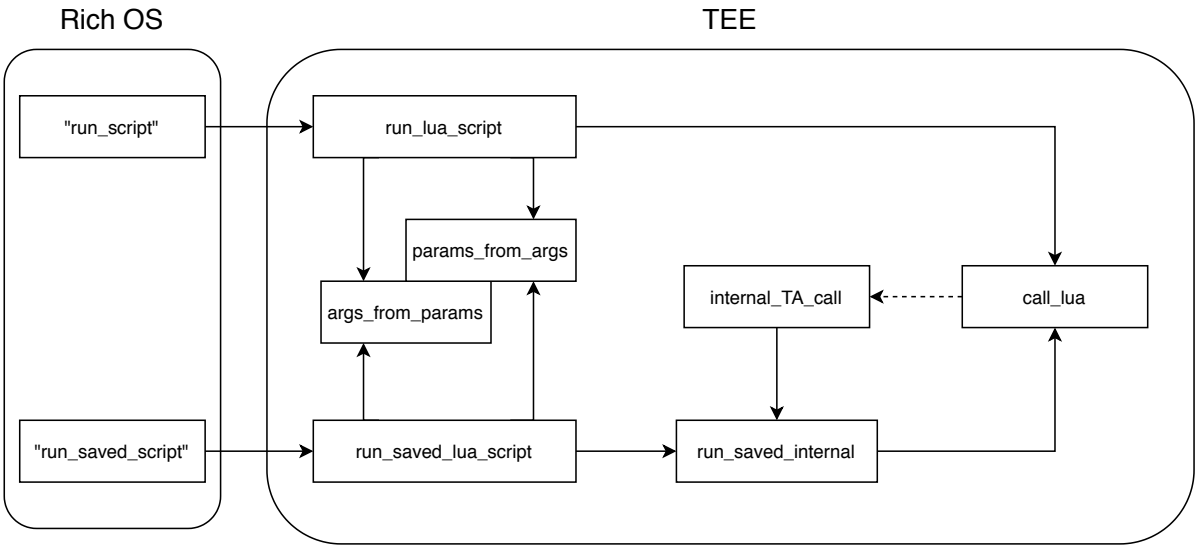

Figure 4.2: The call graph of the internal functions in our system

4.6 Call Flow in the Trusted Application

After invoking TEEC_InvokeCommand on the rich side, OPTEE will context switch context to the TEE and call TA_InvokeCommandEntryPoint, which calls one of our internal functions. Figure 4.2 shows the relationships between these functions, excluding `args_from_stack` and `stack_from_args`. We will also describe the purpose of these functions in the following paragraph.

`run_script`: The entry point invokes `run_lua_script`, which uses a helper function `args_from_params` to extract values from the TEEC_Parameter array into the `input` pointer and the `input_type` flag that `call_lua` expects. After `call_lua` concludes, it uses the helper function `params_from_args` to pass the return values of `call_lua` back into the params array so that the rich world can access them.

`run_saved_script`: The entry point invokes a function called `run_saved_lua_script`, which also uses `args_from_params` to extract arguments. These values are then passed to `run_saved_internal` along with the script name. As the functionality provided by `run_saved_internal` is also needed for internal calls, the separation of `run_saved_lua_script` and `run_saved_internal` was sensible. `run_saved_internal` loads the script from the persistent storage as detailed in Section 3.2.3, and passes the script string as well as the arguments to `call_lua`. For processing the return values, `run_saved_script` utilizes `params_from_args` in the same way as `run_script`.

`internal_TA_call`: This function is exposed to the scripts running on the interpreter. Therefore, it is directly called from the scripts and gets the current interpreter state as an input

parameter. We decided to keep the script state strictly separated and follow our design by having `internal_TA_call` invoke `run_saved_internal` to call the predeployed Lua script. As discussed in section 3.2.3, we need to serialize the Lua arguments before passing them to `run_saved_internal` and deserialize the return values. For this, `internal_TA_call` uses the functions `args_from_stack` and `stack_from_args` in a similar way to `call_lua`. After serializing the arguments, the TA calls `run_saved_internal`, which loads the script from the persistent storage and passes both the script string and the arguments to `call_lua`. If our objective were to avoid the extra effort of serializing and deserializing the data twice, sharing interpreter state between scripts or defining functions that directly transfer data from one state to another would be considerations.

We will explore the inner workings of `save_script` in the next section.

4.7 Persisting Scripts

To make deployed scripts persistent, we make use of both the secure file storage of OPTEE and memory buffers. OPTEE allows for the saving of encrypted files to the rich file system, which can only be read by the TA that created them [33]. We use this mechanism to make scripts available for repeated invocation across multiple runs of the interpreter. Whenever a script is sent in via the `save_script` interface, the TA opens a secure storage PersistentObject using the TEE_CreatePersistentObject function provided by the GlobalPlatform API. The object's name is set to be equal to the received `script_id`. The object is then populated with a byte array containing the script and written to the secure storage, overwriting any previously deployed file with the same identifier. When `run_saved_script` is invoked, the TA tries to load PersistentObject from the secure storage with the passed `script_id`. If the loading succeeded, the TA reads the script data from the object and continues to invoke `call_lua`.

Additionally, whenever a TA session is started, we create a hash table that is kept in memory. This table stores key-value pairs of the form (`script_id`, `script_string`). Whenever a script is saved in the file storage, we also store it in this table for the remainder of the TA instance's lifetime. During every invocation of `run_saved_script`, the TA performs a lookup on the provided `script_id` and only hits the secure storage if this lookup was unsuccessful. We will discuss the performance implications of this in Section 5.2.3. A worthwhile addition to the system would be the ability to preemptively load scripts from the storage into the memory buffer on the TA startup. This feature was excluded as this memory system's main purpose was to test performance differences between memory and storage.

4.8 Authenticating and Encrypting Lua Scripts

This section aims to describe the actual algorithms used in implementing the Encrypt-then-MAC scheme discussed in Section 3.2.4. Our goal was to enable TAs to check the source of a trusted script and provide a mechanism to protect the Lua script's content. As this was

built as part of a proof of concept implementation, we do not claim that the combination of algorithms provides security against all attacks. We strongly advise evaluating the security needs of a given project when adopting the presented methodologies.

