TrustFrame, a Software Development Framework for TrustZone-enabled Hardware

João Rocheteau Ramos
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: joao.silva.ramos@tecnico.ulisboa.pt

***Abstract*—With the continuous evolution of technology fields like mobile, embedded systems and ubiquitous computing, the way we interact with several types of devices is ever changing. Today, applications with sensitive data are increasingly used in mobile devices by users, which brings huge security issues. In an attempt to overcome the growing security concerns, some hardware manufacturers are starting to use ARM TrustZone technology. This technology presents itself with enormous appliance potential, since it allows the development of more robust operating systems, achieving better application security. However, research on this matter is quite complicated, due to various incompatible software and hardware solutions, as well as the lack of documentation and support to software development for TrustZone-enabled hardware. Development initiation on any TrustZone-based solution has many barriers: framework selection, choosing compatible hardware, initial environment configuration, programming APIs study and start of development. In this work we study the current state of the art in TrustZone-based solutions. This work paves the way for the development of a complete framework (documentation, development environment and compatibility support) to ease the bootstrap of TrustZone-based software development projects.**

***Keywords*—Mobile Devices, Security, Trusted Computing, ARM TrustZone, GlobalPlatform.**

I. Introduction

Mobile devices handle data that is becoming increasingly valuable and confidential. This data ranges from simple photos or files to our own medical information, banking credentials or even other access credentials, making them an increasingly sought target for a variety of attacks, as reported in [1]. Furthermore, with the popularity of Internet of Things (IoT) rising, more and more devices are expected to be connected, exchanging these types of data. Therefore it is crucial to have mechanisms capable of protecting such data.

Nowadays, the processing of sensitive data is increasingly dependent on highly complex software, with many lines of code. Not only is this true for traditional computational platforms, like desktops and servers, but also for smartphones. Also, by depending on many lines of code, the operating systems may contain vulnerabilities that can be exploited by malware and possibly result in the theft or repudiation of data.

To develop secure and trusted software, new mobile platforms processors have been created that allow the execution of sensitive code, in a secure way, in particular ARM TrustZone [2] technology, present in the most recent ARM processors. It aims at enabling the creation of an execution environment, for protecting the confidentiality and integrity of critical code, allowing that code to be executed isolated from the main operating system (OS). Therefore, in the event of total OS compromise, the application remains secure. This allows most of the operating system and application code to be separated from the critical code, responsible for handling sensitive data or that executes in an isolated environment, creating a logical barrier between them. However, there are still difficulties in running sensitive code in isolated environments.

A. Motivation

Despite the fact that ARM processors are widely available in smartphones, tablets, among other devices, ARM TrustZone functionality has not been fully explored. There are five main reasons for this.

TrustZone security mechanisms are complex. They suffer from compatibility issues between different hardware choices which require a great effort to overcome them, such as the creation of drivers for the target hardware, junction of APIs into a common API and common software development methods, to successfully minimize the impact of using different tools to achieve the same goal. Since TrustZone is rather used by the OS instead of being used by the applications themselves, although they enjoy the benefits of using TrustZone, optimizing the usage of resources by OSs is an important and ongoing research area.

Lack of good hardware documentation. Hardware manufacturers provide documentation on the hardware they sell, but sometimes it is not good enough. For developers with little or no experience, searching for answers in these documents may well be the same as trying to find a needle in a haystack. Some of these manufacturers also provide a community forum, allowing software developers to ask questions, but quite often, these questions do not receive the appropriate answer or do not get answered at all, which obliges the software developer to continue searching for solutions, wasting more time. As a consequence, these

difficulties present themselves as a major challenge for the entire software development process.

Execution environment related difficulties. There already exist some frameworks capable of using ARM TrustZone technology, such as Genode OS Framework [3]. These development frameworks require the developer to initially configure them, setting the necessary toolchain, verifying the compile targets, and so on, before they can start using them. Again, for developers with little or no experience, just to install and configure a development framework can be a very difficult step, since the developer is not familiar to these types of environments and their requirements. However, they advertise their support of a certain development board and may require additional effort in configuring the environment, to use a hardware feature that may not be fully implemented already. These constraints bring even more difficulties, mainly due to the lack of good documentation for the framework being used. In this case, the developer is faced with the problem of having to explore both the hardware documentation and the framework documentation, which can easily be overwhelming for someone with little experience. Similarly to the case above, Genode provides some documentation, but some of it does not contemplate some probable issues that may appear for hardware other than the ones they mention.

Development board related difficulties. Every development board has its own software, that was designed to work for that specific board and not for any other. This leads to more constraints for the developer, since it sometimes requires different approaches to obtain the same desired outcome. This concern escalates if the developer introduces the selection of hardware, since there are multiple choices available that allow the use of TrustZone and take advantage of its benefits. Each board may have its own way of doing the same step, such as the communication with the development host with proper software or manually, through the use of MicroSD cards, the difference in command targets and image mounting steps. These concerns lead the developer to consider aspects like compatibility between the board and the host. The availability of proper drivers that remove or diminish these constraints is not always assured.