In our simplified setup, the "source" of scripts is a python program responsible for encrypting and signing the Lua scripts that will be executed in the TEE. To perform the cryptographic algorithms in the python script, we use the PyCryptodome [34] package. On the trusted side, the TA has access to the OPTEE implementations of the algorithms, as they are specified in the GlobalPlatform Internal TEE API [11].

Our shared secret is a 256-bit key that is hardcoded into both the python script and the TAs source code. For the generation of the two keys necessary for encryption and MAC generation, we employ an HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [35]. While not included in the GlobalPlatform API, OPTEE offers an extension that enables HKDF by adding a new operation type to the TEE_DeriveKey API function [36]. Alongside the shared secret, we also provide a random, non-secret 128-bit salt for the HKDF extraction.

Encryption and signing. We use the first of the two resulting keys to encrypt the provided Lua script symmetrically according to the Advanced Encryption Standard (AES) Cipher Algorithm [37]. We specifically cipher the data using counter mode, using a random, non-secret 64-bit initialization vector or "nonce". We then concatenate the nonce and the ciphertext according to the Encrypt-then-MAC procedure. The result is used together with the second generated key to generate a MAC using a SHA512 checksum [38]. We then append the salt, MAC, nonce, and ciphertext to get the byte array that represents our secured script. To summarize, the Python script takes in regular .lua files and generates .luata files, which consist of data organized in the following format:

[Salt (16 Bytes)][MAC (64 Bytes)][Nonce (8 Byte)][AES encrypted Lua script]

Signature checking and decryption. This process is the same for both `run_script` and `save_script`. After HKDF is used to generate the two keys from the shared secret and transmitted salt, the TA generates the SHA512 HMAC over the nonce and ciphertext. If the result matches the transmitted HMAC, the application proceeds to decrypt the ciphertext using the second key. The resulting plaintext is the Lua script that can then be passed on to `call_lua` or saved to the secure storage.

4.9 Rich World Application

This section will cover the implementation of our system on the untrusted side. We also used the C language in this part of the application, as OPTEE implements C interfaces to interact with trusted applications, like they are defined in the TEE Client API Specification [12].

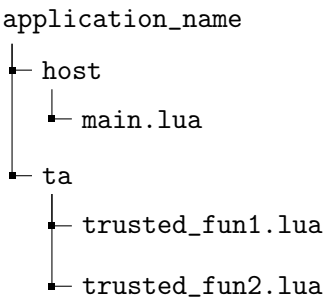

Figure 4.3: The file structure for applications using both the trusted and untrusted side

To run scripts on the untrusted side, we need a language interpreter. For simplicity, we decided to use the Lua scripting language on the rich side as well. As the libc of OPTEE on the rich side has similar limitations as on the trusted side, certain functionality would have to be reimplemented. By choosing Lua, we could reuse the same modified interpreter that was employed on the trusted side. As a result, we defined a unified "Lua" module that we included with both applications during the build process. Using Lua also meant that we could use the same (de)serialization methods for arguments in both worlds, further reducing implementation effort. While the usage of two distinct scripting languages is possible by design, our experience shows that using the same scripting language in both worlds results in an easier development process.

The next paragraph will go over each significant step that happens during the rich world application's execution.

We invoke the rich interpreter via the command line, passing a folder with the structure shown in Figure 4.3 as an argument that contains both the secure and non-secure scripts. We can specify two optional arguments -s and -u. -s tells the interpreter to prefer `save_script` when calling TAs, as the interpreter sends in the scripts using `run_script` on every call by default. -u tells the interpreter to use unencrypted .lua files for TA scripts instead of our custom .luata format. This argument was used for testing the encryption feature and should be disabled in real-world deployments.

After being started, the application opens a session with the trusted interpreter. It then proceeds to load all the trusted scripts from the `/ta` folder and passes them to the `save_script` interface. Even if we decide to send in the scripts with every invocation to a trusted function, we still need to ensure that internal calling between scripts is functional. Thus, preemptively sending in all scripts to the TA is required. Next, a Lua state is set up in the same way as described in `call_lua`. We bind the C function `TA_call` to the interpreter, which enables scripts to invoke TA scripts. The main.lua from the `/host` folder is then loaded in and executed. Whenever the main script calls a TA function, the interpreter serializes the arguments and invokes either the `run_script` or `run_saved_script` interface, depending on the presence of the -s flag. When the call to the TA returns, the return value is deserialized and put on

the main script's stack. After the execution of main.lua concludes the session with the TA is closed, and the program exits.

By exposing our TA's interfaces to scripts running on the untrusted side, our implementation now allows us to develop applications that use both sides in a single scripting language. If we wanted to add the ability for scripts in the rich world to call regular TAs as well, we could achieve this by adding more C bindings similar to TA_call.

4.10 Summary

We chose OPTEE for the TEE implementation because it is open source and well documented. Because Lua offers a great C API and an implementation of the Lua interpreter that could run on OPTEE already existed, we decided to use Lua as our system's target language. We documented the changes made to the interpreter while porting, most notable were the reimplementations of functionality normally found in the unavailable (in OPTEE) math.h and string.h. We proposed mappings in the tables 4.2, 4.3, and 4.4 to link our interfaces from the system design section to the arguments passable via TEEC_InvokeCommand. Additionally, we visualized the way data is passed through our internal functions in Figure 4.2. Using the internal functions offered by OPTEE enabled us to make scripts persist in the secure storage and implement cryptographic measures to decrypt .luata scripts and check their authenticity. For the untrusted side of our implementation, we reused the Lua interpreter from the trusted side. With the rich world interpreter included, our implementation has shown that it is possible to compose applications using the TEE entirely from scripts targeting language interpreters. In the next chapter, we want to show that this approach is viable by going over basic performance measures and an example application.

5 Evaluation

This chapter will assess the proposed system's practicability by detailing the performance tests we conducted on the reference implementation. Furthermore, it aims to give a better understanding of the tradeoffs between different approaches discussed in the design section. The presented results are primarily intended to be compared among themselves and should not be seen as indications of the absolute performance of C or Lua in general.