Finally, **development boards evolution**. Some are now obsolete compared to new ones. The hardware has changed, introducing new hardware features and possibly new primitive instructions to be used. Also, their capabilities have been improved with better hardware, consequently being able to sustain more applications running, both in number and complexity, through the exploration of those added resources. This creates a gap between some of the first models used for this type of work and the new ones that are being released, that come already with changes, both in hardware and software, leaving the older models behind.

B. *Goals and Requirements*

The overall goal of this work is to conceive and implement a Software Development Kit (SDK) that eases the development process, as well as the testing of security applications for ARM TrustZone. The provided system should be compliant with the following requirements:

Ease the development process of new solutions. Instead of wasting time and resources on creating a custom platform that serves their intentions, the provided system should allow developers to jump straight to the development of their solution and start exploring the capabilities at their disposal.

Compatible implementation with API standards. Since the development of solutions for ARM TrustZone began, the interest of this type of research led entities, like GlobalPlatform, to develop specifications for projects on this matter. This non-profit organization has been developing specifications to facilitate the secure deployment and management of applications, in isolated environments. To keep the system current on this theme, this system should provide compatibility with these specifications.

Work on real hardware. This system should work on real development boards, where more and more developers may use and test it, in order to develop their own solutions. To allow such use and testing, this system will be used and tested for the NXP i.MX53 QSB board (formerly known as Freescale i.MX53 QSB) [4], since this is a regularly used development board for ARM TrustZone projects.

Efficient execution of applications. To allow developers to fully develop their idealized solutions, without having to sacrifice some of its features, the system should provide good scalability and usability. The execution of applications on top of it should present good overall performance and efficiency results.

It is intended that the system is not a final product, but instead serves as a basis on which future works might extend the current capabilities.

II. Background

This section provides some necessary background information about ARM TrustZone, GlobalPlatform API and resource management of Linux and Genode, since they are the main topics covered in this work.

A. *ARM TrustZone*

ARM TrustZone [2], [5] technology is a hardware security technology present into recent ARM processors. It consists of security extensions to a System-On-Chip (SoC), providing a security framework that allows the partitioning of both hardware and software resources. These security extensions cover processor, memory and peripherals, granting system developers the ability to execute trusted services, ranging from a simple library to a complete operating system (OS), isolated from the main OS.

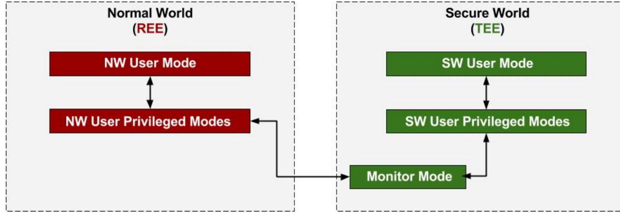

Figure 1. ARM processor modes with TrustZone security extensions.

The resource partitioning allows the coexistence of two separate worlds, *normal world* and *secure world* (see Figure 1). The normal world or Rich Execution Environment, henceforth referred to as REE, is where all user applications are installed and execute, with limited access to the device's capabilities while the secure world or Trusted Execution Environment, henceforth referred to as TEE, is where trusted applications run with full access the its resources. The memory address spaces are independent for each processor mode, creating an isolated ambient that runs parallel to the device's operating system, exploring the concept of privileged and unprivileged mode already present in earlier ARM processors. Between these processor modes lies a switch mechanism known as *monitor mode*, in which both virtual cores switch context through time slices, saving their current state in separate virtual memory, so that the normal world cannot tamper with the secure world's memory. The memory regions used for both worlds can be hard-wired or configured. This configuration must partition the resources available, so that any non-secure access, to secure memory or device, is treated as a security violation, causing an external abort to the core.

B. GlobalPlatform API

To allow system developers to build security services for ARM TrustZone, ARM has initially provided her own TrustZone API called TZAPI. The API defined the interfaces in non-secure world, being publicly available. On the contrary, the API for the secure world was private and closed, leading to it not being accepted by the security industry. As a result GlobalPlatform [6], a non-profit, member driven association , saw an opportunity along with Trusted Computing Group [7], an association formed by AMD, HP, IBM, Microsoft and Intel, are working on creating a joined TEE standard, although GlobalPlatform has its own TEE standard since 2011.

This API is divided into two parts, TEE Client API and TEE Internal API, for the normal world and secure world, respectively. The normal world application calls the TEE Client API, which then communicates with the secure world and sends relevant data. The secure world internally redirects this data to the desired trusted application, using the TEE Internal API.

C. Resource Management

In the Linux, both user and kernel memory are independent and isolated. This isolation is due to the way the address spaces are used by the system, i.e., instead of using the hardware physical memory, the address spaces are virtualized, becoming abstracted from the physical memory available. This allows many virtual address spaces to exist, each assigned to a certain process. To avoid the inefficient use of physical memory the operating system must take actions, like saving the physical memory by only loading virtual pages that are currently being used by the executing program. This process is called Paging, but in Linux it is normally called Swapping. It consists of moving a page from physical memory, that has not been used frequently or for quite some time, to a special file called the swap file. This special file is located in a slower storage device, like a hard disk. This action results in leaving the most used or frequently used pages available in physical memory. Access to the swap file is very slow, compared to the speed of both processor and physical memory.

In Genode [3], unlike Linux, the Genode OS simply arbitrates the access to such resources and allows the delegation of authority over resources between components. The low-level physical resource are represented as dedicated services, provided by the core component, which is the first user-level component, directly created by the kernel. At boot time, the core component creates an initial RAM session with a balance containing all the available physical memory for the init component, the first and only child of core. A child component does not have the authority over init's RAM session nor the RAM sessions of any siblings. Each child may continuously subdivide their budgets in order to spawn grandchildren using the same procedure as their parent did before. The opposite procedure occurs in case a certain RAM Session is closed.

III. Related Work

Several of the TrustZone-based solutions focus on assuring services with more security measures, like authentication mechanisms, one-time passwords, use of cryptology, while others focus on exploring virtualization features. Others focus on simplifying their design, reducing their trusted computing base (TCB), in turn, reducing the attack surface they face.

A. ARM TrustZone-based solutions

TrustOTP [8], a one-time password solution that focus on offering secure one-time passwords (OTP) tokens for smartphones that achieve both the flexibility of software tokens and the security of hardware tokens, through the use of ARM TrustZone. The basic concept behind OTPs is the automatic generation of numeric or alphanumeric string of characters that authenticates the user for a single

transaction or session to an authentication server, being this way resistant against replay attacks.

Trusted Language Runtime [9], [10], or TLR, is a system that protects the confidentiality and integrity of .NET mobile applications from OS security breaches, using the ARM TrustZone hardware features, to provide an isolated trusted environment. It provides runtime support for the secure component based on a .NET implementation for embedded devices, offering productivity benefits of modern high-level languages, such as strong typing and garbage collection, to application developers.

B. Development Frameworks for ARM TrustZone

The Genode OS Framework [3] is a tool kit created by Genode Labs, with the purpose of allowing the creation of operating systems with extended security measures. The architecture in which Genode was build resembles ARM platforms, which allows Genode OS to take full advantage of CPUs with TrustZone technology, at a level that allows Genode to be used as a TrustZone monitor, through a hypervisor, that leverages the protection mechanism of ARM TrustZone. This allows program execution to execute in sandbox environments, with access control execution equivalent to normal world-secure world environments, i.e., granted only the access rights and resources that it requires to fulfill its purpose. In Genode, each component has a budget of physical resources assigned by its parent, unlike traditional operating systems, which create an abstraction from physical resources. This resource management allows the use of dedicated resources by the components, that are within the designated limits, or to assign parts of their resource quota to its children. The usage and assignment of budgets is managed by each component rather than a global policy of the OS kernel. Each component can also communicate with other components and trade resources if need be, but only according to specific rules, reducing the attack surface of security-critical functions, compared to other operating systems.

IV. ARCHITECTURE

In this chapter we will go through the capabilities and design of TrustFrame. We will explore in detail the functionality of the system and its components.

As stated previously, the main goal of this system is to offer a SDK, that is easy to use for both development and testing of applications, capable of using ARM TrustZone without the need for additional effort from the developers that use the system.

In terms of requirements, this system should meet the following requirements:

1) **Ease the development process** by allowing developers to immediately start developing their solutions, saving time and effort in the process.
2) **Compatible with the GlobalPlatform API specifications** by offering new possibilities for developers that are already familiarized with the specifications and for those starting on this journey.
3) **Work on real hardware**, allowing more and more developers to use and test the developed solutions by themselves, assuming they use the same hardware, required by the system.
4) **Allow efficient execution of applications on top of it**, allowing developers to build their desired solutions, without having to sacrifice some of its capabilities, due to performance limits.

A. System Components

Throughout this document, we already discussed some of the major components of this system, namely its two OSes, Linux in the normal world and Genode OS in the secure world and the hardware in which the system runs, the NXP i.MX53 QSB. We now head into further detail around these components, as well as other details not yet fully explored, but also present as seen in Figure 2.

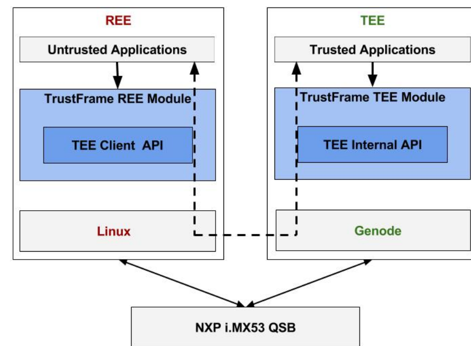

Figure 2. TrustFrame architecture overview.