5.1 Experiment Setup

We conducted our experiments on a Raspberry Pi 3 Model B with Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB of RAM. The setup was running the OP-TEE version 3.8.0, and the platform=rpi flag was set when building OPTEE from source.

For performance measurements, execution times are given in system time (determined using the gettimeofday [39] syscall), meaning they specify the amount of time passed based on a real-world clock. We also measured the elapsed CPU time during our experiments but found no significant difference compared to system time.

To make our results more reliable, we repeated every trial 50 times and averaged the results. We found this to be sufficient across all performance tests as the variance in our data points was generally statistically insignificant.

All of the time-values were measured on the rich OS side. We took one timestamp right before a call to the TA was made, and one directly after the function returned. We did not perform any time measurements inside the trusted environment because the standard library's time measuring functionality was not available in the TEE. As our goal was to show the system's general practicability, we found this to be sufficient, as these durations are what a user of the system will care about in the end. OPTEE does, however, offer a more sophisticated framework for benchmarking and profiling the performance of TEEs [40]. We recommend using this framework for a more detailed understanding of what parts have the most significant impact on the performance.

5.2 Input and Storage

This section evaluates the mechanics for sending scripts into the TEE and storing/loading them on the trusted side. We aim to answer the following questions:

- How long does it take for scripts to be ready for execution when sending them into the TEE?
- How long does it take to put a script into the TEE persistently?
- How does the execution time differ for the three methods of loading scripts: sending them in, loading them from memory in the TEE, loading them from the secure storage?
- How much overhead does our encryption schema introduce for loading in the scripts?

5.2.1 Loading Scripts into the Interpreter

To get a general idea of how much time it takes for scripts to reach the interpreter in the TA, we sent in three Lua scripts of differing sizes. To isolate the time it took to load in the files, we ran a slightly modified version of our TA. Once the TA reached the point where the scripts would be loaded into the interpreter, it instantly returned to the rich world, where a timestamp would be taken.

Looking at the summarized data in Figure 5.1, we can determine that the constant overhead for loading in a script is 0.42 ms in the average case. Furthermore, the data points hint at a linear correlation between the file size and input time, which would be expected. However, to get the exact relation between file size and loading time, more measurements would need to be taken.

# of Lines	Size	Average Elapsed Time (ms)	Standard Deviation
1	14 Bytes	0.42	0.039
10000	97.67 KiB	2.17	0.084
100000	976 KiB	19.1	0.358

Table 5.1: Loading times in relation to file size

After being sent into the TEE, a script needs to be loaded into the Lua interpreter via the `load_buffer` function before execution. To see how much overhead this process introduces, we extended our modified TA from the last experiment: It now also loads the input scripts into the interpreter state and then returns to the rich world. To see if there are performance differences for varying statement types, we sent in three different scripts. All three consisted of 10.000 lines of code (LoC). One file repeated a statement that incremented a variable, one was made up of code comments, and the last one contained empty do end blocks. (do end creates a separate variable scope in Lua. This experiment was chosen to measure the effect of control flow statements on the loader.) The examined scripts were close in file sizes.

Figure 5.2 shows that even though the line count and filesize were similar across all three files, loading times differed significantly. These results indicate that the overhead introduced

Line Type	Average Elapsed Time (ms)	Standard Deviation
Comments	4.56	0.072
Controlflow	21.22	0.43
Increment	72.64	0.340

Table 5.2: Loading times including load_buffer

by `load_buffer` is highly dependant on the type of lines, with more complex expressions likely introducing an even higher overhead. Furthermore, it is noteworthy to compare the processing time of the 10.000 increment script with the values in the previously discussed Table 5.1. We see that the time to send the file into the TEE is insignificant compared to the time it takes to load the script into the interpreter.

We conclude that optimizing scripts for `load_buffer` will result in more significant performance gains than reducing file sizes. While the Lua script loader's internal workings are out of the scope for this work, we can state that changing the number of comments and control flow statements will have little impact on a script's loading time in our system.

In terms of usability, it is up to the target application to determine if loading times like these are acceptable. However, we do expect scripts used in real-world applications to be significantly smaller than the 10.000 lines of code we used in our experiments.

5.2.2 Storing Scripts Persistently

For building persistent APIs, it is not only important to assess how long it takes for scripts to arrive in the environment, but also how much time it takes to save them to our persistent data structures. To measure this time, we ran the TA `save_script` function with the small and medium-sized scripts used in the previous experiments as inputs (due to memory limitations of our TA setup, we were unable to evaluate this functionality with the 100.000 LoC script).

Besides assessing the differences caused by filesize, we also wanted to specify the performance difference between our two proposed methods of persisting scripts: keeping them in memory with a dictionary or storing them in the trusted storage.

Table 5.1 shows the results of our experiment. As expected, saving the scripts in the trusted storage takes significantly longer than storing them in the dictionary kept in memory (278x for the small script and 70x for the medium-sized one). Note that there are only a limited number of cases where one could utilize this performance. If scripts should persist across multiple TA runs and instances, they eventually need to be saved to the trusted storage.

5.2.3 Sending in vs. Memory vs. Storage

In the previous paragraph, we examined the performance that can be expected when storing scripts persistently. Arguably, the time it takes to access the stored scripts is even more critical

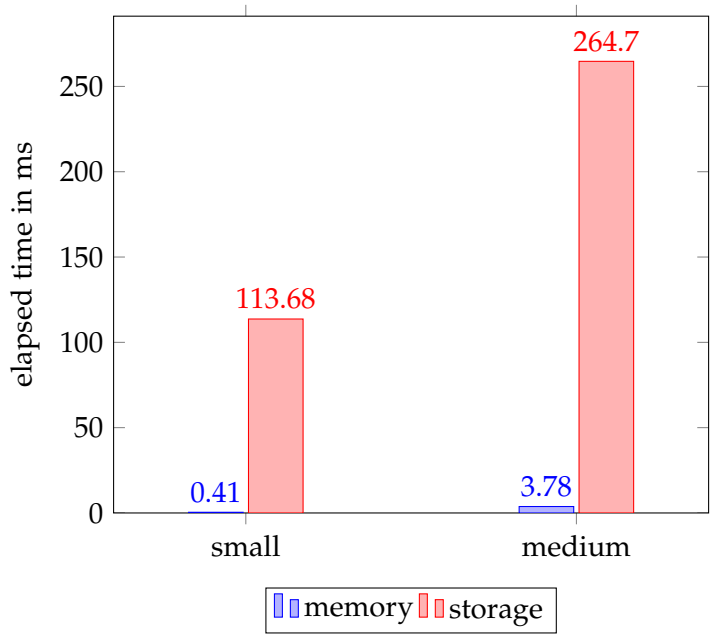

Figure 5.1: Elapsed time to store scripts persistently, small (1 LoC) and medium (10.000 LoC)

for the system's operation. Thus, we compared the time it took to load scripts in the following ways:

- Passing scripts in with the TA call
- Loading scripts from memory
- Loading scripts from the storage as the first storage access
- Loading scripts from the storage as a consecutive storage access

We witnessed a significant performance difference when comparing the first time a TA accessed the secure storage compared to the subsequent reads. Thus, we measured these cases separately. We again used the small and medium-sized Lua scripts and returned to the rich world before the script would be loaded into the interpreter. Figure 5.2 compares the four cases listed above in terms of elapsed time.

Similar to our results for storing scripts, it can be seen that accessing the storage is significantly slower than accessing the scripts in memory (81x for the small script and 42x for the medium-sized one when looking at the first access). For implementations, this means that if a script is to be invoked multiple times, it could be beneficial to load it from the storage into memory at the startup of the TA.

It makes sense that the measured times for sending in the script and loading it from the data structure in memory are similar. Both methods are reading the string directly from memory

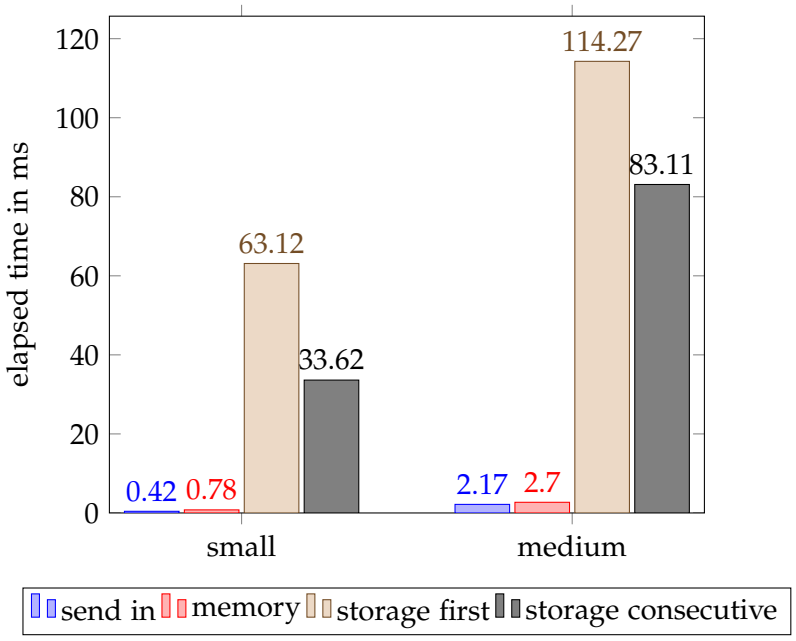

Figure 5.2: Elapsed time to load persistent scripts, small (1 LoC) and medium (10.000 LoC)

available to the TA, with the persistent memory access adding overhead for performing a lookup to find the script. Thus, if we only look at the performance inside the TA, sending in scripts with every call to the TA is the most performant option of getting scripts to the interpreter.

We do not have practical implications for the difference between the first and consecutive storage accesses but reserve the exploration of this behavior for future work.

5.2.4 Encryption Overhead

Lastly, we were interested in how much overhead our Encrypt-then-MAC schema introduced for loading in scripts. The used scripts and TA setup were the same as the experiment of Section 5.2.2. Figure 5.3 compares the loading times for plaintext scripts and ones that were sent in encrypted.
We can see that checking the signature and decrypting the files before loading them adds a significant waiting time before the scripts are ready to be run. Our current setup does not support hardware acceleration for the employed security algorithms, which is one reason for the lower performance. Inefficiencies in our usage of the OPTEE APIs could be another plausible explanation for the slowdown, as optimization was not a priority in the reference implementation. A more in-depth review of the security features provided by OPTEE would give us a better understanding of what causes the slowdown.

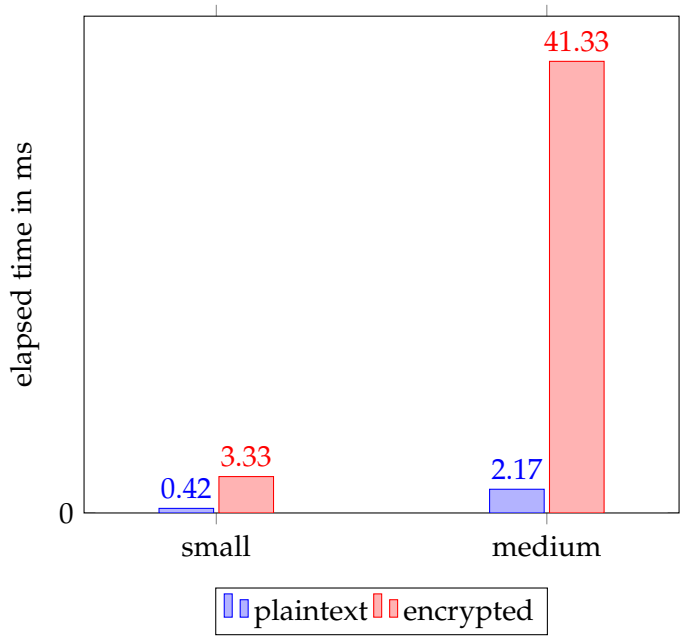

Figure 5.3: Loading time difference for plaintext scripts and encrypted ones, small (14 Byte) and medium (97.67 KiB)

5.3 Performance Measures

This section aims to overview the performance tradeoffs that we make by running Lua scripts on our system over native TAs written in C. We will look at the following three scenarios to compare performance:

- Execution in the TEE
- Execution on the rich OS of OPTEE
- Execution on an unmodified Raspberry Pi OS

The first scenario aims to capture the system's general performance, while the latter two serve to provide references that can be compared against these results. We were primarily interested in the relationship between execution time and the number of executed instructions, as well as the performance difference between Lua and C. Both the Lua script and the C program consist of a single for-loop that performs an increment operation on a variable with every iteration. We measured the execution times of both the Lua script and the C program from 0 up to 1.000.000 loop iterations with a step size of 10.000. To prevent the C compiler from optimizing the loop, we put an asm("") statement in the loop body [41]. This statement has no effect other than preventing loop unrolling.