In terms of execution flow, the system starts with the boot of the SoC used in this work, the NXP i.MX53 QSB. By default, the board boots in the secure world, where it will load Genode's micro kernel, containing 20KLOC, which compared to Linux's 2016 expectancy of around 20MLOC, is quite an achievement. This micro kernel was specifically developed for Genode, thereby reducing the complexity of the trusted computing base compared to other kernels. As seen in the previous chapter, Genode is a collection of small building blocks, where new add-ons can be seen as services to the kernel core. One of these services already provided is the *tz_vmm*, which is Genode's TrustZone demo. In this demo, a VM session interface is used to allow a user-level virtual machine monitor (VMM) to affect the whole state of the CPU of a non-secure virtual-machine (VM), initiate a world switch to the non-secure world, and obtain the VM's state, after an exception-triggered return. To take advantage of Genode's existing work with TrustZone, the *tz_vmm* demo was used as a foundation in this system. After Genode's init

process finishes, the *tz_vmm* service immediately prepares to boot the normal world and initiates a world switch,, where it will load a custom Linux kernel. Since Genode merges the Linux image with its own generated image, in our setup, we changed Genode to use a custom Linux image, compiled by us, instead of the original precompiled image, therefore static. This allows features to be added to the kernel, and allowed us to add our system call which is responsible for the data transfer between both worlds. Another interesting fact is that the Linux root filesystem is built using an initial RAM disk[1], which allowed us to add our binaries, that would be used in the normal world, like a client application. When Linux finishes booting, it is ready to be used, like any Linux distribution, and at this point, the developer can them call his program.

In the normal world, a client application invokes the TEE Client API, implemented by us, which in turn calls our system call, in order to transfer the client application's data. This system call is responsible for executing security verifications, creating the shared memory region, executing the *smc* instruction and finally, returning any received data back to the client application. In the secure world, each time a client application executes and a system call initiates a world switch, the *tz_vmm* receives the normal world data to be handled. It then calls our GlobalPlatform API handler, responsible for preparing the received data and redirecting the client application request to the correct function call from the TEE Internal API, implemented by us, where that data will be manipulated and a response will be prepared to be sent back to the normal world.

The client application (CA) is the normal world side of the application that will use ARM TrustZone in our system. This part focuses on calling the GlobalPlatform TEE Client API, using the specification's data types as arguments. The trusted application (TA) is the secure world side of the application that that will use ARM TrustZone in our system. This part focuses on calling the trusted services, specified by the client application. The system already has an example of a Hello World program, in which it calls the GlobalPlatform API, compatible with its specifications. This example is the combination of the client application that runs on Linux with the trusted application that runs in Genode.

B. System Details

During the initial bootstrap phase of the NXP i.MX53 QSB, our bootloader, *U-Boot*, loads the Genode kernel binary along with additional boot modules into the physical memory. These boot modules are chunks of data, like the ELF images of Genode's components: core, init, all components created by init, and the configuration of the init component. After this task finishes, *U-Boot* transfers control to the kernel, where the kernel then passes information about the physical resources and the initial system state to the core component. The core component then begins to execute its purpose, by making low-level physical resources (physical memory, processing time, device resources, initial boot modules, and protection mechanisms) of the SoC available to other components in the form of services. Then, it creates the init component, using its own services, and delegates all physical resources, and control over the execution of all subsequent component nodes, which can be further instances of init.

[1]Initial RAM disk (initrd) is an initial root file system that is mounted prior to when the real root file system is available.

In terms of memory layout, the development board used in this work contains two memory banks of 512MB each, making a total of 1GB. During the development of this work, it was noted that only one of those memory banks was in fact mapped by Genode, namely bank 0. Although we do not know why only one of the memory banks was mapped, we suppose that it may have been due to the lack of need, given by the simplicity and low memory requirements needed by the Linux kernel used, as well as Genode itself. The used memory bank goes from address 0x70000000 to 0x90000000, and is divided in half for each world, i.e., the memory addresses from 0x70000000 to 0x7FFFFFFF are mapped as secure and the memory addresses from 0x80000000 to 0x8FFFFFFF are mapped as non-secure. This memory mapping results in the assignment of 256MB for the secure world and 256MB for the normal world. Figure 3 illustrates the memory layout described above.

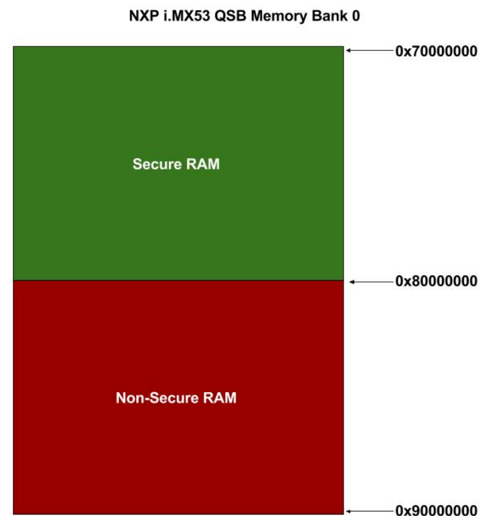

Figure 3. Memory mapping for the i.MX53 QSB bank 0.

V. IMPLEMENTATION

This section provides in-depth details over the implementation for this work. We will explore details concerning both the normal and secure world.

A. Normal World Details

If a normal Linux kernel booted without some modifications, the system would hang during boot. These modifications are required, in order to avoid access permissions issues that cause the system to hang. These issues would occur by executing sensitive instructions in some kernel component drivers like IPU, GPIO, as well as trying to access other peripherals, deemed critical. To allow the kernel to boot without these issues, the Genode staff had to modify these drivers, introducing hypercalls that occur whenever the Linux kernel tries to access a device, which is not assigned to the normal world. When this happens, an external data abort occurs, and the control is passed to the secure world, so that the device may be emulated. The basic idea of emulating device access is allow the hypervisor to pass control to the TrustZone virtual machine monitor (VMM) as soon as the non-secure OS accesses an address outside the allowed physical address boundaries. Afterwards, the VMM may then inspect the address and the program counter of the non-secure OS that raised the access violation. Based on the program counter value, the VMM can then fetch the faulting instruction, decode it and emulate it in software, bypassing the issue that did not allow the Linux kernel to properly boot.

Since we could now add functionalities to our kernel and see them reflected immediately after booting both Genode and Linux, we created a custom system call. Our system call allowed us to immediately switch from the normal world to the secure world when necessary. The major benefit was the simplicity introduced, since instead of having to replicate the same assembly instructions, over and over again, we only needed to use a single line of code to call the system call (see Figure 4).

```
// SMC System Call
SMC_SystemCall(INVOKE_COMMAND, args);
```

Figure 4. Function call for the SMC system call.

Considering we could now exchange data between both worlds, we started to implement the GlobalPlatform APIs for both worlds. We began by the normal world, where we researched how other frameworks implemented the API specifications according to their own concept. This research was done, since the client side of the API would surely be similar, due to its independence from the normal world OS and also to understand better how the specifications could be applied.

Since we needed to pass the data from the client application, through the system call, to be sent to the secure world, we opted to create a data container. This container contains all the arguments and parameters necessary for the secure world to manipulate them and return an accurate response. We followed the same principal for each function that comprises the TEE Client API. Each function executes the following tasks: receive the client application's arguments, store those arguments in the data container, call our system call and wait for a response. All the received arguments are stored into the data container, separated by certain special characters, that work as delimiters. These characters exist to allow the secure world to parse the entire container and, based on the character encountered, assign each piece of data to the correct destination. When the secure world receives the whole container, it will need to assign the correct parameters to the correct session and so on, which it does using this method, achieving an accurate assignment of the received data. After the API function finishes preparing the arguments, it calls our system call and transfers the prepared data container to the system call. Since the system call is located into the kernel space, it allows us to retrieve the process identifier that executed the program. The system call then appends that identifier to the received container and stores the data in the shared memory region. This step is always done to allow the secure world to differentiate between two processes that use the same parameters, like sessions identifiers and such values, that could normally overcome isolation barriers.

B. Secure World Details

Following the implementation of the normal world details, we started to prepare Genode for our requirements. We began by looking into the *tz_vmm* component, in order to understand how we could receive the normal world data container and use it in the secure world. We created a simple secure world handler that is called each time Genode's interrupt handler receives a specific argument, defined by us. This argument, as referred previously, is also sent in one register, which will allow identifying what to do when the argument is received. Since Genode already contains three defined behaviors for different types of interrupts, ours was the fourth, hence the reason the number four is sent in a register. This allows Genode to redirect the interrupt from our *smc* to our handler. This handler will then retrieve the arguments passed in the other registers, map the shared memory region using the received address with the size of the region and extract the data that was stored in it. From this point forward, our secure world handler contains the data sent from the normal world and is able to pass it down to our GlobalPlatform API handler, serving as a bridge between the normal world and our GlobalPlatform API handler.

After we retrieve the data, we must then call our GlobalPlatform API handler. This handler was developed as a service to Genode, just like *tz_vmm* is, which allowed us to call it using Genode's RPC interface, adapted for our needs. Using this interface, we can pass the normal world data to the handler, which then reverse engineers the process executed by the TEE Client API. By detecting the special characters sent along with the data, the handler is able to

partition the received data container and assign the data correctly. This allows it to identify which operation was executed in the normal world and, based on it, execute the respective TEE Internal API function, passing the necessary arguments. Besides passing the necessary arguments, this handler is responsible for handling data updates. These updates occur after a secure world service manipulates the data it receives, and so, this handler ensures an overall synchronization between all the secure world components that must deal with the data. In the end, the *tz_vmm* component retrieves the latest data updates and sends them back as a response, to the normal world.

As for the implementation of trusted services, a new developer may replace the single source file, that contains our examples of trusted services, by his own, provided that he adds the special function, containing the redirection to his trusted services. By looking at our example, he can easily replicate the redirection mechanism. This special function is required, since it is called by the *TA_InvokeCommandEntryPoint* function, from our TEE Internal API implementation. The secure world developer may even add his services to the existing ones, and add a redirection entry to the special function, without replacing the provided source file, further reducing his workload and, possibly, allowing him to easily test his solution, without having to worry about other details, like calling the special function correctly or know which Genode libraries he must use besides the ones needed for his services.