We chose this test program over more complex benchmarks as the modified interpreter does not offer all of Lua's functionality yet. Additionally, using a small program in size prevents

loading times (discussed in Section 5.2) from heavily influencing the result. Even though loading time will have to be considered for the system's operation in the real world, these experiments aim to compare the pure execution time as closely as possible.

5.3.1 Performance: OPTEE Trusted Execution Environment

For measuring the performance in the TEE, the C program was realized as a TA function that would run as soon as the TA was invoked. Furthermore, we passed the Lua script to the TA with each call to the `run_script` function to minimize the loading overhead.

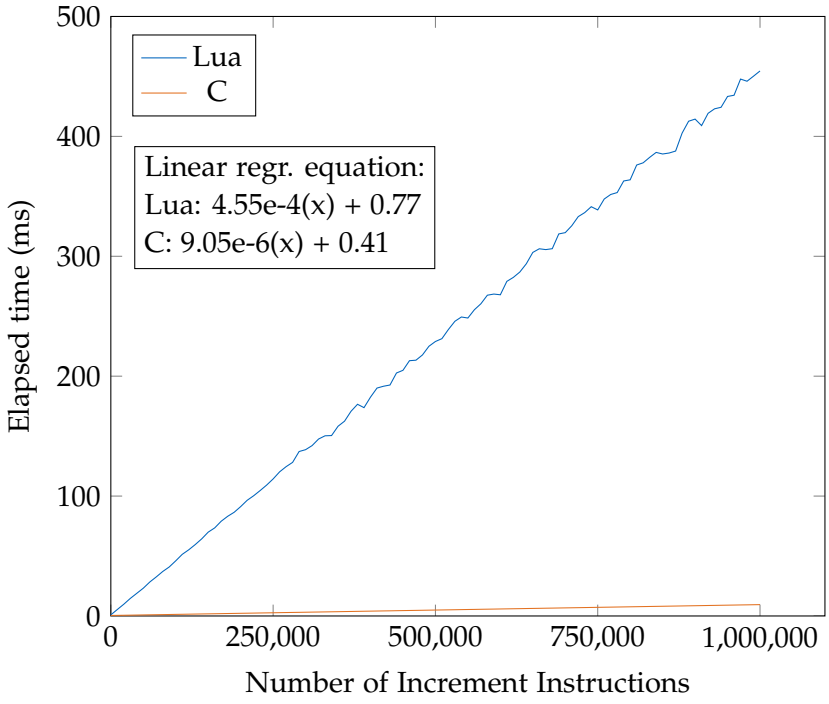

Figure 5.4: Performance tests conducted on the trusted side

Figure 5.4 shows the performance of the two tested programs in the TEE. We can determine that the initial overhead that our interpreter introduces compared to C is 0.916ms on average, as that is the difference in the case where zero increments happen. After that, the execution times increase linearly with the number of loop iterations for both the Lua script and C program. The execution time of the Lua script is, on average, 43.2x higher than the time it takes to run the C code.

5.3.2 Performance: OPTEE Rich World

To see if the execution in the TEE introduces additional overhead, we repeated the comparison on the untrusted side. As discussed in the implementation section, we used the same modified

version of the Lua interpreter on both the trusted and untrusted side. Looking at Figure 5.5, we can see that the difference in performance between C and our Lua interpreter (58.4x on average) is comparable to the experiment conducted on the trusted side. This finding indicates that running our code in the trusted environment does not amplify the performance difference between Lua and C. However, both Lua and C suffer a performance decrease of around 2x (1.82x for C, 2.41x for Lua) compared to the trusted side. At the time of writing, we can provide no plausible explanation for this phenomenon, which is why the exact cause of the performance difference is the subject of further research.

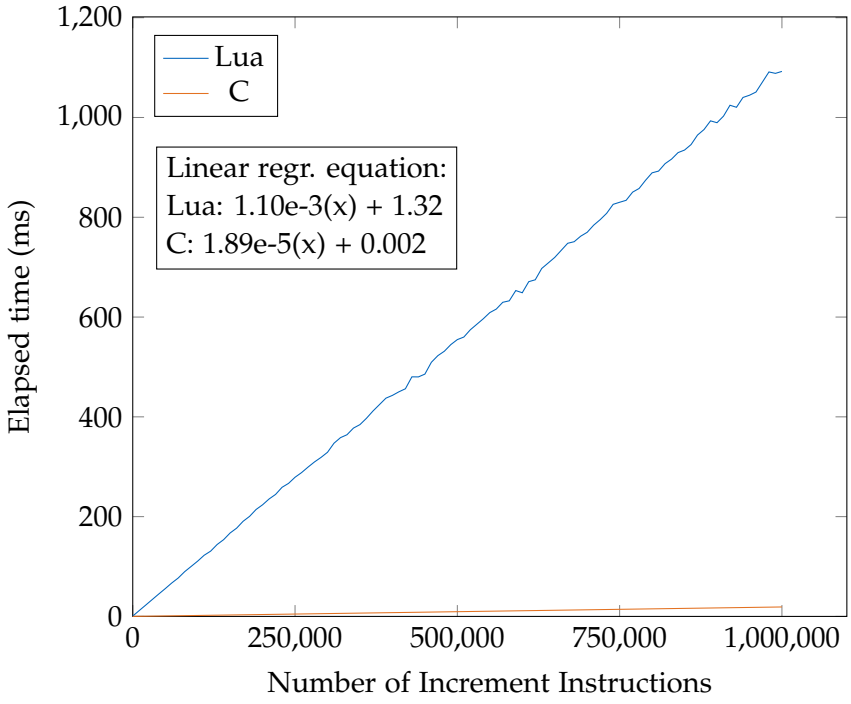

Figure 5.5: Performance tests conducted on the untrusted side

5.3.3 Performance: Raspberry Pi OS

To get an additional reference point as to how Lua compares to C in performance, we also ran the two programs on Raspberry Pi OS (Version 5.4) [42]. To execute the Lua scripts, we used an unmodified Lua Interpreter (Version 5.3.3) [43]. The results of this experiment can be seen in Figure 5.6. While the execution times for the C program are similar to the ones measured on OPTEE, the Lua interpreter is, on average, 2.2x times faster than in the previous experiment.