Our Hello World example executes three different trusted services, each called after the other. The first service executed is the sample we encountered, which receives a certain value from the normal world CA and increments it, returning the new value back to the CA. The second service is similar to the first, but instead of a value, returns a date. The third service uses a different data type, instead of the previous value type. This service requires the CA to prepare a data buffer, containing the desired data, which in this example is a string. This data buffer is then sent the same way to the secure world, where the trusted service receives the string and responds with another string, similar to a TCP/IP acknowledge response.

VI. Evaluation

In this section we provide our evaluation methodology and setup, as well as the results for our prototype.

A. Methodology

Since our solution uses ARM TrustZone, we sought to measure the execution times in each world, along with time measures of our system call, in order to understand the impact it has on the performance of the overall system. We divided these measurements into different sets:

Execution of services in the normal world. In this test we limit our measurements to the normal world. We execute the client application, but instead of invoking the TEE Client API, consequently calling our system call, we simply execute the services solely in the normal world.

Execution of normal world services with secure world invocations. We continued to measure the three services in the normal world. The difference in this test is the invocation the GlobalPlatform TEE Client API functions that will always be executed in every client application, namely the pair `TEEC InitializeContext` and `TEEC FinalizeContext` functions and the pair `TEEC OpenSession` and `TEEC CloseSession` functions. Since every client application must invoke these functions, we measured what impact would the invocation of those functions have, along with the three services, running solely in the normal world.

Execution of secure world services. Afterwards, instead of executing the service in the normal world, like the previous test, we execute it in the secure world, by invoking the `TEEC InvokeCommand` function in the client application.

Execution of the entire system. Finally, we measured the performance of the entire system, from point to point. Starting from the beginning of the client application execution in the normal world, consequently calling our system call, executing commands in the secure world and receiving the response. Each service is measured separately, along with the invocation of the GlobalPlatform API in both worlds.

As for the objective of these tests setups, we aimed at measuring the execution time of:

Every service in the normal world, with no invocations to the secure world.

Each service in the normal world, plus the invocation of the GlobalPlatform API functions to the secure world, without accounting for the overhead introduced by setting and getting the updated data.

The invocation of the GlobalPlatform API functions to the secure world, with the desired service being called in the secure world, without accounting for the overhead introduced by setting and getting the updated data.

Each service in the secure world, plus the invocation costs and the overhead introduced by setting and getting the updated data.

B. System Overhead

The overhead introduced by the system was calculated by removing the service execution time average from the overall time average. Throughout the testing of the system, we noticed that the invocation of the `TEEC InitializeContext` and `TEEC FinalizeContext` functions had minimal impact on overall performance, with estimates of two nanoseconds. The same was observed from the invocation of both the `TEEC OpenSession` and `TEEC CloseSession` functions, with averages between one and two nanoseconds. The biggest time contribution belongs to the `TEEC InvokeCommand` function, mainly due to the

fact that when this function is called and all the data is sent, it must be parsed in the secure world, to be assigned and used accordingly. The measurements for each of these functions already contain the execution costs of the corresponding functions of the secure world, i.e., from the TEE Internal API, due to the small added cost implied.

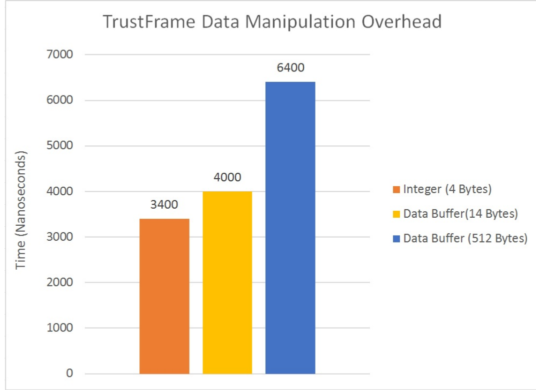

Figure 5. Time overhead added for different amounts of manipulated data.

For the manipulation by value, as used when executing the increment value service, we observed an average overhead time of 3400 nanoseconds. For the data buffer manipulation, the overhead varied based on the amount of data bytes being manipulated. To measure the variation, we tested with different amounts of bytes being manipulated. The first two measurements contained 8 and 14 bytes of data, presenting an average overhead of 4000 nanoseconds, while the third measurement contained 512 bytes of data being manipulated, presenting an average overhead of 6300 nanoseconds. The biggest time contribution for these overhead results is the cost of updating and retrieving the GlobalPlatform parameters, manipulated by the services, along with the data exchange between the *tz_vmm* component and our GlobalPlatform module. Each service must call the appropriate parameter updater, based on the type of data the service manipulated, i.e., by value, like in our increment service, or through the data buffer like our string copy service. Before setting and retrieving the updated data, a cycle is executed, based on the normal world process and session identifier, along with the secure world session identifier, assigned when a new session is opened. This cycle is responsible for providing a considerable contribution to the final overhead. If the service manipulated the data buffer parameter, a small amount of time is added to the overhead, due to the use of an extra *memcpy* function, that is only used for this type of parameters. Figure 5 contains the introduced overhead for the different types of data manipulation used in the services, described above.

The overhead added by our system was already expected. This is due to the fact that, instead of only copying the same code from the normal world to the secure world, we also added more code that must execute before and after the service executes. This is required, not only due to our implementation of the GlobalPlatform specifications, but also due to the requirements of Genode's RPC service. Even though the system overhead brings the nanosecond execution time of both increment value service and print date service to the microsecond execution time, using the system, the added overhead is imperceptible to the developer, since the time scale is still at a few microseconds, which is still very fast. In the data buffer manipulation overhead, although the amount of bytes changed from 14 bytes to 512 bytes, around 37 times more, the overhead variation was roughly 1,6 times, which is insignificant at this scale of time.

C. Service Comparison

Figure 6. Increment service performance results in both worlds.

The increment service consisted into incrementing the received value. Since this is a simple task, it was executed several times, as stated previously. An overview of these results can be seen in Figure 6, where we partition the time costs described above.

We started by measuring the service running isolated in the normal world, which required 19 nanoseconds, on average, to execute. We then measured the costs of invoking the GlobalPlatform API functions, at the same time the service was being executed. Since the `TEEC InitializeContext` and `TEEC FinalizeContext` functions and the `TEEC OpenSession` and `TEEC CloseSession` functions must always execute one time only, we measured their invocation cost, along with the increment service, which resulted in an additional cost of three nanoseconds, on average, to the previous 19 nanoseconds, totalling 22 nanoseconds. We then measured the invocation of all the GlobalPlatform API functions, first while running the service in the secure world and then without running the service at all. This allowed us to retrieve the just the cost of invoking the `TEEC InvokeCommand` function, where we obtained and average of 126 nanoseconds, totalling an average of 148 nanoseconds for this service.

Afterwards, we measured the service executing solely in the secure world, where we isolated the cost of invoking the GlobalPlatform API functions. We obtained the same time average of three nanoseconds, for the all the GlobalPlatform API functions, excluding the `TEEC InvokeCommand`. For the command invocation, the measured an average of 129 nanoseconds and for the service in the secure world we measured an average of 21 nanoseconds, totalling an average of 153 nanoseconds, a slight difference from the previous test, due to small error margins.

The string copy service consisted into copying a received string, which could be used in the secure world, for any means, and copying a response string, to be sent back to the normal world. For this service we executed the same tests, but using different payloads, to ascertain the performance costs of increasing the amount of data manipulated in the secure world. We measured the service using 8 bytes, 14 bytes and 512 bytes. The major differences between this service and the remaining two are that this service manipulates data from a data buffer parameter, specified by the GlobalPlatform API standards. We used the *memcpy* function more often, as a solution, which brought an increase in the time average, compared to the previous services.

In terms of the service execution isolated in the normal world, using the service to copy a total of 8 bytes resulted in an average execution time of 2049 nanoseconds. The same service copying a total of 14 bytes only took around 1,17 times more, about 2405 nanoseconds, although the amount of data increased 1,75 times. As for the final test, copying 512 bytes, the average time was 2863 nanoseconds, which is an increase of roughly 1,19 times the execution time of the same service copying 14 bytes of data, even though the amount of data increased about 36.5 times.

When executing the same service in the secure world, using the service to copy 8 bytes of data resulted in an average time of 241 nanoseconds for the invocation of the GlobalPlatform API functions and an average of 2177 nanoseconds, totalling 2418 nanoseconds, on average. When using 14 bytes of data with this service, the execution time obtained, from the invocation of the GlobalPlatform API functions, was 327 nanoseconds and for the service was 2232, totalling 2559 nanoseconds, on average. Finally, when using 512 bytes of data with this service, the average execution time for the invocation of the GlobalPlatform API functions was 671, while the service only took 2687 nanoseconds to execute, totalling 3358 nanoseconds, on average.

The difference in the execution time of the service isolated, is due to the observed standard deviation of 190 nanoseconds, over the service's execution time, which slightly changes the ratios calculated above. The time difference between executing the service, using 14 bytes and 8 bytes, only varies roughly 1,05 times, while the time difference increases 1,31 times, when using 512 bytes of data. Figure 7, contains an overview of these results.

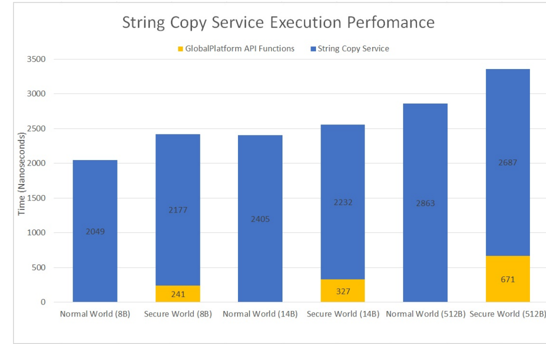

Figure 7. String copy service performance results in the normal worlds.

D. Service Overview

The total execution times for the increment service and print date service remain with minimal differences, due to the fact that they manipulate the received values from the parameter structure of the GlobalPlatform specifications in the same way. For the increment service, the average total time is 3528 nanoseconds, while the average total time for the print date service is 3558 nanoseconds. As for the string copy service, the introduced overhead varies, based on the amount of data being manipulated. For the manipulation of 8 bytes of data, the average total time is 6272 nanoseconds, while for the manipulation of 14 bytes, the total time slightly increases to 6375 nanoseconds, on average. Finally, for the last scenario, the total time of the string copy service, while manipulating 512 bytes of data is 9711 nanoseconds, on average. Figure 8 contains the total execution time for each service.