This result shows us that running our modified Lua interpreter on OPTEE further increases the overhead Lua scripts usually have compared C. However, determining the exact cause of this increase is not trivial. One possible source of the slowdown could be the modifications

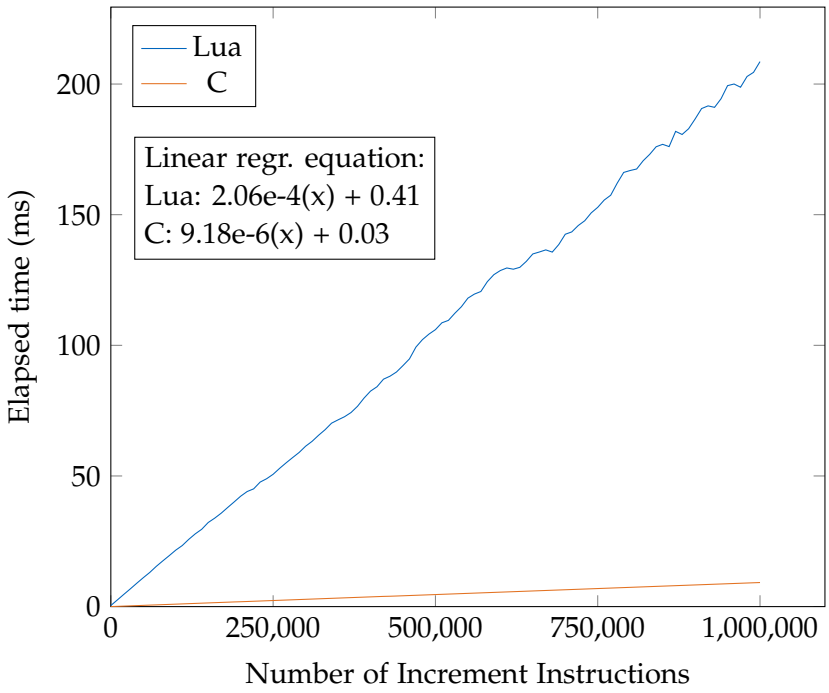

Figure 5.6: Performance tests conducted on Raspberry Pi OS

made to the Lua interpreter. Similarily, OPTEE's implementation of standard C functions used by the interpreter could be less performant than the ones provided by Raspberry Pi OS.

The modified interpreter uses OPTEE specific library functions and cannot operate on Raspberry Pi OS. In turn, the regular Lua interpreter does not run on OPTEE due to the reduced libc. Thus, we could not test the two variables "interpreter runs on OPTEE" and "script runs on modified interpreter" independently. A more fine-grained analysis of both OPTEE and the Lua interpreter's modifications would be necessary to get better insights into the exact cause of the amplified overhead.

5.3.4 General Observations and Summary

Our experiments showcased the expected performance overhead when using our system over regular TAs. We further demonstrated that the overhead falls in the same order of magnitude as the performance difference between Lua and C on non-OPTEE machines.

For further comparison, in experiments testing the LuaJIT, it was found that C is about 75 times faster than the plain Lua interpreter for benchmarks involving mathematical operations [44]. While the performance difference also falls in the same order of magnitude as our observations, we could not replicate math-based benchmarks, as our modified interpreter was still missing this functionality. Implementing these mathematical operators and running such benchmarks for better comparability constitutes crucial future work.

It depends on the real-world use case if the overhead that our system introduces compared to regular TAs written in C is tolerable. Note that in real-world scenarios, singular Lua scripts will likely perform a much smaller number of operations than in our experiments. In such cases, the performance difference in comparison to regular TAs will not be as critical. Furthermore, we want to highlight that Lua's worse performance can be circumvented for frequently used functionality in the TEE. As the Lua interpreter allows developers to expose C functions to Lua scripts, the TA can be modified to include C bindings for operations where performance is important.

5.4 Example Application

In the following section, we will look at a simple application that is meant to present an example use case of our system and show how the Lua scripts interact in practice.

Assume an application on the rich os side is running a database and authenticates users via passwords. The database stores its password hashes using the MD5 message-digest Algorithm (MD5) [45]. We chose this algorithm to construct a simple Lua program that demonstrates the system's capabilities. The actual security it provides in the case of password leaks was not a focus of this example. Say the developers of the application wants to delegate the verification of the password to the TEE. For this, they write two scripts similar to the ones shown in the figures 5.7 and 5.8. The trusted script is responsible for the generation of the hash and the comparison with the database value. The developers know that the system already has a deployed Lua script that generates MD5 hashes, so they can use the functionality by utilizing the internal calling interface. To replicate this for our example, we reused an open-source implementation of the MD5 algorithm written in pure Lua [46]. (We deliberately chose to use a premade open-source script for this. Even though our TA does not offer all the Lua interpreter's functionality in its current state, this example shows that it already manages to run decently complex Lua scripts.)

Whenever the application verifies a password hash, it makes a call to the TA and passes the script, the entered password, and the hash into the TEE. On the trusted side, the script calls the predeployed MD5 script on the entered password, causing the TA to load the script from the internal storage and execute it. After the hash is generated, the trusted script compares the generated hash to the one provided by the database. It then returns a flag to the script running on the rich side, indicating whether the hashes matched or not.