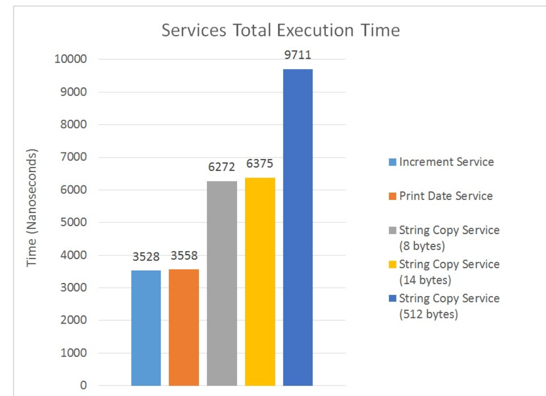

Figure 8. Total execution performance for the provided services.

Although the amount of data manipulated varied considerably between each service, as well as the way our GlobalPlatform module handled the data, the difference in the total time, did not escalate proportionally to the increase in the amount of data being handled. These results show

that our system provides very good scalability, even if larger amounts of data are used within our module, as seen by the results of varying from a simple integer of 4 bytes, being incremented, to a 512 bytes string, being copied.

VII. Conclusion

Some of the presented works related to ARM TrustZone technology focus on providing small, but trusted, services that run in the secure world, isolated from a traditional operating system running in the normal world. These services range from authentication mechanisms, one-time passwords, use of cryptology or virtualization features to reduce their attack surface by minimizing the trusted computing base. Other solutions led to the creation of development frameworks, capable of exploring TrustZone features. However, there are still difficulties in using such platforms, due to the lack of support or good documentation, to the lack of availability of their source code.

In this work, we proposed the development of TrustFrame, a toolkit capable of allowing the development of new TrustZone-based solutions, that could ease the difficulties of developers and serve as a basis for new TrustZone projects in the future. We based our solution on Genode, an existing development framework capable of using ARM TrustZone, where we combined the use of a supported and easily available framework, with the API standards devised by GlobalPlatform, an organization that has been dedicating itself into the development of standardized specifications for trusted execution environments, where trusted solutions execute. Our system was tested using three different types of services, as a proof of concept, with different complexity levels among them.

By analyzing our evaluation results, we realized that, although our system introduced some overhead in the overall performance, that impact is still minimal and insignificant to developers and their solutions. The devised services remained with high speed performance results, while running in the secure world and using the GlobalPlatform API specifications.

A. Future Work

As for future work, the developed solution enables many possibilities. These include the following:

Extend the GlobalPlatform API. Since not all the API specifications were implemented during this work, it is still possible to extend the current implementation and add new functionalities. The GlobalPlatform specifications account for the use of cryptography, arithmetic, trusted storage, as well as other APIs, which were not implemented in this work, leaving room for future work.

Optimize the GlobalPlatform module. During the evaluation tests, it was noted that the system introduced an overhead to the overall execution of a service. Optimizing the data flow and reducing the copy of data, within this module, would surely have an increase in the overall performance results.

Code production automation. During the development of our solution, we noticed that the client application contained duplicated source code, for example, when calling the `TEEC InvokeCommand` function. Automation of the generation of duplicated code, similar to the RPC systems, would benefit future development projects, ultimately aiding the developer.

Increase compatibility and portability of the system. By using Genode, this platform is a candidate for other development boards, since Genode itself supports other development boards. Extending the support to other development boards would surely reach broader audiences and increase interest in this type of work. Although we already support Linux as a rich OS, supporting more REE OSes, like Android, would open new possibilities for more future work. Finally, use of a new toolchain, compatible with this work, could provide better access to the toolchain and further support. This way, issues, like the ones described earlier, that led to compilation and availability problems, making it hard to find the appropriate toolchain, would be mitigated.

References

[1] Educause, "7 Things You Should Know About Mobile Security," pp. 1–2, 2011.

[2] Arm, "ARM Security Technology. Building a Secure System using TrustZone Technology," p. 108, 2009.

[3] N. Feske, "Genode Operating System Framework 16.05."

[4] N. Semiconductor, "i.MX53 Quick Start-R Board."

[5] J. Winter, "Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms," *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, pp. 21–30, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1456455.1456460\nhttp://doi.acm.or

[6] GlobalPlatform, http://globalplatform.org/, 2016, [Online; accessed 17-October-2016].

[7] T. C. Group, http://www.trustedcomputinggroup.org/, 2016, [Online; accessed 17-October-2016].

[8] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP : Transforming Smartphones into Secure One-Time Password Tokens," *Ccs*, pp. 976–988, 2015.

[9] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM trustzone to build a trusted language runtime for mobile applications," *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*, no. i, pp. 67–80, 2014.

[10] ——, "Trusted language runtime (TLR): enabling trusted applications on smartphones," *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, pp. 21–26, 2011.