On average, running this example application took 892.9 ms in total (sd: 13.01 ms). Of this time, 347.8 ms were spent executing scripts in the TEE (sd: 0.582 ms). While the execution time might seem high for this particular operation, it should be kept in mind that this was run on lower-end hardware, we did not optimize the MD5 script for performance, and we included the time to load all the scripts from the rich storage. Additionally, the two trusted scripts were also deployed via the `save_script` interface first instead of being already present

in the TEE. We predict that specialized real-world setups with pre-deployed scripts can achieve much greater performance.

```
-- Lua script running in the Rich OS

entered_password = "connectedmobility"
stored_hash = "bb96d9aa8db126749770da804eb1076e"

arg = entered_password..","..stored_hash

if TA_call("password_match", arg) == 1 then
    --print("Success") -- continue to authenticate user
else
    --print("Invalid password") -- abort
end

return 0
```

Figure 5.7: The Lua script running on the untrusted side.

```
-- Lua script running in the Trusted Execution Environment
arg = ...
-- Extract the password and hash values form the input string
pwd, hash = arg:match("([^,]+),([^,]+)")
if hash == internal_TA_call("md5", pwd) then
    return 1
end
return 0
```

Figure 5.8: The Lua script running in the TEE

6 Conclusion

In this thesis, we surveyed the usage of language interpreters to facilitate the dynamic loading of programs in TEEs. We first established that using scripts as TAs had the advantages of being platform-independent and portable. We then proposed a TA that enables the execution of scripts in a TEE by embedding a language interpreter. In the design section, we further discussed how to deal with passing arguments to the script, how the interpreter state is handled, and how to enable the persistent deployment of scripts to the TEE. Lastly, we complemented the TA with an interpreter on the non trusted side, so that scripts could consume the functions offered by the scripts running in the TEE.

Our implementation demonstrated that running interpreters inside of TEEs is possible on Arm TrustZone. While doing so, we also showcased a combination of technologies (Lua, OPTEE) that could be used to build a system like the one proposed. Additionally, our discussion of the implementation highlighted design challenges (interpreter modifications, mapping arguments to TEE Params) and presented patterns (reuse of internal functions) that generally apply to implementations of such a system.

Our evaluation examined various baseline properties of our system and provided performance estimates for our core operations. We found that the performance decrease of our system compared to native TAs falls in the same order of magnitude as slowdown typically found in comparisons of interpreted languages and C. Finally, we demonstrated our system's ability to host semi-complex programs by running an example application that included a pure Lua implementation of the MD5 hashing algorithm.

Looking at both our system's properties and the evaluation, we conclude that the utilization of language interpreters offers more flexibility than using regular TAs, but suffers a performance overhead. This overhead is potentially higher than the expected slowdown introduced when one uses scripting languages instead of C. With further research, we hope that this performance gap can be better understood or even significantly reduced. It will be interesting to see what kind of applications can make use of the tradeoff that we introduced.

Alongside this work and the concepts presented in it, we also release our implementation as an open-source repository. We hope that these artifacts provide a base for further research in the area of language interpreters running in trusted execution environments.

6.1 Future Work

The scope of this thesis was limited. The main goal of our system design was to describe a small set of features required to fulfill the goals laid out in the design section, our implementation served mainly as a proof of concept, and the evaluation section primarily focused on the fundamental properties of the system. Logically, there are various ways in which future work on the topic could be conducted:

- Completing the port of the modified Lua interpreter to include all functions of regular Lua. Having all Lua functions available would allow us to run common benchmarks on our interpreter to get a better idea as to how the performance compares to regular TAs.
- Extending our system design and TA implementation with functions useful for operation, such as the ability to delete deployed scripts and preload persistent scripts from storage to memory. Evaluating the latter feature could give insight into what optimizations would work well for building persistent APIs from scripts.
- Finding more use cases for a system like ours could yield insights into useful features and possible limitations that were not considered in this work.
- A closer survey of platforms other than OPTEE and Arm TrustZone could help us categorize what concepts presented in this work apply across all architectures and which are results from the technologies used. Similarly, looking at other scripting languages for implementations might reveal some characteristics that would make them especially suitable for a system like ours.
- A more thorough evaluation of our reference implementation using the OPTEE benchmarking framework would help us sort out the open questions raised in the evaluation section. Furthermore, this kind of analysis would help profile our system, find the performance bottlenecks, and help with optimizations to further close the performance gap to regular TAs.

List of Figures

List of Tables

Bibliography

[1] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[2] S. P. Bayerl, F. Brasser, C. Busch, T. Frassetto, P. Jauernig, J. Kolberg, A. Nautsch, K. Riedhammer, A.-R. Sadeghi, T. Schneider, E. Stapf, A. Treiber, and C. Weinert. "Privacy-preserving speech processing via STPC and TEEs". In: *PPML 2019, Privacy Preserving Machine Learning Workshop, CCS 2019 Workshop, November 15, 2019, London, UK*. London, UNITED KINGDOM, Nov. 2019. URL: http://www.eurecom.fr/publication/6098.

[3] T. Lee, J. Song, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, and Z. Lintao. "Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX". In: May 2019, pp. 1–17. ISBN: 978-1-4503-6169-9. DOI: 10.1145/3300061.3345447.

[4] H. Wang, E. Bauman, V. Karande, Z. Lin, Y. Cheng, and Y. Zhang. "Running Language Interpreters Inside SGX: A Lightweight,Legacy-Compatible Script Code Hardening Approach". In: July 2019, pp. 114–121. DOI: 10.1145/3321705.3329848.

[5] F. Guerin, T. Kärkkäinen, and J. Ott. "Towards a Programmable World: Lua-based Dynamic Local Orchestration of Networked Microcontrollers". In: Oct. 2019, pp. 13–18. ISBN: 978-1-4503-6933-6. DOI: 10.1145/3349625.3355441.

[6] N. Langley. *Write once, run anywhere?* https://www.computerweekly.com/feature/Write-once-run-anywhere. Accessed: 2020-10-10.

[7] W. C. Luiz Henrique de Figueiredo Roberto Ierusalimschy. *Lua: an Extensible Embedded Language*. https://www.drdobbs.com/open-source/lua-an-extensible-embedded-language/184410014. Accessed: 2020-10-10.

[8] R. Ierusalimschy. "Part IV · The C API". In: *Programming in Lua*. Lua.org, 2003.

[9] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. 2015, pp. 57–64.

[10] GlobalPlatform. *GlobalPlatform*. https://globalplatform.org/. Accessed: 2020-10-10.

[11] GlobalPlatform, Inc. *GlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50 (Target v1.2)*. https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf. Accessed: 2020-10-5.

[12] GlobalPlatform, Inc. *TEE Client API Specification v1.0 | GPD_SPE_007*. https://globalplatform.org/specs-library/tee-client-api-specification/. Accessed: 2020-10-5.

[13] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan. "Open-TEE - An Open Virtual Trusted Execution Environment". In: (June 2015).

[14] S. Pinto and N. 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.

[15] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares. "Lightweight multicore virtualization architecture exploiting ARM TrustZone". In: Oct. 2017, pp. 3562–3567. DOI: 10.1109/IECON.2017.8216603.

[16] P. Mehrotra. "Cross-device Access Control with Trusted Capsules". MA thesis. The University of British Columbia, 2019, p. 74.

[17] Arm. *Arm TrustZone Technology*. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed: 2020-10-5.

[18] Linaro. *Open Portable Trusted Execution Environment*. https://www.op-tee.org/. Accessed: 2020-10-5.

[19] TrustedFirmware.org. *OPTEE Documentation: Supported Platforms*. https://optee.readthedocs.io/en/latest/general/platforms.html. Even though this document is from the OPTEE documentation, it mentions the existence of differences in the platforms that support TEEs in general.

[20] TrustedFirmware.org. *OPTEE Documentation: Signing of TAs*. https://optee.readthedocs.io/en/latest/building/trusted_applications.html#signing-of-tas. Accessed: 2020-10-5.

[21] J. Wiklander. *Github comment specifying that there are no plans to provide a complete libc for TAs in OPTEE*. https://github.com/OP-TEE/optee_os/issues/2411#issuecomment-399011984. Accessed: 2020-10-5.

[22] Python Software Foundation. *Python/C API Reference Manual*. https://docs.python.org/3/c-api/index.html. Accessed: 2020-10-5.

[23] GlobalPlatform, Inc. *2.1.6 Instance in TypesGlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50 (Target v1.2)*. https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf. Accessed: 2020-10-5.

[24] GlobalPlatform, Inc. *2.2.2 Trusted Storage API for Data and Keys in TypesGlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50 (Target v1.2)*. https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf. Accessed: 2020-10-5.

[25] R. S. P. Leach M. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. RFC Editor, July 2005. URL: https://tools.ietf.org/rfc/rfc4122.txt.

[26] GlobalPlatform, Inc. *3.2.4 TEE_UUID, TEEC_UUID in TypesGlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50 (Target v1.2)*. https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf. Accessed: 2020-10-5.

[27] M. Bellare and C. Namprempre. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm". In: vol. 21. Dec. 2000, pp. 531–545. DOI: 10.1007/3-540-44448-3_41.

[28] TrustedFirmware.org. *Raspberry Pi 3*. https://optee.readthedocs.io/en/latest/building/devices/rpi3.html. Accessed: 2020-10-10.

[29] R. Ierusalimschy. *Lua is fast*. https://www.lua.org/about.html. Accessed: 2020-10-5.

[30] TrustedFirmware.org. *Q: Is multi-threading supported in a TA?* https://optee.readthedocs.io/en/latest/faq/faq.html#q-is-multi-threading-supported-in-a-tal. Accessed: 2020-10-5.

[31] W. C. Roberto Ierusalimschy Luiz Henrique de Figueiredo. *Lua 5.1 Reference Manual*. http://www.lua.org/manual/5.1/manual.html#lua_load. Accessed: 2020-10-5.

[32] P. Rapin. *Data Dumper*. http://lua-users.org/wiki/DataDumper. Accessed: 2020-10-5.

[33] TrustedFirmware.org. *OPTEE Documentation: Secure storage*. https://optee.readthedocs.io/en/latest/architecture/secure_storage.html. Accessed: 2020-10-5.

[34] H. Eijs. *PyCryptodome's documentation*. https://pycryptodome.readthedocs.io/en/latest/. Accessed: 2020-10-5.

[35] P. E. H. Krawczyk. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. RFC Editor, May 2010. URL: https://tools.ietf.org/rfc/rfc5869.txt.

[36] TrustedFirmware.org. *OPTEE Documentation: GlobalPlatform API, HKDF*. https://optee.readthedocs.io/en/latest/architecture/globalplatform_api.html#hkdf. Accessed: 2020-10-5.

[37] K. M. U. Blumenthal F. Maino. *The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model*. RFC 3826. RFC Editor, June 2004. URL: https://tools.ietf.org/rfc/rfc3826.txt.

[38] T. H. D. Eastlake 3rd. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. RFC 4634. RFC Editor, July 2006. URL: https://tools.ietf.org/rfc/rfc4634.txt.

[39] D. Eckhardt. *gettimeofday(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/gettimeofday.2.html. Accessed: 2020-10-5.

[40] TrustedFirmware.org. *OPTEE Documentation: Benchmark framework*. https://optee.readthedocs.io/en/latest/debug/benchmark.html. Accessed: 2020-10-5.

[41] D. S. Maia. *How to prevent GCC from optimizing out a busy wait loop?* https://stackoverflow.com/questions/7083482/how-to-prevent-gcc-from-optimizing-out-a-busy-wait-loop. Accessed: 2020-10-5.

[42] RASPBERRY PI FOUNDATION. *Raspberry Pi OS (previously called Raspbian)*. https://www.raspberrypi.org/downloads/raspberry-pi-os/. Accessed: 2020-10-5.

[43] lua.org. *Lua.org Download area*. https://www.lua.org/ftp/. Accessed: 2020-10-5.

[44] E. Klausmeier. *Performance Comparison C vs. Lua vs. LuaJIT vs. Java*. https://eklausmeier.wordpress.com/2016/04/05/performance-comparison-c-vs-lua-vs-luajit-vs-java/. Accessed: 2020-10-5.

[45] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. RFC Editor, Apr. 1992. URL: https://www.ietf.org/rfc/rfc1321.txt.

[46] A. Baldwin. *Pure lua MD5 Implementation*. https://gist.github.com/evilpacket/3647908. Accessed: 2020-10-5.